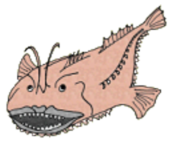
Final Design Document

Project WADT (Web App Deployment Tooling)

Sponsor Chris Fischer



Group A40

Dustin Allen   Daniel Armas   Rithik Dhakshnamoorthy
Jacob Plotz   Luke Samuel Sandoval   Jereme Saunders

24 November 2025

Contents

1 Executive Summary

Imagine you have an epiphany: you need to practice SQL injection or test a newly learned web-security technique, and you need an intentionally vulnerable web application right now. For most students and web-security practitioners, the next hour is spent searching through GitHub, cloning repositories, troubleshooting dependency errors, configuring Dockerfiles, and resolving version mismatches before any real security testing can begin.

The setup process becomes the main barrier, not the learning itself. This challenge represents a broader problem in cybersecurity education: the difficulty of transforming theoretical knowledge into practical skill when technical overhead is disproportionately high.

The WADT project was created to eliminate this barrier and enable students to focus on learning rather than fighting their environment.

The purpose of WADT is to provide a unified, frictionless platform for instantly deploying intentionally and unintentionally vulnerable web applications in a safe, reproducible, and disposable environment. The system directly addresses a critical need within universities, training programs, and self-guided learners by allowing users to interact with vulnerable applications without the time-consuming effort of traditional setup.

Instructors often devote large portions of class time to environment configuration before any instruction takes place. Organizations running cybersecurity bootcamps or workshops face similar challenges on a larger scale—hundreds of machines must be aligned, configured, and verified before exercises can begin. WADT reduces this extensive preparation to a matter of seconds, transforming a costly operational bottleneck into a streamlined, repeatable workflow.

The WADT platform accomplishes this by offering a unified toolchain that abstracts away environment configuration, dependency management, manual Dockerfile creation, and cross-platform inconsistencies. Instead of requiring users to navigate obscure GitHub repositories or resolve local environment conflicts, WADT presents each vulnerable application as an instantly deployable sandbox.

Users simply choose an app from a curated catalog and launch it through a standardized and reliable interface. This approach fundamentally changes the user experience: labs become available on demand; debugging setup issues becomes unnecessary; and learning objectives can be met more efficiently and more consistently.

WADT is built on a robust open-source stack designed for extensibility, reproducibility, and long-term maintainability. Applications are containerized and orchestrated using

Docker, ensuring that each session runs in a clean, isolated environment with predictable behavior. This consistency is crucial when teaching students delicate, nuanced concepts such as injection vulnerabilities, authentication bypasses, cryptographic misconfigurations, or insecure session handling.

The modular system design allows new applications to be onboarded quickly through lightweight deployment manifests, reducing maintenance overhead and enabling continuous expansion of the application catalog.

The objectives of this project are threefold. First, WADT aims to reduce friction in cybersecurity education by enabling instant access to vulnerable applications. This directly improves instructional efficiency and increases the time students spend on practical learning. Second, the project establishes a scalable and modular catalog of applications, supporting an expanding ecosystem of security labs that can grow with educational needs. Third, WADT prioritizes security, reliability, and user experience—critical elements for any platform used in controlled penetration testing and vulnerability analysis. These objectives collectively support a long-term vision of accessible, on-demand cybersecurity instruction that mirrors the tools and environments used by industry professionals.

The technical approach integrates a Django-based backend responsible for orchestration logic, container lifecycle management, and secure API endpoints. A React-based frontend provides intuitive controls for launching and managing application instances. Underneath these layers, Docker handles the actual container deployment, sandboxing, and cleanup routines.

This layered architecture separates concerns cleanly and supports maintainability, testability, and scalability. Automated expiration of containers, structured system logging, configurable deployment parameters, and an extensible manifest format ensure that WADT can evolve alongside new security curriculum needs and additional application types.

Beyond its technical accomplishments, WADT delivers clear business and educational value. By reducing the time required to launch vulnerable labs from hours to seconds, WADT increases instructional efficiency and reduces support burdens on instructors and teaching assistants.

Students benefit from reliable, consistent environments where debugging setup issues is no longer a prerequisite for learning. Academic institutions benefit from lower operational overhead, increased lab reproducibility, and the ability to offer more sophisticated security exercises without scaling complexity.

Over time, this contributes to a more skilled cybersecurity workforce—an increasingly important societal need given the rapid growth of cyber threats and the ongoing shortage of hands-on practitioners.

In summary, WADT transforms vulnerable web application deployment from a tedious, error-prone process into a fast, accessible, standardized experience. By removing setup barriers, it allows learners and professionals to focus on meaningful security practice rather than configuration hurdles. The system successfully meets its core objectives and establishes a strong foundation for future enhancements, including role-based access controls, network configuration management, and automated vulnerability reachability checks.

WADT is well-positioned for broader testing, institutional adoption, and continued development as a scalable, reliable platform for cybersecurity education and experimentation.

2 Problem Statement

In cybersecurity education and training environments, students, instructors, and practitioners frequently rely on vulnerable web applications to develop hands-on skills in exploitation, analysis, and secure development practices. However, the current process for accessing and preparing these applications is fragmented, time-consuming, error-prone, and sometimes frustrating. Learners must often search through external repositories, manually assemble dependencies, troubleshoot inconsistent instructions, and resolve compatibility issues before they can begin meaningful security work.

The environment issues significantly hampers learning efficiency and creates substantial barriers to entry for individuals new to web security or simply trying to gain new skills. The users most affected include undergraduate students enrolled in cybersecurity courses, teaching assistants responsible for supporting these courses, instructors preparing lab materials, and professionals seeking repeatable environments for skill development.

The core problem is that vulnerable web applications, which should serve as accessible practice tools, are instead difficult to prepare and maintain. The process can require extensive setup time, specialized configuration knowledge, and repeated manual intervention. When each user must independently set up their own environment, inconsistencies between machines, operating systems, and dependency versions become inevitable.

An individual may need to set up a new environment for each situation which takes time and computer resources to set up. These inconsistencies often lead to environments that behave unpredictably, making it challenging for instructors to deliver uniform learning experiences and for students to replicate demonstrations or complete assignments reliably. What should be a straightforward path to learning instead becomes a technical burden that distracts from the actual educational objectives.

The impact of this problem is both instructional and operational. Students lose significant amounts of time resolving issues unrelated to the concepts they are meant to learn, which reduces the depth and quality of their hands-on practice. In some courses, entire class periods are consumed by troubleshooting instead of exploring vulnerabilities or analyzing system behavior. This results in reduced learning outcomes, increased frustration, and uneven student performance.

Instructors and teaching assistants must devote additional hours each week to helping students configure their environments, creating bottlenecks that frustrate both learners and staff. Across larger programs or cohort-based training sessions, this problem scales into hundreds of cumulative hours spent on setup rather than education.

From an institutional perspective, the inconsistency of environment preparation undermines the reliability of assessments, demonstrations, and practical exercises. When students cannot reproduce the same application behavior, instructors struggle to evaluate performance fairly or ensure that students are meeting course objectives.

This lack of repeatability also limits the institution's ability to modernize or expand its cybersecurity curriculum, as the administrative burden of maintaining numerous vulnerable applications becomes unsustainable without standardization.

At its core, the problem is not the lack of vulnerable applications themselves, these are widely available and published extensively, but the difficulty and inconsistency involved in making them usable in an educational context. The central challenge is the absence of a reliable, repeatable, and accessible method for preparing practice environments that behave predictably across many different users and over time.

This misalignment between educational needs and practical accessibility creates a barrier to cybersecurity skill development, one that directly impacts students, instructors, programs, and workforce readiness.

3 User Stories

WADT serves a wide range of users who each interact with the system in different ways and for different purposes. User stories illustrate how various stakeholders engage with the platform, what goals they seek to accomplish, and how WADT supports those goals. These narratives highlight both everyday usage and exceptional scenarios to ensure the system addresses practical needs across instructional and exploratory contexts. User stories also help clarify operational and functional expectations for our product that help to guide design decisions and expected system behavior.

3.1 Student User Stories

A student launching their first vulnerable web app

An underclassmen cybersecurity student opens the WADT interface after hearing in lecture about SQL injection techniques. The student has never cloned a repository or configured a Docker image before. They navigate to the catalog, choose an intentionally vulnerable blog application, and press the "Deploy" button.

Within seconds, the system spins up an isolated container instance. The student interacts with the web app directly through a provided URL and begins practicing query manipulation without ever having to troubleshoot dependencies or install packages. The student repeats this process with other applications to compare different vulnerability patterns, confident that each deployment is clean and reproducible.

A student preparing for an exam or certification

An upperclassmen student studying for an applied cybersecurity exam needs rapid access to practice environments in order to diagnose and possibly fix vulnerabilities. Instead of manually rebuilding labs for setup, they use WADT to deploy known vulnerable applications repeatedly, resetting containers between attempts.

The system's predictable behavior gives the student a stable environment to test their skills in injection, authentication bypass, and insecure session handling. The ability to rapidly deploy multiple apps back-to-back supports structured, focused study sessions that would otherwise be slowed by setup overhead.

A remote learner with limited hardware

An undergraduate computer science student only has access to an older laptop with limited processing power needs to participate in their online cybersecurity course. Traditional vulnerable app environments are too heavy for their machine. By accessing WADT through a web browser on a lab computer, the student launches remote container instances hosted on the central server, bypassing local hardware constraints.

They can complete all labs, interact with vulnerable applications, and submit assignments without needing to install or run any resource-intensive components on their own device. WADT thus provides equitable access for students across a wide range of computing environments.

3.2 Instructor User Stories

Setting up a classroom lab session

An instructor is planning an in-class workshop on cross-site scripting. Instead of preparing custom VM images or relying on students to install tools in advance, the instructor uses WADT to curate a list of approved vulnerable applications. At the start of class, students are instructed to log in, select the appropriate application from the pre-filtered catalog, and deploy it.

The entire class is ready to begin within minutes. The instructor can rely on consistent application behavior across all student instances, making it easier to teach, demonstrate, and assess learning outcomes.

Monitoring activity and offering guidance

During a lab period, an instructor monitors student progress by observing container deployment logs and instance status. When a student encounters unexpected behavior, the instructor checks whether the issue is due to misuse, container expiration, or application runtime errors. The system's structured logging allows the instructor to trace events and help students without needing to replicate their exact steps manually.

The instructor can advise students to reset or re-deploy containers when necessary, ensuring a smooth instructional flow. If absolutely necessary, the instructor can intervene directly in a student's container to shutdown their environment if they suspect misuse.

Creating new assignments or labs

When designing a new assignment around insecure direct object references, the instructor identifies a relevant application and verifies that it behaves as expected within a WADT deployment. They then write assignment instructions referencing the platform's deploy URL and reset functionality.

Because WADT standardizes deployment, the instructor is confident that all students will interact with an identical environment, reducing ambiguity and minimizing grading complications.

3.3 Teaching Assistant and Support Staff User Stories

Providing troubleshooting assistance

A teaching assistant responds to student messages in a course discussion forum. In previous semesters, support requests often involved dependency conflicts, incorrect virtual machine configuration, or issues setting up virtual environments.

With WADT, most issues instead involve conceptual misunderstandings or misuse of platform features, which the TA can quickly diagnose by reviewing container logs or guiding the student to redeploy. The TA experiences a reduced support burden and can focus more on helping students improve their security skills.

Ensuring lab readiness before class

A TA responsible for lab infrastructure checks the WADT dashboard the day before a scheduled assignment release. They verify that all approved applications deploy properly, that container expiration and cleanup processes are functioning, and that no stale containers remain active.

WADT gives the TA confidence that the environment will be ready for the next day's class without requiring manual VM imaging or environment resets.

3.4 System Administrator User Stories

Managing resource usage and container lifecycle

A system administrator responsible for the server hosting WADT monitors system load and storage utilization. They rely on WADT's automated expiration policies to remove inactive instances and prevent resource exhaustion.

If an unusually high number of deployments occurs during a bootcamp week, the administrator uses the platform's administrative controls to adjust cleanup frequency or enforce deployment limits. This oversight ensures the system remains stable during periods of heavy usage.

Adding new vulnerable applications

A system administrator receives a request from an instructor to add a newly published vulnerable application into the WADT catalog. Using the deployment manifest template, the administrator builds a lightweight descriptor file that defines the container image, environment variables, exposed ports, and reset behavior.

After testing the new application's deployment and runtime behavior, they add it to the catalog. The modular design allows applications to be introduced quickly without modifying the core system.

3.5 Security Researcher and Practitioner User Stories

Testing new tools or prototypes

A security researcher developing a scanning tool needs predictable, repeatable vulnerable targets. Instead of curating their own set of local containers, the researcher uses WADT to spin up multiple application instances representing different vulnerability classes. They run their scanner against each instance and compare results across repeated deployments, benefiting from the consistency of WADT's standardized environment.

