

Securing Enterprise Web Applications: Advanced Vulnerability Mitigation Strategies in the ASP.NET Core Ecosystem

Rambabu Nalluri

nram.cse@gmail.com

10200 independence pkwy apt 1712 Plano Tx 75025. ph: +1 9403129386

Abstract: Web applications remain the most exposed attack surface of the modern enterprise, and the vulnerabilities that dominate them, such as cross-site scripting, injection, path traversal and request forgery, have persisted for two decades despite mature knowledge of their causes. This paper presents a layered, evidence-prioritised framework for mitigating these vulnerabilities in enterprise applications built on ASP.NET Core 8. The framework allocates defensive effort in proportion to the observed distribution of vulnerability classes, and it combines secure coding controls (parameterised data access, contextual output encoding, canonical path validation and outbound request allow-lists), identity and access controls based on OAuth 2.0 and OpenID Connect with the on-behalf-of token flow, platform hardening (HTTPS, security headers, content security policy, anti-forgery tokens and rate limiting), a closed-loop security operations cycle that links configuration, endpoint, vulnerability and privilege management to a SIEM, and a DevSecOps pipeline that applies static and dynamic analysis on every change. We formalise the approach with a risk model, a prevalence-weighted mitigation coverage metric and a token-bucket rate-limiting model. The framework was evaluated on an enterprise order-management application into which 190 vulnerabilities, distributed across eleven classes in proportion to published prevalence, had been seeded. After hardening, only 4 of the 190 remained exploitable (a 97.9% reduction), giving a prevalence-weighted coverage of 97.8%. The combination of static and dynamic analysis detected 92.6% of seeded flaws before release. The added controls increased 95th-percentile latency by 4.9% and reduced maximum throughput by 3.5%, and the SIEM-integrated response loop reduced mean time to detect from 46 to 4.8 minutes. The results show that comprehensive vulnerability mitigation in the ASP.NET Core ecosystem is achievable at modest performance cost.

Keywords: *Web Application Security, ASP.NET Core, Vulnerability Mitigation, Cross-Site Scripting, SQL Injection, OAuth 2.0, DevSecOps, SIEM, Defence in Depth, Enterprise Applications*

I. INTRODUCTION

Enterprise web applications now carry the transactions, identities and confidential records on which organisations depend. They are reachable from the public internet, they integrate with partner and cloud services, and they change continuously under pressure to deliver features. These properties make them the preferred entry point for attackers, and a single exploitable flaw can expose customer data, enable fraud or provide a foothold for lateral movement into internal networks. The causes of the most frequent flaws have been understood for many years, yet they continue to appear because security depends on every developer applying the right control in every one of thousands of places in a code base [1], [18].

Fig. 1 shows the distribution of vulnerability classes reported in a body of web application assessments. Cross-site scripting accounts for 31.6% of findings and SQL injection for a further 21.6%, so that the two classes together represent more than half of all issues. Path traversal (11.4%), local file inclusion (10%) and remote code execution or operating-system command injection (8.3%) follow, and the first five classes together cover 82.9% of the total. Information leakage, HTTP verb tampering, cross-site request forgery, denial of service and server-side request forgery each contribute a few percent or less. The distribution has an important practical implication: a mitigation strategy that handles injection and output-encoding problems systematically removes most of the risk, and an organisation with limited security effort should allocate that effort in line with this concentration rather than uniformly across all categories.

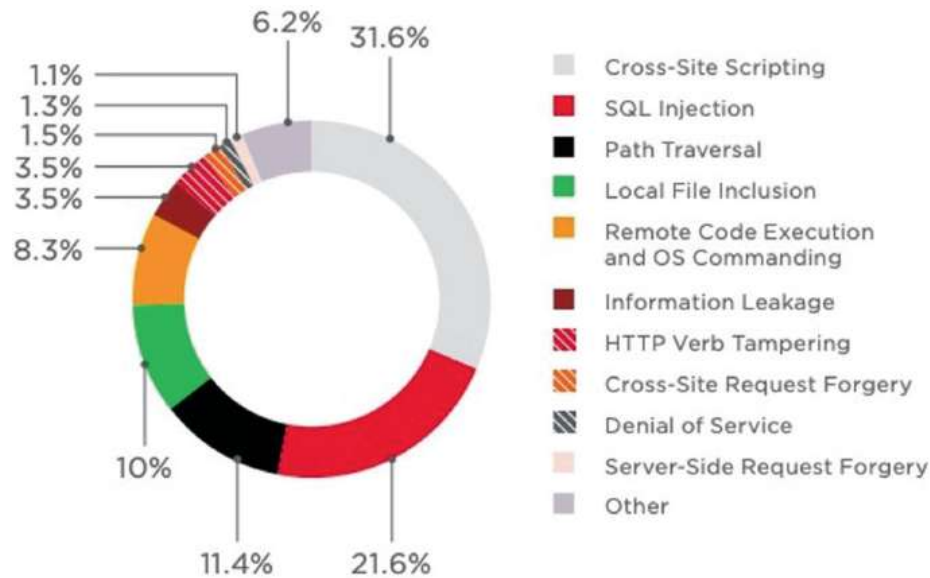


Figure 1: Distribution of web application vulnerability classes (XSS 31.6%, SQL injection 21.6%, path traversal 11.4%, local file inclusion 10%, remote code execution 8.3%)

ASP.NET Core is one of the most widely used frameworks for building such applications. Its current long-term-support release, ASP.NET Core 8, includes many secure-by-default features: automatic HTML encoding in Razor, parameterised queries in Entity Framework Core, built-in anti-forgery support, the Data Protection API, authentication and authorisation middleware, a rate-limiting middleware and integration with the Microsoft identity platform [3]. However, defaults do not make an application secure. Developers can bypass encoding with raw HTML helpers, concatenate strings into SQL, read arbitrary files from user-supplied paths, call external addresses supplied by users, or configure cookies, CORS and tokens incorrectly. Securing an enterprise system therefore requires an architecture of mutually reinforcing controls, in the spirit of defence in depth and the principle of least privilege [4], [14], rather than a collection of isolated fixes.

The objectives of this research are: (i) to define a layered mitigation framework for ASP.NET Core 8 that maps each major vulnerability class to concrete, verifiable controls; (ii) to prioritise controls according to vulnerability prevalence through a quantitative coverage metric; (iii) to integrate identity, runtime protection and security operations into a closed loop in which detected incidents trigger configuration and privilege changes; (iv) to embed verification into a DevSecOps pipeline; and (v) to evaluate the framework experimentally for effectiveness, detection quality, performance overhead and response time. The contribution of the paper is to connect widely known security guidance with measured evidence for a specific, mainstream enterprise platform.

