National College of Ireland

Enhancing Security of WordPress Containers on AWS:A Multitool Vulnerability Analysis.

MSc Research Project

Msc in Cloud Computing

TEJASWINI DHANESH KUMAR

StudentID:X23288957

School of Computing
National College of Ireland

Supervisor: SAI EMANI

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	TEJASWINI DHANESH KUMAR
Student ID:	X23288957
Programme:	CLOUD COMPUTING
Year:	2024-25
Module:	MSc Research Project
Supervisor:	SAI EMANI
Submission Due Date:	10-08-2025
Project Title:	**Enhancing Security of WordPress Containers on AWS:A Multitool Vulnerability Analysis.**
Word Count:	9903
Page Count:	22

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Tejaswini Dhanesh Kumar
Date:	12th September 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Enhancing Security of WordPress Containers on AWS:A Multitool Vulnerability Analysis.

TEJASWINI DHANESH KUMAR
x23288957
RESEARCH IN COMPUTING
National College of Ireland
YouTube URL: https://youtu.be/C166DzuwS3E
Deployed Application URL: wordpress-application-1853364263.us-east-1.elb.amazonaws.com

ABSTRACT

There are two main risks that containerised WordPress on AWS has; vulnerable container images and application-layer attacks. Such implementation is organised as the layered defence, which incorporates continuous scanning of the Amazon ECR/Inspector images with additional AWS WAF and EventBridge/Lambda automation. SQL-injection (SQLi) and cross-site-scripting (XSS) tests resulted in 100% blocking of SQLi payloads, and 90.95 percent mitigation of XSS with its managed and internal WordPress-specific rules (/wp-login.php, xmlrpc.php). CVEs scanned on image scan prior to deployment; activated automated alerts, optional responses. Operational overhead was very low: WAF assessment <1 ms per request, ALB routing=20 ms, Lambda notifications=300 ms after the event and ECR scan=30-60s/image. Compared to a baseline where no WAF or scanning would be in place, the framework significantly increases protection with minimal overhead cost to application runtime, and is a reusable closed-loop pattern that can be applied to any other container-based web application running on AWS.

INTRODUCTION:

1. RESEARCH BACKGROUND:

The proliferation of cloud-native technologies has revolutionized web application deployment by enabling unmatched agility, scalability, and affordability(Waseem et al., 2024). Among these technologies, containerization—through software such as Docker—is the dominant underpinning element of modern infrastructure. Simultaneously, WordPress remains the most dominant content management system (CMS), powering more than 40% of websites globally as of 2025(Schäferhoff, 2025). As more usage is seen for containerized WordPress production on Amazon Web Services (AWS), application developers and organizations increasingly rely on services such as Amazon ECS, ECR, and Lambda to run and scale applications in production. Even as these advances occurred, security concerns developed simultaneously. WordPress's extensibility through themes and plugins contributes to its attack surface, with approximately 90% of WordPress vulnerabilities originating from plugins and 6% from themes (Watson, 2025). Container environments introduce additional risks such as insecure base images, image sprawl, and runtime misconfigurations. Old security models—static Virtual Private Clouds (VPCs), basic Identity and Access Management (IAM), and perimeter-only defense—are not adequate for today's threat environment, where both application-layer attacks (e.g. SQL injection, cross-site scripting) and container-level exploits must be addressed simultaneously (Wallace, 2025). This situation demands a multi-layered, proactive security solution that converges detection, protection, and remediation capabilities tailored to containerized WordPress infrastructure.

2. PROBLEM STATEMENT:

Current security tools for AWS containerized WordPress are largely siloed and reactive. Single services such as Elastic Container Registry (ECR) for vulnerability scanning, Web Application Firewall (WAF) for filtering bad traffic, and Lambda for event-driven automation are provided by AWS. These are mostly used in silos, resulting in fragmented processes, manual intervention, and late threat response. For instance, ECR can warn on a weak image but cannot automatically impose runtime limits or trigger countermeasures. Similarly, WAF protects against standard web attacks but is not container-aware, and Lambda is underleveraged for security management.This integration lack between security elements creates primary protection vulnerabilities, contributes to operational overhead, and diminishes the resiliency of WordPress applications deployed in the cloud. The problem is not that there are not enough tools, but that there is a lack of an integrated, automated security framework that brings together application-layer and container-layer protections.

3. RESEARCH QUESTION:

How can AWS-native services—specifically Amazon ECR, AWS WAF, and AWS Lambda—be integrated into a unified security framework to detect, mitigate, and respond to WordPress-specific application and container-level threats in a scalable and automated manner?

4. RESEARCH OBJECTIVE:

The primary aim of this research is to create, design, and evaluate a holistic, multi-layered security framework for containerized WordPress deployments on AWS. The model will integrate Amazon Elastic Container Registry (ECR) for continuous vulnerability scanning, AWS Web Application Firewall (WAF) for application-layer protection, and AWS Lambda for automated threat response. Specifically, the research seeks to assess the framework's efficacy in detecting and preventing common WordPress-specific vulnerabilities such as SQL injection, cross-site scripting (XSS), and container image vulnerabilities. Furthermore, the research seeks to measure the performance overhead imposed by the framework in terms of latency, resource consumption, and system scalability under simulated workloads. Finally, the framework's performance will be contrasted with a baseline containerized WordPress implementation lacking inherent security controls, thereby enabling a comparative assessment of security efficacy and operational efficiency.

5. RESEARCH CONTRIBUTIONS:

This work has a number of important contributions to the field of cloud and container security. First, it proposes a novel, all-encompassing security model that combines vulnerability detection, real-time protection, and automatic mitigation in particular for WordPress containers on AWS. The work also presents a successful proof-of-concept of this model with real attack simulations, supporting evidence-based measurement of both effectiveness and scalability. In addition, the study quantifies the performance impact of the planned framework, providing critical data on the trade-off between security enhancement and system performance in a containerized environment. Through a focus on WordPress's unique architectural aspects in containerization, the study fills a critical literature gap where end-to-end security solutions lack development. Finally, the model built by this research develops an off-the-shelf model that can be customized to safeguard other content management systems or microservice applications on cloud-native platforms, thus extending the limits of state-of-the-art cloud security best practices.

I. LITERATURE REVIEW:

a. Local Development Environment: MAMP vs XAMPP:

The local stacks help to set up a secure sandbox ahead of cloud Infrastructure deployment Both tools, AMP and XAMPP, distribute LAMP packages on WordPress; whereas, MAMP, exerts its priorities on a simple macOS workflow, XAMPP is more versatile and multi-platform. To minimise setup friction and allow focussing on

security testing as opposed to tooling, MAMP was chosen to provide parity with the PHP / MySQL versions used later on in containers in production.

b. Cloud Platform Selection: AWS vs Other Cloud Providers:

The choice of AWS was driven by the overall richness of integrated services and developed security / compliance controls which adhere to AWS design pillars of security (Amazon Web Services, 2024). Its native developer/security tooling affords the possibility of a one-vendor DevSecOps strategy, which will be vital on a micro-research project with little bandwidth to integrate disparate dashboards or IAM constructs (Amazon Web Services, 2024).

c. Cloud Platform Selection for WordPress Hosting:

Using AWS, your WordPress can exist both on EC2/Lightsail and in the containers through ECS/EKS, allowing you to grow coverage from a single VM-hosted to a multi-container-based setup within a single security boundary (Amazon Web Services, 2024). This project is moving toward the use of containers in an attempt to provide standardisation of the build, scan images and reduce the complexity of rolling out repeatable builds.

d. AWS Native Services vs Third-Party Alternatives for Security:

Less cross-cloud credentials and the ability to do all telemetry, controls and automation within an AWS IAM and EventBridge/Lambda. ECR offers confidential, policy-controlled image management and vulnerability scanning that are closely coupled in the build/deploy processes. WAF is directly attached to ALB/CloudFront and can use managed rule groups and custom rules in code form (Amazon Web Services, n.d.-b).

e. Container Security and DevSecOps Strategies:

Research and recommendations highlight security-enhancing techniques: securing images, least privilege run-time, robust cluster/cloud management, and security monitoring (NIST, 2017). The four layers the layered model illustrates are classified in the 4C model as Code, Container, Cluster, and Cloud (Cloud Native Computing Foundation, n.d.). OWASP mentions the most common app-layer risks (SQLi/XSS) which according to them should be covered by WAF policies (OWASP Foundation, 2021).

f. Containerization and AWS Elastic Container Registry (ECR):

ECR acts as the secure registry: images are stored privately, scanned on push, continuously via Inspector and high-severity findings are event-raised to facilitate automation. This directly facilitates the supply-chain management and operates security to the left in the production line (Amazon Web Services, 2024).

g. Web Application Security: AWS Web Application Firewall (WAF):

