



Architectural Design and Deployment Considerations for a Three-Tier Browser-Based Intrusion Detection System

Oluwasegun Godwin Bamisaye

Department of Cyber Security, Faculty of Computing, Air Force Institute of Technology, Kaduna, Nigeria

Corresponding Author Email: godwinbamisaye3@gmail.com

ORCID: <https://orcid.org/0009-0001-5429-5874>

Osichinaka Chiedu Ubadike

Department of Aerospace Engineering, Faculty of Air Engineering
Air Force Institute of Technology, Kaduna, Nigeria

Email: oc.ubadike@afit.edu.ng

ORCID: <https://orcid.org/0000-0002-5186-0911>

Freeman Bitrus

Department of Cyber Security, Faculty of Computing, Air Force Institute of Technology, Kaduna, Nigeria

Email: freemanbitrus70@gmail.com

ORCID: <https://orcid.org/0009-0005-9964-5128>

Samson Adeyinka

Department of Cyber Security, Faculty of Computing, Air Force Institute of Technology, Kaduna, Nigeria

Email: samson.adeyinka@afit.edu.ng

ORCID: <https://orcid.org/0009-0005-9361-7337>

and

Adeniran Kolade Ademuwagun

Department of Cyber Security, Faculty of Computing, Air Force Institute of Technology, Kaduna, Nigeria

Email: kolade.ademuwagun@afit.edu.ng

ORCID: <https://orcid.org/0000-0002-0206-2120>

Abstract

Browser-based intrusion detection is commonly presented in the literature as a machine learning problem, with emphasis placed on classifier accuracy, precision, and recall. This paper instead examined it as a software engineering and deployment challenge. Specifically, it investigated the architectural decisions required to implement a browser-based intrusion detection system that is maintainable, scalable, and compatible with the operational constraints of modern browser extension platforms. Using a three-tier browser-based intrusion detection framework as a case study, the paper analysed the engineering decisions made across the Browser Extension Layer, Backend Intelligence Layer, and Dashboard and Storage Layer. At the client side, the study discusses how the Chrome Manifest V3 service-worker execution model influences state management, permission scoping, and resource utilization. At the backend, it examined request handling, machine learning model loading, and stateless API design to support efficient inference and horizontal scalability. The persistence layer is analysed with respect to database schema design, concurrent write handling, and audit-trail integrity using PostgreSQL. Rather than evaluating detection performance, the paper focused on deployment architecture, separation of concerns, scalability, and maintainability, highlighting the trade-offs associated with each engineering decision. The analysis identified practical deployment challenges, including ephemeral service-worker state, per-process machine learning model memory overhead, and development-specific configuration constraints, that must be addressed before a prototype can be deployed in production. By documenting these implementation-level decisions and their operational implications, the paper provided a practical reference for researchers and software engineers designing browser-native intrusion detection systems and contributes to the relatively limited literature on end-to-end architectural design for browser-based security frameworks.

Keywords: Service worker constraints, browser-based security tooling, Browser-based intrusion detection, Chrome Extension, Manifest V3, Intrusion Detection System (IDS), Real-time security monitoring, Software architecture.

Introduction

Most published work on browser-level intrusion detection foregrounds the machine learning model at its core, comparing classifiers on accuracy, precision, or recall against a labelled dataset. That framing answers an important question, whether a given model can distinguish malicious from benign flows, but it leaves a different, equally consequential question mostly unanswered: what does it actually take to run such a model as part of a working system inside a real browser, talking to a real backend, over a real network, in a way that a development or security team could deploy and maintain? Systematic reviews of the intrusion detection literature have repeatedly observed that continuous, real-time monitoring paired with disciplined feature engineering is a baseline requirement for deployable systems, not a detail to be resolved after the modelling work is done (Arnob *et al.*, 2025), and separate surveys note that the literature remains comparatively thin on end-to-end, browser-native architectures, with most published work concentrated at the level of an isolated subsystem or algorithm (Hozouri *et al.*, 2025).