Demonstrating vulnerabilities to a non-technical audience

A practitioner giving a presentation to stakeholders needs a safe, controlled demonstration environment. They deploy an application through WADT and walk the audience through a simplified exploitation workflow. Because the environment is isolated and disposable, the practitioner can safely demonstrate risky techniques without exposing production systems or requiring complex setup.

3.6 Exceptional and Edge Case User Stories

A user forgetting to shut down instances

A student deploys multiple applications for exploration but forgets to stop them before logging out. WADT's expiration mechanism automatically terminates these containers after 24 hours, freeing resources and preventing unintended long-term usage. When the student returns, they can redeploy fresh instances with no impact on their workflow.

A deployment failure due to invalid configuration

If a user attempts to deploy an application whose manifest contains an error, WADT detects the issue and presents a clear error message rather than leaving the user to debug the underlying container logs. Administrators are alerted to the issue so they can patch the manifest. The user experience remains smooth, and the system prevents the propagation of broken configurations.

3.7 Summary of User Story Themes

Across all user roles, several common themes emerge. Users value the ability to deploy vulnerable applications quickly, reliably, and consistently. They rely on WADT to minimize setup friction, provide a stable environment, and support repeatable workflows.

Whether the user is a student, instructor, administrator, or researcher, WADT enhances their capabilities by standardizing vulnerable application deployments and reducing the operational overhead typically associated with cybersecurity labs. These stories collectively illustrate the breadth of real-world situations WADT is designed to support.

4 Project Description

4.1 Overview

The WADT project allows students and instructors to browse a catalog of ready-to-deploy vulnerable webapps and manage them through a graphical interface. WADT is the first step in the learning pipeline that ensures students and instructors are on the same page in minimal time. Their efforts can shift to investigating the app itself, while the container-based architecture of WADT ensures they have reproducible sandboxed builds available at the click of a button.

4.2 The Setup Timesink Problem

Cybersecurity students want to practice exploiting and patching vulnerable webapps. When they sit down to practice, their time is eaten up by installation and setup before they can start hacking.

Cybersecurity instructors want to provide labs for their students. These labs require research, setup, installation, and testing. The instructor must take the time to guide each student through the same prerequisite steps before the real learning can begin.

Both parties are capable of following a list of instructions to set up a target webapp. These skills are not the desired target of practice. However, requiring setup work on a per-lab basis means wasting learning time.

4.3 Project Goals and Objectives

WADT aims to eliminate redundant setup work by providing a standardized environment where vulnerable web apps can be deployed instantly in isolated containers. Building on this foundation, WADT introduces a modular deployment toolkit that streamlines lab creation and management.

Each app is launched with its dependencies in a reproducible containerized setup, supported by command-line scripts for rapid provisioning, teardown, and scenario resets. Network and volume isolation ensure clean separation between labs, while customization hooks allow instructors to inject or remove vulnerabilities as needed.

This tooling empowers educators to design focused, resettable exercises and enables students to concentrate on exploit development and remediation without setup distractions.

5 Personal Motivations

5.1 Dustin Allen

My motivations for this project was to further my skills in Python and more experience with working on API's specifically. Along with this I was really interested in the idea of creating something that deals with security issues. Being able to work on a completely different API infrastructure and having more freedom to experiment and learn on my own has me very excited to see what I can create and how the API side of the project can be created.

With an interest in full-stack development this also gives me a good chance to research and learn about industry standards along with making sure to implement them into the project not only to benefit the project but also to have in a portfolio to share during future job interviews.

I also believe that this project has lots of potential to reach stretch goals and to implement features that could introduce very nice quality of life changes for users. This project is also important to me as it has the potential to help get others more interested in computer science and specifically in the cybersecurity sector.

Personally I had wished that I had learned more about cybersecurity earlier in life and with this project it will help others have a tool to more easily learn about and gain experience in identifying security issues with real world examples. This also gives me the chance to experience more about cybersecurity as well through testing the vulnerable apps, and understanding the research that my group members do on the vulnerable apps that we choose to dockerize for this project.

Since I was chosen to be project manager and work on the backend portion of the project, my ideas are to first work with my fellow backend developer to categorize and separate the functionalities into API's that we will need. To this effect I want to have more collaboration between our sections to make sure that we are testing our code together so that we prevent compatibility issues later.

I believe that communication and each department sharing the important details on how features work will be important in ensuring that we get the project done on time and will aid in fixing issues that arise quickly and efficiently. As project manager my main goal is to not micro-manage and to use the weekly meetings as check-ins to make sure we're on schedule. And if we fall off schedule then my idea is to figure out the root of the issue and make sure that we communicate more to prevent falling off schedule again. I feel that project manager is a good way to push myself to be more knowledgeable about all facets of the design processes.

Aside from those motivations I also have some aspirations and features I would want to propose to the group in the form of stretch goals and collaborative goals to help the team reach the most effective and efficient project that we can make. I hope to make a team environment where all my fellow members feel comfortable sharing their ideas and thoughts on what we do. This way we can help identify issues on the personal items we work on and can give each other ideas on what might work better or just to encourage each other on good work done.

Our sponsor's encouragement for us to come up with good ideas and additions to the project that we would find interesting and that's within scope of the project has also motivated me a lot to do more research on possible additions to the project. I plan on challenging myself on the API side to add features such as a health check for the dockerized apps in case of crashes, and some other features that would help with stability to ensure we aren't using too many resources by running the apps.

For me having the support of our sponsor to take on our own initiatives for the project has made this much more interesting and also will give me more things to talk about in future interviews.

5.2 Daniel Armas

My motivations for this project center on advancing my growth in Cybersecurity and Networking, while also developing my UI/UX skills. Although I am still relatively new to the cybersecurity and networking field, it is the area that excites me most. When I learned that this project was recommended for those interested in cybersecurity, I knew it was the right fit.

One of the aspects that drew me in was the opportunity to work with Docker containers, a technology I have not yet used but have been eager to explore. I would also like to strengthen my skills in frontend development, as I have opportunities outside of senior design as a contractor building websites, and I want to gain the skills necessary to excel in that role.

I am especially interested in collaborating with the Docker and backend teams to integrate Docker into a web-based solution. While my current ideas are limited due to my unfamiliarity with Docker, I am committed to learning quickly and contributing to building an efficient, reliable, and user-friendly service.

Having worked with simulated testing environments through certifications, I was always a little curious how it worked. This project presents an exciting chance to design and improve upon that experience, creating a solution that is practical and effective.

5.3 Rithik Dhakshnamoorthy

When I began considering options for senior design, I wanted a project that would give me the chance to develop skills I hadn't yet mastered. I liked a lot of the projects shown to me that are sometimes related to computer science, but this project stood out because it was new, open-ended, and offered plenty of space for some creativity. It also introduced technologies I didn't work with before, which makes it an excellent challenge. Cybersecurity and Docker stood out to me, since both are widely used in today's industry and tied to my future work after graduation.

This semester I am taking several cybersecurity courses, which has given me a stronger academic foundation in the subject. However, most of that learning has been theoretical, focused on concepts, frameworks, and case studies rather than doing hands-on application. Because it revolves around intentionally vulnerable web applications, it provides a unique opportunity to connect what I am learning in class with practical, real-world scenarios.

I see this as a chance to reinforce my coursework by applying security principles directly, gaining experience with how vulnerabilities are created, exploited, and mitigated in systems that are similar to professional environments.

Another reason I chose WADT was the chance to work and understand computer engines like Docker. At the time, I had no direct experience with containerization, but I was aware of its growing importance in professional environments.

As part of the Docker team, my role is to dockerize vulnerable web applications so they can be deployed consistently and reproduced across different systems. This responsibility allows me to learn how to manage different services, simplify builds, and keep the project dependable for teaching purposes. I am motivated by the challenge of building deployment tooling that not only coordinates multiple services but also supports security testing in a controlled environment.

Finally, I liked how the sponsor, who was a recent graduate who had guided other senior design teams before, gave us the freedom to make additions to our project that can improve of how it looks more better. I plan on learning and interacting more with Docker and cybersecurity, not only to strengthen my understanding of containerization and orchestration, but also to explore how these technologies intersect with modern security practices.

By experimenting with Docker in different deployment scenarios and studying common attack vectors, I aim to gain a deeper awareness of how vulnerabilities arise in web applications, such as misconfigured containers, insecure defaults, or unpatched dependencies. In the end, these motivations can help me reach my goals of getting a professional job in the future after graduation.

5.4 Jacob Plotz

I chose WADT because our sponsor Lt. Fischer titled his introduction slide `$whoami`. I thought, "seems like a mighty technical fellow!" Then I saw the rest of the pitch and my prediction was verified.

WADT is run by a sponsor with a successful track record of Senior Design projects, available expertise and mentoring, and a keen interest in a slick MVP. Our scope is well-defined and achievable with room for improvement. We get to make a site that solves a real problem, involves a heap of technical challenges, and I can explain it to my grandma without needing to stop talking to regain my strength. I would contend that I have found the Holy Grail of Senior Design Projects. I will enjoy writing maps and zips until the cows come home.

My main goal is to transform Docker from a pesky whale that lingers unused on the cybersecurity student's desktop into a useful tool by abstracting away its presence. The name of our game is "hide the whale."

The project is an interesting challenge because we are building a very transparent tool: Docker itself already uses a client/server architecture with a REST API available. There is a sense in which we are creating a client/server architecture over top of the existing architecture.

We have been able to gain insight from the design of Docker itself and by investigating the REST API it provides (and the SDK+CLI that wrap around it).

5.5 Luke Samuel Sandoval

I chose to take on the challenge of Dockerizing our vulnerable web applications because containerization has become a foundational skill in modern software engineering and cybersecurity. As I approach graduation and prepare to enter industry, I want hands-on experience with the same tooling used to deploy, isolate, and scale real-world systems. This project offers a direct way to build those skills while contributing something genuinely useful to students who need reliable practice environments.

There is a clear need for this work. Today, students often spend more time configuring machines, troubleshooting dependency issues, and reconciling outdated instructions than actually practicing security concepts. By packaging intentionally vulnerable applications into clean, reproducible containers, we can remove the setup tax that slows down learning and give students an environment where they can immediately focus on exploitation, testing, and experimentation.

I am also motivated by the opportunity to apply container orchestration techniques in a realistic, production-minded setting. The project's structure and access to an AWS instance provide an environment that balances experimentation with real operational constraints. This allows me to explore scalability, automation, and reliability in a way that mirrors the workflows found in industry platforms.

My technical vision includes building a curated catalog of vulnerable applications, each defined with a lightweight deployment manifest that simplifies onboarding new apps. By using Python bindings and the Docker Engine REST API, the system will expose one-click lifecycle actions such as deploy, restart, stop, reset, and scheduled expiration after 24 hours. Overall, we will implement structured logging for container events to support auditing and instructor oversight, as well as a frontend UI that makes these operations intuitive.

The sponsor's long-term vision of as enabling network configuration from the UI and integrating vulnerability checks align with my interest in creating tooling that is both educational and operationally sound. Longer term, I hope to incorporate role-based access controls, per-session launch tokens, and a small registry of approved base images to maintain reproducibility and security.

Ultimately, my motivation is to build a system where a first-time user can launch a vulnerable web application in seconds without wrestling with configuration, while I gain practical experience with the technologies and workflows I expect to use in my career.

5.6 Jereme Saunders

One of my primary goals heading into senior design was to push myself into areas where I had less experience and to expose myself to technologies and practices that I knew would be valuable in my future career. I considered several projects that offered opportunities to learn new skills, but this one stood out. It was a brand-new project with plenty of room for creativity, and it presented multiple opportunities to work with unfamiliar technologies in a meaningful way. The focus on security and the chance to gain experience with Docker made it especially appealing and directly applicable to future software development work.

Although I had learned about security in various classes, I had very little practical experience with it being a central focus of a project. In most of my previous work, security often took a back seat to other priorities. This project, however, is inherently tied to security due to its nature as an educational tool designed around vulnerable web applications. That made it an ideal opportunity to gain first-hand experience in a field that is becoming increasingly important as people become more reliant on technology.

Another major factor in my decision was the involvement of Docker, as this was the only project pitch that was explicitly working with it. I had zero experience working with Docker, but I have heard of it and learned of the increasing popularity of it with development in a professional setting. This project felt like the perfect chance to become familiar with a technology that I will almost certainly encounter again after graduation.

I was also drawn to the idea behind the project itself. I liked the idea of a project that was made by students for other students. It also helped that the sponsor was a recent graduate who had worked with other senior design teams.

That familiarity with the process made the project feel approachable, and I felt much more comfortable committing to a project with a sponsor who understood both the technical and academic expectations. Choosing a senior design project is a long-term commitment, and knowing I would be working with someone experienced in the process made the decision to work on this project much easier.

6 System Pipeline Diagram

The following activity diagram (figure 1) presents the high-level overview for users using WADT. The diagram demonstrates the main interaction loop that enables users to move from authentication to hands-on practice.

This emphasizes WADT's primary goal of eliminating setup overhead while providing a streamlined and intuitive interface. Once a user has been authenticated, they can deploy any available applications in a single step and immediately begin testing. Container management involves providing users the capability to reset or inspect containers as needed without restarting the entire process. When the session ends, either by a user manually stopping or automatic timeout after 24 hours, all resources are cleaned up to maintain system integrity.