L7 filtering with L7 Rules: WAF has managed rule groups containing patterns relating to SQLi/XSS, as well as WordPress-specific probes (e.g. /wp-login.php, xmlrpc.php) and custom rules (Amazon Web Services, n.d.-b). Because the majority of WordPress vulnerabilities start at the plugin / theme layer, a WAF layer also makes sense to reduce exploitability at the edge until a patch is ready (Patchstack, 2025; WPScan, n.d.; Wordfence, 2025).

h. Summary of Alignment with Project Objectives:

Examining the chosen tools and platforms – MAMP for local development, AWS for cloud deployment, and individual AWS services like ECR and WAF – we notice each decision is firmly rooted in modern best practices. MAMP facilitated rapid local development on macOS, an approach widely recommended for its ease in that environment. AWS, as the cloud platform, provided the extensive service offerings and stability required to .allowing tight integration and vulnerability scanning of container images, and the use of AWS WAF gives the application an essential protection layer against web threats (Amazon Web Services, 2024; Amazon Web Services, n.d.-b). following industry security best practices. Together, these choices form a cohesive approach: the utilization of mature technologies to supply a safe, direct pipe from development to production. Each

component supports the larger objective of the thesis – the creation and protection of a cloud-hosted web application with contemporary DevSecOps tools and practices. The synthesis of these tools (i.e., building in MAMP, containerizing for AWS ECR, and shielding with AWS WAF) echoes the marriage of security and development, which is the thesis goal of delivering a secure and solid application deployment. In totality, this literature review justifies the reasons for which every technology was selected above others and how, together, they address the conundrums of modern web application deployment. Each choice – be it MAMP, AWS, or single services like ECR and WAF – is validated as a solution to the respective layer of the system, hence forming a good foundation for the implementation and analysis in the later chapters of the thesis.

i. WordPress-Specific Threats and Security challenges in Container Environments:

Most known vulnerabilities are introduced by the plugin ecosystem of WordPress; the most common are SQLi/XSS types (Patchstack, 2025; WPScan, n.d.; OWASP Foundation, 2021). Containers can increase risk confirmation by an insecure image onto a huge scale however can limit the range of a blast when segmentation and the least privilege are executed (NIST, 2017). The combination of rules of WAF and image scanning minimizes both vectors (Amazon Web Services, n.d.-b).

j. AWS Security Tools

The blueprint employs ECR (image scanning), WAF (L7 filtering) and EventBridge-Lambda to provide auto-notifications/mitigations. CloudTrail and CloudWatch can be utilised to log/metric controls to help validate controls and support incident investigation (Amazon Web Services, 2024).

k. DevSecOps and Automation

Security is inbuilt- image-scanning during build, enforcing policy as code, and triggering automated response to events (Amazon Web Services, 2024; Cloud Native Computing Foundation, n.d.). This decreases exposure times and average mitigation.

l. Research Gaps and Contributions

Other previous work separately considers WordPress app-layer security or container security. Few examples of how to incorporate ECR/WAF/Lambda into a closed loop on WordPress. This paper fills that gap with an end-to-end, measurable prevention, protection, and response (Amazon Web Services, 2024; Cloud Native Computing Foundation, n.d.).

m. Research Opportunities

The next steps are to scale the ecosystem with runtime anomaly detection and drift control (e.g., even more Inspector/ Guard Duty integrations) and perhaps policy as code guardrails around WordPress plugins- all in the same AWS event pipeline (Amazon Web Services, 2024).

n. Recent Advances in Cloud-Native Container Security:

Notwithstanding the abundance of existing knowledge about cloud, container, and WordPress security, there are still some areas this thesis would seek to fill:

WordPress in DevSecOps Context: Typical WordPress deployments (especially for small to medium-sized websites) have yet to embrace DevSecOps processes in full. Much of the WordPress security literature is focused on the use of security plugins, hardening the PHP application or using managed hosting with one-click capabilities, rather than simply examining WordPress as an application inside a CI/CD pipeline with tight security gates. There is not a lot of literature on how WordPress development can be integrated into an in-date DevSecOps pipeline (containerization, automated testing, scanning, continuous deployment). This thesis bridges the gap by explaining how a WordPress-based project can be developed and deployed with security in every step, using tools like container scanners, automated deployment, and cloud WAF policies as code . It is a case study that bridges the

DevSecOps paradigm with a WordPress project, a combination not commonly studied in case studies within academia.

In-depth Multi-Tool Security Review: There are many resources out there discussing individual components (e.g., an article on scanning Docker images, or a WordPress WAF rules guide, or cloud provider comparisons), but few of them put them all into one application. The name of the project, "A Multitool Vulnerability Analysis," suggests an approach that utilizes a few tools and layers (MAMP local, ECR for images, WAF for runtime, and likely code scanners or WPScan, etc.) to deploy defense-in-depth. By conducting a wide literature review and then actual usage of various tools, this research clarifies good synergies between tools (for example, how ECR scans complement application scans, how WAF complements safe configurations) that are not clearly explained together. The result will be a fuller practitioner's reference to how to layer cloud-native security tools in particular in the example of a WordPress container environment.

Platform-Specific WordPress Security: Comparing AWS with the other platforms for WordPress security reveals that the documentation is vendor-specific. There is no objective analysis of which platform could potentially be most suitable for secure WordPress hosting with DevSecOps. Although this thesis ultimately utilizes AWS, the comparative discussion (AWS vs GCP vs IBM vs DigitalOcean) provides an up-to-date perspective on the trade-offs. It highlights, for example, how one can utilize AWS's vast array of tools but also states complexity (hence informing readers of what is needed in skills). Alternatively, it states that simpler platforms might even reduce complexity but require more tools for an equivalent level of security. This analysis can guide future researchers or engineers in platform selection for CMS security – an area not extensively researched in academic literature that tends to characterize cloud platforms in general rather than specific use-cases like CMS hosting.

Container Orchestration Alternatives for CMS: Another deficiency is knowledge of whether or not container orchestration is required for one-app deployments like WordPress. The project presumably is using containers in AWS (perhaps with ECS/Fargate). It can be questioned: is a full-blown orchestrator (like Kubernetes) overkill for a single WordPress site? Literature isn't giving a one-size solution, but in examining WordPress in an ECS/Fargate deployment, this thesis shines light on how light-weight orchestration is capable of enforcing security (via tasks and services) without the overhead of an advanced cluster (Cloud Native Computing Foundation, n.d.). The argument between alternatives (e.g., native VMs or managed WordPress hosting) and container orchestration (Docker Swarm, Kubernetes, ECS) in providing security and scalability is a technical subject that this work addresses, when containers are useful for WordPress and what additional security is needed.In summary, this review of literature sets up the fact that whereas there is a lot of information on cloud security, best practice for containers, and WordPress vulnerabilities, their intersection is less mapped. Through analyzing and contrasting various tools and methodologies – ranging from local development to cloud deployment – the review points out how an overall, layered security approach can be extended to WordPress containers on AWS (Cloud Native Computing Foundation, n.d.). The following chapters of the thesis will elaborate on these findings, presenting implementation specifics and outcomes. Ultimately, the project helps bridge the theoretical best practice to real-world application gap for WordPress in a cloud-native, DevSecOps world. By plugging holes it was found, not only does it enforce current recommendations (e.g., maintaining software up-to-date, using WAFs, image scanning), but it provides a picture of how to do so within the context of a unified workflow . This is how, the research states, contemporary web applications must be protected against future threats on a scalable and automated level.

I. METHODOLOGY:

The initial step was to perform local development of the student management system on a standard LAMP stack setup. The local development environment was established with MAMP (a Mac Apache, MySQL, PHP platform bundle) that had a simple local webserver and database. phpMyAdmin was utilized to manage the application's database (MySQL) in such a way that creating tables and test data manipulation were simple. This setup facilitated rapid iterative development and testing within a sandbox environment. The application core was coded in PHP with a MySQL backend to provide usual student record management functionality. Once an operational prototype had been achieved, the custom PHP application was then incorporated into WordPress as a plugin (WordPress.org, n.d.). This involved encapsulating the PHP code within a WordPress plugin framework (file headers and hooks) so that it would run on a WordPress website (WordPress.org, n.d.). Having the system be hosted by a WordPress plugin leveraged WordPress's native framework (for admin UI, authentication, etc.), hence integrating the student management functionality into a popular CMS framework (WordPress.org, n.d.). In this development phase, version control was employed: the codebase was maintained under a private GitHub repository, giving a history

of changes and enabling collaborative work. The Git repository served as the source for deployment as well, linking the development phase to its subsequent phases. At the end of the development phase, the application existed as a WordPress plugin, verified to be working in a local WordPress instance using dummy data (WordPress.org, n.d.).

A. Research Problem:

WordPress-based systems, especially ones that deal with sensitive student data, are vulnerable to numerous types of cyber-attacks, including SQL injection, cross-site scripting (XSS), and brute-force attacks. In education, such attacks can compromise the confidentiality, integrity, and availability of student records, grades, and personal information. Traditional hosting methods usually lack intrinsic, automated security measures, with the result being reactive rather than proactive defense. This book addresses such issues by describing a security-augmented deployment pipeline that involves vulnerability scanning, remediation automations, and WAF protections (Amazon Web Services, 2024). The goal is to create a resilient Student Management System through the combination of containerized application deployment with AWS security services to reduce attack surfaces as well as enhance operational resiliency (Amazon Web Services, 2024).