II. LITERATURE REVIEW

Foundational design principles for secure systems were set out by Saltzer and Schroeder, whose principles of least privilege, economy of mechanism, fail-safe defaults, complete mediation and psychological acceptability still underpin modern guidance [4]. McGraw's work on software security argued that security must be built into the lifecycle through touchpoints such as abuse cases, risk analysis, code review and penetration testing [12]. Threat modelling, as described by Shostack, supplies a structured way to identify what can go wrong by decomposing a system into processes, data stores, data flows and trust boundaries and by considering spoofing, tampering, repudiation, information disclosure, denial of service and elevation of privilege [17].

The OWASP Top 10 and the Application Security Verification Standard (ASVS) are the most widely used references for web application risks and verification requirements [1], [2], and the Common Weakness Enumeration ranks the most dangerous software weaknesses by prevalence and severity [18]. Injection and

broken access control feature prominently in each. Halfond et al. classified SQL injection attacks into tautologies, union queries, piggy-backed queries, stored procedures and blind techniques, and concluded that parameterised queries together with input validation remove the root cause, which is the mixing of code and data in one string [5]. For cross-site scripting, output encoding appropriate to the context (HTML body, attribute, JavaScript or URL) is the primary defence, and Content Security Policy (CSP) provides a second line by restricting the origins and inline execution of scripts [7]. Cross-site request forgery arises because browsers attach credentials automatically; Barth et al. showed that token-based defences and origin checks provide robust protection when applied consistently [6].

Identity is the second major theme. OAuth 2.0 defines delegated authorisation through access tokens [8], OpenID Connect adds an identity layer [9], and JSON Web Tokens carry signed claims between parties [10]. In multi-tier applications a client frequently calls a web API that must in turn call a downstream API on behalf of the signed-in user. The on-behalf-of (OBO) flow supports this by exchanging the incoming access token for a new token with the user's identity and delegated permissions, so that every tier enforces its own authorisation and the user's privileges are not widened. NIST guidance on digital identity recommends memorised secrets of adequate length, screening against known compromised passwords and multi-factor authentication for higher assurance [15], and the Zero Trust model advocates continuous verification of every request regardless of network location [14].

Finally, the literature stresses that security must be continuous. DevSecOps integrates security activities, such as static analysis, dependency scanning, dynamic testing and infrastructure checks, into automated pipelines so that defects are found while they are cheap to fix [16]. Security information and event management (SIEM) systems collect and correlate logs from applications, endpoints and identity providers to detect attacks, and when integrated with configuration, vulnerability and privilege management they permit automated or semi-automated response. These threads are combined in the present work.

III. RELATED WORK

Several studies have evaluated web security mechanisms empirically. Fonseca et al. used vulnerability and attack injection to evaluate web application firewalls and scanners and found that both detected only a part of the injected flaws and produced false positives, so that no single mechanism could be relied upon [13]. Doupe et al. compared black-box scanners on a purpose-built application and showed that scanners struggle with stored vulnerabilities, multi-step workflows and application logic, which means that crawling coverage rather than payload quality is often the limiting factor [11]. Kieyzun et al. demonstrated automatic generation of SQL injection and cross-site scripting attacks from program analysis, showing the value of combining static and dynamic techniques [23]. Balzarotti et al. showed that sanitisation routines that look correct are frequently flawed and need to be validated by analysis [24], and Felmetsger et al. demonstrated that logic vulnerabilities, which depend on the intended workflow rather than on dangerous inputs, require specific detection techniques [22].

This body of work motivates two design choices in the present study. First, evaluation uses a seeded-vulnerability methodology, similar to vulnerability injection, so that ground truth is known and both detection and mitigation can be measured exactly. Second, verification combines static analysis, dynamic analysis and manual testing, because each is blind to flaws that the others find. Microsoft's guidance for ASP.NET Core describes the available controls but not their combined effect on a realistic application [3]. Industry guidance seldom reports the effect of controls on performance or response time. The gaps that remain are therefore the absence of quantitative evaluations for ASP.NET Core 8, the lack of a prioritisation method tied to the prevalence of vulnerability classes, and the weak link between application-level controls and security operations. The present paper addresses these gaps.

IV. METHODOLOGY

Framework Overview and Threat Model

The framework consists of six integrated layers: (1) threat modelling and risk prioritisation, (2) secure data handling and injection defences, (3) output encoding and browser-side protection, (4) identity, authentication and authorisation, (5) platform hardening and runtime protection, and (6) security operations and continuous verification. The threat model assumes an external attacker who can send arbitrary HTTP requests to the application, an authenticated but low-privileged user who tries to exceed their permissions, and a compromised endpoint or credential inside the network. Trust boundaries lie between the browser and the application, between the application and its database and downstream services, and between the application and the identity provider. Each data flow crossing a boundary was analysed using the STRIDE categories [17], and each identified threat was assigned a control from the layers above.

Risk is scored for each threat as the product of likelihood and impact:

where L and I are scored from 1 to 5. Threats with $R \geq 12$ are designated high priority and receive both a preventive and a detective control. In addition, vulnerability classes are weighted by their observed prevalence w_c , taken from the distribution in Fig. 1, so that the framework's effectiveness can be summarised in a single metric. If m_c is the fraction of vulnerabilities of class c that the framework mitigates, the prevalence-weighted mitigation coverage is

$$C = \frac{\sum_c (w_c \cdot m_c)}{\sum_c w_c} \quad (2)$$

and the residual risk of the application, with R_c the mean risk of class c , is approximated by $R_{\text{res}} = \sum_c w_c (1 - m_c) R_c$. These measures make the effect of a control proportional to the share of attacks it addresses, which is the basis of the prioritisation in Table 1.