More detailed activity diagrams will follow in the System Architecture section to provide more substantial details of the notable activities, namely for backend deployment of containers, interactions with containers, and container cleanup. These diagrams have been separated to provide clarity.

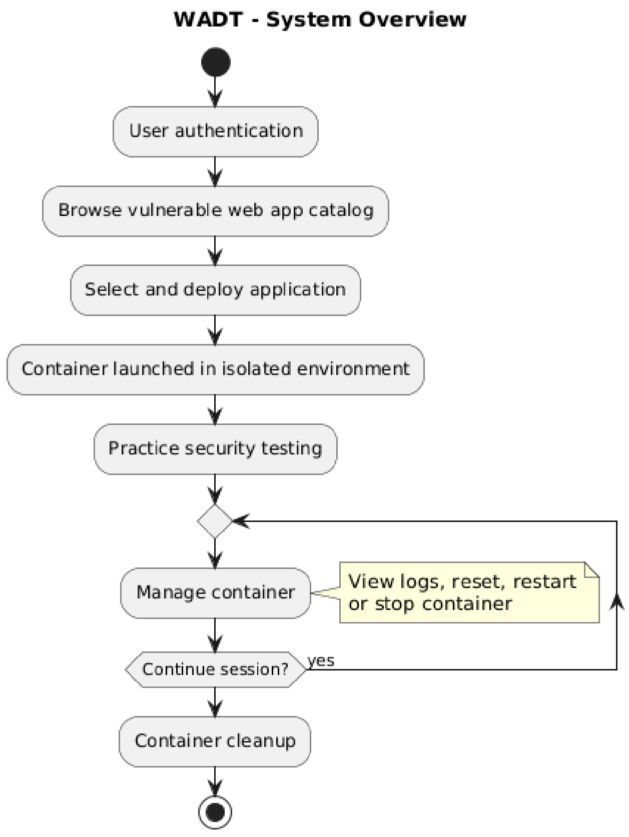

WADT - System Overview
User authentication
Browse vulnerable web app catalog
Select and deploy application
Container launched in isolated environment
Practice security testing
Manage container
View logs, reset, restart or stop container
Continue session?
yes
Container cleanup

Figure 1: System Overview

7 Broader Societal Impacts

The WADT project has implications that extend beyond its technical implementation. By lowering barriers to cybersecurity practice, standardizing deployment workflows, and creating a safe environment for exploring vulnerabilities, the platform supports broader goals in education, ethics, public safety, and digital accessibility.

The following section outlines the societal impacts of WADT and how the project contributes to the development of a more secure, informed, and equitable technological landscape. We are mainly building for a classroom environment, but the implications of our work extend beyond that as our code is open source and has minimal technical requirements.

7.1 Advancing Cybersecurity Education

A central societal benefit of WADT is its ability to improve the accessibility and effectiveness of hands-on cybersecurity education. Students often face steep learning curves when configuring environments for security practice, with setup errors consuming valuable instructional time. WADT eliminates this friction by providing a streamlined interface that instantly deploys vulnerable applications in reproducible containers. As a result, students and cybersecurity learners can engage more deeply with core security concepts rather than technical logistics.

Furthermore, instructors can leverage a consistent platform to design labs, assessments, and demonstrations without worrying about dependency issues or system inconsistencies. This creates a more stable foundation for curriculum development and improves both the quality and scalability of cybersecurity instruction.

Another key factor when considering cybersecurity education is the ability to keep up with the rapidly evolving threat landscape. WADT's modular architecture allows new vulnerable applications to be added easily, enabling educators to incorporate emerging vulnerabilities and attack vectors into their teaching. This adaptability ensures that learners are exposed to relevant, up-to-date challenges that reflect real-world security concerns as they are uncovered.

7.2 Strengthening the Future Workforce

As cyber threats grow in both frequency and complexity, industries increasingly depend on professionals with practical, tool-driven experience. WADT contributes to workforce readiness by exposing students to real-world technologies such as Docker, container orchestration, structured logging, and distributed system workflows. This alignment with industry practices ensures that learners who train on WADT gain marketable skills and familiarity with the same tools used across engineering, DevOps, and security operations teams.

By creating an environment that mirrors professional infrastructure patterns, the platform helps bridge the gap between academic theory and operational reality. This ultimately supports a more resilient and capable cybersecurity workforce.

7.3 Promoting Safer Software and Responsible Practices

WADT helps cultivate a culture of safe and ethical experimentation. Many aspiring developers and security practitioners lack access to environments where they can explore vulnerabilities without risk. By providing isolated containers that can be reset, expired, and audited, WADT encourages responsible hacking practices and discourages unsafe experimentation on unauthorized systems.

The platform also empowers future engineers to understand how vulnerabilities emerge in the first place. As students gain experience interacting with unintentionally vulnerable components, they become better equipped to follow secure coding practices, evaluate risk, and recognize problematic design patterns. This knowledge, when carried into industry, can contribute to reducing the prevalence of preventable vulnerabilities in consumer and enterprise software.

7.4 Ethical Considerations

Although WADT supports learning in a safe environment, working with vulnerable systems inherently raises ethical questions. It is essential that any platform facilitating exploitation techniques reinforces proper intent, scope, and responsibility. WADT's architecture—isolated containers, automatic cleanup, controlled user sessions, and audit logging—provides guardrails that promote ethical use while minimizing the potential for misuse.

The project also encourages best practices in ethical hacking education, including responsible disclosure, respect for system boundaries, and an emphasis on consent-based testing environments. By embedding these lessons early, WADT helps shape ethical instincts and responsible behaviors in the next generation of cybersecurity professionals.

7.5 Environmental and Sustainability Impact

While WADT relies on compute resources, its design can contribute to more sustainable educational infrastructure. The use of short-lived containers that automatically expire after 24 hours prevents resource waste and minimizes idle CPU usage. Automated cleanup routines ensure that unused environments do not persist indefinitely, helping reduce the platform's overall energy footprint. When considering large-scale deployments across multiple institutions, these efficiencies can add up to significant reductions in resource consumption. Developing a system that takes energy and resource into account will only become more important as cloud computing continues to grow.

Containerization itself is also more resource-efficient than running full virtual machines for each student. By packaging only the minimal runtime environment needed for a specific lab, WADT reduces overhead, encourages lighter deployments, and supports more sustainable use of cloud or on-premise compute resources. Over time, institutions adopting containerized labs may reduce electricity consumption, cooling demands, and hardware wear associated with heavier virtualization strategies.

7.6 Accessibility and Educational Equity

WADT can help broaden participation in cybersecurity by reducing technical barriers that disproportionately affect beginners or students without prior systems administration experience. By removing setup challenges, the platform allows learners of varying backgrounds, including those without powerful personal hardware, to participate fully in hands-on labs.

Additionally, because WADT can be hosted in the cloud, students who rely on public devices, low-cost laptops, or university lab computers can still access full-featured environments. This supports a more equitable academic experience and expands opportunities for students in under-resourced programs.

7.7 Long-Term Maintainability and Community Benefit

Beyond the classroom, WADT has potential value for the broader security community. Its modular design allows new vulnerable applications to be onboarded with minimal overhead, enabling the platform to grow into a lasting resource. If released openly or adopted by multiple institutions, WADT could become a shared ecosystem for standardized, safe, and easily deployable security labs.

This fosters cross-institution collaboration, reduces duplicated effort, and supports long-term sustainability. By contributing tooling and documentation that others can build upon, the project reinforces the open-source philosophy that underpins much of modern cybersecurity research and development.

8 Minimum Viable Product

8.1 Scope and Limitations

WADT is intended to be used in an instructional setting as part of a lab. While all code developed for the project is free and open-source, the site and the AWS instance it runs on are not accessible to the general public. The team members and sponsor are the only individuals expected to access the site and instance while it is in early development.

Later in development, our users will access both the WADT site and the instance through their instructor similar to how we have accessed these two resources. We expect that users (assisted by instructors) are capable of reproducing our instructions for accessing the instance and authenticating to access the site when we provide these documents.

The current scope of WADT does not provision for multiple concurrent users at a time with containers managed on a per-identity basis. Each container may be associated with one and the same identity, under the assumption that students will actively collaborate and share management of the containers.

8.2 Hosting and Security Considerations

Hosting vulnerable applications on a public, internet-facing server is considered a bad practice. Special precautions must be taken to ensure that malicious actors cannot compromise the instance through a vulnerable container, and that the containers themselves are not compromised and used for criminal purposes.

A viable solution may be to avoid hosting the site at all. It is technically possible for the site to be kept local-only and require each student to download and run the project from their own machine. However, this would largely defeat the purpose of the project as it would require its own setup and installation steps.

Another option may be to install and configure the project local-only on the AWS instance but avoid exposing the instance to the public internet. Each student would be required to connect to the instance through a VPN. The preceding two solutions are dead simple to the point of being below the level of sophistication expected of a 9-month project.

A more feasible solution would be to host the site and add an extremely restrictive security configuration. The instance could be configured to only accept incoming traffic from and send outgoing traffic to a small number of IPs or an IP range explicitly listed in a whitelist. This solution is ideal if the machines used in the lab have IP addresses that fall within a specific range or are unlikely to change over a semester-long course. Under this scheme, the target webapp containers would be blocked from making requests to third parties. Discretion would be left up entirely to the students (a rogue student would have the ability to change the whitelist).

8.3 Vulnerable App Selection and Containerization

In order to give our users a variety of options to choose from, we want to select a diverse set of vulnerable web applications. There is an existing registry of vulnerable web applications known as the OWASP Vulnerable Web Applications Directory (VWAD). This registry contains a variety of web applications that are intentionally designed to be vulnerable for educational purposes. We will select at least 7 web applications from this registry that cover a range of vulnerabilities and technologies.

The web applications listed in this registry record the technologies used as well as the specific vulnerabilities they contain. This information will help us ensure that we are providing a comprehensive and varied set of applications for our users to work with.

Creating a quality docker image for each web application is a non-trivial task. We will need to ensure that each image is reliable (it always deploys when required) and reproducible (the application source code and initial state are immutable and consistent). This may involve creating custom Dockerfiles, configuring the applications correctly, and testing the images thoroughly.

8.4 Technical Debt and Maintainability

While we may assume one identity controls all containers, it may still be smarter in the long run to design for the possibility of multiple concurrent users, even if this isn't made possible by our hosting configuration.

Since ”make it support multiple users” is an excellent goal if this project returns to Senior Design, we will design for the possibility of this extension by programmatically tracking container IDs and associating them with the user's identity and actions. This design decision also makes it easier for us to manage state and tell the user ”You cannot stop a container you haven't deployed!” even in the single-user version of WADT.

As a stretch goal, future teams that work on this project would be able to extend our API instead of rewriting it.

8.5 ”Do One Thing Well”

WADT is a stage in a learning pipeline. Pipelines are made possible by small, focused applications that do one thing well and rely on standard interfaces. WADT is an abstraction for managing containers. It is not an abstraction over the process of finding API endpoints, making requests, patching, and testing. Therefore, WADT should use conventional input and output formats and get out of the way of the user's preferred tools.

If all the user needs is a URL, provide the URL in an easy-to-copy format. Nothing else.

8.6 Definition of Done

- At least 7 target vulnerable webapps are listed on the site with icons, names, and descriptions. An image for each webapp has been created and tested for reliability (always deploys when required) and reproducibility (application source code and initial state are immutable and consistent). Each target app passes automated tests ensuring that the desired vulnerable endpoints are confirmed reachable.

- Container limits. Each container is allotted a specific amount of compute resources and network bandwidth. Each container is only kept running for a maximum of 24 hours from the last user interaction, at which point it will be automatically downed.

- Status indicator. Each webapp indicates its state. State indication matches actual container behavior (does not indicate stop when container with specified ID is still running, etc.).

- Deploy feature. Users can deploy any of the target webapps. Deploy is accompanied by connection information, assisting users in making requests to the vulnerable application.

- Stop feature. Users can stop any webapp. Stop only functions for containers which are currently running.

- Restart feature. Users can restart each webapp. ”Restart” in this context means stop the webapp the student has been working on patching and start it again.

- Reset feature. ”Reset” means to stop the version of the app the student has been working on, and deploy the original image before any modifications were made.

- Instructor controls. Site admins can view logs with the status of each container. Each deploy, stop, and restart action is logged. Admins have the option of deploy, stop, and restart, and their actions override the wishes of users (while preventing race conditions and state inconsistency). Instructors have the option to ”disable” a container, meaning that the container leaves control of the students and only the instructors can manage it. When a container is disabled, students are unable to make any state changes to the container.

- Student-facing logging. Student-facing logs must provide a window into the container's behavior, without burdening the student with unnecessary Docker-specific details. For students, no single action in the UI should require scrolling the view of the logs (everything resulting from one action should fit on one view). All major interactions through the UI should be logged with timestamps. The logger must preserve history from the moment the container is started to the moment it is manually downed or auto-downed after 24 hours. In the event of an error resulting from a failed attempt to patch a vulnerability, the student must receive the full error message.