This gap matters because the browser extension environment imposes constraints that have little to do with model accuracy and everything to do with software engineering. Manifest V3, the current Chrome extension platform, replaced long-lived background pages with event-driven service workers that can be terminated by the browser at any time and must be prepared to resume without relying on in-memory state persisting between events. An extension's webRequest observation logic, its API request handling, and any local caching it performs must all be designed around this lifecycle rather than around the assumption of a continuously running process. On the server side, a backend that must apply machine learning inference to a continuous stream of browser-originated requests has to make deliberate choices about where and how models are loaded, how requests are handled statelessly enough to scale horizontally, and how detected events are persisted in a way that remains both auditable and efficient to write under concurrent load. None of these decisions show up in a confusion matrix, yet each one determines whether the resulting system can survive contact with a real deployment. Prior evaluations of browser-extension-based security tools have flagged exactly this kind of engineering gap in practice: extension-based phishing detectors relying on continuous, client-side analysis have reported measurable browser performance degradation under sustained load (Prasad *et al.*, 2024), and heavier server-side ensemble architectures have been observed to introduce latency instability once deployed at production scale (Almuhanna & Dardouri, 2025), a symptom, in both cases, of engineering decisions rather than modelling ones. This paper addresses that gap directly by using a working three-tier browser-based intrusion detection framework as a case study and asking a specifically architectural research question: what engineering decisions, at the extension, backend, and persistence tiers, are

required to make a browser-based intrusion detection system maintainable, horizontally scalable, and compatible with the operational constraints of a modern browser extension platform? The machine learning components embedded in the framework are treated here purely as one module within a larger system; their classification performance is out of scope for this paper and is addressed elsewhere in this research programme.

Related Literature

Browser Extension Platforms as an Engineering Constraint

Research examining the browser extension ecosystem has increasingly treated the extension platform itself, rather than any specific detection technique, as a subject worth studying. Surveys of the malicious browser extension threat landscape document how extension permission models, update mechanisms, and store review processes shape what an extension can and cannot safely be trusted to do (Singh *et al.*, 2025), while analyses of covert data-collecting extensions show that even legitimate-looking client-side code can misuse broad host permissions if the extension's own architecture is not deliberately scoped (Vimal *et al.*, 2024). These findings motivate treating permission scoping and execution-model compliance as first-order engineering concerns for any security-purposed extension, this framework's own extension included, rather than as incidental implementation detail.

Browser Extensions as Security Tooling Delivery Mechanisms

Several studies have used the browser extension as the delivery mechanism for a security capability. Machine-learning-based phishing extensions performing real-time URL analysis have demonstrated that client-side inference is technically feasible, but have also reported that sustained, high-throughput analysis inside the extension process measurably affects browser performance (Prasad *et al.*, 2024), an outcome consistent with the general principle that computationally heavier logic is better placed in a backend service than inside the extension itself. Malware-detection extensions built around embedded machine learning classifiers extend this delivery pattern to a broader class of threats (Sriram *et al.*, 2025). A related architectural pattern pairs a lightweight browser extension acting as a sensor with a separate backend service performing the heavier analytical and risk-scoring work (Xue *et al.*, 2024), a division of labour that closely parallels the three-tier separation examined in this paper.

Backend Architectures for Real-Time Security Analytics

Systematic reviews of intrusion detection systems converge on continuous, low-latency, real-time monitoring as a non-negotiable requirement for deployable

systems (Arnob *et al.*, 2025), but comparatively few studies examine how that requirement should shape backend software architecture specifically. Work on ensemble-based network intrusion detection has shown that heavier server-side model architectures can introduce latency instability once exposed to production-scale traffic volumes (Almuhanna & Dardouri, 2025), underlining those decisions about model loading, request handling, and computational staging carry real operational consequences distinct from a model's offline accuracy. Separate research into the interpretability of machine-learning-based intrusion detection systems has identified a persistent tension between explainability and computational overhead; more transparent models tend to demand more processing resources, a trade-off that has direct implications for how such models should be deployed and staged within a latency-sensitive backend (Mohale & Obagbuwa, 2025). In-browser, vision-based defences against page manipulation attacks further illustrate that browser-native, real-time architectures are an active area of systems research in their own right, not merely a deployment target for models developed independently of the browser context (King *et al.*, 2025).

Research Gap

Across this body of work, detection technique and extension use-case are described in reasonable depth, but the specific engineering mechanics of running a three-tier, browser-to-backend-to-database security pipeline, service worker lifecycle handling, model-loading strategy, and schema design for audit logging are rarely documented as a subject in their own right. Surveys note the shortage of end-to-end, browser-native architectures in the literature (Hozouri *et al.*, 2025), but this shortage extends specifically to systems-level treatment: papers that do describe end-to-end pipelines typically do so in service of reporting detection metrics, with the architectural decisions themselves treated as implementation footnotes rather than analysed for their deployment consequences. This paper addresses that gap by making the architecture, and the reasoning behind its specific engineering decisions, the primary subject of study.