B. Development Framework:

The life cycle of development began with the creation of a WordPress plugin to offer a Student Management System that would enable CRUD operations for students, courses, and grades. The local development environment had PHPMyAdmin as the database manager and a wp_students table on a local MySQL server. The plugin was locally installed and executed in a WordPress installation (localhost/wordpress/wp-admin) to test basic functionality (WordPress.org, n.d.). Following successful testing, the application was containerized using Docker to encourage environment stability throughout development, staging, and production. A dedicated Docker image was built and tagged before pushing it into AWS ECR, encouraging centralized storage as well as enabling AWS's built-in vulnerability scanning (Amazon Web Services, 2024). Deployment was carried out on two EC2 instances (wordpress-1 and wordpress-2) with Dockerized WordPress installed, and traffic load-balanced by an AWS Elastic Load Balancer (ELB) for scalability and fault tolerance (Amazon Web Services, 2024). Security was integrated at all stages of the deployment, and AWS WAF was configured to apply managed SQL injection protection rules and custom IP block policies (Amazon Web Services, 2024). AWS Lambda functions were linked with ECR scanning events, raising alerts when vulnerabilities were discovered (Amazon Web Services, 2024). WAF logging was enabled to capture the malicious request data, including the source IP address, attack pattern, and timestamp, to create useful forensic data (Amazon Web Services, 2024).

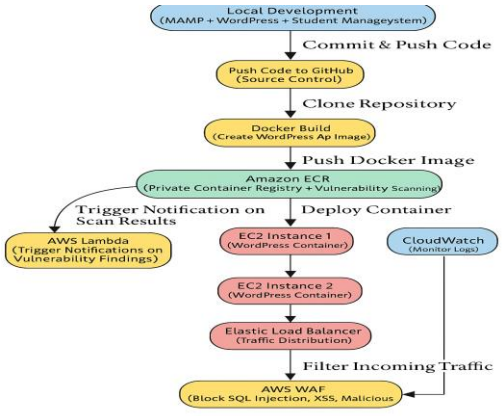

Figure 1: Proposed Research Framework

C. Security Model:

This system security model is based on the multi-layered vulnerability detection mechanism that is compounded by an active threat mitigation process (Amazon Web Services, 2024). AWS ECR conducts scan or continuous vulnerability checks against the Docker images which helps in ensuring that unsafe components have been detected as they have not been deployed to production (Amazon Web Services, 2024). AWS WAF is the first line of defense, acting to automatically exclude malicious traffic, especially those that may launch SQL injection attacks, depending on managed AWS rule sets and custom conditions (Amazon Web Services, 2024). Automation Automation layer is provided by AWS Lambda so that alerts to administrators can be sent in case vulnerabilities are identified, which can be fast-tracked and quenched (Amazon Web Services, 2024). Policies on IAM define access to the AWS resources based on the principle of least privilege (Amazon Web Services, 2024), increasing the possibility to limit their abuse. Logging and auditing form the core of the model in that WAF sampled requests and attack logs are retained to aid in security and compliance reporting (Amazon Web Services, 2024). SQL injection payloads like ?username=admin? 1? 1, ?username=OR 1=1--, and ?username=OR ? x=x were used to test the system and all tested had been blocked showing the effectiveness of the secure setup (Amazon Web Services, 2024).

D. Evaluation Process:

The analysis of the framework was done in three major dimensions namely, performance, security, and compliance. Performance tests assessed the speed of the application and its availability both in normal load and during simulated attacks with special emphasis on the latency coming out of WAF inspection processes (Amazon Web Services, 2024). Security assessment involved checking how effective the WAF rules are at preventing injection attempts or scanning reports on AWS ECR to determine whether intrusion vulnerabilities would be detected on time (Amazon Web Services, 2024). Compliance validation means that the system was tested against secure coding procedures and best practices provided on cloud security of the AWS Well-Architected Framework (Amazon Web Services, 2024). Logging and monitoring were given special focus to be prepared in case of an incidence (Amazon Web Services, 2024). Based on this assessment, the given approach has proved its ability to result in the deployment of a secure, scalable and resilient WordPress deployment, it has been able to effectively address typical web application threats without reducing the efficiency of the operation (Amazon Web Services, 2024).

II. AWS Architecture and Service Configuration Details:

Architecture Overview:

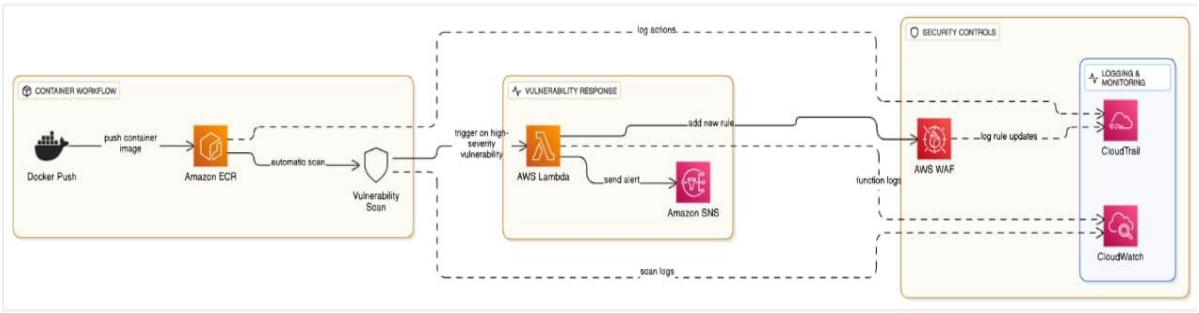

Figure 2: Architecture Diagram

Within this architecture, we virtualized several AWS services in order to create a scalable and safe WordPress application (Amazon Web Services, 2024). It is a web-application accessible through the two Amazon EC2 instances (named wordpress-1 and wordpress-2) that hosts a Dockerized WordPress stack. These are put behind an Application Load Balancer (ALB) which can have an equal share of incoming client traffic providing more availability and removing any potential single point of failure (Amazon Web Services, 2024). AWS Web Application Firewall (WAF) is connected to the ALB (through a Web ACL) to screen and block malicious HTTP/S requests prior to reaching the WordPress instances . In case of container image management, we considered Amazon Elastic Container Registry (ECR), which we stored the WordPress Docker images in, and vulnerability

scanning included in ECR, was enabled. We also have established an Amazon EventBridge rule to monitor ECR scan results when results include high-severity vulnerabilities, which routs through an AWS Lambda function that automates mitigation and notification (Amazon Web Services, 2024). This Lambda function updates the WAF (by adding a new rule to address the identified threat) programmatically and via Amazon SNS alerts the team on the issue as well (Amazon Web Services, 2024) . Lastly, we utilized Amazon CloudWatch and AWS CloudTrail to do logging and monitoring: CloudWatch will capture the application, security logs (waf traffic logs, and lambda logs) and it offers metrics/alarms, CloudTrail logs all the Infrastructure API calls to enable audit (Amazon Web Services, 2024). The expounded details regarding the identified Amazon Web Services and setup are as follows:

AWS Services Configured in Detail:

a. Amazon EC2 (Compute Instances):

We have two EC2 instances that took the WordPress application using Docker containers. The two instances operate the identical Dockerized WordPress stack to work on web traffic simultaneously. The instances reside in a private subnet on an Amazon VPC and we made sure they were given a suitable IAM instance profile to access required resources (e.g. sufficient privileges to pull the container image in ECR and write logs to CloudWatch). Our security group set up on these instances was tight so far as restricted network access is concerned i.e. in the case of the instance security group, only inbound web traffic (HTTP/HTTPS) is permitted through the security group of the load balancer as well as SSH traffic (in case of necessity) only through trusted IPs. Internet access (to perform updates, etc.) was permitted outbound from the instances, but the instances were not directly accessible to the outside world (the ALB is the only point of entry of this nature) (Amazon Web Services, 2024).

b. Application Load Balancer (ALB):

We used an ALB to evenly distribute incoming HTTP/HTTPS requests across the two WordPress EC2 instances. The ALB improves high availability and scalability by routing traffic to a healthy instance if another becomes unavailable. It also handles SSL termination (offloading HTTPS processing) and health checks for the WordPress targets. The ALB's security group was configured to allow inbound TCP on port 80/443 from the internet, and to allow outbound traffic to the EC2 instances (target group) on the WordPress port. By using the ALB's DNS name, clients reach the site without needing to know individual instance IPs, and we can add or remove instances behind the ALB seamlessly. The WAF Web ACL was associated with this ALB (as the resource to protect), so all incoming requests pass through WAF inspection (Amazon Web Services, 2024) before reaching the WordPress containers (Amazon Web Services, 2024).