- Instructor-facing logging. Must show timestamped logs of all actions taken through the UI by students. Must show container status changes. Uptime for each container must be available to instructors.

8.7 Stretch Goals

- Instructor-facing controls to add new containers. The admin interface provides a method to upload a custom Dockerfile and add that image to the catalog. Instructors will be able to add container metadata (name, description, tags, category, etc.). This new container will function identically to the containers built-in to the site.

- Automatic patch checking. User communicates which API endpoints they wish to patch, and the site automatically checks periodically if that endpoint is still vulnerable. Status reports are generated showing which of a set of endpoints are still reachable at the time the report is generated.

- UI allows editing of network settings per-container. Users can customize port number on a per-container basis through the UI.

- System resource usage display. Admin UI displays, in addition to container uptime, CPU, memory, and network usage for each container.

- Custom colorschemes. Make site compatible with widely used color scheme file formats (minimum: .Xresources, .itermcolors, Windows settings.json, and .yml used for GNOME terminal and xfce4 terminal). Students will be able to use the same colors for WADT and for their terminal.

- Custom interactions window. Users can issue (arbitrary) commands to the container through the site. Users may edit files with a text editor through the site. This stretch goal is extremely ambitious. The security vulnerabilities potentially resulting from exposing a shell to a container are numerous and severe. Additionally, embedding a full terminal emulator inside of a webapp is a difficult task.

8.8 Tech Stack

- **Django** is a web framework for Python that makes use of the Model View Template (MVT) architecture. Common middleware, a database administrator interface, class-based views, and URL routing are a few of the tools the project uses which are included in Django. Django can be extended with the Django Channels library for handling tasks and the Django REST Framework for creating APIs that don't fit the MVT architecture.

- **Python** is needed to use Django. Its support for functional and object-oriented programming is reflected in the availability of Django's functional views or class-based views. Its available, easy-to-use libraries for interfacing with Docker blend well with Django idioms within the project.

- **JavaScript** adds an interactive element to the website's frontend. It enables use of the React framework.

- **React** is a frontend library for JavaScript. It's components offer additional power and control over the relatively plain HTML template syntax of Django. Stateful components of the UI (logs, timers, status indicators) can easily be modeled with React features.

- **Docker** containerizes vulnerable webapps and has actively developed SDKs for Python. It ensures that each vulnerable webapp runs in a separate, isolated environment.

- **AWS** provides scalable hosting for the project.

9 Backend Research

9.1 Hosting Provider

9.1.1 DigitalOcean

DigitalOcean droplets are small-sized virtual machines that are billed on a monthly basis. The team has previous experience using DigitalOcean droplets for hosting, making a Basic droplet a potential choice. However, since WADT is a SaaS application, in DigitalOcean's own terms a Basic droplet may not suffice. Additional tiers of compute and memory would impose an unfair cost on the student group.

9.1.2 BuyVM

BuyVM virtual machines are cheaper than a DigitalOcean droplet, but lack the support and convenient admin UI of DigitalOcean or AWS. BuyVM would allow maximum control and potentially a relatively unique hosting stack comprised of a hardened *BSD distribution. However, this route was not chosen for the first iteration of WADT due to the high learning curve.

9.1.3 AWS

AWS EC2 was recommended by our sponsor and provided and paid for by our sponsor. This made it an attractive option for the team, even if the pricing model creates risks and additional considerations regarding extensive network traffic. AWS benefits from extensive documentation and high configurability. Ultimately, a single instance was chosen to start with the option of scaling to additional instances as usage of the project grows.

9.2 Host Operating System

9.2.1 OpenBSD

OpenBSD is a security-focused UNIX system. While using a security-focused system for a cybersecurity-focused project is a good idea, the relative scarcity of documentation meant a steep learning curve for the team.

Considering a *BSD distribution was quickly invalidated as the team determined that Docker is a Linux-specific application. Docker relies on Linux kernel namespaces as part of its core design, meaning that to use a *BSD distribution would require dramatic changes in other parts of the project, effectively invalidating WADT's original purpose. A BSD-based project would make use of Jails instead of Docker. Since users expect to manipulate Docker containers with WADT, changing to BSD would require a change in our MVP, making it more than a mere technical decision.

9.2.2 CentOS

CentOS is a Linux distribution which is used internally for the labs run by our sponsor. For the purposes of our users, the main functional difference they will experience between CentOS and other choices of distribution is the package manager. Key project pieces (Docker, NGINX, etc.) are available for every Linux distribution the team considered, the only difference between them being familiarity.

It was decided to optimize for our developer team's productivity in the beginning, and since no member had previous experience with CentOS, it was decided to postpone its adoption. WADT can be migrated to CentOS for lab instructors and students once its core features are developed.

9.2.3 Amazon Linux

Amazon Linux is a distribution provided by Amazon specifically for EC2 instances. Its primary advantage for the team would be easy integration with IaC tools like Terraform and CloudFormation.

Amazon Linux is not ideal for either developers or users in terms of previous experience and familiarity. It is not used internally within labs and was not used for testing and early development.

9.2.4 Ubuntu

Ubuntu was chosen to promote a fast ramp-up and make it easy for every member of the team to contribute. Ubuntu is an ideal choice, since it supports all elements of the project tech stack and is widely-used.

Specifically, Ubuntu by default enforces an externally-managed Python environment which effectively necessitates using virtual environments to manage project dependencies. This ensures no developers have conflicts between project-specific packages and system-wide packages and reduces dependency conflicts.

PostgreSQL is also available in Ubuntu's default repositories, simplifying project setup when compared to CentOS or Amazon Linux.

9.3 General Security Concerns

Security was one of the biggest concerns as hosting intentionally vulnerable applications inherently has many risks that need to be narrowed down and handled before pushed to a live server.

Docker has some very strong security primitives that will be utilized to help the application be as secure as possible. Linux namespaces will be used to help ensure that each container has its own isolated process tree, filesystem, and network stack, ensuring that the containers are self-contained and unaware of the host system. A dedicated, isolated Docker bridge network will be used for each student's session. Containers will not be attached to the default bridge or the host network.

Using this will aid in preventing these containers from scanning or attacking other services on the host machine/other containers. This essentially enforces a "deny by default" policy for communication within the containers and system itself.

One of the other main concerns was the containers not shutting down on time or a student potentially forgetting to shut down the container itself. Not only would this potentially lead to high overhead costs, it would also potentially leave an opening for a security breach to happen. Due to this, a life time limit will be implemented on the containers to automatically run a stop command to ensure that the containers do not run for extended periods of time. This will help us cleanup our resources along with limiting the potentials for misuse.

9.4 Incoming Requests

Arbitrary command execution vulnerabilities are present for a number of selected target vulnerable webapps.

Suppose for a case example, the Damn Vulnerable Web Application (DVWA) selected by the team. DVWA includes a PHP remote code execution lab. Since arbitrary command execution is built-in to the target webapp, the slightest misconfiguration of the underlying Docker container will compromise the server.

Example exploitation steps:

1. Misconfigure Docker containers. In this example, a user uses a DVWA compose file with `privileged: true` and `pid: "host"` configured for each container

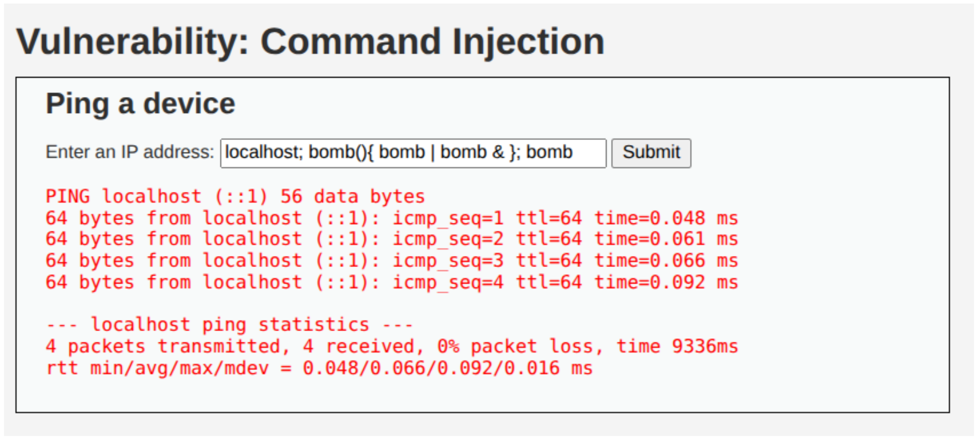
2. User starts DVWA with `docker compose up` with root privileges

3. User configures DVWA built-in security level to low using environment variables, config file, or webpage

4. User accesses command injection vulnerability in browser at /vulnerabilities/exec

5. User enters `localhost; ls /dev` to verify injection vulnerability. Devices on the host are listed. In this instance, the shell runs as user www-data, meaning that exploitation possibilities are roughly the same as a non-root user on the host. In the event of a privilege escalation vulnerability, the user may gain root access

6. For this demonstration, the user runs a fork bomb (figure 2)



Vulnerability: Command Injection

Ping a device

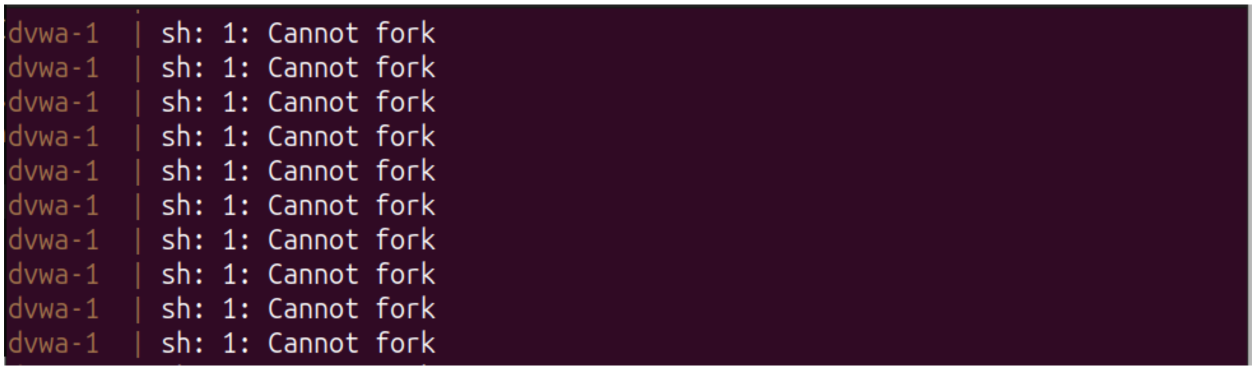
Enter an IP address: localhost; bomb(){ bomb | bomb & }; bomb [Submit]

```
PING localhost (::1) 56 data bytes
64 bytes from localhost (::1): icmp_seq=1 ttl=64 time=0.048 ms
64 bytes from localhost (::1): icmp_seq=2 ttl=64 time=0.061 ms
64 bytes from localhost (::1): icmp_seq=3 ttl=64 time=0.066 ms
64 bytes from localhost (::1): icmp_seq=4 ttl=64 time=0.092 ms

--- localhost ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 9336ms
rtt min/avg/max/mdev = 0.048/0.066/0.092/0.016 ms
```

Figure 2: Command injection - fork bomb setup

7. A significant slowdown occurs on the host system (figure 3). In principle, a user could perform this attack on the WADT site to take it offline



```
dvwa-1   | sh: 1: Cannot fork
dvwa-1   | sh: 1: Cannot fork
dvwa-1   | sh: 1: Cannot fork
dvwa-1   | sh: 1: Cannot fork
dvwa-1   | sh: 1: Cannot fork
dvwa-1   | sh: 1: Cannot fork
dvwa-1   | sh: 1: Cannot fork
dvwa-1   | sh: 1: Cannot fork
dvwa-1   | sh: 1: Cannot fork
dvwa-1   | sh: 1: Cannot fork
```

Figure 3: Command injection - fork bomb execution

9.5 Outgoing Requests

In addition to concerns that users could compromise the WADT site through any one of its containers, there is the additional concern that the containers could be manipulated to perform malicious outgoing requests. Containers, if compromised, could participate in DDoS, botnets, or be used to distribute illegal content, to name a few examples.

This problem is especially difficult to solve since blocking all outgoing requests will break any dependencies that target vulnerable webapps have on external services.

While a simple whitelist based on a search could whitelist each domain that a given target vulnerable app depends on, it would be easily exploitable by users. Given that users need write access to application source code in order to apply certain kinds of bugfixes, users would be able to add whatever external domains they wish, and therefore have them whitelisted.

As an example, the OWASP Juice Shop webapp makes external requests to CloudFlare to provide cookie consent forms. Disabling all outgoing requests and responses would break this dependency. In principle, individual apps cannot be relied on to provide an exhaustive list of the third-party domains they attempt to access. Whitelisting all domains that appear in a regex search has a few notable issues:

- Easily exploitable. Any form of write access to the source code of the application means full control of the whitelist
- Not dynamic. Any attempts to patch vulnerabilities by modifying source code which contacts third parties will be blocked until the whitelist is manually updated
- Not complete. Users may still make malicious requests to the domains that the application already requires

These problems are still an open issue for the team until a better solution is discovered.

