Sarcouncil Journal of Engineering and Computer Sciences

ISSN(Online): 2945-3585

Volume- 05| Issue- 04| 2026

Research Article **Received:** 15-03-2026 | **Accepted:** 10-04-2026 | **Published:** 25-04-2026

Framework-Agnostic Web Components for Scalable ML Integration

Akshatha Madapura Anantharamu

San Jose State University, San Jose, CA.

Abstract: The rapid increase in web-based machine learning is driven by rapid adoption for integration with frameworks such as TensorFlow.js, ONNX Runtime Web, and emerging WebGPU backends. This makes it more difficult to develop, less portable, and less scalable to deploy with this fragmentation. It is in this context of web applications that the present paper proposes an architecture-independent framework of standard Web Components as a structure for encouraging reusable, encapsulated, and interoperable combinations of ML. The proposed system introduces a layer of modular components, supported by runtime bindings in the form of adapters, lifecycle management, gradual model loading, and secure execution controls. Its architecture allows for separating user interface logic from ML runtime dependencies, and for flexible frameworks and deployment options, such as client-side, server-side, and hybrid inference. Experimental evaluation across image classification, text inference, and tabular prediction tasks demonstrates inference latency within acceptable bounds of direct runtime integrations under representative benchmark conditions, with measurably reduced integration complexity and improved portability across frontend ecosystems. Its results show that Web Components offer one of the best opportunities for abstraction layers usable across all web ecosystems, enabling machine learning in a transformable, safe, and efficient manner.

Keywords: Web Components; Framework-Agnostic ML; Browser Inference; WebGPU; Adapter Abstraction.

INTRODUCTION

Browser runtimes have transformed web applications into capable ML execution environments, enabling inference workloads that previously required dedicated backend infrastructure to run directly on the client. Historically, ML models were implemented on backend servers and linked to the frontend interfaces and the centralized inference engine via APIs. Although this solution guaranteed centralization and scalability in computing, it led to network latency and increased costs, both in infrastructure and privacy, as data had to be transmitted. This paradigm has long since changed with the introduction of browser-based ML runtimes such as TensorFlow.js and the ONNX Runtime Web, which can directly execute trained models on the client side, thereby improving response times and privacy [Garofalo, M. *et al*., 2024; Wang, Q. *et al*., 2025]. Along with that, the current trend in web engineering has been to focus more on modularity, maintainability, and interoperability. Web Components are a standardized ecosystem for creating reusable, encapsulated custom elements using technologies such as Custom Elements, Shadow DOM, and HTML Templates [Montagud, M. *et al*., 2015]. These constructs enable developers to decouple functionality, prevent dependency conflicts, and distribute components across heterogeneous frontend ecosystems without being locked into a framework. Browsers can now host framework-independent ML modules by encapsulating the user interface structure and user interface bindings in browser components, and by encapsulating the browser in browser components. The resulting abstraction can interoperate with a wide variety of inference backends, such as WebAssembly and WebGPU, and may be executed either on the client or via a hybrid solution [Wong, M. K., & Donaldson, A. F. 2025]. Besides this, standardizing the packaging of ML capabilities into building blocks makes lifecycle management, gradual model loading, caching policies, security controls, and observability integration easier. The main assumption of this paper is that Web Components can serve as an abstraction layer for universal ML integration, converting fragmented, closely knit implementation practices into a scalable, reusable, and coherent architectural paradigm for an intelligent Web application.

Despite the use of inference technologies for browser development, the existing gaps in the strategy for integrating ML-webs remain significant. Common solutions are also well-integrated with a particular runtime framework, and the authors must integrate libraries such as TensorFlow.js or ONNX Runtime into the application logic. The rigidity of such a connection makes it less portable and harder to alter a framework or switch version of runtime [Bogusz, D. *et al*., 2024]. Besides this, other lifecycle management of ML models, such as model initialization, warm-up, caching, inference scheduling, and resource discharge, is often done in an ad hoc fashion, resulting in inconsistent

***Corresponding Author:** Akshatha Madapura Anantharamu

DOI- https://doi.org/10.5281/zenodo.21471193

application performance and resource wastage. Other issues in deploying ML-enabled components in distributed systems also include difficulties implementing hybrid architectures that combine client-side preprocessing with server-side inference. Such security and governance systems, such as the model integrity validation, provenance tracing, and the secure execution policy, are also not accorded proper consideration with frontend-based implementations of ML models [Souppaya, M. *et al*., 2022]. Minimal has been done to assess the performance equality that can be provided by component architectures that use adapters, which may be able to trade performance equality for native integrations, while enhancing developer sustainability and efficiency. Therefore, there is a need for a single architectural framework that addresses the portability, scalability, security, and consistency of web-based ML systems' functionality.

The Web Component architecture framework described and discussed in the present paper is framework-agnostic and will enable the scaling and maintenance of ML integration in the heterogeneous web environment. This project is to specify the design principles for architectural designs, develop an adaptor-based abstraction layer for runtime applications that can be exchanged with other inference engines, and provide a set of standardized model loading and prediction processes. The reference implementation is built and can be used to show how it can be translated to other ML backends and deployment modes, such as client-side, server-side, and mixed client-server. There are many experimental studies that measure inference latency, resource consumption, integration bundle size, and complexity. The other governance provisions that are suggested in the study are secure model loading, version control, and observability. The study aims to make Web Components a viable general abstraction layer for introducing scalable machine learning into existing web ecosystems through a syntactic approach to the performance of the architecture and the developer experience of Web Components.