System Architecture Overview

The framework under study is organized as three tiers: a Browser Extension Layer responsible for client-side observation and metadata capture, a Backend Intelligence Layer responsible for preprocessing and inference, and a Dashboard and Storage Layer responsible for persistence and administrative oversight. This separation of concerns is itself an engineering decision, it keeps the browser-resident component, the tier most exposed to user-facing performance costs, as small and simple as possible, while concentrating heavier computation and stateful storage in server-side components that can be scaled,

monitored, and updated independently of the client. The remainder of this paper examines each tier in turn, focusing specifically on the implementation-level decisions that determine whether the architecture is deployable in practice.



Fig. 1. Three-tier architecture: the Browser Extension Layer communicates with the Backend Intelligence Layer over a metadata-only REST/JSON API, which persists results to the Dashboard and Storage Layer via ORM/SQL.

Browser Extension Layer: Engineering Constraints and Decisions

The Manifest V3 Service-Worker Constraint

The extension is built against Manifest V3, Chrome's current extension platform, which replaced the persistent background pages of earlier extension versions with event-driven service workers. A service worker is not guaranteed to remain resident in memory; the browser may terminate it between events and restart it on demand, which means any state held only in JavaScript variables during the worker's lifetime can be lost without warning. This has direct consequences for a security extension whose core task is tracking in-flight network flows, since a flow's start time and identifying details must be available when that flow later completes, potentially after the service worker has been recycled. The extension's network-monitoring logic addresses this by keeping in-flight flow state in a lightweight in-memory structure keyed by request identifier for the (typically short) duration of a single request-response cycle, while using `chrome.storage.local`, a persistence mechanism that survives service-worker restarts, for anything that must outlive an individual flow, specifically, the rolling list of recent alerts surfaced to the user.

Bounded Local Storage as a Resource-Management Decision

Because `chrome.storage.local` persists across service-worker restarts, it is a natural place to cache alerts, but it is not unlimited, and an extension that appends to it indefinitely risks unbounded growth over a long browsing session. The extension addresses this with an explicit cap on the number of retained alert records, discarding the oldest entries once the cap is reached. This is a deliberate resource-management decision rather than an incidental implementation detail: it bounds the extension's storage footprint regardless of how long the browser session runs or how many alerts are generated, which matters for an extension intended to run continuously in the background of ordinary browsing.

Permission Scoping and the `webRequest/declarativeNetRequest` Split

The extension requests two distinct network-related capabilities: the observational `webRequest` API, used to detect when requests begin and complete so that flow-level timing and counts can be derived, and the `declarativeNetRequest` API, used to enforce a small, pre-declared set of blocking rules against known-bad domains without the extension needing to inspect or act on the content of every request in real time. This split reflects a meaningful engineering distinction: `webRequest` observation is used strictly for timing and metadata capture, while blocking decisions that do not depend on behavioral analysis are delegated to the declarative rule engine, which Chrome can evaluate more efficiently than equivalent imperative logic running inside the extension's own event handlers. Host permissions are similarly scoped to the origins the extension actually needs to reach, the browsing target and the local backend endpoint, rather than requested more broadly than the extension's function requires, consistent with the permission-minimization concerns raised in studies of browser extension trustworthiness (Singh *et al.*, 2025; Vimal *et al.*, 2024).

Delivering Telemetry from an Ephemeral Process

Once a flow completes and its metadata has been assembled, the extension transmits it to the backend using the `fetch` API with the `keepalive` flag set. This is a specific concession to the service-worker lifecycle: a standard fetch request can be interrupted if the initiating context is torn down before the request completes, and `keepalive` instructs the browser to attempt to complete the request even if the service worker that initiated it is subsequently terminated. Combined with a high scheduling priority on the request, this choice reflects a deliberate trade-off in favour of telemetry delivery reliability over strict adherence to the service worker's nominal execution window.