9.6 Cross-Site Scripting (XSS)

XSS created problems for the original scope of WADT. These problems were discovered when researching target vulnerable webapps. OWASP hosts a Juice Shop instance that is shared between all users on the public internet. For this instance, all XSS challenges are disabled because of the potential consequences of XSS for users.

The Juice Shop project has taken the time already to detect container-based environments and disable the ”danger zone” challenges to prevent XSS attacks on other users. However, for other target projects, or for real vulnerabilities, the demarcation of vulnerabilities into ”safer” and ”dangerous” is not preemptively done. A full audit of each selected application of the WADT project is outside of the scope of Senior Design.

As a result of these constraints, the WADT project is limited to a single-user scope. In principle, XSS attacks on other users cannot occur if only one user is active on the system.

In this scenario, the user would have to deliver a malicious payload to the target vulnerable app that would harm their own machine. However, in practice, small groups of students are going to collaborate and be supervised by an instructor. The level of trust placed in the students with the potential for delivering malicious payloads to the target apps is similar to the level of trust to grant access to the original instance, meaning that this problem requires a social rather than technical solution.

9.7 2-Factor Authentication

In addition to previous security methods, it is desirable to add 2-factor authentication upon login. There are multiple approaches which solve the problem.

9.7.1 pyotp + pass + pass-otp

This solution implements TOTP (Time-based One Time Password) authentication, a form of 2-factor authentication, that relies on SSH access to the AWS instance which runs the site. This restriction ensures that the only users who can manipulate containers through the site *already* have full server access as a prerequisite.

The setup steps are as follows:

1. Use `pyotp` to generate a shared secret
2. Display the shared secret as a URI to the user on the WADT webpage. Other display options include QR code
3. User logs in to the instance over SSH
4. User uses `pass` with the `pass-otp` extension to store the URI

The usage steps are as follows:

1. User logs in to the instance over SSH
2. User runs `pass otp [secret-name]` to generate a one-time sign in code
3. User enters sign in code (or copies it) to the WADT site
4. User completes signin

While this solution is minimal, it relies on existing libraries and services for the core parts of TOTP, sidestepping documented issues with roll-your-own TOTP like floating point imprecision when calculating the time input.

This solution is also compatible with existing, more standardized authentication providers like Microsoft Authenticator, making it a sensible addition to the project.

All users who access WADT may be required to have SSH access to the underlying server as a prerequisite if it is ensured that only the `pass` program on the server is configured for MFA.

9.7.2 Traditional Authentication Providers

Traditional auth providers are available, most notably Microsoft Authenticator and Google Authenticator, that can provide the same service as `pass-otp` for users. However, since these traditional providers are not linked to the underlying hosting, they do not provide the same kind of security that the previous solution affords.

Traditional 2FA providers are primarily concerned with verifying the identity of users. Microsoft authenticator in and of itself will verify that the individual attempting a sign-on is the identity they claim to be. However, a solution with `pass-otp` also incidentally verifies that the individual attempting sign-on has access to the resources the WADT project is looking to protect.

9.8 Containerization Tools

9.8.1 OCI Compliance

The Open Container Initiative (OCI) maintains a group of three specifications which containerization tools are expected to follow to be compliant to the same standards as Docker. Since Docker was taken as a baseline technology for the project, only other OCI-compliant containerization tools were considered.

9.8.2 Podman

Podman is a drop-in replacement for Docker which is both daemonless and rootless. It differs in its underlying architecture: Podman is pod-based (see figure 4). A pod is an infra container combined with a number of regular containers.

For comparison, plain Docker functions interacts with plain containers, not pods. Docker Compose can be used to run multiple containers together for a project.

Podman's daemonless design means that it does not require a running background process (daemon) to function. Instead, Podman relies on systemd. This is in contrast to Docker, which requires a daemon with root privileges.

Podman is also rootless, meaning that system root privileges are not required by any part of Podman to run containers. This feature is the biggest draw of Podman over Docker for a security-focused project. With Podman, WADT would never require root privileges to run containers or be forced to interact with a privileged daemon process.

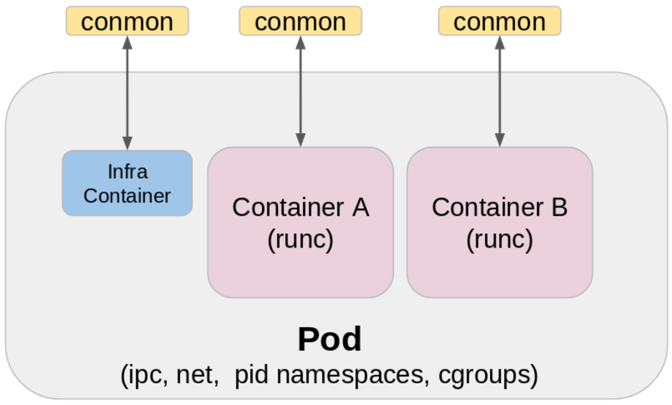


Figure 4: Pod architecture [6]

Although Podman is intended as a drop-in replacement for Docker, there are minor differences which ultimately led to the team's selection of Docker instead. Docker can perform pulls from DockerHub with short image names, while Podman requires additional configuration to do so. So while `docker pull postgres` will run successfully, `podman pull postgres` will throw an error.

These small differences are likely to create cognitive friction for users of WADT.

9.8.3 Docker

Docker is a widely-used set of container tools. A number of features made Docker the ideal choice for WADT.

- Official, extensive SDK with available low-level API access
- Images already available for numerous applications. While any OCI-compliant container engine can run these images, in practice they are built and tested for Docker
- DockerHub integration

9.9 Programming Language and Web Framework

9.9.1 Python and Flask

Flask is a micro web framework for Python. Since it's a micro framework, it lacks certain key features the team finds valuable for developing. Primarily, there is no built-in admin interface for interacting with the project's database given a model (it must be added using Flask Admin).

WebSockets are a key part of the project to provide live status updates for each container. Unfortunately, the support for WebSockets with Flask is relatively dated. The Flask-Websockets repository was archived last year, and the tutorial for Flask-SocketIO, an alternative library, was last updated in 2014.

The ecosystem of extensions to Django was found to be more actively developed. Since the project is timeboxed to a 9-month time frame, the "batteries included" features of Django were chosen over Flask.

9.9.2 Python and Django

Django is a web framework for Python which was chosen for its sensible architecture and useful features. Django's ORM and admin panel made configuring the database a one-day job, while its migrations feature ensures that any changes are tracked along with the project.

Django's MVC architecture and template system enforces clear separation of concerns and makes prototyping fast and easy. Combined with its powerful ORM, simple views can be implemented in a few lines of code (5).

```
def index(request):
    container_catalog = Container.objects.order_by("name")
    context = {"container_catalog": container_catalog}
    return render(request, "wadtapp/index.html", context)
```

Figure 5: Django queries and template system make simple pages simple to write.

9.10 Docker Integration

9.10.1 Overview

Docker integration lies at the core of the WADT project. Every feature from container deployment to logging, resetting, and shutting down a container relies on Docker's ability to build and manage isolated environments quickly and consistently. While previous sections discuss why Docker was chosen over alternatives like Podman, this section focuses on how Docker is practically integrated into our system architecture and development workflow.

At a high level, Docker serves as the layer that bridges our backend application logic and the intentionally vulnerable web applications being deployed for users. When a user interacts with the WADT frontend, such as clicking “Deploy,” “Stop,” or “Reset”, these requests travel through our Django-based API to the Docker engine via the python-on-whales library. The backend then performs the necessary Docker operations, returning live container states, logs, and connection details to the frontend. This structure allows us to abstract away all of Docker’s complexity while maintaining full control and observability over the containerized environments.

Docker integration most importantly ensures reproducibility. Each vulnerable web application is built from a deterministic Dockerfile and versioned image, which guarantees that students and instructors see identical environments across sessions. This is vital in cybersecurity labs where even small configuration differences could lead to inconsistent or misleading results. By leveraging Docker’s lightweight image system, we achieve repeatable deployments without sacrificing speed, isolation, or complicated library version overhead.

9.10.2 Integration with Backend (Django + Python-on-Whales)

The WADT backend, built with Django, uses the python-on-whales library to communicate directly with the Docker engine. Python-on-whales is a high-level, Pythonic interface to Docker that wraps the standard Docker CLI, allowing developers to invoke container operations using clean Python syntax while retaining access to the full range of Docker functionality.

When a user initiates a deployment from the frontend, Django processes the API request and uses python-on-whales to start a container from the appropriate image. Each container is instantiated from a predefined image stored locally on the AWS instance. Metadata such as the container ID, image name, network configuration, and timestamps are recorded in the project’s database, ensuring synchronization between Django’s logic layer and Docker’s runtime state.

One major reason for selecting python-on-whales over the traditional Docker SDK docker-py is its built-in thread safety and high-level abstractions. WADT often handles multiple container deployments simultaneously—one per vulnerable web application, and in the future, potentially one per user session. Python-on-whales manages Docker operations safely in multi-threaded or asynchronous contexts, preventing race conditions and socket contention that can occur when using the lower-level SDK. This makes it ideal for our multi-user educational environment, where concurrency is common.

The backend uses the library to manage all stages of the container lifecycle, including creation, execution, stopping, and log retrieval. Example commands include:

```
from python_on_whales import docker

# Create and start a container
container = docker.run("bkimminich/juice-shop", detach=True, name="student_1")

# Retrieve logs
logs = docker.logs(container)

# Stop and remove container
docker.stop(container)
docker.remove(container)
```

These operations are wrapped in Django view functions or background tasks, ensuring consistent control flow and error handling. Additionally, since python-on-whales mirrors the actual Docker CLI syntax, developers can more easily reason Docker commands and debug them both in our local environment and in our production environment.

9.10.3 Security Considerations

Security within this integration layer is enforced by restricting which Docker operations Django is permitted to execute. Containers are never launched with privileged flags or mounted host directories, and Docker's network isolation ensures that containers cannot access the host or each other unless explicitly configured. This keeps intentionally vulnerable applications securely contained while preserving functionality for testing and instruction.

9.10.4 Container Lifecycle Management

Every container deployed through WADT follows a clearly defined lifecycle managed by the backend. Understanding this lifecycle is crucial for maintaining predictable behavior, preventing resource exhaustion, and ensuring a secure sandboxed experience for users.

1. **Creation:** When a user selects a vulnerable web application and clicks "Deploy," Django instructs Docker to create a new container from the corresponding image. Environment variables, network parameters, and resource limits (CPU, memory, bandwidth) are configured during this stage.

2. **Execution:** Once created, the container is started in detached mode. The user is provided with access details, such as connection URLs and exposed ports. Docker isolates the container using Linux namespaces and cgroups, ensuring strong separation from the host.

3. **Logging:** All container logs are streamed through the backend using the docker.logs() function. These logs are relayed to both student and instructor dashboards, allowing visibility into application behavior, debugging messages, and exploitation attempts.

4. **Timeout and Auto-Stop:** Containers are automatically stopped after a set period of inactivity (e.g., 5 hours). This prevents long-running containers from consuming excessive resources or remaining exposed unnecessarily. The backend periodically polls Docker for container statuses to enforce these limits.

5. **Reset and Cleanup:** Users can reset their environment at any time, triggering Django to stop and remove the existing container and redeploy a clean instance. When containers are stopped or deleted, associated volumes and networks are also removed to avoid leftover data or stale configurations.

This lifecycle model ensures reliability and safety. It keeps container environments limited in lifespan, prevents resource leaks, and allows users to repeatedly practice deployments and exploitations in clean, reproducible environments.

9.10.5 Future Improvements and Stretch Goals

While Docker currently provides the foundation for container management in WADT, several future enhancements could further increase performance, scalability, and flexibility.

A major planned improvement is the implementation of per-user dynamic container allocation. Currently, containers are managed centrally, but expanding this to associate containers with specific user sessions will allow simultaneous isolated environments for multiple students. Python-on-Whales concurrency support and thread-safe design make this a feasible upgrade path.

Another enhancement would involve adopting Docker Compose or Kubernetes for orchestration. Docker Compose could simplify management of multi-container environments, while Kubernetes would enable automatic scaling, health monitoring, and load balancing—features especially useful for larger educational deployments. Since python-on-whales supports Compose commands directly, this transition could be made incrementally without rewriting backend logic.

Finally, the team could also automate image rebuilding and vulnerability patching. A nightly rebuild process could use docker.build() to refresh base images, integrate new security updates, and revalidate application configurations. Combined with continuous secret scanning and monitoring, these improvements will make WADT more robust and sustainable as an educational security platform.

We also could host a wider array and wider number of vulnerable web applications by automating the Dockerfile creation process. By programmatically generating Dockerfiles from application metadata, we could rapidly expand the catalog of available targets without manual intervention. This would greatly enhance the learning opportunities for students and ease the work of whoever wants to contribute a vulnerable web application to our platform.