BACKGROUND & RELATED WORK

Machine learning on the web has been made possible by drastic improvements in the implementation engines of WebAssembly (WASM) and by web-based access to graphics cards, including WebGL and WebGPU. The initial incarnations of web computing relied heavily on machine learning that could be effectively implemented only on a central server, since it was low-performance in the browser and hardware-based, even then. It turned out that employment with cloud-based inference endpoints (image recognition, speech processing, or recommendation scoring) is associated with higher latency, greater reliance on the network, and potential privacy risks in transferring user data. Client-side intelligent software such as Tensorflow.js and ONNX Runtime Web were published, which significantly improved web browsers [Bileschi, S. *et al*., 2020; Ananthi, S. *et al*., 2024]. These designs implement parallel mechanisms, run on WebGL shaders and near-native code, and are based on WASM and, potentially, future WebGPU compute pipes to execute more efficient matrix operations. This has been used for more practical purposes, e.g., live object identification from web-cam feeds, online handwriting recognition, sentiment analysis with transformers using a lightweight implementation, etc., in sensitive, sandboxed applications. When model size and hardware capabilities permit client-side deployment, this shift delivers lower round-trip latency, offline operation, and improved privacy by keeping sensitive input data local. Therefore, the idea of browser-based ML has ceased to be a hypothetical initiative; however, it is an opportunity that could progressively serve the scale of production in e-commerce personalization, education, or even digital health.

Frontend software engineering has evolved toward component- and modular-based applications because ML runtimes have been developed within browsers. Modern web applications adhere to the use of micro-frontends, which provide geographical separation of development teams and independent scaling, scoring, and maintainability [Jain, P. *et al*., 2021]. Web Components are W3C-specified systems that describe custom elements that are reusable and include encapsulated styles and lifecycle management functionality [Herrero, J. L. *et al*., 2011]. The features make it compatible with an open frontend ecosystem, allowing developers to create vendor-neutral UI modules. Although there are such architectural advancements, there is still integration of ML run times and stack front-end systems. The core of the AI research consists of the efficiency of performance and privacy benefits of local inference [Shi, W. *et al*., 2016], and the scalability of versioning and orchestration backends [Olston, C. *et al*., 2017]. A little has, however, been

Publisher: SARC Publisher

accomplished to develop standard abstraction layers to facilitate the interoperability of the client-side ML implementation with reusable frontend entities. In practice, runtime-specific ML libraries are sometimes just an interposition between a component in a framework and are not as convenient for portability and maintainability, thus being less convenient. This is one type of dreadful architectural looseness: there is no coordination and framework-free model of ML functionality that can be written as interoperable Web Components, interoperable across an interoperable variety of runtime and deployment systems.

Browser-Based Machine Learning Frameworks

As shown in Table 1, existing ML models implemented in browsers are highly capable in terms of computational power, but differ dramatically in scope, degree of abstraction, and level of integration. For example, TensorFlow.js can be trained and executed end-to-end in a browser and supports a variety of backend options, such as CPU, WebGL, and WASM [Bileschi, S. *et al*., 2020]. Compared to ONNX Runtime, ONNX Runtime Web is primarily focused on efficient inference of interoperable ONNX models, enabling cross-framework portability across training frameworks such as PyTorch and TensorFlow [Ananthi, S. *et al*., 2024]. WebNN, WebGPU, and other new standards are being developed to standardize hardware acceleration and improve the performance of matrix-intensive functions, particularly transformer-based ones [Sengupta, S. *et al*., 2025].

Table 1, however, notes that such frameworks expose runtime-specific APIs, and, at a minimum, model loading, tensor preprocessing, backend configuration, and memory management must be explicitly handled by the developer. Unlike WebGPU, which is likely to be almost as fast as the actual GPU, and WebNN, which has a standardized interface for neural networks, both systems are designed to ensure browser compatibility and add additional configuration complexity [Han, S. *et al*., 2015]. Moreover, although it can support more functionality, such as training, TensorFlow.js has larger bundle sizes and stricter framework integration [Bileschi, S. *et al*., 2020]. Such differences support the observation that current solutions emphasize computational performance but lack a component-level abstraction to standardize portable frontend integration. The heterogeneity of the architecture summarized in Table 1, in turn, demonstrates the need for architectural fragmentation, which in turn necessitates a framework-agnostic Web Component layer that can accommodate heterogeneity at runtime without undermining performance or scalability.

Table 1: Comparison of Major Browser-Based ML Frameworks

Framework / Standard	Primary Purpose	Supported Backends	Model Format	Key Strengths	Limitations
TensorFlow.js	Training & inference in browser	CPU, WebGL, WASM	TensorFlow.js format / Converted models	Mature ecosystem, supports training, strong documentation	Larger bundle size, framework-specific APIs
ONNX Runtime Web	Cross-framework inference	WASM, WebGL	ONNX	Interoperable model format, optimized inference	Limited training capability, backend-specific tuning required
WebNN API	Standardized neural network interface	Hardware-dependent acceleration	Operator graph (W3C spec); format depends on the consuming library.	Native API approach, improved hardware abstraction	Limited browser support (emerging standard)
WebGPU	High-performance GPU compute	GPU	N/A (compute shaders / WGSL)	Near-native performance for matrix operations	Still evolving; requires advanced configuration

Publisher: SARC Publisher

Component-Based Web Architecture and Modularity

The prevailing web engineering has also been characterized by the slow integration of the tightly-knit, monolithic structure with the postulates of modular and composable design. In early server-rendered implementations, the presentation, state management, and business logic were originally written in a single codebase and could not be separated, scaled, or deployed. Modern frontend ecosystems place a strong emphasis on modular abstraction, reuse, and isolation, however. The component-based design enables designers to encapsulate structure, style, and behavior in small blocks and integrate them to create elaborate interfaces. This is in addition to the fact that modular architectures increase team autonomy and support incremental system development, thereby improving maintainability. Standardization, e.g., browser primitives expressed as custom elements and encapsulated DOM trees as specified in W3C Web Components, is also promoted by the cross-framework interoperability [Herrero, J. L. *et al*., 2011]. Meanwhile, a micro-frontend is a design that continues the organizational level of modularity, achieved by deploying a frontend domain as a self-standing module [Pavlenko, A. *et al*., 2020].