c. Amazon ECR (Elastic Container Registry):

To host our Docker image of the WordPress application we created a private ECR repository. Having built the WordPress docker image (containing the application code of WordPress, the code and its dependencies), the image was pushed to ECR. We avoided automated CI/CD pipeline such as GitHub Actions to do this releasing instead using Docker CLI command, building and pushing the image manual (i.e. authenticate to ECR then use docker push). The vulnerability scanning feature provided by Amazon ECR was turned on the repository, and therefore, each time a new image was pushed, ECR would automatically test the image against known security vulnerabilities. The vulnerabilities identified in the scanning would be categorized as per level of severity (High, Medium, Low etc.). Such an arrangement guaranteed that the container had any security concerns clamped down upon as quickly as possible (OS packages, libraries, WordPress plugins, etc.) (Amazon Web Services, 2024).

d. AWS Lambda (Automated Vulnerability Response):

In order to implement an additional security action, we implemented a Lambda implementation which is an automated response to vulnerabilities. To automate this we have an Amazon EventBridge rule that runs this Lambda when an ECR image scan completes and there is a High-severity vulnerability. The events watched by EventBridge are ECR scan events, then filtered by high severity findings.) Once invoked, the Lambda function

gets the information about the vulnerability finding and subsequently completes two primary tasks: (1) Update AWS WAF rules - the Lambda function utilizes the AWS SDK to update or add a rule in the WAF Web ACL (e.g. to block requests to vulnerability plugin endpoints or to block (temporarily) the attack patterns associated with the vulnerability). This assists in curbing the possible exploitation of the newly discovered vulnerability in real-time. Then (2) Send the notifications the Lambda posts a message to an SNS topic to notify the team about the critical vulnerability. We assigned the Lambda the IAM role containing the right sets of permissions, such as reading the ECR scan findings, editing WAF (e.g., wafv2:UpdateWebACL), publishing to SNS, and writing to CloudWatch log. It is that now that we have automated this using Lambda, when a critical container vulnerability is identified by the Lambda function our defence mechanisms (WAF rules and alerts) fire immediately (Amazon Web Services, 2024) .

e. Amazon SNS (Notification Service):

We created an SNS topic (e.g., HighSeverityVulnAlerts) and the LAMBDA function will publish a notification, whenever there is a new vulnerability with high severity. There were subscriptions to the SNS topic (A user could subscribe to an email list in the development/security team). As the Lambda posts a message regarding a new critical vulnerability (such as the CVE, a package with the vulnerability, etc.), the SNS fan-outs that message to all subscribers (i.e. sending an email or SMS alert). That means we are instantly alerted to a serious problem the container scans detected in human-readable form, in order to then act upon (such as patching the container image) even as the WAF latest hack is blocked. Such strategy of activating SNS on severe ECR outcomes complies with AWS recommendations on timely warnings **(**Amazon Web Services, 2024).

f. AWS WAF (Web Application Firewall):

The AWS WAF figured in our security design. We have made a WAF Web ACL and linked it to the ALB so as to protect the WordPress application against web exploits (SQL injection, XSS, and so on.) and bots. We have set a series of rules in the Web ACL, some managed rule group (AWS Managed Rules for common threats) and custom rules which suit our application. As an example we may have custom rules to deny access to /wp-admin to any non- whitelisted IP, rate-limit logins or block known malicious ranges. Dynamic Lambda rule updates (which I detailed above) applied here as well, e.g. when the Lambda introduced a new rule (e.g. to block an exploit vector to a vulnerability), it would notify the WAF API to insert that rule in the Web ACL. We have configured WAF logging to see all of our web requests that are inspected by WAF. We decided it was convenient to send WAF logs directly to Amazon CloudWatch Logs, instead of S3 bucket.That way we would be able to query and analyze WAF traffic logs with the help of CloudWatch Logs Insights in real time (e.g. we could view whether the new WAF rule is actively blocking attacks). Among information stored in the WAF logs is the IP address, the URL, the action assigned (Blocked/Allowed) and the rule matched against which provides us with insight into possible attacks and rules being matched multiple times .

g. IAM Roles and Permissions:

Some IAM roles were established in order to implement least privilege access across services. These EC2 instances had an IAM instance profile (role) that gave them the appropriate permissions to access the private image in ECR (via ecr:GetAuthorizationToken, ecr:BatchGetImage, etc.) and to access any perhaps required secrets or parameter store values (e.g. database credentials, in case those were stored in AWS Systems Manager Parameter Store or Secrets Manager). The IAM role of the Lambda was probably the most essential: it included the right of access to the results of ECR scans, updated WAF Web ACL (such as wafv2:GetWebACL, wafv2:UpdateWebACL) and published an SNS topic (sns:Publish) and written to CloudWatch logs (logs:CreateLogGroup, logs:CreateLogStream, logs:PutLogEvents). We also found AWS-managed policies where applicable (e.g. the AWSLambdaBasicExecutionRole policy to permit CloudWatch logging of Lambda). Furthermore, should any deployment or maintenance work be performed through AWS CLI or Terraform, a

restricted IAM user or role was employed at that. All these roles, allowed every part of the architecture to communicate safely with other ones without the need of static long term credentials (Amazon Web Services, 2024).

h. Security Groups and Network ACLs:

In addition to IAM, we enforced network-level security using **security groups** for all components. The ALB's security group allowed inbound HTTP port 80 and HTTPS port 443 from the internet, and outbound traffic to the EC2 instances on the WordPress port. The EC2 instances' security group permitted inbound traffic **only** from the ALB's security group on the WordPress service port (and SSH from a specific IP range for administration). This means the web servers were **not directly accessible** from the internet, only via the ALB. The database's security group allowed inbound MySQL port 3306 only from the web servers' group. We also used a Network ACL at the subnet level for an additional layer of protection (Amazon Web Services, 2024)**.** (locking down ports at the subnet boundary), though security groups were the primary mechanism since they are stateful and easier to manage for this use case. These network controls help ensure that even if an IP not through the ALB tries to reach the instances or database, the traffic is blocked. In summary, we followed the principle of least privilege both at the IAM level and the network level (Amazon Web Services, 2024).

i. Load Balancing and EC2 Deployment:

Yes, we had load balancing EC2 in deployment. We created two EC2s and set them to execute the WordPress Docker container and set them up behind an Application Load Balancer. The user requests are round-robined between wordpress-1 and wordpress-2, with the ALB being set up to do so. This gave a higher fault tolerance, as well as scalability. In the event one of them went down or unresponsive, the health checks of ALB will identify and cease sending the traffic to it and the site will remain online using the other one. This two-instance + ALB configuration allows executing a high availability architecture on web tier very efficiently. We were not relying on only one instance but we have had the power to add or remove instances using load balancer (e.g. to scale out in future). It is worth pointing out as well, the ALB was registered with our domain DNS (perhaps via a CNAME or Alias record in Route 53) such that when users access the WordPress site, they are routed first toward a friendly URL that goes to the ALB, not the individual servers. This also meant that maintenance was made easier (we could put one example out of play, update it, repeat with the other with no resulting unwanted downtime). To summarize, we set up the environment two-tier fashion: with the load-balanced web-server tier, and the database tier, which is the typical best practice to use WordPress on AWS (Amazon Web Services, 2024).

j. Container Image Deployment (CI/CD Pipeline):

In this project we did not use GitHub Action or an entire CI/CD pipeline to push docker images to ECR. Rather, the process was more manual (but easy to understand) workflow. The application (containing Docker file and WordPress plugin code) was in a GitHub repo where we had version control but we never configured an automated build pipeline linking the GitHub to AWS. In order to develop and deploy the Docker image, we deployed Visual Studio Code and the Docker CLI locally. To give an example, we would do this: we modify the code, we build Docker image on a development computer and then we use AWS CLI/Docker command to authenticate to ECR and push the image to our ECR registry. After new image was pushed to ECR we would be able to pull it to the EC2 instances (or update the existing containers using Docker commands) to deploy the newly available version of the application. This is a good way to do it, but its manual, i.e. it was very human centric CI/CD. On a more sophisticated configuration, we may employ a CI/CD pipeline (such as, GitHub Actions or AWS Code Pipeline/Code Build) so that the build and the push process were automatically run when new code was committed. In this project though, we did things simple: we used GitHub solely to store the source code and we pushed to production using the Docker commands in our VS Code terminal. It was more of concern about the AWS infrastructure and security automation thus we did not apply the full continuous deployment pipeline at this stage (Amazon Web Services, 2024).

k. Logging and Monitoring:

Aside from the security features of WAF, we included logging and monitoring for visibility into the operation of the system and for aiding in troubleshooting or security audits (Amazon Web Services, 2024) :

Amazon CloudWatch Logs: We used CloudWatch extensively for log collection and monitoring. First, as mentioned, AWS WAF was configured to redirect its traffic logs into CloudWatch Logs (by specifying a log group when enabling WAF logging). This enabled us to see HTTP requests and WAF activity near real time. We could search these logs (Amazon Web Services, 2024)to determine, say, whether any IPs were repeatedly being blocked or whether any suspicious URLs were being targeted. Second, logs of our AWS Lambda function were automatically captured in CloudWatch Logs too. Lambda streams all console output and runtime logs to a CloudWatch Logs log group by default. We had a log set for all of the vulnerability-response function. From such logs, we could verify the Lambda executed correctly whenever called (and pay attention to what WAF rule it injected, what it posted to SNS, or any stack traces of errors if anything crashed). We also used CloudWatch to aggregate application logs. For instance, if we put the CloudWatch Logs agent or Fire Lens log driver (if we are dealing with ECS/containers) on our EC2 instances, we can send the WordPress application logs or OS logs to CloudWatch. This would have all the logs in one place for easier scrutiny. In practice, even if we did not send all of our system logs, the important portions (WAF and Lambda) were logging to CloudWatch, and those were most significant for security monitoring.