In summary, Docker integration—implemented through python-on-whales is central to WADT's architecture. It enables safe, concurrent, and repeatable container management while providing a clear, maintainable Python interface for future growth.

9.11 python-on-whales

Python-on-Whales is a clean, one-to-one bridge between the Docker command-line interface (CLI) and Python. It acts as a direct wrapper, letting developers control Docker with familiar Python syntax instead of juggling terminal commands. In short — if you know Python, you already know how to use Python-on-Whales.

What makes it stand out is its balance between simplicity and full Docker coverage. It mirrors nearly every Docker feature while offering Pythonic conveniences and type safety.

Some of its main features include:

- Support for the latest docker features
- Support for Docker stack, services, and Swarm
- Progress bars and progressive outputs when like: pulling, pushing, etc
- Support for other CLI commands such as `docker cp`
- SSH support for remote daemons.
- Contains a fully typed API
- All docker objects and the docker client are safe to use with multithreading and multiprocessing

Python-on-Whales basically turns docker automation into something readable, testable, and smooth to integrate with larger python projects, no weird bash script required. More details from python-on-whales can be found on their official website [5].

9.12 Go, and Go-sdk

Go is an open-source programming language developed by Google, built for efficiency in cloud computing, networking, and large-scale distributed systems. It's commonly used for developing command-line interfaces (CLIs), backend web services, and site reliability tools.

AWS SDK for Go (Go-SDK) is a software development kit that integrates Go applications with Amazon Web Services. It provides Go developers with the ability to interact directly with AWS APIs, making it useful for managing cloud resources and automating deployments.

Since our project uses AWS for hosting, we briefly considered using Go with the AWS SDK for backend operations. However, we ultimately decided against it, as Go wasn't part of our primary backend stack. Sticking with our existing setup allowed us to keep the development process simpler and avoid unnecessary integration overhead.

9.12.1 docker-py

`docker-py` has two problems that make it a poor choice for the project:

- Lack of built-in support for Docker Compose. This means additional libraries must be installed for a number of the selected target vulnerable apps
- Lack of guarantee of thread-safety for core data structures. The DockerClient class provided by `docker-py` is not documented as thread-safe, and appears to store mutable state that affects program execution. Compared to python-on-whales, which makes a guarantee of thread-safety for the core client data structure, docker-py requires more overhead to manage

9.13 Vulnerable Web Application Selection

9.13.1 Selection Criteria

The goal of the WADT project is to provide a catalog of intentionally and unintentionally vulnerable web applications that together expose students to a broad, representative set of real-world security issues. We are aiming to maximize educational value while remaining practical to deploy and maintain. We selected target applications according to the following criteria:

- **Vulnerability breadth:**
  Each application should expose multiple common vulnerability classes (for example, the OWASP Top 10) so students can practice exploiting and patching different kinds of flaws.

- **Technology diversity:**
  The set should span several technology stacks (e.g., JavaScript/Node, Python/Django, PHP, LAMP, PostgreSQL) to expose students to language and framework-specific security issues and tooling.

- **Educational maturity:**
  Preference for well-documented, actively maintained projects with clear lab-friendly features (hints, adjustable difficulty, reset functionality).

- **Container friendliness:**
  Applications must be reliably containerizable and suitable for running in isolated, resource-constrained environments.

- **Practical relevance:**
  The application scenarios should reflect realistic application classes (API-driven services, single-page apps, e-commerce platforms, CMS, legacy stacks) so students learn transferable skills.

9.13.2 Intentional vs. Unintentional Vulnerabilities

Intentional (Designed) Vulnerabilities

Intentional or designed vulnerable applications are created specifically to teach security concepts. They are instrumented to contain a wide variety of exploitable flaws, often include challenge scaffolding or hints, and are typically packaged to be easy to reset and reproduce. Examples in our catalog (e.g., OWASP Juice Shop, DVWA, PyGoat) fall into this category. These apps prioritize discoverability and repeatability of vulnerabilities so students can quickly attack and patch vulnerabilities to varying extents.

Unintentional (Real-world) Vulnerabilities

Unintentional vulnerabilities are real production defects discovered in mainstream software: they were not created for teaching but were found and disclosed (often with CVEs). Including intentionally insecure apps alongside examples of unintentional vulnerabilities (e.g., a historical path traversal or file upload RCE) gives students exposure to the kinds of high-impact bugs that affect real services. These examples teach vulnerability discovery techniques, incident response workflows (patching, CVE analysis, key rotation), and the ethical responsibilities of disclosure and remediation. Although these vulnerabilities may be harder to exploit or reset, they provide invaluable context for understanding the stakes of security in practice. Hints and path to reproduce vulnerabilities will be offered to aid students.

9.13.3 Selected Applications

9.13.4 OWASP Juice Shop

Overview and stack

OWASP Juice Shop is a modern intentionally insecure web application implemented with a Node.js/Express backend and an Angular frontend. It is designed as a single-page application that emulates contemporary web development patterns.

Educational value

Juice Shop covers a very wide set of vulnerabilities (many items from the OWASP Top 10) in a modern application context: authentication/authorization flaws, insecure direct object references, broken access control, XSS, CSRF, injection issues, and client-side security problems. Because it is SPA-based, it is particularly good for teaching students about the interaction between frontend frameworks and backend APIs, token storage, and client-side attack surfaces.

Deployment notes and Docker considerations

For WADT we will prefer pinned image tags or build images from a known commit to ensure reproducibility. Typical deployment will require exposing the application port and mapping any data volume used for persistence. Resource usage is moderate; containers may be run with constrained CPU and memory limits without impairing lab exercises.

Why chosen

Juice Shop provides a modern, realistic lab with a polished UI and a comprehensive set of built-in challenges. It is ideal for intermediate-to-advanced students who need practice against up-to-date web stacks.

9.13.5 PyGoat

Overview and stack

PyGoat is a Django-based intentionally vulnerable web application implemented in Python and Django. It is generally smaller in scope than other applications but focuses on server-side vulnerabilities common to Python web frameworks.

Educational value

PyGoat is useful for students who want to explore issues in Python/Django applications specifically: insecure ORM usage, improper authentication/authorization, template injection, and misconfigured middleware. Its smaller codebase makes it easier for students to read the source, trace control flow, and implement fixes.

Deployment notes and Docker considerations

PyGoat can be containerized with a lightweight image based on an official Python base image. The database backend for PyGoat is typically SQLite or a lightweight DB by default, but we may run it against PostgreSQL in a secondary container (or a single container with the DB bundled).

Why chosen

PyGoat focuses on server-side Python/Django security, offering a clear environment for students to practice code-level remediation and gain familiarity with a widely-used backend framework.

9.13.6 Damn Vulnerable Web Application (DVWA)

Overview and stack

DVWA is a classic PHP/MySQL vulnerable web application widely used in education. It intentionally exposes a set of vulnerabilities and allows the security level to be adjusted to tune difficulty.

Educational value

DVWA is particularly well-suited for beginners and intermediate students because exercises are simple to understand and reproduce: SQL injection, cross-site scripting (XSS), command injection, file inclusion, and authentication bypasses. The adjustable security level makes

DVWA a flexible tool for progressive lab assignments, starting from smaller exploits and advancing to realistic variations.

Deployment notes and Docker considerations

DVWA is typically deployed as a LAMP-style container or with separate containers for PHP and MySQL. Because it is lightweight, DVWA is easy to run on constrained hosts. We must ensure default credentials and debug features are documented and reset between sessions.

Why chosen

DVWA's long-standing presence in security education, its adjustable difficulty, and its simplicity make it an ideal entry point for students learning web vulnerabilities and exploitation basics.

9.13.7 Ghost CMS (Unintentionally Vulnerable Example)

Overview and stack

Ghost is a widely used open-source blogging and publishing platform built on Node.js. It represents a realistic production-grade content management system (CMS) that students might encounter in the wild.

Vulnerability detail

A historical path traversal vulnerability (CWE-22) was disclosed for Ghost (CVE-2023-32235). Affected versions prior to 5.42.1 allowed an attacker to read arbitrary files on the web server (for example, system configuration files) without authentication by exploiting an improper input path normalization issue.

Educational value

Including Ghost with this specific CVE gives students a real-world case study in how a seemingly small file-path handling bug can lead to critical information disclosure. Labs can cover:

- How path traversal vulnerabilities arise (unsafe file path concatenation, insufficient input validation).
- How to reproduce the issue in a controlled environment using crafted requests.

- Incident response steps: responsible disclosure, patching strategy, and post-exposure mitigation (rotate secrets, review logs).
- Differences between exploiting intentional educational vulnerabilities and analyzing disclosed CVEs in production software.

Deployment notes and Docker considerations

Because this is a production-grade application, resource and configuration considerations (storage for themes, filesystem access, and user content directories) must be handled carefully to avoid leaking host state. For lab purposes, the container will be run in an isolated network with seeded content and no external persistence unless explicitly required.

Why chosen

Ghost demonstrates a realistic, unintentional vulnerability in a contemporary platform and teaches practical incident-handling skills in addition to exploitation techniques.

9.13.8 PrestaShop (Unintentionally Vulnerable Example)

Overview and stack

PrestaShop is an open-source PHP e-commerce platform used for online stores. It is representative of a PHP-based web applications that manages business workflows.

Vulnerability detail

A high-severity vulnerability (CVE-2023-3882) affecting PrestaShop versions such as 8.1.1-apache was disclosed that allows arbitrary file write and potentially remote code execution. Typical exploitation involves a 'Zip Slip' or unsafe file upload path that permits an authenticated or misconfigured flow to write a webshell or malicious file. This enables arbitrary command execution on the server.

Educational value

PrestaShop with this CVE illustrates real-world risks in file handling and upload features on e-commerce platforms. Lab objectives include:

- Demonstrating how improper validation on file uploads and archive extraction can lead to arbitrary file write and RCE.

- Teaching safe file handling patterns (validate file names, use safe extraction libraries, enforce strict upload directories with no execute permissions).
- Practicing mitigation steps for production systems: patching, credential rotation, integrity checks, and rollbacks.

Deployment notes and Docker considerations

For lab reproducibility we will ensure database seeding is deterministic. Because this scenario can involve writeable filesystem areas, containers will be carefully sandboxed (no host mounts, strict filesystem permissions) to avoid any risk to the host.

Why chosen

PrestaShop is a high-impact, real-world example of how file handling vulnerabilities in feature-heavy web applications can lead to full server compromise. It complements our intentional-vulnerability apps by demonstrating production failures.

9.13.9 vAPI

Overview and stack vAPI (Vulnerable Adversely Programmed Interface) is a self-hosted lab that shows common API security issues from the OWASP Top 10. It uses PHP and MySQL, built with the Laravel framework. The project includes Postman collections for testing and supports Docker Compose and Helm for deployment.

Educational value vAPI helps users learn about real-world API vulnerabilities like broken access control, data leaks, and insecure input handling. Each issue is set up as a hands-on challenge, making it useful for students, ethical hackers, and security testers. Postman files make it easy to follow and repeat tests. Some vulnerabilities stimulated in vAPI are lack of resources, injection flaws, and broken authentication.

Deployment notes and Docker considerations vAPI comes with a Docker Compose setup that runs both the Laravel app and MySQL database. Users should set up the `.env` file with correct database settings and match passwords across files. Using volumes helps save logs and exploit data. After setup, the app runs at `http://localhost/vapi/`.

Why chosen vAPI was chosen because it focuses on API-specific security issues, which are common in modern apps. It's easy to deploy, well-documented, and recognized by the security community. It fits WADT's goal of offering practical, container-based labs for learning and testing.

9.13.10 NoSQL Injection Vulnerable App (NIVA)

Overview and stack NIVA is a simple web application built to demonstrate NoSQL injection vulnerabilities in MongoDB. It uses Java as the backend language and connects to a MongoDB database using the official driver. The app allows users to search for contact records by email, and includes both secure and insecure query implementations to show how different coding practices affect security.

Educational value This app is designed to help users understand how NoSQL injection works and why it matters. It shows how attackers can manipulate query inputs to access or modify data they shouldn't be able to. By comparing safe and unsafe query methods, learners can see the impact of poor input validation and insecure query construction. NIVA is useful for students, ethical hackers, and developers who want hands-on experience with NoSQL security issues. Its vulnerabilities are Data extraction, NoSQL Injection, and Privilege escalation.

Deployment notes and Docker considerations NIVA is easy to run using Docker. A prebuilt image is available on Docker Hub, and the app can be launched with a single command: `docker run -p 8080:8080 aabashkin/niva`. Once running, it's accessible at `http://localhost:8080`, where users can test both secure and insecure endpoints. No extra setup is required, but you can add volumes if you want to save logs or test data. The app runs as a standalone container, making it simple to reset and reuse in different testing scenarios.