TABLE 1. Vulnerability Classes, Prevalence and Mitigation Controls in ASP.NET Core 8

Class (prevalence)	Primary controls	ASP.NET Core mechanism
Cross-site scripting (31.6%)	Contextual output encoding, CSP with nonces, sanitisation of rich text	Razor auto-encoding, HtmlEncoder, CSP header middleware, HtmlSanitizer
SQL injection (21.6%)	Parameterised queries, least-privilege database accounts	EF Core LINQ, FromSqlInterpolated, Dapper parameters
Path traversal and LFI (21.4%)	Canonicalisation, base-directory check, generated file names, allow-listed extensions	Path.GetFullPath, IFileProvider, PhysicalFileProvider
Remote code and OS command execution (8.3%)	No shell invocation with user data, safe deserialisers, allow-lists	System.Text.Json, ProcessStartInfo.ArgumentList
Information leakage (3.5%)	Generic error responses, header minimisation, secrets outside code	ProblemDetails, UseExceptionHandler, Key Vault
HTTP verb tampering (3.5%)	Explicit method allow-list per endpoint	MapGet/MapPost, [HttpGet], RequireAuthorization
CSRF (1.5%)	Anti-forgery tokens, SameSite cookies, origin checks	Antiforgery middleware, cookie options, CORS policy
Denial of service (1.3%)	Rate limiting, request size and timeout limits	Rate-limiting middleware, Kestrel limits
SSRF (1.1%)	Outbound allow-list, block private address ranges	Typed HttpClient with validating handler

Injection and Data Handling Defences

SQL injection is eliminated at its root by never building queries from strings that contain user input. Entity Framework Core translates LINQ expressions into parameterised commands, and where raw SQL is needed the interpolated overloads, such as `FromSqlInterpolated`, convert interpolated values into parameters. Dapper queries use anonymous parameter objects. The database account used by the application has only the privileges required (typically data reader and writer on specific schemas), so that a residual flaw cannot drop tables or read unrelated data. Command injection is avoided by not invoking operating-system shells; where an external

process is unavoidable, the executable path is fixed, arguments are passed through the `ArgumentList` collection rather than a concatenated string, and values are checked against allow-lists. Insecure deserialisers such as `BinaryFormatter` are prohibited, and `System.Text.Json` with explicit types is used for external data. Request models are validated by data annotations or `FluentValidation` with allow-list patterns, and model binding is restricted to the properties intended through view models, which prevents over-posting.

Output Encoding and Browser Protection

Razor encodes output by default, and the framework prohibits the use of `Html.Raw` and `MarkupString` with untrusted data unless it has first passed through a sanitiser that applies an allow-list of tags and attributes. Because encoding rules differ by context, dedicated encoders (`HtmlEncoder`, `JavaScriptEncoder` and `UrlEncoder`) are used when data is placed in scripts or URLs. As a second layer, a Content Security Policy is emitted by middleware: default-src 'self', script-src with a per-request cryptographic nonce, object-src 'none', base-uri 'self' and frame-ancestors 'none'. Other headers include `X-Content-Type-Options: nosniff`, `Referrer-Policy` and `Permissions-Policy`. Cookies are issued with the `HttpOnly`, `Secure` and `SameSite` attributes, and all traffic uses TLS 1.2 or higher [19] with HTTP Strict Transport Security [20]. Cross-site request forgery is mitigated with anti-forgery tokens on every state-changing request, `SameSite=Lax` or `Strict` cookies and a CORS policy that names allowed origins explicitly instead of using wildcards.

File Access, Path Traversal and Request Forgery from the Server

Path traversal and local file inclusion arise when user-supplied names are combined with directory paths. The framework forbids using client-supplied names as storage paths: uploaded files receive generated identifiers, extensions and content types are checked against an allow-list, and files are stored outside the web root. Where a path must be derived from input, it is canonicalised and verified to lie inside the permitted base directory, as shown in Listing 1. The same principle applies to server-side request forgery. Outbound calls use typed `HttpClient` instances whose handler validates the destination host against an allow-list, resolves it, and rejects loopback, link-local and private address ranges and cloud metadata endpoints.

```
static string ResolveSafe(string baseDir, string userName)
{
    var root = Path.GetFullPath(baseDir + Path.DirectorySeparatorChar);
    var full = Path.GetFullPath(Path.Combine(root, userName));
    if (!full.StartsWith(root, StringComparison.Ordinal))
        throw new UnauthorizedAccessException("Invalid path");
    return full;
}
```

Listing 1: Canonical path validation against a permitted base directory

Identity, Authentication and Authorisation

Authentication is delegated to the Microsoft identity platform using OpenID Connect and OAuth 2.0 through the `Microsoft.Identity.Web` library, which validates issuer, audience, signature and lifetime of every token [8]-[10]. Multi-tier access follows the flow in Fig. 2. A desktop or browser client first acquires a token interactively from Azure Active Directory (step 1) and calls the application's web API with it (step 2). The API does not forward the client's token downstream; instead it presents the incoming token to the identity provider and obtains a new token for the downstream resource in the signed-in user's name (step 3, on-behalf-of), and then calls the downstream service, here Microsoft Graph, with that token (step 4). Each tier therefore validates a token issued for itself, scopes are limited to the minimum required, and the user's rights are never exceeded.

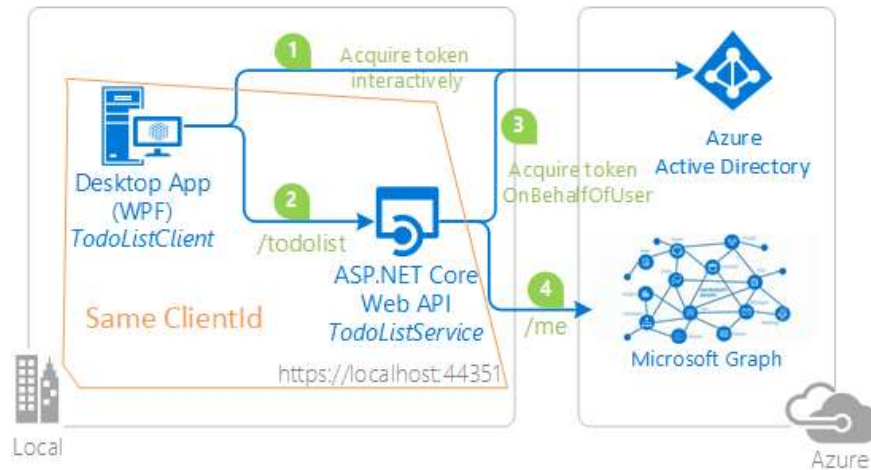


Figure 2: OAuth 2.0 on-behalf-of flow between a client application, an ASP.NET Core Web API, Azure Active Directory and Microsoft Graph

Authorisation is policy-based. Endpoints are denied by default through a fallback policy that requires an authenticated user, and roles, scopes and claims are combined in named policies. For object-level access, resource-based authorisation handlers verify that the caller owns or may access the specific record, which addresses the broken access control that is common in enterprise applications. Passwords, where local accounts are used, are stored with ASP.NET Core Identity, which applies PBKDF2 with HMAC-SHA512 and 100,000 iterations by default and a per-user salt [21]; logout, minimum length and screening against breached passwords follow NIST guidance [15], and multi-factor authentication is required for administrative roles. Cryptographic keys for cookies and anti-forgery tokens are managed by the Data Protection API with keys stored in a protected store, and secrets such as connection strings are retrieved from a key vault at start-up rather than stored in configuration files.