Table 1. Extension-Layer Engineering Decisions

Constraint	Mechanism Used	Rationale
Ephemeral service-worker lifecycle	In-memory flow map keyed by request ID	State only needs to survive a single request/response cycle
State must outlive worker restarts	<code>chrome.storage.local</code> with a 25-alert cap	Persists across restarts while bounding storage growth
Efficient blocking vs. observation	<code>webRequest</code> (timing) / <code>declarativeNetRequest</code> (blocking)	Avoids imperative content inspection for known-bad domains
Reliable telemetry delivery	<code>fetch()</code> with <code>keepalive</code> + high priority	Request survives worker termination mid-flight

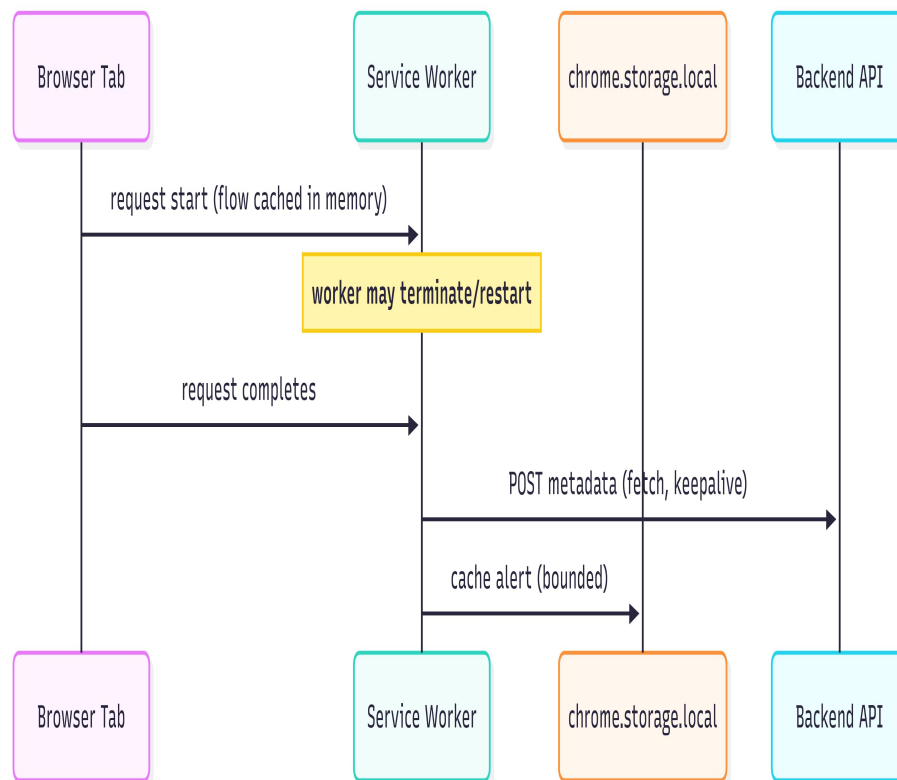


Fig. 2. Service-worker lifecycle across a flow's start and completion events, showing the risk of state loss and the role of chrome.storage.local.

Backend Intelligence Layer: Engineering Decisions

Framework Selection and API Design

The backend is built on Django, exposing its detection endpoint through Django REST Framework's class-based `APIView` pattern. This choice reflects a pragmatic engineering trade-off rather than a performance-driven one: Django's batteries-included design provides request routing, serialization, and, critically, a ready-made administrative interface that the Dashboard and Storage Layer reuses directly, which meaningfully reduces the engineering effort required to stand up a working system compared to assembling an equivalent stack from smaller, more specialized components. The detection endpoint itself is designed to be stateless at the request level: each incoming request carries the complete set of behavioral metadata needed to produce a classification, and the view does not depend on any server-side session state associated with the requesting browser. This statelessness is what allows the endpoint, in principle, to be scaled horizontally behind a load

balancer without requiring sticky sessions or shared in-memory session stores between backend instances.

Model Loading Strategy and its Memory Implications

The trained classification and anomaly-scoring models are deserialized with Joblib and loaded as class-level attributes on the detection view, meaning they are loaded once when the backend process starts rather than being reloaded on every incoming request. This is an important latency decision, avoiding per-request model-loading overhead, but it is not without cost: because the models are bound to the process rather than shared externally, each backend worker process running behind a production application server (for example, multiple Gunicorn or uWSGI workers serving the same Django application) holds its own independent copy of both models in memory. This is a direct and quantifiable trade-off between inference latency and per-process memory footprint, and it is one that must be explicitly accounted for when deciding how many backend worker processes a given deployment can run concurrently on a given host. Deployments that need to scale request throughput by adding more worker processes will proportionally increase total model memory consumption, a constraint that does not appear at all in an offline evaluation of the model's classification accuracy but is central to whether the backend can actually be scaled out in production.