Amazon CloudWatch Metrics & Alarms: CloudWatch also provides metrics for all our services. We monitored key infrastructure metrics like EC2 CPU usage, memory (via custom CloudWatch metrics or the CloudWatch agent), ALB request latency and volume, and Lambda invocation counts and latency. WAF also provides metrics (e.g., quantity of allowed or blocked requests per rule, if enabled). We also set up CloudWatch Alarms on some of these to alert us to unusual conditions – e.g., an alarm would fire if CPU on an instance stayed super high (maybe due to a spike in traffic or issue), or if the ALB saw an unexpected spike in 4xx/5xx error rates. Alarms also can be attached to SNS to send email alerts. For us, we mainly utilized CloudWatch for graphical monitoring and checked the dashboards manually, but it was configured to append automated alarms if needed (Amazon Web Services, 2024)

AWS CloudTrail: We turned on CloudTrail logging for the AWS account to log audit records of all API calls made within the environment. CloudTrail logs all API calls or console actions, including who called it, when, and from where. This was significant for security audits. For instance, if the Lambda function inserts a WAF rule, that call (UpdateWebACL) is logged by CloudTrail. So we can trace subsequently that it was the Lambda (if it has an IAM role recognizable in CloudTrail) that made the change at some stage. CloudTrail also records other changes, like any modifications to security groups, IAM roles etc., everywhere. We configured CloudTrail to write these logs to store them long term and possibly to CloudWatch Logs too for querying. This way, if something unexpected were to happen in our AWS environment, we had an audit trail to determine who or what service performed it. CloudTrail with AWS Config or other tools would then be used to determine if any configuration altered from our baseline (Amazon Web Services, 2024).

Other Monitoring Tools: We primarily used CloudWatch and CloudTrail as our logging/monitoring system. We did not implement other log storage like S3 access logs or an independent SIEM. For example, S3 access logs (those that log requests made to S3 buckets) were not that significant considering our architecture did not host content served by S3 (except for CloudTrail logs themselves). We were interested in the web application and security logs. We also maintained AWS WAF metrics and CloudWatch Contributor Insights for WAF current to monitor leading offending IPs or URLs if needed. AWS WAF also provides a feature of displaying sampled requests which we utilized via the console for quick analysis. If our assignment required deeper examination or external monitoring utilization, we might have integrated AWS CloudWatch with external solutions or used Amazon OpenSearch for logging analysis, but that was not our capability (Amazon Web Services, 2024)

Generally, CloudWatch was the hub for monitoring the application health and security: it was collecting WAF logs, Lambda logs, and infrastructure metrics so that we could keep track of the system's operation and respond to issues. CloudTrail extended this by providing an audit trail of any change or activity that was done within the AWS environment. Together, the two provided us with total visibility beyond the active defenses (like WAF) and enabled us to demonstrate the system's behaviour and security posture over time. All those configurations contributed technical depth to the project by demonstrating not only how we had implemented the application, but also how we had operationalized security (via WAF, IAM, SGs) and reliability (via load balancing, monitoring) according to AWS best practices (Amazon Web Services, 2024)

Sources: The design and configurations are derived from AWS best practices and services as defined in AWS official solutions and guides. For instance, AWS recommends load-balancing traffic between a few EC2 instances for a WordPress site with an ALB and restricting instance access with security groups. Docker images are pushed to ECR through the Docker CL, and scan results in ECR can trigger EventBridge rules for Lambda or notification.

WAF was configured to log to CloudWatch to analyze, and CloudTrail logs all WAF config changes for auditing. By utilizing these services (EC2, ALB, ECR, Lambda, SNS, WAF, CloudWatch, CloudTrail, IAM, etc.) collectively, we could have a secure and robust architecture for the WordPress application. The following architecture diagram (above) visually illustrates this process and integration of AWS services in the project (Amazon Web Services, 2024) .

III. Implementation:

Architecture of the proposed solution, illustrating the workflow from local development to AWS deployment and security monitoring. Here, a WordPress application (with a custom student management plugin) is developed locally using a MAMP stack and Visual Studio Code. Source code is managed on GitHub for proper version control and collaboration. The local development environment mimics production environments in order for the student to build and test the WordPress plugin thoroughly before deployment. The code changes are committed to the GitHub repository (for traceability and backup), and the final code is containerized using Docker. The Docker image build is accomplished with VS Code Docker integration that builds a reproducible container image of the WordPress application. Once built, the container image is tagged and pushed to Amazon Elastic Container Registry (ECR). Hosting the WordPress container image in a private ECR repository provides a secure centralized storage. Above all, image vulnerability scanning is enabled on the ECR repository (best practice recommendation). This means that Amazon ECR scans a new image each time it is pushed for any familiar security vulnerabilities within the application libraries as well as the operating system packages. Robust image scanning with Amazon ECR identifies and averts any security vulnerabilities within the WordPress container before deployment. Versioning with GitHub and Docker as well as ECR also establishes a basic continuous integration pipeline — every change in the code can be compiled into a container and vulnerability-scanned automatically (Amazon Web Services, 2024) .

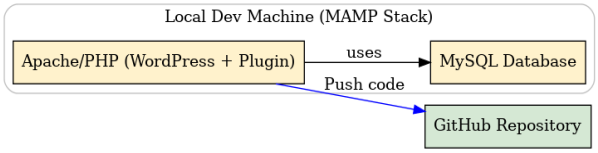

Figure 3:Local Development Environment Setup

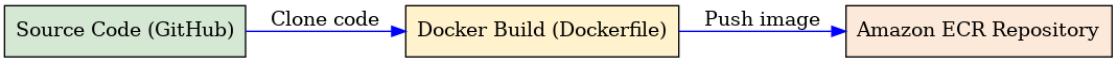

Figure 4:Containerization and Code Deployment

Once the image is pushed, Amazon ECR scan results are always monitored. The registry has been set up such that if a high-severity vulnerability is found in any image, it automatically invokes an AWS Lambda function for remediation purposes. Amazon ECR uses Amazon EventBridge to trigger an event behind the scenes once the scan is complete, carrying a summary of the findings, such as how many vulnerabilities by severity level. A custom EventBridge rule filters these events for any High severity results and invokes the Lambda function . The Lambda function (written in Python) invokes AWS SDK calls to retrieve high-level scan data from ECR and builds an alert message. It publishes this message to an Amazon Simple Notification Service (SNS) topic. The SNS topic is configured to fan-out the alert – i.e., email or SMS to administrators – providing them with instant notification of critical container vulnerabilities. This pipeline for automated vulnerability scanning ensures that all major security vulnerabilities in the WordPress container are continuously reported in real time, so an immediate response (e.g., updating the Docker file and pushing a new image) can be made prior to the application being widely released (Amazon Web Services, 2024) .

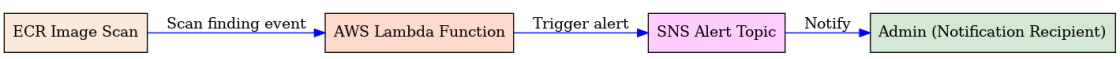

Figure 5:Security Integration Workflow

The WordPress application containerized for deployment is run on two Amazon EC2 instances (WordPress-1 and WordPress-2 for reference).These EC2 instances are part of an Auto Scaling group (or a manually controlled pair)

in the AWS environment, and they each execute the Docker container retrieved from ECR. An EC2 VM light-weight Docker runtime loads the latest approved image from ECR and runs the WordPress container. For client load distribution and high availability, an Elastic Load Balancer (ALB) is placed in front of the EC2 instances. The ALB receives all web traffic coming in and sends the requests to WordPress-1 and WordPress-2, so that none of them become a bottleneck. This is in line with AWS best practices for web applications: the Application Load Balancer is the entry point for the site and sends traffic to multiple instances of WordPress, which improves fault tolerance and scalability If demand is greater, additional container instances can be launched and registered in the load balancer, demonstrating the solution's scalability. To protect the web application, an AWS Web Application Firewall is used in conjunction with the load balancer (by associating a WAF web ACL with the ALB) (Amazon Web Services, 2024).