Why chosen We chose NIVA because it focuses on NoSQL injection, which is a type of vulnerability that's becoming more common as NoSQL databases grow in popularity. The app is lightweight, easy to deploy, and clearly shows the difference between secure and insecure code. Its Docker compatibility and minimal setup make it a great fit for WADT's goal of providing practical, container-based labs for learning and experimentation.

9.13.11 VulnerableApp-facade

Overview and stack VulnerableApp-facade is a Java-based tool that helps manage and launch multiple vulnerable applications from one place. It's built using Spring Boot and provides a web interface and API to control apps like OWASP Juice Shop. Instead of running each app separately, this facade lets users start, stop, and switch between them easily. It acts like a central dashboard for security labs.

Educational value This app has vulnerabilities like SQL injection, weak login flows, and session handling. This tool is useful for learners, trainers, and testers who want to explore different types of web vulnerabilities without setting up each app manually. By combining several vulnerable apps into one interface, it can save time and helps users to focus on learning. It also supports tracking progress and organizing exercises, which is helpful in classroom or workshop settings. Users can compare how different apps handle similar vulnerabilities and learn from the differences.

Deployment notes and Docker considerations VulnerableApp-facade

supports Docker and includes a Docker Compose file to run the facade along with selected vulnerable apps. Users can choose which apps to enable by editing the configuration file. The default port is `9090`, and the web interface is available at `http://localhost:9090`. The setup is straightforward, and no customization is needed to get started.

Why chosen We chose VulnerableApp-facade because it makes working with multiple vulnerable apps much easier. Instead of setting up each app manually, users can launch everything from one place. It's simple to use, works well with Docker, and supports a wide range of popular security labs. This makes it a great fit for WADT's goal of building a flexible and user-friendly environment for learning, testing, and teaching web application security.

9.13.12 BodgeIt Store

Overview and stack The BodgeIt Store is a purposely insecure web application created to help people learn about web security. It's written in Java and uses JSP (JavaServer Pages) for the frontend. The app runs on a servlet container like Apache Tomcat and uses an in-memory database, which means it doesn't need any external database setup. It simulates a basic online store with features like user login, product listings, and a shopping cart. These features are intentionally built with security flaws to make them easier to test and exploit.

Educational value BodgeIt Store includes common web vulnerabilities like SQL injection, XSS, CSRF, and IDOR, all exposed by default for testing. This app is great for beginners who want to practice finding and understanding common web vulnerabilities. It includes examples of issues like cross-site scripting (XSS), SQL injection, cross-site request forgery

(CSRF), and insecure direct object references. There are also hidden pages and functions that can be discovered through testing. One helpful feature is the scoring page, which shows which vulnerabilities have been found, making it useful for self-paced learning or classroom challenges. It's a good starting point for anyone new to ethical hacking or penetration testing.

Deployment notes and Docker considerations It is easy to run using Docker, which makes the app simple to set up and reset. A prebuilt Docker image is available on Docker Hub. To start the app, users can run the command:

`docker run -p 9090:8080 psiinon/bodgeit`.

After that, the app will be available at `http://localhost:9090/bodgeit`. Since it uses an in-memory database, all data is lost when the container stops, which is useful for resetting the environment between tests. If needed, users can attach volumes to store logs or other data. No extra configuration is required, making it quick to get started.

Why chosen We chose the BodgeIt Store because it's simple and easy to use, especially for people who are just starting to learn about web security. Even though it's no longer actively updated, it still works well and covers many of the most common web vulnerabilities. Its Docker support makes it easy to include in WADT's container lab environment. Overall, it's a usable tool for teaching and practicing basic security testing skills.

9.13.13 Damn Vulnerable Restaurant

Overview and stack Damn Vulnerable RESTaurant is a Python-based API training app built with FastAPI and PostgreSQL. It includes REST and GraphQL endpoints and simulates a restaurant system with user roles and data access.

Educational value The app teaches API security through hands-on challenges. Users explore vulnerabilities like broken authentication, insecure access control, and data exposure while progressing through game-like levels. It also has vulnerabilities such as Broken Function Level Authorization (BFLA), Broken Object Level Authorization (BOLA), and Server-Side Request Forgery (SSRF).

Deployment notes and Docker considerations It supports Docker Compose and includes scripts for developer and hacker modes. The API runs on port `8091`, with Swagger and Redoc docs available. Data persists across sessions, and volumes can be added for logging.

Why chosen We chose this app for its modern stack, interactive design, and strong focus on API-specific flaws. It fits WADT's goal of providing practical, containerized labs for security learning.

9.13.14 Ghost Kit (Unintentionally Vulnerable Example)

Overview and stack

Ghost Kit is a WordPress plugin that extends the block editor with page builder blocks, motion effects, and layout tools. WordPress uses many third-party plugins to add features, and since WordPress runs much of the web, any plugin vulnerability can affect a lot of sites. Ghost Kit integrates directly into the PHP based WordPress stack, running on PHP/MySQL, and is commonly deployed in production sites for design flexibility.

Vulnerability detail

A medium severity vulnerability (CVE-2025-9992) was disclosed by Wordfence in August 2025. Versions up to and including 3.4.13 are affected. The flaw lies in the icon block title field, which fails to sanitize user input and does not escape output. As a result, authenticated users with Contributor or higher privileges can inject arbitrary JavaScript into posts. When these posts are published or viewed, the malicious script executes in the browser of any visitor. This is a classic Stored Cross Site Scripting (CWE-79) scenario. Exploitation requires only Contributor access, which is often granted to content creators, making the attack realistic in multi author environments.

Educational value

Ghost Kit illustrates how seemingly harmless design features can introduce serious security risks. Lab objectives include:

- Demonstrating how stored XSS works in practice: a Contributor submits a payload, an Admin publishes it, and the script executes.
- Teaching secure coding practices for plugin developers: always sanitize input, escape output, and enforce strict role based permissions.
- Practicing mitigation steps: updating to Ghost Kit up to and including 3.4.14, auditing user roles, and deploying a Web Application Firewall to block malicious payloads.
- Highlighting the importance of plugin patch management in WordPress ecosystems, where third party code is often trusted without review.

Deployment notes and Docker considerations

For reproducibility, the lab uses a WordPress container seeded with Ghost Kit. A Contributor account is created to inject the payload, while an Admin account publishes the post. The Dockerfile installs required PHP extensions, which include mySQL and soap, and copies the vulnerable plugin into the container. Containers are sandboxed with no host mounts and strict file permissions to isolate the exploit. This ensures that the stored XSS remains contained within the lab environment and cannot affect the host system.

Why chosen

Ghost Kit is a strong unintentionally vulnerable example because it demonstrates how real world plugin flaws can compromise production systems. Ghost Kit was not designed for training due to its vulnerability being discovered in the wild. This makes it valuable for teaching students and practitioners how to handle actual plugin security failures.

9.13.15 Deployment and catalog-level considerations

For each selected application, we plan to:

- **Prefer reproducible images:**
  Use images from known commits to ensure identical lab environments across sessions.
- **Use ephemeral containers by default:**
  Configure deployments so containers can be reset or destroyed and recreated from a clean image to avoid state drift between student sessions.
- **Manage persistence intentionally:**
  When persistent state is required (for example, seeded content in Ghost or database state for PrestaShop), provide documented seed fixtures and automated reset scripts.
- **Resource planning:**
  Assign resource limits (CPU, memory) to each container type and test concurrent runs to validate the AWS instance sizing for class workloads.
- **Documentation and lab guides:**
  Provide short, focused lab instructions and expected learning outcomes for each app so instructors can map exercises to teaching objectives.

9.13.16 Summary

Our chosen catalog (OWASP Juice Shop, PyGoat, DVWA, Ghost CMS, and PrestaShop) provides a balanced mix of intentionally crafted educational targets and real-world, unintentional vulnerabilities. This mixed approach allows students to practice both guided exploitation and code-level fixes in pedagogical environments, while also learning to analyze, reproduce, and respond to actual CVEs found in production software. The container-based WADT architecture and the python-on-whales backed orchestration are well-suited to host this catalog in a reproducible, isolated, and instructor-friendly manner.

9.14 Asynchronous Support

9.14.1 Celery

Celery is a task queue that's compatible with Django. While it is capable of efficiently handling the asynchronous message-passing needed to send real-time status messages for each container, it involves many features that aren't needed for the project.

Celery requires the choice of a message transport (another dependency and another source of complexity). Its performance features are excellent, even beyond the scope of the MVP. ”Millions of tasks a minute” [4] are not likely to happen within the project scope.

Due to the high complexity associated with a sophisticated solution for concurrent computation, the team opted for a simpler solution, Django Channels.

9.15 Django Channels

Django Channels is a project that will take Django and extend its abilities beyond HTTP so it may handle websockets, chat protocols, IoT protocols, and more. Django channels comprise of multiple packages:

- Channels: the Django integration layer
- Daphne: the HTTP and Websocket termination server
- asgierf: the base ASGI library, and channels redis: the Redis channel layer backend.
- channels-redis: a Redis-based backend for message passing

In our project, Django Channels is used to eliminate the need for the client to repeatedly poll the server for updates. Instead, it keeps a live connection open, enabling real-time, two-way communication through WebSockets. Rather than following the traditional request-response cycle — where the client constantly sends “any updates?” messages — Django

Channels allows the server to push updates to the client as they happen. This drastically cuts down on network overhead and server load, since the client only receives data when something actually changes.

By maintaining an open, event-driven connection, Django Channels creates a faster, more responsive interaction between the client and server. The result is smoother, real-time feedback and an overall better user experience.

9.16 Backend Architecture

Django makes use of a Model, View, Template Architecture. A model is an abstraction over a database. A view performs an action. A template is a skeleton of a page which can be filled out with data.

A Django application is almost always interacting with the user by running view functions/methods in response to browser requests. These views typically render templates by supplying data to them, often using models to parse raw data.

Django encourages developing views (pages) that render a template by supplying it with data. The view behaves as a bridge between data (represented by models) and templates which dictate the look and feel of a page.

A key limitation of the MVT architecture with Django is the relative difficulty of integrating useful Django idioms with the React frontend library. Both tools aim to accomplish the same tasks in different ways, and may clash in ways that must be resolved on a per-project basis.

9.16.1 Django MVT

It is possible to develop frontend pages using React and serve the resulting files with Django. This approach places Django first, React second, as ultimately the Django parts of the application are responsible for routing and serving static files.

As a result, the team can still use the MVT architecture to design pages, while making a case-by-case decision for each page if React is needed. Simple pages can make use of class-based views to remain simple, while complex components can be handled in an appropriate React component rather than trying to push the limits of Django templates. It's the best of both worlds. Or is it?

The issue is that Django is not able to serve JavaScript source files as written. The React parts of the application must first be bundled. This entails new development steps and new sources of bugs.

Another issue: bundled assets tend to be quite heavy. As you might expect, keeping source code alongside its dependencies results in large bundle sizes by default. While there are standardized tools for fixing this issue, such as the webpack bundle analyzer, the team did not want to introduce large bundle sizes.

Since React was an add-on to the initial sponsor provided scope, if its introduction resulted in the necessity of serving bundled static JavaScript files of multiple megabytes, we may violate the requirement of efficiency.

In summary, while the conventions of the MVT architecture enforce a clear separation of concerns, and therefore good design, disregarding these conventions on a case-by-case basis and opting for React instead effectively nullifies the benefits of consistency and clarity that were sought.

The further consequences of a more complex build pipeline and the potential for huge assets led to the team opting to develop a more restricted Django application, as described next.

9.16.2 API-only

Rather than attempt to use the full MVT design of Django, it is possible to limit its responsibilities in the application to more traditional REST API scope. A series of endpoints that send/receive JSON can be written using the Django REST Framework.

Under this design, the frontend and backend are almost completely decoupled. While this makes the codebase easier to understand, it also leads to an evil of writing software: unnecessary duplication and reinventing the wheel.

Consider the container catalog page of WADT as an example. This page simply lists out the available containers for a user to select. You may imagine now that it is just an unordered list of hyperlink text. Under one possible scheme of implementation (figure 6), you are completely correct.

However, under the API-only design, the backend could not be responsible for rendering finished HTML. Instead, the backend view would essentially return the results of a database query as JSON.

Conceptually, this design change means that portions of frontend code need to parse and display data, repeating the work performed by Django's built-in template system. Django interacts with data through the model abstraction, hence why data is accessed using Python attribute syntax (`container.name`). Under the REST API design, frontend JavaScript code would first have to treat data as a series of bytes sent over a network connection, then parse it into JSON, then convert into native programming language objects.

```
{% if container_catalog %}
<ol class="catalog">
        {% for container in container_catalog %}
      <li class="entry">
        <h2><a href="{% url 'workbench' container.id %}">
          {{ container.name }}</a></h2>
        <p class="status">Status: <span class="indicator">
          {{ container.status }}</span></p>
        <p class="description">{{ container.description }}</p>
      </li>
        {% endfor %}
</ol>
```

Figure 6: Excerpt of a container list page developed using Django template language

9.16.3 Why Django REST Framework? (DRF)

These flaws are made up for by the batteries included style of DRF. Sure, it may be necessary to create boilerplate JavaScript to take this approach (or far more likely, generate it with an LLM and subject it to human review and testing). But these compromises wash away when DRF provides features that make it much faster to develop and communicate.