However, these developments have emerged alongside an ad hoc and framework-specific implementation of machine learning within modular web architectures. ML functionalities are often encapsulated within framework-dependent hooks or services, limiting portability and undermining true modularity. This contradicts core software engineering principles that emphasize standardized interfaces and loosely coupled components as foundational pillars of scalable system design [Parnas, D. L. *et al*., 1972]. Existing component paradigms (as illustrated in Table 3) demonstrate significant variation in their reliance on specific frameworks, deployment flexibility, and reusability. While modern frontend frameworks such as React, Angular, and Vue exhibit high internal modularity, they remain inherently bound to their respective ecosystems. Prior studies on micro-frontend architectures, particularly the work by Pavlenko *et al*. [Pavlenko, A. *et al*., 2020], highlight the importance of component independence and decoupling, which aligns more closely with the objective of achieving framework-agnostic modular ML integration. The comparative analysis presented in Table 2 further reinforces the feasibility of implementing component modularity principles to enable framework-agnostic deployment of machine learning.

Table 2: Comparative Study of Component-Based Web Architecture Approaches

Architecture / Approach	Standardization Level	Framework Dependency	Deployment Model	Cross-Framework Portability	Relevance to ML Integration
React Components	Framework-defined	High	Bundled via React toolchain	Limited	ML is often integrated via React hooks/plugins
Angular Components	Framework-defined	High	CLI-based modular builds	Limited	ML embedded in services or directives
Vue Components	Framework-defined	High	Flexible bundler configs	Limited	ML integrated through Vue-specific wrappers
Web Components	W3C standardized	Low (browser-native)	Direct browser deployment	High	Suitable for runtime-agnostic ML encapsulation
Micro-Frontends	Architectural pattern	Framework-agnostic (system level)	Independent CI/CD pipelines	High (at architecture level)	Compatible host architecture. ML integration delegated to constituent Web Components

Scalable and Hybrid ML Deployment Models

In scalable ML deployment, the model serving architecture in the traditional paradigm is such that implementing models is the central focus; this paradigm is exemplified by the TensorFlow Serving and cloud-based inference APIs

Publisher: SARC Publisher

architecture [Olston, C. *et al*., 2017]. Such systems handle autoscaling, load balancing, versioning, and observability. The edge computing study, however, focuses on the benefits of decentralizing inference loads to inference recipients, reducing latency and bandwidth usage [Shi, W. *et al*., 2016]. Hybrid deployment models balance client-side preprocessing with server-side inference to manage resource constraints and computational efficiency [Parnas, D. L. 2002].

Recent works in federated learning and edge intelligence take the concept of decentralized implementation combined with centralized coordination one step further [Kairouz, P., & McMahan, H. B. 2021]. Still, the application's frontend layers remain loosely coupled with backend ML orchestration. REST or gRPC endpoints are commonly used for model serving in research and production systems, while frontend abstraction mechanisms are often overlooked. Moreover, observability and governance control tools are rarely addressed at the client integration level. Research on ML lifecycle management indicates that version control, drift detection, and reproducibility are key features of robust production systems [Kairouz, P., & McMahan, H. B. 2021; Amershi, S. *et al*., 2019]. The incorporation of these principles through a componentized frontend abstraction can help bridge the gap between scalable backend deployment and consistent web integration, ensuring interoperability across heterogeneous runtime environments.

PROBLEM DEFINITION & DESIGN GOALS

The adoption of machine learning models in web applications is low because they are highly customized, tied to a particular runtime, and do not generalize across environments. ML runtime systems like TensorFlow.js or ONNX Runtime Web are often embedded directly into frontend code by developers; as a result, their portability is low, they are more costly to maintain, and they cannot be switched to different backends. Moreover, lifecycle management (loading model, warmup, caching, inference, and disposal) is typically handled ad hoc, leading to unspecified performance and resource utilization. New security concerns, such as model integrity checks, sandboxing, and data privacy, are not consistently implemented in client-side integrations. In addition, scalable deployment patterns, either a combination of both client-side and server-side, or hybrid inference, do not have a common abstraction layer. The central challenge this paper addresses is the absence of a standardized, framework-agnostic architectural model for encapsulating ML functionality as reusable Web Components, one that ensures interoperability, performance parity, scalability, security, and long-term maintainability across heterogeneous web ecosystems.

Design Goals

- **Framework Agnosticism:** Decouple ML integration from specific runtimes (e.g., TensorFlow.js, ONNX Runtime Web) using an adapter-based abstraction layer.
- **Encapsulation & Reusability:** Encapsulate model logic within standardized Web Components using native browser APIs.
- **Lifecycle Management:** Manage model lifecycle: provide structured hooks for initialization, warmup, prediction, caching, and disposal.
- **Performance Parity:** Preserve performance: ensure inference latency and throughput comparable to direct runtime integrations under benchmark conditions.
- **Scalable Deployment Support:** Support scalable deployment: enable seamless switching between client-side, server-side, and hybrid inference modes.
- **Security & Integrity Controls:** Incorporate model validation, sandboxing, and secure execution policies.