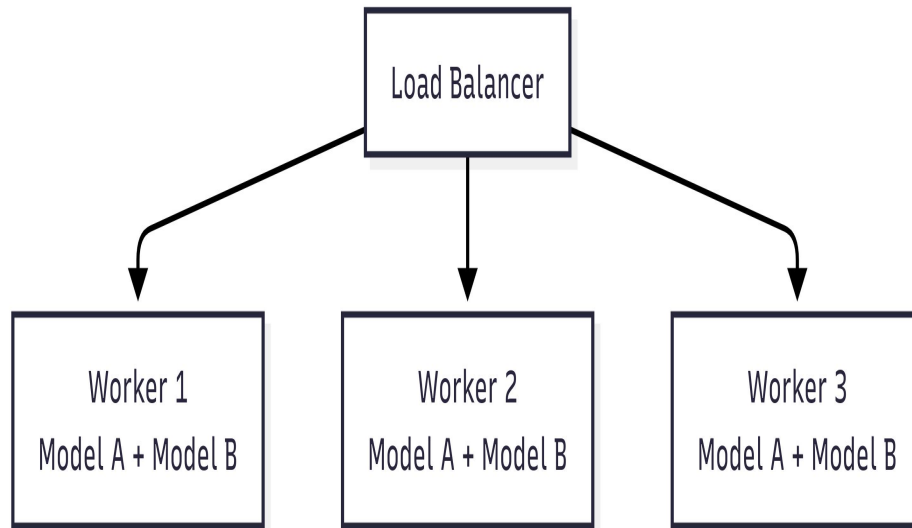


Fig. 3. Each backend worker process independently preloads its own copy of both models, trading per-process memory for lower inference latency.

Staged Detection as a Latency-Reduction Pattern

The detection pipeline is structured in sequential stages: a fast heuristic check runs first, matching backend requests against unambiguous, well-known attack markers, and only requests that are not resolved by this fast path are passed on to the

machine learning models. This staging is an engineering pattern as much as a detection strategy, it exists specifically to avoid invoking the two heavier model-inference calls for the subset of requests whose disposition is already unambiguous, reserving that computational cost for the cases that actually require it. This kind of short-circuit evaluation is a standard latency-reduction technique in request-handling pipelines generally, applied here in a security-specific context.

Resilience to Malformed Input

Because the backend's detection endpoint receives data from a client process, the browser extension, that it does not fully control, the request-handling logic is wrapped in explicit exception handling that returns a structured error response rather than allowing an unexpected input shape to raise an unhandled exception. This is a modest but necessary piece of defensive engineering: a backend that crashes or hangs on malformed telemetry from a single misbehaving or outdated extension instance would represent a much more serious deployment risk than a single dropped detection event.

Dashboard and Storage Layer: Schema and Deployment Considerations

Minimal Schema as a Design Decision

Detected anomalies are persisted through a dedicated Django model whose fields are deliberately narrow: a timestamp, the source URL, the prediction outcome, a threat label, a numeric anomaly score, and the small set of behavioral metadata values that produced that outcome. No field in this schema is capable of holding page content, form data, or any other payload-derived value, which means the persistence layer's privacy posture is enforced structurally, at the schema level, rather than relying on downstream policy to avoid writing sensitive fields that the schema would otherwise permit. This is a meaningful engineering decision distinct from, though complementary to, the extension and backend's avoidance of payload access in the first place: even if a future change to the detection logic attempted to pass additional data through to the logging call, the schema itself would need to be explicitly altered to accommodate it, which introduces a natural review point.

Database Choice and Concurrent Write Handling

The persistence layer uses PostgreSQL rather than a lighter-weight embedded database. This choice is directly tied to the deployment model: because anomaly records can be written concurrently by multiple backend worker processes handling requests from many browser clients simultaneously, the storage layer needs robust support for concurrent writes and transactional integrity, properties that a production relational database provides more reliably under concurrent load than an

embedded, single-writer alternative. For a system whose stored records function as a security audit trail, transactional guarantees are not a convenience but a requirement, an inconsistent or corrupted anomaly log would undermine the dashboard's usefulness for incident investigation.

Administrative Dashboard as an Engineering Reuse Decision

The administrative dashboard is built on Django's built-in administrative interface rather than a separately engineered front-end application. This is again a deliberate trade-off: it sacrifices some interface customization in exchange for a substantial reduction in development effort, since the administrative interface for reviewing, filtering, and annotating logged anomalies is generated largely automatically from the underlying data model rather than hand-built. For a system whose dashboard exists to support internal security review rather than end-user interaction, this trade-off favours engineering economy without a corresponding loss of core functionality.