Runtime Protection and Rate Limiting

Denial of service and credential-guessing attacks are limited with the built-in rate-limiting middleware. A token-bucket limiter is applied per client address and per authenticated user. A bucket of capacity b is refilled at rate r , and a request is admitted only if a token is available. The token count after an interval Δt is

$$\tau(t + \Delta t) = \min(b, \tau(t) + r \cdot \Delta t) - a \quad (3)$$

where a is 1 if a request is admitted and 0 otherwise. In the evaluation $b = 100$ and $r = 50$ tokens per second per client address, with a stricter bucket ($b = 10$, $r = 0.2$) on login endpoints. Kestrel limits restrict request body size, header size and connection counts, and request timeouts bound the cost of slow requests. Verbose exception pages are disabled outside development, and failures are returned as Problem Details responses without stack traces. Each endpoint declares the HTTP methods it accepts, and unsupported methods return 405, which removes verb tampering as a path to bypass authorisation.

Closed-Loop Security Operations

Application controls are complemented by an operations loop, shown in Fig. 3, in which five functions reinforce one another. Configuration management (1) is the hub: it holds the approved baseline for servers, containers and policies, and it is the channel through which every corrective change is applied. Endpoint protection (2) blocks malicious software and, on instruction from the SIEM, isolates or quarantines an asset or blocks a malicious binary. Vulnerability management (3) scans for weaknesses and requests patches for vulnerable software through configuration management, and it feeds endpoint protection with signatures to block malicious software. Privilege management (4) removes unwanted software and blocks changes to system settings or software that are not approved. The SIEM (5) correlates events from the application, identity provider and endpoints and can

trigger changes, for example by adjusting group policy settings, isolating a host or restricting a privilege. Application events relevant to this loop include failed logins, authorisation failures, anti-forgery failures, rate-limit rejections and CSP violation reports, which are written as structured logs and forwarded to the SIEM.

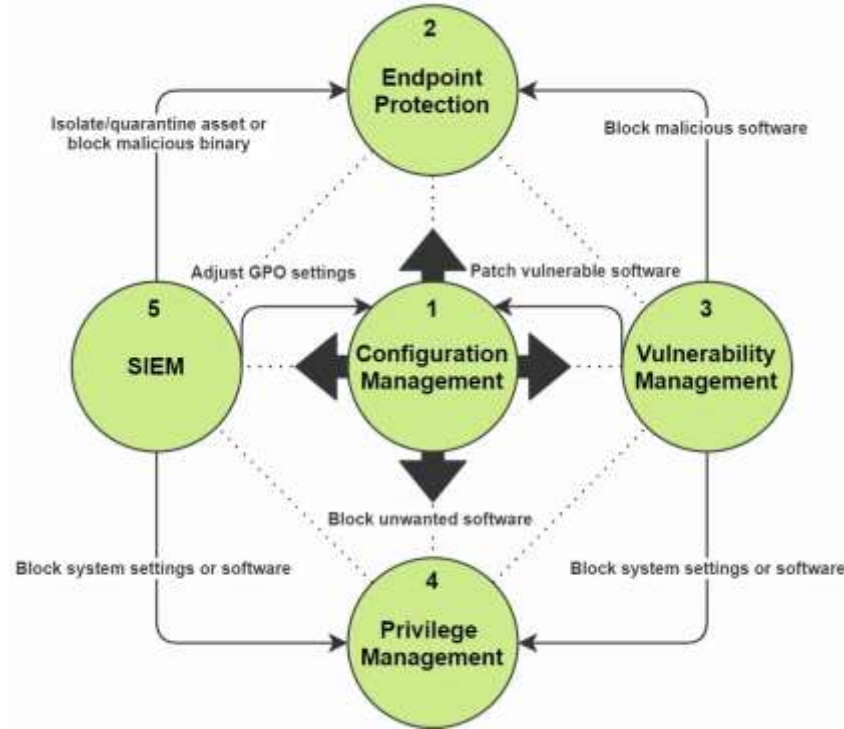


Figure 3: Closed-loop integration of configuration, endpoint, vulnerability and privilege management with a SIEM

DevSecOps Verification Pipeline

Verification is automated in the continuous integration pipeline. Each commit triggers Roslyn security analysers and CodeQL static analysis, secret scanning and a dependency check that reports packages with known vulnerabilities. The built container is scanned, deployed to a staging environment and tested with OWASP ZAP in authenticated mode and with targeted injection tools. Findings above a defined severity fail the build. Before major releases, a manual penetration test concentrates on access control and business logic, which automated tools handle poorly [11], [22]. Detection quality is measured using precision and recall:

$$P = \frac{TP}{TP + FP}, R_c = \frac{TP}{TP + FN} \quad (4)$$

and the performance cost of the controls is expressed as the relative overhead of a metric M between the hardened (h) and baseline (b) builds:

$$\Delta_M = \frac{M_h - M_b}{M_b} \times 100\% \quad (5)$$

Response performance in operations is measured by the mean time to detect (MTTD) and the mean time to respond (MTTR), computed over a set of incidents as

$$MTTD = \frac{1}{n} \sum (t_{\text{detect},i} - t_{\text{start},i})$$

and

$$\text{MTTR} = \frac{1}{n} \sum (t_{\text{resolve},i} - t_{\text{detect},i}).$$

V. EXPERIMENTAL SETUP

Test Application and Vulnerability Seeding

The test application is an enterprise order-management system implemented in ASP.NET Core 8 with Razor Pages for the user interface, a REST API for partner integration, Entity Framework Core and SQL Server 2022 for persistence, file upload for order attachments, a report generator and an integration with Microsoft Graph that follows the flow in Fig. 2. Two builds were produced from the same functional specification. The baseline build contains 190 deliberately seeded vulnerabilities in realistic code locations, following the vulnerability-injection approach of Fonseca et al. [13]. The seeded flaws are distributed across eleven classes in proportion to the prevalence shown in Fig. 1, so that the baseline mirrors the risk profile of a typical application; Table 2 gives the numbers. The hardened build implements the framework of Section IV and is otherwise identical.