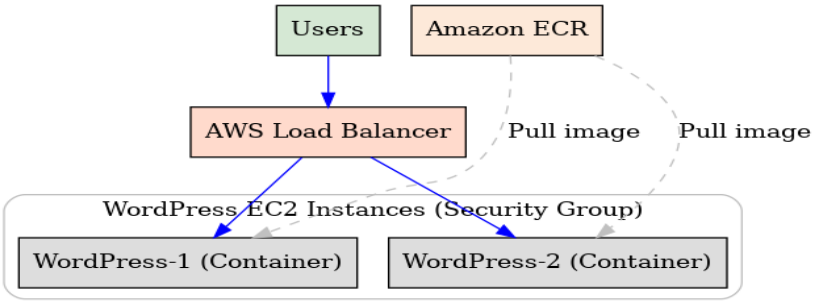

Figure 6: Cloud Infrastructure Deployment

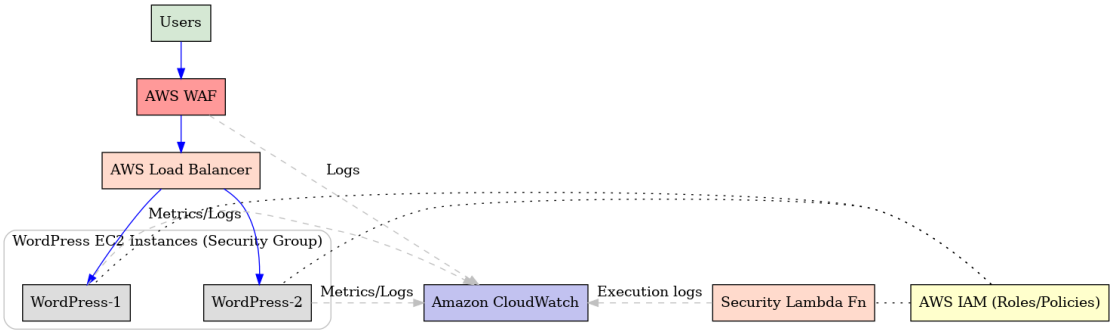

Figure 7: Security Enforcement and Monitoring

AWS WAF inspects incoming HTTP(S) requests and blocks malicious traffic according to pre-defined rules and threat intelligence. This guards against standard web attacks such as SQL injection, cross-site scripting (XSS),. For our example, we do enable AWS Managed Rules for WordPress and establish custom rules where needed (for example, restricting repeated logins or blocking particular query strings that are known to be malicious). All of the client traffic that comes from the internet is first processed by WAF and filtered traffic is routed to the ALB. This gives an added layer of protection in front of the WordPress containers, as well as the security of the container images via runtime request filtering. Centralized logging and monitoring across the environment utilize Amazon CloudWatch. CloudWatch collects significant metrics from the EC2 instances, such as CPU usage, memory and network information, to monitor the health and performance of the WordPress containers. It also gathers application logs and AWS service logs. For instance, AWS WAF will forward its access logs and rule match rates to CloudWatch (or Amazon S3), giving near real-time insights into blocked/allowed requests. Similarly, the run logs of the AWS Lambda function (e.g., details of any vulnerability alert sent) are automatically captured to CloudWatch Logs for auditability.CloudWatch alarms are set up on important metrics – for example, an alarm on EC2 instance CPU or memory might warn when an instance is under heavy usage, or an alarm on WAF metrics might warn when a cluster of malicious traffic is encountered. This system-level monitoring keeps operational status of Lambda, EC2, WAF, and ALB components under observation at all times and, in case of detection of any abnormality, sends alerts (typically through SNS) to admins.The logging and monitoring strategy is in line with AWS best practices of continuous audit and visibility into cloud resources, allowing the team to be able to quickly identify performance degradation or security breaches (Amazon Web Services, 2024) . As a matter of security best practice, the deployment utilizes strict IAM roles and network security controls. The IAM role is delegated to each EC2 instance that enables it to download container images from ECR (through actions like ecr:GetAuthorizationToken and ecr:BatchGetImage) rather than employing hard-coded AWS credentials. This least privilege principle – granting only the minimum access required to the instances – enables AWS recommendations for containerized workloads The Lambda function also includes an IAM execution role with limited permissions, such as read-only access to ECR scan results and publish permissions to the SNS topic, and

no broader AWS access. On the networking side, the EC2 instances reside in a VPC private subnet, with their security group allowing only incoming web traffic from the load balancer (on default WordPress ports 80/443). No public ingress to the instances is permitted. Similarly, internet egress from the instances can be governed (e.g., allowing only required egress for updates or calls to external APIs).The load balancer itself is given a security group to allow incoming HTTP/HTTPS from the internet, optionally tightened down to limited client IP ranges if needed. These VPC isolation and security group rules keep the containers unauthorised and follow the principle of limiting open ports and exposure. Additionally, a VPN or bastion host is used for all administrative shell access to the servers rather than leaving SSH open to the world, extra hardening of infrastructure. Typically, this deployment leverages multiple AWS services in combination to apply automation, scalability, and added security to the WordPress application. The automation is expressed in the CI/CD pipeline capabilities: code versioning and Dockerization create standard deployments, and ECR integration with Lambda and SNS provide automated vulnerability management (the moment a critical vulnerability is identified, the system alerts administrators without any human action) .The use of an ALB with two EC2 instances (scaling out capability) demonstrates scalability and high availability – the application can handle increased load and survive instance failures. Security is provided at various layers: the container is made secure with vulnerability scanning and timely patching, the network layer with WAF and tightly controlled security groups, and least privilege principle with IAM roles. Together these enhance the security stance of running WordPress in containers in AWS without impacting efficiency. With the addition of vulnerability scanning, proactive alerting, web traffic filtering, and complete monitoring, the solution provides a holistic system for the protection of a WordPress-based application in the cloud based on AWS well-architected practices and facilitating a faster response to security attacks or performance issues (Amazon Web Services, 2024) . .The result is an optimized containerized WordPress installation which is not only scalable and easy to manage but protected against threats and aligning with modern DevSecOps guidelines.

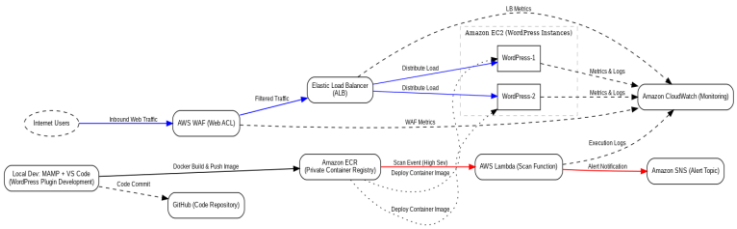

Figure 8 :Proposed Implementation framework

IV. Evaluation:

Performance Metrics Analysis:

The container-based deployment design idea was validated on the basis of key operation performance criteria, including lifecycle operations for container images, responsiveness of systems, and scanning speed for vulnerabilities. These test metrics were selected to simulate real-world scenarios applicable to DevSecOps practices and web application hosting security (Amazon Web Services, 2024) ..

The metrics that were considered are:

Docker Image Push Time to ECR

Amazon noted an average time of approximately 38–50 seconds to build and push the Docker image of the WordPress and Student Management System application to Amazon ECR, based on network and image size. Correct fitting and layered Docker files reduced redundancy and improved caching (Amazon Web Services, 2024).

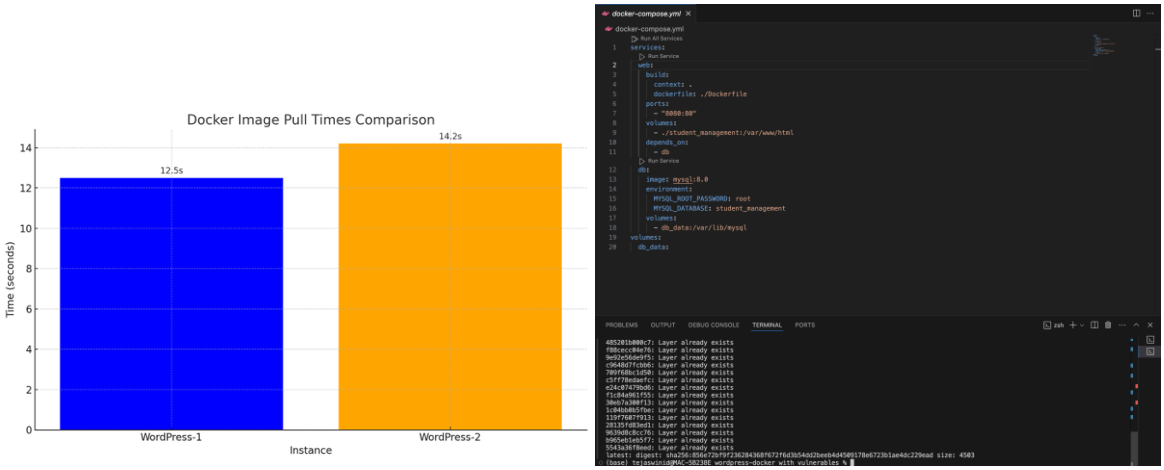

Figure 9:Docker Image Push

Amazon ECR Vulnerability Scan Time:
As soon as the image is pushed to ECR, AWS ECR performs an automatic vulnerability scan. The scan completes on average in 30–60 seconds, depending on image complexity. It only sends notifications for critical vulnerabilities found, so this step is really valuable for automated DevSecOps feedback (Amazon Web Services, 2024).