The Feedback Loop as an Architectural Component

The dashboard allows an administrator to mark a specific logged anomaly as a false positive, and this annotation is intended to feed periodic model recalibration. From a systems perspective, this closes the loop between the persistence layer and the backend's detection logic: the storage layer is not a passive archive but an active input to the system's ongoing operation. Implementing this loop reliably requires the schema to retain enough information about each flagged event, the metadata values that produced the original classification, to make later recalibration meaningful, which is part of why the schema described in Section 6.1, though minimal, retains the underlying behavioral values rather than only the final prediction.

Table 2. Persistence-Layer Schema Summary (AnomalyLog)

Field	Purpose	Design Rationale
Timestamp	Time of detection (auto)	Supports chronological audit review
Source_url	Originating flow URL	Links an alert back to its browsing context
Prediction / threat_label	Pipeline outcome and category	Drives dashboard filtering and triage
Score	Anomaly confidence value	Supports threshold recalibration
Duration, Payload_mean	Underlying behavioral metadata	Retained (not just the verdict) to support feedback-loop retraining; no content or PII fields exist by design

Deployment Topology and Scalability

Independent Scaling of Client and Server Tiers

Because the Browser Extension Layer and the Backend Intelligence Layer communicate exclusively through a well-defined REST API, the two tiers can, in principle, scale independently. The extension is a static, install-once client distributed to individual browsers, its resource footprint does not grow with the number of users in aggregate, only with the intensity of a single user's own browsing activity. The backend, by contrast, must handle aggregate load across every deployed extension instance, which makes it the natural target for horizontal scaling, adding additional backend worker processes or instances behind a load balancer, as deployment size grows. The statelessness of the detection endpoint (Section 5.1) is what makes this horizontal scaling viable without requiring session affinity between a given browser and a specific backend instance.

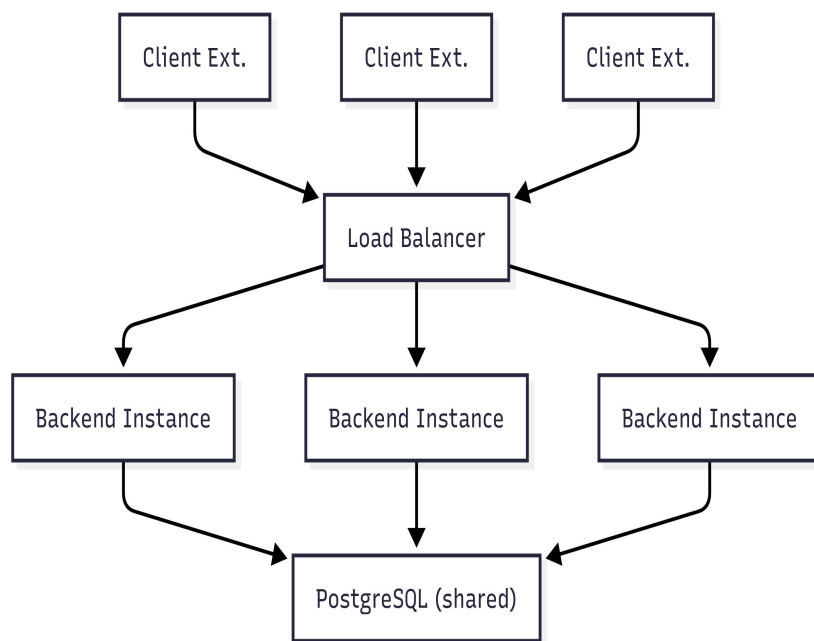


Fig. 4. Deployment topology: the extension client fleet scales with user count, while stateless backend instances scale independently behind a load balancer, sharing a single PostgreSQL database.

Development-Configuration Assumptions that Require Hardening

The current implementation's host permissions and API endpoint configuration are scoped to a local development target, communicating with a backend running on localhost. This is an appropriate configuration for development and controlled testing, but it is not a production-ready deployment topology on its own: moving from a local backend to a network-accessible one introduces

requirements that the prototype does not yet address, including TLS termination for the REST API, some form of authentication or origin verification so that the detection endpoint cannot be invoked by arbitrary third-party clients, and explicit cross-origin configuration appropriate to the browsers and hosts involved. None of these are architectural changes as such, the three-tier separation and the REST API contract between tiers remain unchanged, but they are necessary hardening steps between the current prototype and a production deployment, and are identified here explicitly as such rather than left implicit.