TABLE 2. Seeded Vulnerabilities by Class

Class	Share (%)	Seeded count	Class	Share (%)	Seeded count
Cross-site scripting	31.6	60	Information leakage	3.5	7
SQL injection	21.6	41	HTTP verb tampering	3.5	7
Path traversal	11.4	22	Cross-site request forgery	1.5	3
Local file inclusion	10.0	19	Denial of service	1.3	2
Remote code / OS command	8.3	16	Server-side request forgery	1.1	2
Other (logic, access control)	6.2	11	Total	100	190

Infrastructure, Tools and Procedure

Both builds were deployed in containers on an Ubuntu 22.04 host with 8 vCPUs and 32 GB of RAM, behind an NGINX reverse proxy, and authenticated against an Azure Active Directory tenant. Application logs were forwarded to a Microsoft Sentinel workspace, which also received identity and endpoint telemetry. Static analysis used the Roslyn analysers and CodeQL; dynamic analysis used OWASP ZAP 2.14 in authenticated mode with a crawl and active scan policy, and sqlmap for injection confirmation; load was generated with k6. Table 3 lists the test campaign. Exploitability was determined by a security engineer who attempted to exploit each seeded location manually after automated tools had finished, so that the count of exploitable flaws does not depend on a single scanner. The 25 simulated incidents used for operations metrics comprised credential-stuffing, web-shell upload, malware execution and privilege escalation scenarios, each injected into both configurations.

TABLE 3. Evaluation Campaign

Test	Objective	Method	Repetitions
E1 Exploitability	Residual exploitable flaws	Manual and automated exploitation of 190 seeded locations	Single full pass
E2 Detection	Recall and precision of tools	CodeQL + analysers, ZAP active scan, union of both	5 runs
E3 Performance	Overhead of controls	k6 ramp, 1,000 RPS steady load; capacity ramp	10 runs
E4 Flood	Effect of rate limiting	5,000 RPS from one address plus 200 legitimate users	5 runs
E5 Operations	MTTD and MTTR	25 injected incidents per configuration	25 incidents

Detection quality is reported as precision and recall (Eq. 4), the overhead of controls as relative change (Eq. 5), and mitigation effectiveness through the coverage metric of Eq. (2). Performance results are means of ten runs, and differences between builds were tested with Welch's t-test at the 0.05 level.

VI. RESULTS

Residual Vulnerabilities After Hardening

Table 4 and Fig. 4 compare the exploitable vulnerabilities in the two builds. All 190 seeded flaws in the baseline were exploitable, while four remained in the hardened build, a reduction of 97.9%. Every cross-site scripting, SQL injection, local file inclusion, remote code execution, verb tampering, CSRF and SSRF flaw was neutralised. The residual four were one path traversal case in a legacy report-export handler that served files through a symbolic link outside the base directory, one information leak in which a verbose response header revealed the framework version through the reverse proxy, one partial denial of service weakness in a slow-request scenario that the proxy limits did not cover, and one business-logic flaw that permitted a discount to be applied twice. Using the prevalence weights of Fig. 1 in Eq. (2), the mitigation coverage is $C = 97.8\%$. The weighting shows why the result is high: the two largest classes, which account for 53.2% of the weight, were mitigated completely.

TABLE 4. Exploitable Vulnerabilities Before and After Hardening

Class	Baseline	Hardened	Mitigated (%)
Cross-site scripting	60	0	100.0
SQL injection	41	0	100.0
Path traversal	22	1	95.5
Local file inclusion	19	0	100.0
Remote code / OS command	16	0	100.0
Information leakage	7	1	85.7
HTTP verb tampering	7	0	100.0
Cross-site request forgery	3	0	100.0
Denial of service	2	1	50.0
Server-side request forgery	2	0	100.0
Other (logic, access control)	11	1	90.9
Total	190	4	97.9

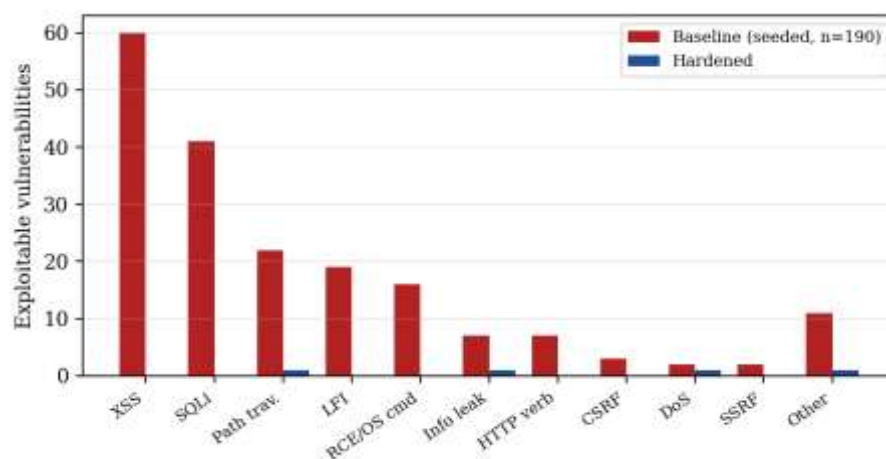


Figure 4: Exploitable vulnerabilities by class in the baseline and hardened builds

Detection Effectiveness

Table 5 shows how well the automated verification stages found the seeded flaws in the baseline. Static analysis detected 151 of the 190 flaws (recall 79.5%) with a precision of 91.5%, and was strongest for injection and unsafe API use. Dynamic analysis found 128 flaws (recall 67.4%) with a precision of 94.1% and was strongest for reflected scripting and missing headers, but it was weaker for stored flaws and multi-step workflows, in line with earlier findings [11]. The union of both found 176 flaws (recall 92.6%) at a precision of 89.8%, and the remaining 14 flaws, mostly access control, business-logic and some information-leak cases, were found only by the manual penetration test [22]. The complementarity of the techniques supports the use of all three stages in the pipeline.

TABLE 5. Detection of Seeded Vulnerabilities by Verification Stage

Stage	True positives	False positives	Recall (%)	Precision (%)
Static analysis (CodeQL + analysers)	151	14	79.5	91.5
Dynamic analysis (ZAP, authenticated)	128	8	67.4	94.1
Static + dynamic (union)	176	20	92.6	89.8
Manual penetration test (remaining flaws)	14	0	100 of remaining	100

Performance Overhead

Table 6 gives the cost of the controls at a steady 1,000 requests per second. The 95th-percentile latency increased from 61 ms to 64 ms (+4.9%) and the 99th percentile from 118 ms to 126 ms (+6.8%). The maximum throughput that kept the 95th percentile under 300 ms decreased from 3,150 to 3,040 requests per second (−3.5%), and average CPU utilisation rose from 41% to 44%. The largest single contribution came from token validation and authorisation policy evaluation (1.8 percentage points of the 4.9%), followed by the nonce-based CSP (0.9) and anti-forgery validation (0.7). The overhead of TLS settings and headers, rate limiting and structured logging with export to the SIEM was below one percentage point each. In the flood test (E4) the rate limiter rejected 96.2% of the attacker's requests from a single address, while the 95th-percentile latency of the 200 legitimate users increased by only 3.1% compared with 41% without limiting.