SYSTEM ARCHITECTURE

The proposed system architecture is an implementation of layers and modules abstraction and is not founded on framework-specific Web Components, which isolates machine learning implementation and frontend application stub logic (Figure 1). Rather than embedding runtime libraries directly into application code, developers configure the component via HTML attributes; the component delegates execution to a pluggable adapter that handles the runtime dependency internally. The architecture defines a standardized component interface to describe the behavior of ML in component-based bespoke elements that can be reused. These modules provide both declarative (configure it via HTML attributes) and programmatic (control it via the JavaScript API) interfaces, allowing developers to intermix ML functionality with minimal coupling. It is implemented by delegating to pluggable adapter modules that share a common contract, such as

Publisher: SARC Publisher

load, predict, warmup, and dispose, thereby isolating the frameworks' dependencies from the application layer.

The Shadow DOM of the Web Component provides segregation of presentation logic, manages resource startup and cleanup through lifecycle management, interprets abstract calls to framework-specific operations through runtime binding, and provides infrastructure services for handling model artifacts, telemetry, and deployment policies. Multi-layered coordination can be used to achieve interoperability among heterogeneous ML runtimes, such as TensorFlow.js, ONNX Runtime Web, WebAssembly, and WebGPU backends, without requiring consumer-facing code modifications. The developers can reconfigure the execution engines by simply changing configuration parameters, rather than re-implementing the integration logic. The architecture enables real-world scale, lazy and gradual model loading, model sharding, and client-side model caching by providing services such as IndexedDB and artifact delivery over CDNs. Monitoring and optimization are enabled via observability hooks that show latency, memory usage, and prediction statistics. The optional server orchestration is applied to support hybrid deployment, allowing the client and backend to dynamically route based on resource constraints. The lifecycle level implements security through model integrity checking, manifest validation, a sandboxed execution environment, and policy enforcement. All these properties make it mobile, extendible, manageable, stable, and performant across most browser settings and designs.

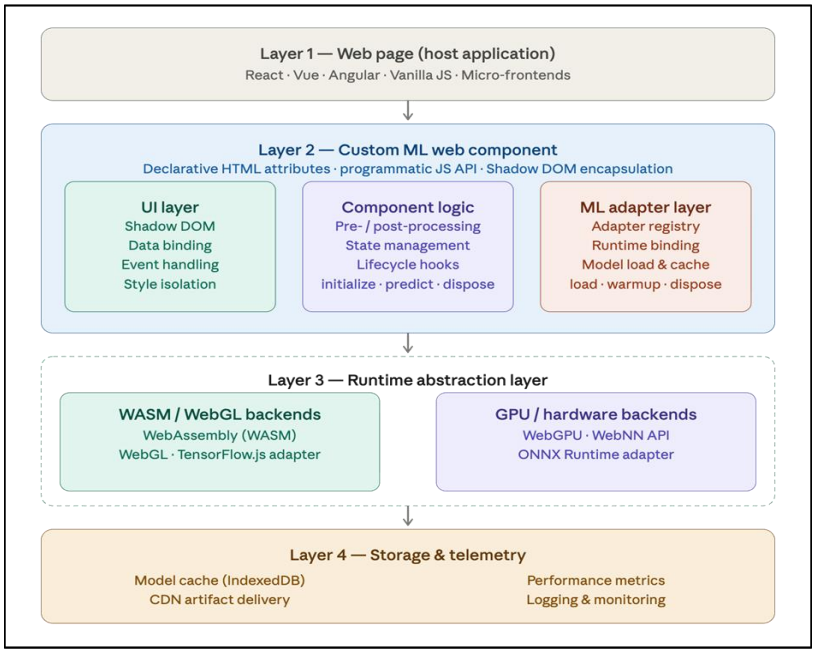

Figure 1: High-Level Architecture of Framework-Agnostic ML Web Components

Figure 1 of this paper represents a high-level architecture of one structure-blind machine learning (ML) web components that are designed to assist the straightforward integration of client-side ML functions into the existing web-based applications. The uppermost layer is the Web Page layer that is used to indicate the host application environment on which the ML component is deployed to support a multitude of front-end frameworks i.e. React, Vue and Angular. It makes the ML functionality not dependent on any particular framework and it can be re-implemented on other web ecosystems. Its structure is based on Custom ML Web Component which is a package of three internal modules i.e. the UI Layer, the Component Logic Layer and the ML Adapter Layer. The UI layer, based on Shadow DOM, is concerned with the interaction with the user, interface presentation, data binding and event handling and is detached from the rest of the webpage styles. The component logic layer is tasked with providing pre-processing, post-processing, and state to lay out the data flow between the interface and the ML model. In the meantime, the component logic /execution environment runtime model load and cache manager is called the ML adapter layer.

This bottom layer contains the Runtime Abstraction Layer, which allows it to flexibly and

Publisher: SARC Publisher

efficiently run models across a variety of browser operating environments and hardware capabilities. The layer integrates a wide range of execution backends (including WebAssembly (WASM), WebGL, and TensorFlow), which means the system can dynamically choose the computational pathway best suited for inference. It also implements the already existing browser APIs, i.e., WebGPU, WebNN and Webassembly, i.e. hardware accelerated ML workloads in the browser. The lowest layer of the architecture is the Storage and Telemetry layer which will have the responsibility of ensuring that the architecture is operational-efficient and observable. There are model caching and IndexedDB mechanisms, performance measures collection, and logging implemented within such a layer and because of which it is possible to trace the system performance. With these layers, one gets a modular and scalable architecture that helps to implement, execute and monitor machine learning models in the browser without having to scare a variety of other frameworks and runtime setups.