Host Environment and Operational Stability

The backend was developed and operated on a Pop!_OS Linux environment, selected specifically for its stability under sustained, high-concurrency network activity, a relevant property given that the backend must remain responsive while continuously receiving telemetry from active browser sessions. During testing, the system was subjected to bursts of high-volume traffic specifically to check that it remained stable and responsive under load conditions heavier than ordinary browsing, a deployment-readiness check rather than a classification benchmark, and one that is a necessary, if not sufficient, precondition for confidence in a production rollout.

Cross-Browser Deployment Constraints

The extension's reliance on Manifest V3-specific APIs, service workers, `'declarativeNetRequest'`, ties its current implementation to Chromium-based browsers, whose extension platform has matured furthest along the Manifest V3 model. Extending the framework to non-Chromium browsers such as Firefox is not simply a matter of packaging the same code differently; it requires re-implementing the extension-layer logic against each target browser's own extension APIs and lifecycle model, which may differ meaningfully in how (or whether) they support equivalent service-worker semantics. This is a genuine architectural constraint on deployment breadth, not merely a configuration detail, and is treated accordingly in Section 9.

Discussion: Engineering Trade-Offs

Table 3 (suggested placement below) summarizes the recurring pattern across every tier examined in this paper: each engineering decision resolves a specific constraint at the cost of a specific, identifiable trade-off, rather than being a free improvement. The ephemeral service-worker lifecycle is addressed by offloading persistent state to `'chrome.storage.local'`, at the cost of that storage needing explicit bounds to avoid unbounded growth. Preloading models at process start removes per-request loading latency, at the cost of multiplying memory consumption across

every backend worker process a deployment runs. Choosing Django and its built-in admin interface accelerates development and reduces the engineering surface area to maintain, at the cost of interface customization. Choosing PostgreSQL over a lighter embedded database provides the concurrent-write and transactional guarantees a security audit trail requires, at the cost of additional operational complexity relative to a zero-configuration alternative. None of these trade-offs is wrong; each represents a defensible choice for a system with these particular requirements, continuous background operation, privacy-constrained data collection, and audit-grade logging, but each is also a concrete point at which a different deployment context might reasonably choose differently, and a future engineering team adapting this architecture should treat each one as a deliberate decision to revisit rather than an implementation detail to inherit unexamined.

TABLE 3. Engineering Trade-off Summary

Decision	Constraint Addressed	Cost Incurred
Offload state to chrome.storage.local	Ephemeral service-worker memory	Requires explicit retention bounds
Preload models at process start	Per-request inference latency	Memory duplicated across every worker process
Django + built-in admin	Development effort / time-to-deploy	Limited dashboard customization
PostgreSQL over embedded DB	Concurrent writes, transactional audit trail	Added operational/deployment complexity

Practical Deployment Applications

The architectural pattern examined here, a lightweight, stateless browser sensor paired with a horizontally scalable backend and an audit-grade persistence layer, generalizes beyond this specific framework to any organization considering browser-native security tooling. Enterprise IT departments distributing extensions through managed Chrome policy can use the same extension/backend separation to keep the client-side footprint minimal across a large, centrally managed fleet of browsers. Security operations teams reviewing flagged activity benefit from a persistence layer whose schema is deliberately narrow, both because it limits what a compromised dashboard could expose and because it keeps the review interface focused on the metadata actually relevant to a decision. Organizations operating in regulated sectors, where audit-trail integrity and data minimization are compliance requirements rather than conveniences, benefit specifically from the schema-level

and database-level decisions discussed in Section 6, which enforce those properties structurally rather than through policy alone.

Conclusion

This paper has examined a three-tier browser-based intrusion detection framework specifically as a software engineering artefact, asking not how accurately its embedded models classify traffic but what architectural decisions were required to make the system deployable, maintainable, and scalable. The analysis shows that meaningful engineering work is concentrated precisely at the seams between tiers and at the platform constraints each tier must respect: the browser extension's accommodation of the Manifest V3 service-worker lifecycle, the backend's model-loading and request-staging strategy, and the persistence layer's schema and database choices all carry consequences for deployability that a purely model-centric evaluation would not surface. Treating these decisions as first-class subjects of study, rather than as implementation footnotes to a detection-accuracy result, is the specific contribution of this paper, and the trade-offs identified here, state persistence under an ephemeral execution model, per-process memory cost for preloaded models, development-configuration assumptions that require hardening before production use, are offered as a practical reference point for engineers building comparable browser-native security systems.