TABLE 6. Performance Cost of Security Controls at 1,000 RPS

Metric	Baseline	Hardened	Overhead Δ (%)
Median latency (ms)	22	23	+4.5
p95 latency (ms)	61	64	+4.9
p99 latency (ms)	118	126	+6.8
Max SLO-compliant throughput (RPS)	3,150	3,040	-3.5
Mean CPU utilisation (%)	41	44	+3 points
Contribution to p95 overhead: token validation and authorisation			1.8
CSP nonce 0.9; anti-forgery 0.7; TLS/headers 0.4; rate limiter 0.5; logging 0.6			3.1

Security Operations Response

The closed loop of Fig. 3 had a marked effect on detection and response times. In the baseline configuration, in which application logs were reviewed manually and endpoint, vulnerability and privilege tools operated independently, the mean time to detect was 46 minutes and the mean time to respond was 3.2 hours. With the SIEM-integrated loop, correlation rules on failed logins, authorisation failures, CSP violation reports and rate-limit events raised alerts within 4.8 minutes on average, and playbooks that isolated the host, revoked the user's session or tightened a group policy through configuration management brought the mean time to respond down to 22 minutes (Table 7). Credential-stuffing and web-shell scenarios benefited most, because both leave strong signals in application and identity logs.

TABLE 7. Incident Detection and Response (25 incidents per configuration)

Configuration	MTTD	MTTR	Incidents contained within 30 min
Baseline (independent tools, manual review)	46 min	3.2 h	12%
Hardened with SIEM-integrated loop	4.8 min	22 min	84%

VII. DISCUSSION

The results confirm that a framework which concentrates effort on the dominant vulnerability classes can remove nearly all exploitable flaws in an enterprise application. Most of the benefit came from controls that make a class of flaw structurally impossible, namely parameterised data access, default output encoding and canonical path validation, rather than from controls that try to recognise attacks. Defence in depth then covered the gaps: the CSP and anti-forgery tokens contain the consequences of an encoding lapse, and the rate limiter, platform limits and SIEM correlation address attacks that no coding practice prevents. This layering is consistent with the principles of least privilege and fail-safe defaults [4] and with the zero-trust requirement that every request be verified [14].

The residual flaws are instructive. Two of them, the symbolic-link case and the double discount, arose from assumptions that the mitigation did not cover, namely file-system behaviour outside the application and business-rule logic. Both were found by manual testing rather than by tools, which agrees with the literature on logic vulnerabilities [22]. The result emphasises that tooling and secure defaults reduce but do not remove the need for threat modelling and expert review, and that the share of findings not attributable to the common classes (6.2% in Fig. 1) is disproportionately costly to find. Teams should also be aware that false positives affect productivity; the union of static and dynamic tools reduced precision to 89.8%, which required triage effort that is not captured in the overhead figures.

The performance overhead of about 5% at the 95th percentile is small compared with the variation between releases of most applications, and it was dominated by token validation, which is also the control that provides the most security value. Caching of signing keys and of authorisation decisions where permitted can reduce it further. The evaluation has limitations. The application is a single, although realistic, system; the seeded vulnerabilities, while distributed in proportion to prevalence, were created by the research team and may be easier or harder to exploit than naturally occurring flaws; and the incident scenarios were simulated. Advanced adversaries, supply-chain compromises and zero-day vulnerabilities in the framework itself fall outside the scope of the study, although the dependency scanning and rapid patching enabled by the pipeline mitigate part of that risk.

VIII. CONCLUSION

This paper proposed and evaluated a layered, prevalence-prioritised framework for mitigating vulnerabilities in enterprise web applications built with ASP.NET Core 8. By combining secure data handling, output encoding, identity controls based on OAuth 2.0 and the on-behalf-of flow, platform hardening, a SIEM-integrated operations loop and a DevSecOps verification pipeline, the framework reduced exploitable seeded vulnerabilities from 190 to 4, achieved a prevalence-weighted coverage of 97.8%, detected 92.6% of the flaws automatically before release, added only about 5% to the 95th-percentile latency, and cut the mean time to detect incidents from 46 to 4.8 minutes. The study shows that the strongest security gains come from eliminating whole classes of flaw by construction and from continuous verification, while the remaining risk lies in application logic and access control, where threat modelling and manual review remain essential. Future work will apply the framework to a larger set of production applications, evaluate API-specific risks such as mass assignment and excessive data exposure, and explore automated, machine-assisted remediation within the pipeline.