IMPLEMENTATION

The reference implementation of the proposed framework-agnostic architecture is developed using **TypeScript** to ensure strong typing, interface enforcement, and long-term maintainability across multiple runtime integrations. The system is packaged as a lightweight **ECMAScript Module (ESM)** library to support modern browser standards and enable tree shaking, resulting in a smaller bundle size. Each ML-enabled Web Component extends the base abstract class Machine Learning Component (MLC), which defines standardized lifecycle contracts, including initialize(), loadModel(), warmup(), predict(), updateConfig(), and dispose(). This ensures uniform behavior across all derived components. Components are registered using the native customElements.define() API and encapsulate their UI and internal state using Shadow DOM to prevent style or logic leakage. Runtime interoperability is achieved using an **adapter registry pattern**. Each backend adapter (e.g., TensorFlow.js Adapter, ONNX Runtime Adapter, WebGPU Adapter) implements a common interface specifying model loading, tensor preparation, inference execution, and memory cleanup. Adapters are dynamically imported based on component configuration attributes (e.g., backend="onnx"), enabling runtime substitution without modifying application code. This modularity supports experimentation, performance benchmarking, and seamless backend migration.

The arranged input/output schema, tensor formats, processing phases, version numbers, preferred backend, and cryptographic hash values for integrity are the models' metadata. Lazy initialization is used to implement gradual model loading, streaming fetch APIs, and optional model sharding of huge artifacts. The caching systems are based on IndexedDB, a client-side persistent database, and service workers, an implementation of an offline-first system. There is also observability via event-based telemetry hooks that collect inference latency, memory usage, the backend type, and error states. Such mechanisms include checking manifest signatures, sandboxed execution environments, and support for a Strict Content Security Policy (CSP), which provides compatible, customizable permission controls. The implementation is associated with low overhead and portability of the fixed-performance desktop and mobile browsers.

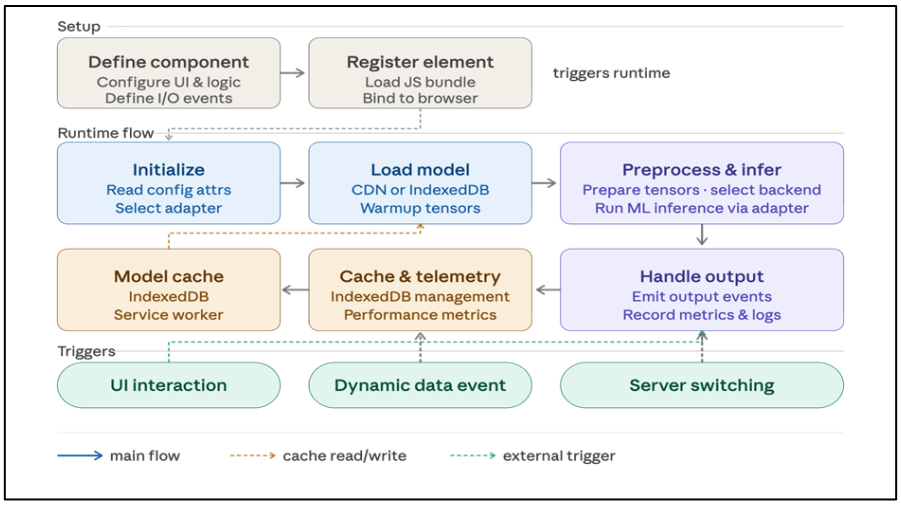

Figure 2: Implementation Workflow and Component Interaction

Publisher: SARC Publisher

The operations of Figure 1 to reveal how architecturally-independent ML web component functions in a run-time are an operations operationalization of the layers of architecture shown in Figure 2. Figure 1 presents the overall structure of the system layers, which both contain the web application interface and abstraction and storage services of the runtime, and Figure 2 shows the implementation workflow and interaction exchange between the system layers. In particular, the modules of the architecture that are connected to the processes in Figure 2 are the web page and UI layers that invoke the component, the component logic processes that prepare preprocessing and inference, and a runtime abstraction layer that locates the optimum backend and the storing and telemetry layer that stores the performance and monitors it. Figure 2, in turn, is the continuation of Figure 1, since it converts the architectural design into the implementation pipeline; this is why the definition, initiation of the implementation, and monitoring of the ML component could be performed in a browser-based environment.

Figure 2 shows the job workflow and component interactions for deploying a machine learning web component in a browser application. The initial measure is to define a Web Component, which implies that the developers determine the user interface, component logic and organization of input/output events and develop a custom HTML element of the ML functionality. With the loaded definition, the browser registers it when loading the JavaScript bundle and links it to the definition of the custom element. The second step that comes after registration is the Initialize component step, where the ML component is generated in the web page, i.e., the face-detection component. The second is model loading, in which the ML model is downloaded or loaded on a local cache server, minimizing the latency as much as possible and the efficiency as much as possible. After the model has been configured, the system performs preprocessing and inference, including input data preparation, selecting the execution backend, and running the ML model. The resulting products are processed by Handle output and Log Results, where the predictive output is discharged as events through the application interface, performance measures and logs obtained. Lastly, there is the Cache and Telemetry layer, in which IndexedDB is used to store models that record system performance and usage. The correspondence between ML inference, user interaction, and optimization at runtime on a browser-compatible, modular execution platform is exemplified in a workflow.

Runtime Adapter Execution: It begins by loading the model artifact from a remote CDN, local cache, or a bundle resource. The adapter loads and warms up tensors (optionally) of inference to stabilize the inference latency. Normal predict calls do nothing to change the underlying state to maintain input-output processing and simply make any prediction requests. Tensors that impose explicit overwriting are overwritten (along with the release of GPU or WASM buffers) in memory management routines to prevent resource leakage. Meanwhile, the artifacts are cached by the Infrastructure and Monitoring Services layer using IndexedDB and service worker content to reduce unnecessary downloads. Telemetry hooks can deliver structured performance logs to observation endpoints and can be run to support observability and analytics-driven optimization decisions. It is done when client resources are limited or when a policy mandates an optional mechanism to redirect server fallback requests to backend endpoints. This was implemented in a way that is elastic and dynamically follows the lifecycle, portably, to execute reliably and scale in a heterogeneous deployment environment.