The contributions of this paper are:

- a) A tier-by-tier account of the engineering decisions underlying a three-tier browser-based intrusion detection architecture, extension, backend, and persistence, documented at the level of specific implementation choices rather than high-level architectural description alone.
- b) An analysis of the Manifest V3 service-worker execution model's implications for a browser extension performing continuous network observation, and of the state-persistence strategy adopted to work within that model.
- c) A discussion of backend design choices, model-loading strategy, request statelessness, and pipeline staging, that determine whether the analytical tier can scale horizontally under load.
- d) A schema-level analysis of the persistence layer's design, showing how field selection and database choice serve both audit-trail integrity and deployment scalability.
- e) A consolidated discussion of the deployment-readiness gaps between the current prototype and a production-hardened system, offered as a practical checklist for teams building comparable browser-native security tooling.

Limitations

The analysis presented here is grounded in a single framework's implementation and has not been validated against alternative architectural choices at each tier; claims about trade-offs are therefore descriptive and comparative in principle rather than benchmarked against competing implementations. The deployment-hardening gaps identified in Section 7.2 have been identified but not yet closed or empirically tested at production scale.

Future Work

Future work should implement and evaluate the production-hardening steps as already identified, TLS termination, endpoint authentication, and cross-origin configuration, and assess their performance impact under realistic multi-instance deployment. A containerized deployment of the backend and persistence tiers would allow more rigorous, reproducible testing of the horizontal-scaling claims made. Finally, re-implementing the extension layer against non-Chromium extension platforms would allow the cross-browser constraints identified in to be assessed directly rather than inferred from platform documentation alone.

References

- Almuhanna, R., & Dardouri, S. (2025). A machine learning approach for anomaly based network intrusion detection. *Frontiers in Artificial Intelligence*, 8, Article 1625891. <https://doi.org/10.3389/frai.2025.1625891>
- Arnob, A. K. B., Chowdhury, R. R., Chaiti, N. A., Saha, S., & Roy, A. (2025). A comprehensive systematic review of intrusion detection systems: Emerging techniques, challenges, and future research directions. *Journal of Edge Computing*, 4(1), 73-104. <https://doi.org/10.55056/jec.885>
- Holdbrook, R. K. B., Kwubeghari, A., & Ezeji, N. G. (2024). An analysis of key tools for detecting cross site scripting attacks on web based systems. *Lecture Notes of the Institute for Computer Sciences*, 54, 187-202. <https://www.researchgate.net/publication/377873080>
- Hozouri, A., Mirzaei, A., & Effatparvar, M. (2025). A comprehensive survey on intrusion detection systems with advances in machine learning and emerging cybersecurity challenges. *Discover Artificial Intelligence*, 5, Article 314. <https://doi.org/10.1007/s44163-025-00578-1>
- King, S., Ozen, I., Subramani, K., Senthivel, S., Vadrevu, P., & Perdisci, R. (2025). PP3D: An in-browser vision-based defense against web behavior manipulation attacks. *arXiv*. <https://arxiv.org/abs/2510.18465>

-
- Singh, S., Varshney, G., Singh, T.K., Mishra, .V and Verma, K., "Malicious browser extensions: threat landscape in 2025," *arXiv preprint*, arXiv:2503.04292, 2025. <https://arxiv.org/abs/2503.04292>
- Sriram, S., Kumar V., and Keerthivasan, T. (2025). Advanced malware detecting Chrome extension using machine learning. Daffodil International University Archive. <https://www.researchgate.net/publication/391218113>
- Mohale, V. Z., & Obagbuwa, I. C. (2025). Evaluating machine learning based intrusion detection systems: Enhancing transparency and interpretability. *Frontiers in Computer Science*, 7, Article 1520741. <https://doi.org/10.3389/fcomp.2025.1520741>
- Prasad, D., Suhasini, S., Parthasarathy, B., & Praveen, J. (2024). Enhancing internet security: a machine learning based browser extension to prevent phishing attacks. 2024 International Conference on Communication. <https://doi.org/10.1109/IC3SE62002.2024.10593201>
- Vimal, B et al.,(2024) "I spy with my little eye: Analysis and detection of spying browser extensions," in *Proc. IEEE European Symp. Security and Privacy*. <https://people.cs.vt.edu/vbimal/publications/spyingext-euroSP18.pdf>
-