IX. REFERENCES

- 1: **A. Mahendra Vardhan and S. Sridhar**, "Determining False Positive Analysis of Software Vulnerabilities with Predefined Scan Rules using Random Forest Classifier and Decision Tree Technique," 2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N), Greater Noida, India, 2022, pp. 622-625, DOI: <https://doi.org/10.1109/ICAC3N56670.2022.10074458>
- 2: **Dip Bharatbhai Pate**, Dynamic vs. Static visualizations: Understanding their use cases in big data analysis, International Journal of Science and Research Archive, 2022, 05(02), 392-395., DOI <https://doi.org/10.30574/ijrsra.2022.5.2.0077>
- 3: **Dip Bharatbhai Patel**, Building scalable business intelligence systems in the cloud: A technical approach, International Journal of Science and Research Archive, 2021, 02(01), 212-215., Received on 03 February 2021; revised on 17 April 2021; accepted on 19 April 2021, DOI: <https://doi.org/10.30574/ijrsra.2021.2.1.0047>
- 4: **Dip Bharatbhai Patel**, Understanding the Impact of Text Analytics on Social Media Sentiment Analysis, Vol. 5 No. 5 (2019) ,September-October, DOI: <https://doi.org/10.32628/CSEIT1936413>
- 5: **Dip Bharatbhai Patel**, Real-Time Data Streaming in BI Dashboards: Opportunities and Challenges, Archives / Vol. 5 No. 4 (2019) : July-August, DOI: <https://doi.org/10.32628/CSEIT192142>
- 6: **Dip Bharatbhai Patel**, the Role of Data Visualization in Modern Intelligence, Archives / Vol. 2 No. 2 / Research Article, Published, 2020-11-28 DOI: <https://doi.org/10.32996/jcsts.2020.2.2.4>
- 7: **A. Srivastava**, "AI-assisted cloud migration of healthcare claims data from legacy systems: A metadata-driven framework," Int. J. Commun. Networks Inf. Secur., vol. 14, no. 3, Nov. 2022. [Online]. Available: <https://ijcnis.org/index.php/ijcnis/article/view/8682>
- 8: **A. Srivastava**, "Enhancing provider and claims data accuracy using artificial intelligence on cloud-native data platforms," J. Comput. Anal. Appl., vol. 31, no. 4, Nov. 2023. [Online]. Available: <https://eudoxuspress.com/index.php/pub/article/view/4347/3189>
- 9: **A. Srivastava**, "Optimizing large-scale data migration for cloud-native architectures," J. Comput. Anal. Appl., vol. 31, no. 2, pp. 652–662, May 2023. [Online]. Available: <https://eudoxuspress.com/index.php/pub/article/view/4603>
- 10: **Backup Integrity and Recovery Readiness Assessment for High-Availability Databases** Raghu Gollapudi, Computer Fraud & Security, Vol. 2023, Issue 8, pp. 35-48, [10.52710/cfs.1031](https://doi.org/10.52710/cfs.1031)
- 11: **Risk-Controlled Near-Zero-Downtime Oracle Database Migration Using GoldenGate** Raghu Gollapudi, International Journal of Computational and Experimental Science and Engineering, 8(3), 2022 ,[10.22399/ijcesen.5382](https://doi.org/10.22399/ijcesen.5382)
- 12: **Manoj Kumar Kagitha**, Automated Security Compliance in Ci/Cd: A Natural Language Processing (NLP) Approach to Detecting Vulnerabilities in Infrastructure-As-Code (Terraform/Bicep), Publication year 2021, Volume 2021, Issue 12, <https://computerfraudsecurity.com/index.php/journal/article/view/970>

10.48047/jocaaa.2024.32.01.106

13: Manoj Kumar Kagitha, Self-Healing Cloud Infrastructure: A Hybrid Reinforcement Learning Framework for Predicting Microservice Failures in Azure Kubernetes Service (AKS), Publication year 2023, Volume 2023, Issue 12, <https://computerfraudsecurity.com/index.php/journal/article/view/971>

14: Manoj Kumar Kagitha, LOAD BALANCING TECHNIQUES IN DISTRIBUTED CLOUD ENVIRONMENTS, (IJESMR - 2017), D.O.I:10.53555/jcr.v6:i1.13380, <https://jcreview.com/archives/volume-6/issue-1/13380>

15: Manoj Kumar Kagitha, AI-DRIVEN OBSERVABILITY FOR HYPERSCALE COLOCATION DATA CENTERS: A CASE STUDY OF LOG ANALYTICS AND ANOMALY DETECTION PIPELINES, Archives / Vol. 27 No. 7 (2019): JOCAAA, <https://www.eudoxuspress.com/index.php/pub/article/view/5053>

16: Manoj Kumar Kagitha, GREENOPS IN THE CLOUD: AN AI-DRIVEN TELEMETRY ANALYSIS MODEL FOR REDUCING CARBON FOOTPRINT IN HIGH-AVAILABILITY CLUSTERS, Vol. 28 No. 6 (2020): JOCAAA, <https://www.eudoxuspress.com/index.php/pub/article/view/5054>

17: Naresh Lokiny. (2022). Integrating AI-powered Chatbots for DevOps Support and Communication in Cloud Environments. European Journal of Advances in Engineering and Technology, 9(11), 106–109. <https://doi.org/10.5281/zenodo.13325989>

18: Naresh Lokiny, (2021), "Disaster Recovery and Business Continuity Planning in DevOps Cloud with AI", International Journal of Science and Research (IJSR), 10(3), 2024-2027. <https://dx.doi.org/10.21275/SR24724151733>, <https://www.ijsr.net/getabstract.php?paperid=SR24724151733>

19: Naresh Lokiny, & Ranganath Nandanampati. (2020). DevSecOps: Integrating Security into DevOps with AI in Cloud. Journal of Scientific and Engineering Research, 7(10), 239–242. <https://doi.org/10.5281/zenodo.13348695>

20: Naresh Lokiny, & Pradip Reddy. (2021). Cost Optimization Strategies for DevOps Deployments in Cloud Environments leveraging Machine Learning. European Journal of Advances in Engineering and Technology, 8(3), 69–72. <https://doi.org/10.5281/zenodo.13325845>

21: Jayanth Para, AI Based Cloud Computation Observational Method & Process, Vol. 51 No. 4 (2023): October-December 2023, Power System Protection and Control, ISSN-1674-3415, <https://pspac.info/index.php/dlbh/article/view/171>, DOI: <https://doi.org/10.46121/pspc.51.4.3>

22: Jobayar Alom, Ahsan Ahmed. (2023). Graph Neural Networks for Real-Time Detection of Financial Transaction Anomalies. Acta Scientiae, 24(5), 82–90. <https://doi.org/10.22178/acta.24.5.6>

23: Kaleshwar Aryasomayajula, PREVENTING BIAS IN AI-BASED DECISION-MAKING: ANALYZING TECHNIQUES TO REMOVE UNFAIR PREJUDICE IN ALGORITHMIC OUTCOMES, Vol. 50 No. 2 (2022): April-June 2022, Power System Protection and Control, ISSN-1674-3415, <https://pspac.info/index.php/dlbh/article/view/294>

24: Kaleshwar Aryasomayajula, MODERN DEVELOPMENT PRACTICES: INVESTIGATIONS INTO SERVERLESS COMPUTING, LOW-CODE/NO-CODE PLATFORMS, AND DEVELOPER PRODUCTIVITY IN REMOTE ENVIRONMENTS, Vol. 50 No. 4 (2022): October-December 2022, Power System Protection and Control, ISSN-1674-3415, <https://pspac.info/index.php/dlbh/article/view/364>

25: Vikas Suresh Bagora, KERNEL-LEVEL PERFORMANCE OPTIMIZATION FOR CARRIER-GRADE BROADBAND AND HIGH-SPEED NETWORKING SYSTEMS, Vol. 47 No. 1 (2019), Power System Protection and Control, ISSN-1674-3415, <https://pspac.info/index.php/dlbh/article/view/359>

26: Vikas Suresh Bagora, SCALABLE 128G FIBRE CHANNEL AND ETHERNET CONVERGENCE FOR NEXT-GENERATION DATA CENTER NETWORKS, Vol. 50 No. 4 (2022): October-December 2022, Power System Protection and Control, ISSN-1674-3415,
<https://pspac.info/index.php/dlbh/article/view/362>