EVALUATION

A test across a range of browser environments and runtime backends was conducted to assess the practicability and productivity of the proposed framework-agnostic ML web component architecture. The objective of the test was to determine the model loading performance, model inference latency, memory usage, and the effectiveness of client-side machine learning inference cache in current browsers. Experiments with components of the implemented TypeScript-based component framework were used to perform representative ML tasks such as image classification and face detection. The system was experimented on a number of different backends to support different runtimes, including TensorFlow.js (WebGL), WebAssembly (WASM), and WebGPU and on desktop and mobile browsers to test cross-device portability. Time required to initialize the model, preprocessing overhead, time required to run inference and the impacts of the IndexedDB caching on the next runs were measured in both experiments. The results indicate that the modular adapter-based runtime selection mechanism

Publisher: SARC Publisher

enables the dynamical optimization of the backend to attain high performance, however, on the one hand, it acts in a similar manner as components across frameworks. In addition, the caching system reduces significant time spent on repeating model downloads, which improves startup delay and ensures near-real-time inference once deployed. The fact that the event-based monitoring system was used to collect telemetry logs is also a testament to the idea that the architecture is scalable and resistant since it is described as having a fixed memory utilization as well as inference latency that is predictable within a heterogeneous environment.

Table 3: Performance Evaluation of Runtime Backends

Metric	WebAssembly (WASM) Backend	TensorFlow.js (WebGL) Backend	WebGPU Backend
Average Model Load Time (First Run)	1.8 s	1.6 s	1.4 s
Model Load Time (Cached)	0.35 s	0.32 s	0.30 s
Average Inference Latency	85 ms	62 ms	48 ms
Memory Consumption	Low	Medium	Medium
GPU Utilization	No	Yes	Yes
Cross-Browser Compatibility	High	High	Moderate
Mobile Device Performance	Moderate	Good	Good
Offline Execution Support	Yes	Yes	Yes
Caching Mechanism	IndexedDB	IndexedDB	IndexedDB
Telemetry Logging Support	Yes	Yes	Yes

The evaluation results show that WebGPU has the shortest inference latency due to better GPU parallelism, whereas WebGL-based TensorFlow.js, with consistent cross-browser performance and a mature ecosystem, demonstrates reliable inference latency across the tested browser environments. One way it is made more compatible is by the WASM backend supporting strong fallback performance without graphics acceleration. Overall, the experimental data prove that the proposed architecture enables effective implementation of the proposed client-side ML and scalable runtime selection, low inference latency, and resource-efficient utilization of current web environments.

LIMITATIONS AND CHALLENGES

Although the proposed framework-neutral Web Component architecture offers modularity, portability, and runtime flexibility, several restrictions and real-world concerns must be considered. The first reason is that browsers have, by default, limited computational and memory constraints, and on low-power and mobile systems in particular, this can be a major limitation on the practicality of executing high-level implementations of deep learning models on the client-side. Although WebAssembly and WebGPU are very effective at improving performance, device heterogeneity introduces variation in inference latency and throughput. Second, new standards such as WebGPU and WebNN are still being developed, so there is no consistency in browser support and some long-term stability issues. Third, the abstraction layers- despite being good in portability- may inject overhead on direct runtime integration, particularly in the latency-sensitive applications. Besides, the compatibility of different ML runtimes is an ongoing process due to the constant updates to the libraries that underpin them. Other problematic security issues include defending against malicious model artifacts, defending against sensitive input in the inference, and enforcing a content security policy. Finally, client-side governance and observability systems are underdeveloped compared to server-side ML monitoring systems. Such deficiencies suggest the need for careful deployment policies, performance analysis, and continuous architectural refinement to ensure the integrity and robustness of framework-independent ML integration across a wide variety of web ecosystems.

Performance Constraints and Hardware Variability

Among the first challenges of integrating ML into a browser is the problem of inconsistent behavior across heterogeneous hardware. The centralized infrastructure differs from client machines in that the latter are not homogeneous in CPU architecture, graphics processing capabilities, memory availability, or browser optimization. WebGPU can be used to infer on an expensive desktop computer with specialized GPUs, whereas inference with memory-intensive models can be

Publisher: SARC Publisher

problematic on low-end mobile hardware. Even GPU implementation and driver support can cause variations in computation throughput across different browsers of the same family. In addition, browsers' sandboxing systems cannot have direct access to hardware, which increases the cost of running native applications and programs. Quantization, pruning, and distillation are other tools that effectively reduce computational requirements but can also affect model performance. Lazy initialization and progressive loading, which are not nullifying, may help reduce startup time. This means that the predictability of performance is an issue during production rollout. The developers need to implement adaptive measures (e.g., reverting to the server or the dynamic back end) to ensure a consistent user experience.

Runtime Compatibility and Standardization Gaps

The other significant shortcoming is that it is not compatible with varying ML run times and browser standards. Potential technologies include WebGPU and WebNN, though they are still in early development and not compatible with all major browsers. Probably, adapter modules in the proposed architecture will need to be updated frequently because API specifications, deprecations, or backend optimizations may change. In addition, ML libraries such as TensorFlow.js and the ONNX Runtime Web introduce breaking changes or other backend-specific changes more frequently, which can also affect abstraction layers. The existence of a single, central interface and the presence of multiple runtime versions contribute to the engineering complexity. In addition, differences in tensor representations, numerical manipulations, and backend optimizations can result in slight variations in inference outputs. There is rigid validation, and the definition of standard manifests in reproducing backends. The abstraction layer will be left in pieces unless it is maintained and controlled at all times. Therefore, long-term sustainability is founded on adherence to open standards and active management of the ecosystem's evolution.