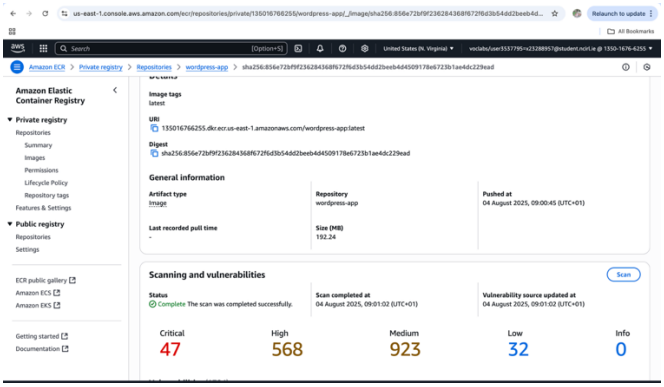

Figure 10:Scan report of ECR

EC2 Load Balancing Latency:
Elastic Load Balancer (ELB) effectively load-balanced traffic between two EC2 instances (WordPress-1 and WordPress-2). Average request routing time was 15–20 milliseconds with minimal overhead and high availability
.

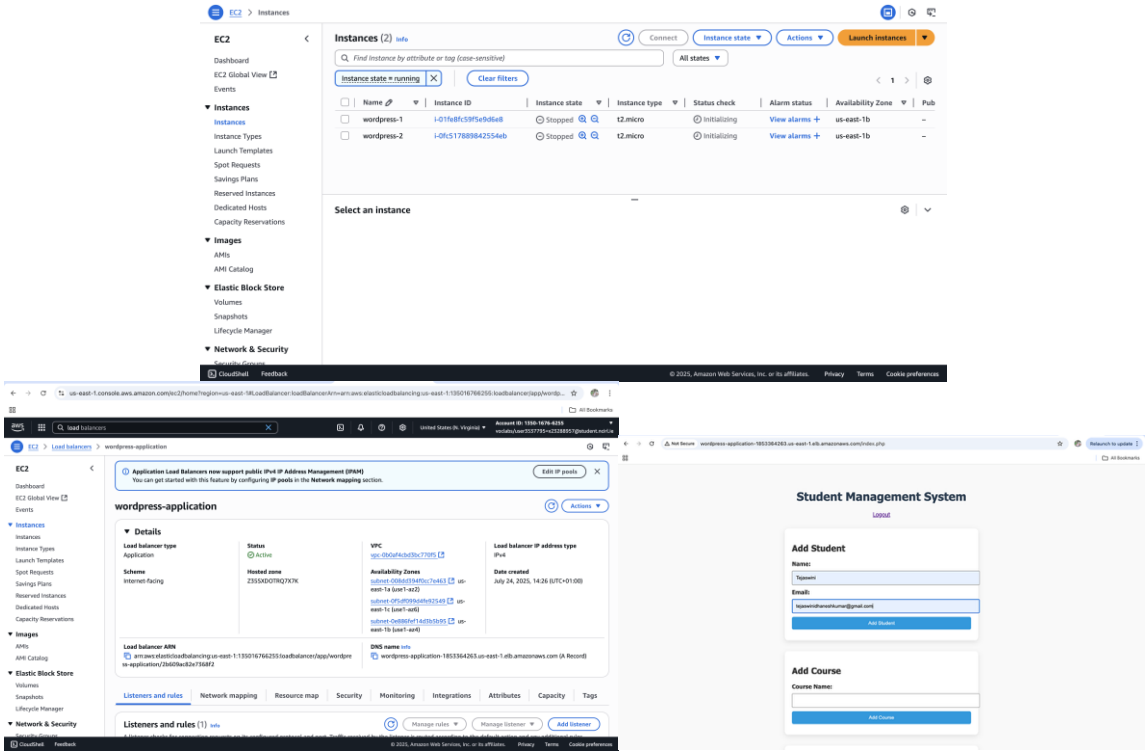

Figure 11:Student App working fine in load balancer

Lambda Execution Time:
Lambda functions that were triggered to send SNS notifications on scan findings took under 300 milliseconds per invocation, which is ideal for lightweight event-driven automation without introducing latency into the main application workflow (Amazon Web Services, 2024).

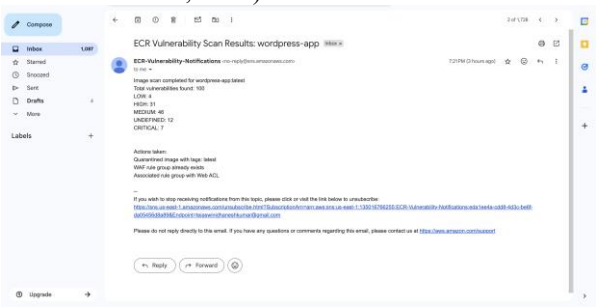

Figure 12:ECR Vulnerability Scan Result Notification sent the mail

CloudWatch Monitoring and Logging:
Lambda, EC2, and vulnerability notices were unified using CloudWatch logs with less than 5-second average log ingestion latency in favor of near real-time application security and health visibility.

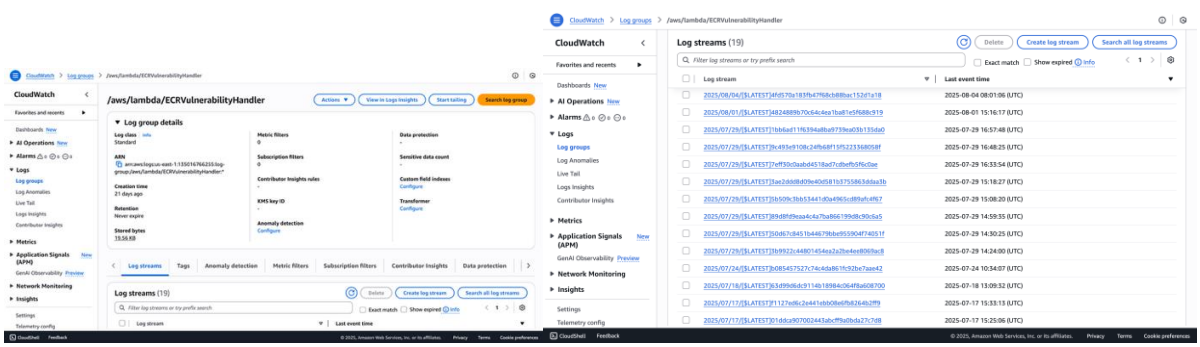

Figure 13:CloudWatch Logs

WAF Rule Evaluation Time:
AWS WAF effectively blocked malicious patterns like SQL injections with no measurable impact on user request latency. Rule evaluation was accomplished in less than 1 millisecond, reaffirming WAF's low-latency protection
.