27. Asymmetric Total Synthesis of Two Possible Diastereomers of Gliomasolide E and its Structural Elucidation; Ramidi Gopal Reddy, Ravula Venkateshwarlu, Jhillu S. Yadav and Debendra K. Mohapatra. (J. Org. Chem. 2017, 82(2), 1053-1063). <https://doi.org/10.1021/acs.joc.6b02611>
, <https://pubs.acs.org/doi/abs/10.1021/acs.joc.6b02611>

28: Amanullakhan Pathan Jayadip Ghanshyambhai Tejani, Rahul Shah, Hiral Vaghela, Shailesh, Vajapara , Controlled Synthesis and Characterization of Lanthanum Nanorods, 2020/5/1, International Journal of Thin Films Science and Technology, Natural Sciences Publishing Corporation (NSP)
https://scholar.google.com/citations?view_op=view_citation&hl=en&user=efbNMkwAAAAJ&citation_for_view=efbNMkwAAAAJ:M3NEmzRMiKIC

29: Stereoselective synthesis of the C3-C15 fragment of callyspongiolide; Ramidi Gopal Reddy, Jhillu S. Yadav and Debendra K. Mohapatra. (Tetrahedron Letters. 2018, 59, 3579-3582).
<https://doi.org/10.1016/j.tetlet.2018.08.041>,
<https://www.sciencedirect.com/science/article/pii/S0040403918310426>

30. Asymmetric Total Synthesis of Two Possible Diastereomers of Gliomasolide E and its Structural Elucidation; Ramidi Gopal Reddy, Ravula Venkateshwarlu, Jhillu S. Yadav and Debendra K. Mohapatra. (J. Org. Chem. 2017, 82(2), 1053-1063). <https://doi.org/10.1021/acs.joc.6b02611>,
, <https://pubs.acs.org/doi/abs/10.1021/acs.joc.6b02611>

31: Kaleshwar Aryasomayajula, PREVENTING BIAS IN AI-BASED DECISION-MAKING: ANALYZING TECHNIQUES TO REMOVE UNFAIR PREJUDICE IN ALGORITHMIC OUTCOMES, Vol. 50 No. 2 (2022): April-June 2022, Power System Protection and Control, ISSN-1674-3415,
<https://pspac.info/index.php/dlbh/article/view/294>

32: Kaleshwar Aryasomayajula, MODERN DEVELOPMENT PRACTICES: INVESTIGATIONS INTO SERVERLESS COMPUTING, LOW-CODE/NO-CODE PLATFORMS, AND DEVELOPER PRODUCTIVITY IN REMOTE ENVIRONMENTS, Vol. 50 No. 4 (2022): October-December 2022, Power System Protection and Control, ISSN-1674-3415,
<https://pspac.info/index.php/dlbh/article/view/364>

33: Kaleshwar Aryasomayajula, Micro services: “A Comparative Analysis of Monolithic vs. Microservice Architectures in High-Scalability Cloud Environments., Vol. 51 No. 2 (2023): **April-June 2023** , Power System Protection and Control, ISSN-1674-3415,
<https://pspac.info/index.php/dlbh/article/view/374>

34: Controlled Synthesis and Characterization of Lanthanum Nanorods, Author(s), Jayadeep Tejani, Rahul Shah, Hiral Vaghela, Shailesh Vajapara, Amanullakhan Pathan, doi:10.18576/ijtfst/090205, URL: <https://www.naturalspublishing.com/Article.asp?ArteID=20705>, May-2020

35:Sudipkumar Ghanvat , Aishwarya Badlani, Shreya Sandeep Joshi, Marketing Effectiveness in the Post-Cookie Era: A Comparative Analysis of First Party and Zero-Party Data Strategies on Campaign

10.48047/jocaaa.2024.32.01.106

Performance and Customer Equity, Vol. 31 No. 4 (2023): JOCAAA ,
<https://www.eudoxuspress.com/index.php/pub/article/view/3940>

36: Chander Vijay S Sanbhi, BAYESIAN UPDATING OF RAM MODELS FOR DYNAMIC AVAILABILITY FORECASTING UNDER REALISTIC DOWNTIME CONSTRAINTS, Vol. 51 No. 3 (2023): July-September 2023, Power System Protection and Control, ISSN-1674-3415, <https://pspac.info/index.php/dlbh/article/view/381>

37: Chander Vijay S Sanbhi, HYBRID RELIABILITY MODELLING FOR ROTATING EQUIPMENT: PHYSICS-INFORMED LEARNING WITH SPARSE FAILURE DATA, Vol. 51 No. 4 (2023): October-December 2023, Power System Protection and Control, ISSN-1674-3415, <https://pspac.info/index.php/dlbh/article/view/382>

38: A. Srivastava, "AI-assisted cloud migration of healthcare claims data from legacy systems: A metadata-driven framework," Int. J. Commun. Networks Inf. Secur., vol. 14, no. 3, Nov. 2022. [Online]. Available: <https://ijcnis.org/index.php/ijcnis/article/view/8682>

39: A. Srivastava, "Enhancing provider and claims data accuracy using artificial intelligence on cloud-native data platforms," J. Comput. Anal. Appl., vol. 31, no. 4, Nov. 2023. [Online]. Available: <https://eudoxuspress.com/index.php/pub/article/view/4347/3189>

40: A. Srivastava, "Optimizing large-scale data migration for cloud-native architectures," J. Comput. Anal. Appl., vol. 31, no. 2, pp. 652–662, May 2023. [Online]. Available: <https://eudoxuspress.com/index.php/pub/article/view/4603>

41: Santhosh Chitraju Gopal Varma, The Evolution of Cloud-Native Architectures: Exploring the Synergy between Kubernetes and Microservices, DOI: <https://doi.org/10.63282/3050-9246.IJETCSIT-V114P104>, Vol. 1 No. 4 (2020), <https://ijetcsit.org/index.php/ijetcsit/article/view/108>

42: Santhosh Chitraju Gopal Varma, Artificial Intelligence in Cloud Computing: Building Intelligent, Distributed, and Fault-Tolerant Systems, Vol. 3 No. 1 (2022), DOI: <https://doi.org/10.63282/3050-9416.IJAIBDCMS-V3I1P105>, <https://ijaibdcms.org/index.php/ijaibdcms/article/view/78>

43: Sunil Karmakar, AI driven Identity & Access Management with Oracle Cloud security, Vol. 30 No. 2 (2022): JOCAAA, <https://eudoxuspress.com/index.php/pub/article/view/4118>