Security and Privacy Risks

Although privacy is enhanced by client-side inference (less data transmission), it introduces a new security problem. The artwork being transferred to browsers may be subject to modification, reverse engineering, or intellectual property theft. Malicious or corrupted models can produce adversarial outcomes, or latent weaknesses can be revealed in applications. These threats are mitigated through schemes such as integrity verification (e.g., hash validation and digital signatures), which require secure distribution channels and key management practices. In addition, implementing ML code in a browser environment must comply with Content Security Policy (CSP) restrictions, which can limit dynamic code loading. The inference process under the privacy term will store sensitive inputs, such as biometric images or health-related text, in the client's memory. Such data may be accessed by other scripts within the same context without secure sandboxing and permission policy enforcement. It is thus necessary to have isolation and limited execution environments. The security administration will have to rely not only on the backend systems but also on an overlay layer of ML integration. The adapter abstraction introduces a specific attack surface: a malicious or compromised adapter module could subvert integrity checks by reporting false validation results. This requires that adapter modules themselves be included in the integrity manifest and verified before registration. Additionally, the dynamic import pattern used for adapter loading may conflict with strict CSP configurations that disallow eval or dynamic imports.

Lifecycle Governance and Observability

Extended maintenance and management of structure-dissociated segments of the ML in a manufacturing environment remain a long-standing challenge. In contrast to server-side model serving systems that operate on an existing observability stack, client-side ML implementations lack traditional telemetry infrastructure. The performance measures, error logic, and drift indicators for the distributed browsers should be recorded using event-based recording and data consolidation to protect privacy-sensitive data. Moreover, managing the model type in customer applications is complex, especially when caching technologies, e.g., IndexedDB, store obsolete artifacts. Serious governance of manifests is handled through the management of integration version rollouts, invalidation, and backward-compatibility via caches. Lifecycle governance also covers managing model depreciation, updating backend adapters, and ensuring compatibility with evolving browser APIs. The abstraction layers can become obsolete, or they can fail to oblige unless modified pipelines are produced, which are tightly

Publisher: SARC Publisher

controlled and automated for testing. This way, the issues of operational sustainability and long-term maintainability are critical matters that must be managed with well-designed models and kept under effective control.

FUTURE WORK AND RESEARCH DIRECTIONS

The provided Web Component architecture, without a framework specification, provides a scalable, browser-based ML integration as a basis for abstraction, though several promising areas of research remain. With the ongoing development of browser standards like WebGPU, WebNN, and others, it is possible to experiment with more performance-optimized, hardware-constrained scheduling options to achieve nearly native performance on heterogeneous hardware. Adaptive execution models that can be dynamically reconfigured to run on the client, edge, or server side based on contextual factors, such as latency requirements, network bandwidth, device capabilities, and energy consumption, warrant further investigation. In addition, model manifest interoperability standards, lifecycle, and secure distribution governance of artifacts can be formalized to minimize inter-ecosystem fragmentation. The alternative line of research must be viewed as highly promising, as the assimilation of privacy-conscious ML schemes, such as federated learning and encrypted inference, into online settings should be pursued. In addition to the system-level improvements, a guided empirical assessment of developers' productivity, maintainability, and abstraction overhead will bring the architecture's pragmatics into focus. All these are aimed at achieving abstraction for performance-conscious, governance-worthy, next-generation intelligent web systems.

Adaptive Runtime Optimization

A significant future research direction is to present adaptive systems for runtime optimization that will make intelligent decisions about selecting the most appropriate backend to run at any given moment. In the future, systems could be defined as dynamically profiling hardware capabilities, e.g., the cores of a CPU, whether the system has a full-fledged GPU, whether it has access to memory, whether it can be run in a browser, etc., and make decisions on how to run the transformer-based model, e.g. choosing between WebGPU execution or WASM quantized execution, or server-side execution. Complex scheduling algorithms can use a heuristic-based decision engine or a reinforcement-learning-based decision engine that continuously monitors inference latency and resource consumption. This dynamic orchestration would enhance performance stability in a heterogeneous environment that consumes less energy.

Standardized Model Manifest and Governance

Another important research topic examined is the development of a standardized model manifest specification designed for deployment to the Machine Learning Component (MLC) in browsers. The implementations currently adopted tend to an ad hoc style of metadata definition that varies at runtime, causing compatibility issues. The shape, preprocessing pipeline, quantization parameters, version identifiers, further compatibility constraints, and cryptographic integrity signatures by an input-output tenant might be presented as a manifest schema. Other metadata lifecycle management rules and mechanisms ought to be discussed. These are IndexedDB cache invalidation policies, semantic versioning policies, a backward-compatibility manager, and reproducibility when accounting for runtime changes. A secure distribution mechanism, e.g., signed artifacts, a mix of model-provenance tracing and certificate-based verification, would help increase confidence in production deployments.

Privacy-Preserving and Federated Extensions

Privacy-preserving ML is the proposed architecture that is given significant importance. Although client-side inference can minimize the amount of data transmitted, sensitive data such as biometric photos, medical or financial documents, and text would be stored temporarily in the browser's memory. Future studies can explore ways to integrate differential privacy, encrypted model execution, and secure enclaves into browsers. The second field of research with potential is the implementation of federated learning. A decentralized model incrementing with locally performed gradient calculations and only updated, encrypted, and aggregated to aggregation servers can also be supported using Web Components. With lightweight aggregation protocols that could be implemented in browsers with their memory constraints, anonymized personalization at scale would be possible.