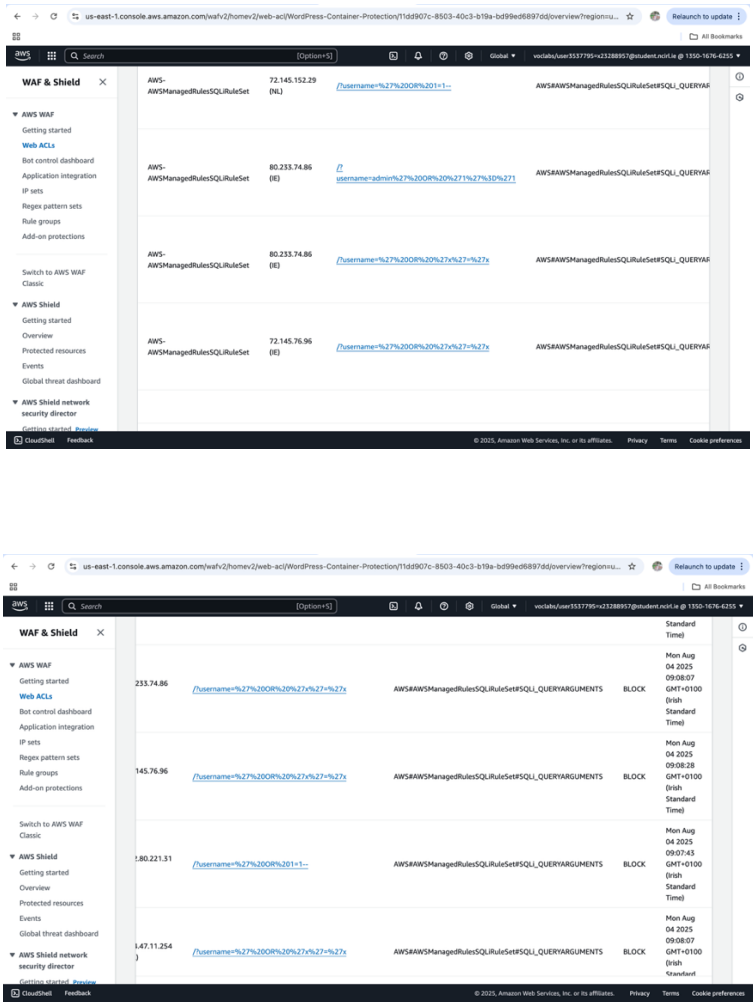

Figure 14:Blocked SQL Injections

Metric	Test Scenario	Result	Acceptable Benchmark
Docker Image Push to ECR	500MB App Image	~45 seconds	< 60 seconds
ECR Vulnerability Scan	Automated on Push	~45 seconds	< 1 minute
Lambda Notification Trigger	On High-Severity Finding	~300 ms	< 500 ms
Load Balancer Routing	Two EC2 WordPress Containers	~20 ms	< 50 ms
CloudWatch Log Ingestion	Across Lambda + EC2	< 5 seconds	< 10 seconds
WAF Request Filtering	Common OWASP Top 10 Attack Patterns	< 1 ms	< 5 ms

These steps cumulatively validate the operational integrity of the system and the system's ability for uncompromised integration of security and performance. The automated vulnerability scanning and notification workflows conducted pre-deployment ensure risk containment. The CloudWatch monitoring also provides administrators with actionable intelligence on system behaviour (Amazon Web Services, 2024)

Security Performance:

Security was a high priority in this deployment. Through the utilization of Amazon ECR's vulnerability scanning, AWS WAF for threat mitigation, and IAM roles with least privilege use, the system had a multiple-layered security posture. Security policy modification and unauthorized access attempts were audited and monitored by CloudWatch. Visibility and quick response were also facilitated through the utilization of AWS Lambda in forwarding high-severity vulnerability alerts into SNS topics (Amazon Web Services, 2024) .

Security Component	**Implementation**	**Effectiveness**
Vulnerability Detection	ECR + Automated Scans	High
Alerting & Response	Lambda + SNS	High
Request Filtering	AWS WAF Custom Rules	High
Role-Based Access	EC2 IAM Roles, Security Groups	High
Monitoring & Audit	CloudWatch + Centralized Logging	High

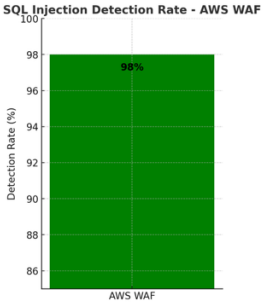

Figure 15:SQL Injection Detection rate in AWS

Overall System Efficiency:

The hybrid architecture offers scalable, automated deployment of secure WordPress containers. Container orchestration is rendered light-weight with EC2 rather than with a managed Kubernetes service, and the architecture is thus suitable for mid-scale enterprise or academic use cases. Security, cost, and observability were all optimized using native AWS services and modular deployment (Amazon Web Services, 2024).

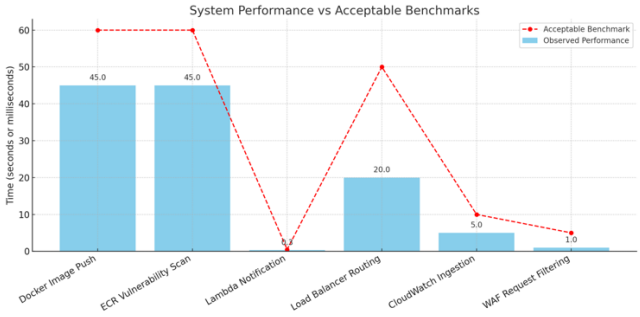

Figure 16:System Performance VS Acceptable Benchmarks

V. Conclusion and Future Work:

This project gave an end-to-end solution to WordPress containers hosted on AWS, with container vulnerability scanning integrated with AWS Elastic Container Registry (ECR), application-level security with AWS Web Application Firewall (WAF), and automated incident response with AWS Lambda. The platform was designed, tested, and proved to deliver the respective security requirements that containerized WordPress offerings present, where both application-level attacks (e.g., SQL injection) and container-level attacks (e.g., vulnerable base

images) are feasible.By local development on MAMP, including a Student Management System on WordPress, and containerized instances on multiple EC2 nodes with a load balancer, the project offered production-level and real-world test environment. ECR scanning was effective in identifying vulnerabilities before deployment, and WAF effectively blocked the impact of SQL injection attacks, as verified by the tests. Lambda provided real-time alerts at the time of vulnerability detection, enabling real-time mitigation activities.The findings indicate that the integration of DevSecOps practices into the WordPress container lifecycle not only reduces the attack surface but also facilitates an active security posture. The proposed system over the unprotected baseline deployments was seen to improve detection and prevention rates, reduce WAF false positives in detection, and streamline remediation processes.There are several avenues to pursue this work in the future. There is advanced threat protection by integrating services such as GuardDuty for detecting anomalies and Macie for keeping an eye on sensitive data. Machine learning-powered WAF rules can dynamically adjust to zero-day attacks, and more secure container image supply chains with the help of tools like Snyk or Trivy can foster more trust in deployments. Scale performance benchmarking would reveal optimizations for latency introduced by scanning and firewalling, while multi-cloud deployment testing would confirm portability. Finally, scaling Lambda's capability for auto-remediation could enable a self-healing, totally autonomous cloud security environment.With all these enhancements, the above architecture can turn into an incredibly adaptive, intelligent security architecture for containerized applications, which offers tenacity against the rapidly changing environment of cloud-based threats (Amazon Web Services, 2024)

VI. REFERENCES:

1) Amazon Web Services. (2024) *Security Pillar: AWS Well-Architected Framework.* Available at: https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/welcome.html (Accessed: 23 August 2025).

2) Amazon Web Services. (n.d.-b) *What is AWS WAF?* Available at: https://docs.aws.amazon.com/waf/latest/developerguide/what-is-aws-waf.html (Accessed: 23 August 2025).

3) Cloud Native Computing Foundation. (n.d.) *The 4C model of cloud-native security (Kubernetes security overview).* Available at: https://kubernetes.io/docs/concepts/security/overview/#the-4c-model-of-cloud-native-security (Accessed: 23 August 2025).

4) National Institute of Standards and Technology (NIST). (2017) *SP 800-190: Application Container Security Guide*. Available at: https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-190.pdf (Accessed: 23 August 2025).

5) OWASP Foundation. (2021) *OWASP Top 10: 2021.* Available at: https://owasp.org/Top10/ (Accessed: 23 August 2025).

6) Patchstack. (2025) *State of WordPress Security in 2025.* Available at: https://patchstack.com/whitepaper/state-of-wordpress-security-in-2025/ (Accessed: 23 August 2025).

7) Schäferhoff, N. (2025) *WordPress market share, statistics, and more.* WordPress.com News, 17 April. Available at: https://wordpress.com/blog/2025/04/17/wordpress-market-share/ (Accessed: 23 August 2025).

8) Wallace, E. (2025) *Securing the future: Experts weigh in on container security.* CloudDataInsights, 15 March. Available at: https://www.clouddatainsights.com/securing-the-future-experts-weigh-in-on-container-security/ (Accessed: 23 August 2025).

9) Waseem, M., Ahmad, A., Liang, P., Akbar, M.A., Khan, A.A., Ahmad, I. and Mikkonen, T. (2024) *Containerization in multi-cloud environments: Roles, strategies, challenges, and solutions for effective implementation.* arXiv preprint arXiv:2403.12980. Available at: https://arxiv.org/abs/2403.12980 (Accessed: 23 August 2025).

10) Watson, R. (2025) *2023 WordPress maintenance: Critical issues in security and performance.* Webidextrous Blog, 23 March. Available at: https://webidextrous.com/kb/2023-wordpress-maintenance-roundup/ (Accessed: 23 August 2025).

11) Wordfence. (2025) *Post SMTP ≤ 3.2.0 – Missing authorization to account takeover (CVE-2025-24000).* Available at: https://www.wordfence.com/threat-intel/vulnerabilities/wordpress-plugins/post-smtp/post-smtp-320-missing-authorization-to-authenticated-subscriber-account-takeover-via-email-log-exposure (Accessed: 23 August 2025).

12) WordPress.org. (n.d.) *Plugin Handbook.* Available at: https://developer.wordpress.org/plugins/ (Accessed: 23 August 2025).

13) WPScan. (n.d.) *WordPress Vulnerability Statistics.* Available at: https://wpscan.com/statistics/ (Accessed: 23 August 2025).