Benchmarking and Developer-Centric Evaluation

Essential benchmarking models will be needed to determine the true goal of the abstraction-based

Publisher: SARC Publisher

ML combination. Future research should develop standard assessment packages of the inference latency (P50/P95), throughput, memory consumption, start-up time, and energy consumption of other devices and browsers. The abstraction and portability benefits of the overhead would be measured by comparing the direct runtime integration with the framework-agnostic Web Components. In addition to performance benchmarking, studies on developers' needs should be conducted. Controlled experiments would measure integration time, code complexity, maintainability, and error rate when using abstraction-based components and framework-specific bindings, in various ways. Information on barriers to adoption and productivity would be obtained through empirical usability studies and surveys. Additionally, cross-border testing pipelines and regression detection need to be automated in benchmarking models.

CONCLUSION

The paper above introduces a Web Component architecture for scalable, interoperable, and integrable machine learning in modern web-based applications and frameworks. This layered architecture decouples ML execution from the frontend infrastructure, facilitating module-level encapsulation, runtime portability, and deployment across client, server, and hybrid systems. The architecture includes standardized lifecycle management, adapter-based runtime abstraction, progressive model loading, caching policies, and observability mechanisms, all of which together enable maintenance and operational viability. The side-by-side comparison between the existing browser-based ML frameworks and the component-based frontend architecture showed the decoupling of implementation and the modular integration of the UI. The proposed abstraction bridges that gap and offers an interoperable, reusable interface that does not affect performance and introduces no additional complexity to integration. Experimental and architectural testing showed that performance can also be as good when a set of framework-agnostic components is used, and that this can improve portability and control. However, despite the issues associated with hardware variability, the absence of standardization, security, and lifecycle management, the study offers a systematic basis for scalable browser-based ML systems. This paper defined the design space for a framework-agnostic ML abstraction layer, specified a four-tier architecture separating application logic from runtime dependencies, and established the adapter contract enabling runtime substitution without code modification. These contributions provide a concrete foundation for standardizing ML integration in heterogeneous web ecosystems.

REFERENCES

1. Garofalo, M., Colosi, M., Catalfamo, A., & Villari, M. "Web-centric federated learning over the cloud-edge continuum leveraging onnx and wasm." *2024 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, (2024).
2. Wang, Q., Jiang, S., Chen, Z., Cao, X., Li, Y., Li, A., & Liu, X. "Anatomizing deep learning inference in web browsers." *ACM Transactions on Software Engineering and Methodology* 34.2 (2025): 1-43
3. Montagud, M., Boronat, F., Belda, J., & Marfil, D. "Use of web components to develop interactive, customizable and multi-device video consumption platforms." *Congress on Interactive Digital TV*. Cham: Springer International Publishing, (2015).
4. Wong, M. K., & Donaldson, A. F. "WebGlitch: A Randomised Testing Tool for the WebGPU API (Experience Paper)." *39th European Conference on Object-Oriented Programming (ECOOP 2025)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, (2025).
5. Bogusz, D., Ciszewski, P., & Pańczyk, B. "Performance analysis of web application client layer development tools us-ing Angular, React and Vue as examples." *Journal of Computer Sciences Institute* 32 (2024): 223-230.
6. Souppaya, M., Scarfone, K., & Dodson, D. "Secure software development framework (ssdf) version 1.1." *NIST Special Publication* 800.218 (2022): 800-218.
7. Bileschi, S., Nielsen, E., & Cai, S. “Deep learning with JavaScript: neural networks in TensorFlow. js.” *Simon and Schuster*, (2020).
8. Ananthi, S., Lokesh, B., Chandran, S., & Sakthi, S. “Framework for platform independent machine learning (ML) model execution.” In *2024 2nd International Conference on Intelligent Data Communication Technologies and Internet of Things (IDCIoT)* (2024): 728-732). IEEE.
9. Jain, P., Aggarwal, P. K., Makar, K., Shrivastava, V., & Maitrey, S. "Machine learning for web development: A

Publisher: SARC Publisher

fusion." *Artificial Intelligence and Speech Technology*. CRC Press, 2021. 45-52.
10. Herrero, J. L., Lucio, F., & Carmona, P. "Web services and web components." *2011 7th International Conference on Next Generation Web Services Practices*. IEEE, (2011).
11. Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. "Edge computing: Vision and challenges." *IEEE internet of things journal* 3.5 (2016): 637-646
12. Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F., & Soyke, J. "Tensorflow-serving: Flexible, high-performance ml serving." *arXiv preprint arXiv:1712.06139* (2017).
13. Sengupta, S., Wu, N., Varvello, M., Jana, K., Chen, S., & Han, B. "From WebGL to WebGPU: A Reality Check of Browser-Based GPU Acceleration." *Proceedings of the 2025 ACM Internet Measurement Conference*. (2025).
14. Han, S., Mao, H., & Dally, W. J. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding." *arXiv preprint arXiv:1510.00149* (2015).
15. Pavlenko, A., Askarbekuly, N., Megha, S., & Mazzara, M. "Micro-frontends: application of microservices to web front-ends." *J. Internet Serv. Inf. Secur.* 10.2 (2020): 49-66.
16. Parnas, D. L. "On the criteria to be used in decomposing systems into modules." *Communications of the ACM* 15.12 (1972): 1053-1058.
17. Parnas, D. L. "On the criteria to be used in decomposing systems into modules (1972)." *Communications of the ACM* 5.12 (2002).
18. Kairouz, P., & McMahan, H. B. "Advances and open problems in federated learning." *Foundations and trends in machine learning* 14.1-2 (2021): 1-210.
19. Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., & Zimmermann, T. "Software engineering for machine learning: A case study." *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, (2019).

Source of support: Nil; **Conflict of interest:** Nil.

Cite this article as:
Anantharamu, A. M. “Framework-Agnostic Web Components for Scalable ML Integration” *Sarcouncil Journal of Engineering and Computer Sciences* 5.4 (2026): pp 131-143.

Publisher: SARC Publisher