What follows is a summary of the benefits of DRF for the project.

- Web Browsable API. DRF ensures that once you've written an endpoint, its spec is converted into a documentation page. The work of the backend team then reduces to adding intuitive, human-readable descriptions of endpoints, saving hours and hours and hours of time and repetitive stress injury.

- Superset of Django. Inter-operates with Django Channels (another key component of the project), and all standard Django features are available since functional views are still accessible.

- Viewsets. DRF allows the backend team to create classes that descend from a DRF-provided template and inherit useful features. See figure 7 for a code excerpt that demonstrates why viewsets save time. Methods to list containers and access a specific container are inherited, and all that is required further is methods to perform project-specific actions related to each container.

- Mixins allow for the creation of reusable viewset behavior. As a project-specific example, the container viewset can be refactored to use mixins which are also inherited by the viewset for log entries. Under this scheme, CRUD behavior for each is defined in a single location, allowing for easier debugging.

- Serializers automate the conversion of models into JSON. DRF allows for the creation of custom serializers that decouple serialization from model spec and usage. In figure 7, the container viewset makes use of a ContainerSerializer class written for the project.

```
class ContainerViewSet(viewsets.ModelViewSet):
    queryset = Container.objects.all()
    serializer_class = ContainerSerializer

    @action(methods=["get"], detail=True)
    def deploy(self, request, pk=None):
        db_container = Container.objects.get(pk=pk)
        image = db_container.docker_url
        real_container = docker.run(image, detach=True)
        db_container.docker_id = real_container.id
        db_container.docker_name = real_container.name
        db_container.save()
        ...
```

Figure 7: DRF viewsets are batteries-included, but still extendable for WADT's use case

9.17 API Testing

API testing is an essential step in the development process. API testing serves multiple functions, and through testing will be a necessity throughout the development process. The most relevant types of testing for WADT are:

- Functionality Testing. This is the most important function of API testing. It answers the simple question, Does the API work as intended? For our project, this includes verifying that container deployment, stopping, restarting, resetting, and logging functions work correctly and return the expected information.

- Security Testing. Given that our application manages containers running intentionally vulnerable web applications, security testing is crucial. We need to ensure that the API itself cannot be exploited to bypass container restrictions and compromise our AWS server

- Interoperability Testing. Our API communicates with Docker through the Python Docker SDK. If there is a mismatch between SDK versions or Docker versions, this could cause errors when attempting to manage containers. It is important that we thoroughly test this to make sure these interactions work as expected.

It is important that we use the right technologies for such an important step in development. The two technologies we considered for API testing are cURL and Postman. Both come with their own benefits and disadvantages.

9.17.1 Postman

Postman is a popular API testing tool with one of its main selling points being its graphical user interface and collaboration tools. The interface simplifies the process of creating, sending, and managing API requests. This makes it especially useful for our projects' collaborative development.

Postman's documentation overview includes resources describing all of the functionalities of the software [7]. The functions that are naoteable for use in our projectinclude:

- User-Friendly Interface: Postman has its own application with a straightfoward interface, making it easy to quickly set up and execute API tests without extensive configuration.
- Collaborative Workspaces: Postman offers shared workspaces where tests can be easily shared with other team members, ensuring consistent and repeatable testing across the development team.
- Scripted Testing with JavaScript: Postman allows automated test scripts to be written in JavaScript, enabling the creation of reusable testing routines and smoother integration.
- CI/CD Pipeline Integration: Postman tests can be run within CI/CD pipelines, allowing for automated testing as part of the deployment process.
- Cost: The basic version of Postman is free to use many helpful features to suit our current testing needs [8].

9.17.2 cURL

cURL is another tool used for testing REST APIs. Unlike Postman, which provides a graphical user interface, cURL operates entirely from the command line. This makes it better for scripting and automating tests.

cURL provides comprehensive documentation for all of its features and capabilities on their online book [9]. The most appealing aspects of working cURL in our situation include:

- No setup required or cost. cURL is preinstalled on most UNIX-based systems, so there is no need for additional configuration prior to testing.
- More versatility. It supports protocols beyond HTTP and HTTPS, such as FTP, FTPS, SCP, and SFTP. However, these are not likely to be needed for our application.
- Efficiency Using the command line can be faster than using an interface, particularly for quick testing or repeated API calls.
- Automation and scripting. cURL can easily be integrated into shell scripts and CI/CD pipelines to more efficiently confirm that our Docker containers and backend APIs are functioning properly.

9.17.3 Summary: cURL and Postman in Our Testing Workflow

Overall, cURL complements Postman well. While Postman is best at collaborative testing and visualization, cURL is better suited for automation of tests as well as integration with the CI/CD pipeline. Together, they provide more flexibility for our testing purposes. It is possible that we use both, however, we will primarily use cURL, as it better integrates with other API documentation tools.

9.18 API Documentation

API documentation is often an overlooked aspect of software development, yet it is one of the most important. Thorough and accurate documentation provides clear communication between the API development team and anyone who needs to interact with the API. This includes front-end developers, testers, and even future developers so they can properly understand the API functions. Without proper documentation, using an API can feel like trying to build a piece of furniture out of the without an instruction manual. While it is technically possible, it can be slow, confusing, and prone to error.

Comprehensive API documentation saves time and preserves sanity, by providing a formal description of how each endpoint works, what data it expects, and what responses it returns. This is especially important for complex APIs, where multiple endpoints interact with each other. For our project, maintaining detailed API documentation will also make it easier for future teams to continue development and troubleshoot any issues in a more efficient way.

We considered two tools for documenting our APIs, Hopscotch and SwaggerHub. Both providesda standardized way to describe REST APIs. However, they differ in cost, accessibility, and integration capabilities within our existing testing structure.

9.18.1 Hoppscotch

Hoppscotch is a free and open-source platform for testing and documenting APIs [2]. It is entirely browser-based, making it easy for all team members to access without any need for additional installation. One of Hoppscotch's biggest advantages for our project is its ability to integrate with our API testing tools such as cURL.

As we test our APIs, Hoppscotch will allow us to document each endpoint directly within the same environment, which can be very convenient. Hoppscotch can also generate cURL commands for each request. This makes it simple to copy those commands into scripts for automated or mode advanced testing in the future. This allows for the ability to keep our documentation and testing closely aligned throughout the development process, rather than writing APIs and documenting at the end. Because cURL can be used in our CI/CD pipeline, this integration also helps to ensure that what we have documented remains consistent with what is being actively tested.

Endpoints can also be grouped into collections which help to organize the different APIs, rather than having one large list of different endpoints. Any documentation, alongside any additional notes, can be easily shared with its Share feature.

9.18.2 SwaggerHub

SwaggerHub is well-known platform for designing and documenting APIs, which incorporates the OpenAPI Specification or OAS. This specification provides a standardized format for describing REST APIs. These descriptions are defined within a YAML or JSON document and are used to define all aspects of the API, such as the endpoints and their corresponding response formats, status codes and example payloads. The goal of these OAS schemas is to provide a structure that can clearly be understood by anyone reading it [10].

SwaggerHub offers additional advanced features such as version control, various collaboration tools, and even automatic documentation generation. It can also be integrated with CI/CD tools to keep API documentation in synch with live deployments.

While SwaggerHub does offer many helpful features, it does have a significant drawback, its high cost. While a 30-day free trial is available, the Team plan, which includes the collaboration features, costs $29 per month [11].

9.18.3 HoppScotch or SwaggerHub?

Both Hoppscotch and SwaggerHub provide suitable options for our API documentation needs. While SwaggerHub offers more professional-grade collaboration and versioning tools, it also comes with a high monthly cost. This is a long-term project, so we would be required to maintain documentation well beyond the free trial period and thus would have to pay $29 per month for the Team package. This high cost makes SwaggerHub not feasible in the long term, as we are college students trying to avoid unnecessarily high out of pocket costs.

Hoppscotch by contrast, is free, yet it still provides the tools we need to properly document our APIs. It also can integrate directly with cURL, allowing us to import cURL commands directly into Hoppscotch. Additionally, it supports real-time documentation alongside testing. This makes Hoppscotch the more practical tool for our API documentation.

9.19 API Performance Testing

Efficiency is paramount to the WADT project. The team aims to optimize these metrics:

- Response time. While the Docker Engine often takes a significant amount of time to complete requests, we must ensure that users experience a sub-200ms response time. Workarounds may be developed to provide preliminary responses before the Docker Engine completes its work, in an async fashion, so that the site isn't perceived as slow due to its dependency on Docker.
- Resource usage. Network usage is especially important to measure and minimize, since it is the most likely avenue for hosting bills to come from. Third-party requests sent by target vulnerable webapps must be monitored.
- Previous two metrics under concurrent user load. While the original scope of WADT allows for a single-user design, in all practicality it will be desirable to extend to multiple concurrent users in the future.

9.20 Implementing Webhooks

Webhooks are one-way calls that originate from the backend and notify another system when a specific event has occurred. They differ from traditional API requests in that they are push-based rather than pull-based—instead of the frontend constantly asking for updates, the backend automatically sends information when something changes. For instance, when a user makes a purchase on Amazon, an API call first records the transaction in the backend's database; once the payment is verified, a webhook is sent to confirm the successful purchase and update the user interface in real time.

In our project, webhooks serve as infrastructure-driven callbacks—functions triggered by the underlying container orchestration system. These callbacks are sent to our backend whenever key container events occur, such as when a container is created, becomes healthy, or is terminated. The frontend then receives the corresponding status updates through WebSockets or Server-Sent Events (SSE), ensuring the interface reflects container activity immediately without the need for repeated polling requests. Each webhook carries structured information relevant to the event (for example, the container name, event type, and associated identifiers), which the backend processes before relaying to the user interface.

Security is a central consideration in implementing webhooks. Every webhook request includes an HMAC (Hash-based Message Authentication Code) signature—a cryptographic hash that verifies the integrity and authenticity of the message. The payload also contains a timestamp, allowing the backend to reject outdated or replayed requests. All webhook traffic is transmitted over HTTPS to protect data in transit. Additionally, minimal error logs are maintained for debugging purposes, with sensitive or privileged information redacted to prevent accidental exposure.

Together, these mechanisms make webhooks a reliable and secure method for synchronizing system state across components. By leveraging webhooks alongside real-time channels like WebSockets, our frontend remains responsive and accurate, reflecting infrastructure changes the moment they occur while maintaining strong security guarantees.

9.21 Secrets Management

9.21.1 Overview

Secrets management refers to the secure handling of sensitive information such as API keys, passwords, encryption keys, and database credentials used throughout an application's infrastructure. Without proper management, secrets stored in code or config files can expose vulnerabilities to attackers. Proper secrets management minimizes these risks by ensuring that secrets are secure, access is restricted, and exposure is quickly detected and remediated.

In our project, secrets management is especially important because we host intentionally vulnerable applications on an Amazon Web Services (AWS) cloud environment. Any leaked credentials, tokens, or access keys could allow malicious users to gain control of containers, the host instance, or associated AWS resources. Therefore, we must adopt proactive scanning and detection measures as part of our security workflow. After all, if the whole point of this application is to teach students about vulnerable web applications, ours has to be secure.

9.21.2 TruffleHog

TruffleHog is an open-source security tool designed to detect secrets and credentials within source code repositories, config files, and commit histories. It scans files for high-entropy strings (values that look like random keys) and known credential patterns (such as AWS access keys, GitHub tokens, or private keys). It can be used locally or integrated into CI/CD pipelines to automatically flag and block commits that contain sensitive information. This allows our secrets management to be proactive rather than reactive. We want our application to be secure from the start, rather than trying to patch vulnerabilities after they are discovered.

TruffleHog's main features include:

- **Pattern-based and entropy-based scanning:** Detects both known secret formats and unknown high-entropy strings that might be secrets.
- **Version-history scanning:** Inspects the entire Git commit history, preventing exposure from older commits.
- **Extensive detector library:** Includes hundreds of predefined detectors for cloud providers, APIs, and services.
- **Integration support:** Can be run manually or automatically in CI/CD workflows to enforce security policies before deployment.

9.21.3 Importance for WADT

For the WADT project, TruffleHog helps prevent accidental credential leakage in our public or shared codebase. Because our team uses AWS for hosting and relies on Docker for containerization, we routinely work with environment variables, API keys, and access credentials that could cause significant damage if exposed.

Integrating TruffleHog allows us to:

- Automatically scan our Git repository before merges or deployments.
- Detect hard-coded credentials or tokens in Django configuration files, Dockerfiles, or environment files.
- Prevent compromised access to our AWS instance or container registry.
- Enforce best practices for secure development across the team.

9.21.4 Integration Plan

We plan to integrate TruffleHog as part of our CI/CD security checks:

1. **Local Pre-Commit Hook:** Developers will install a pre-commit hook that runs TruffleHog before code is committed. If any secrets are detected, the commit will be blocked until the secret is removed or replaced with an environment variable.

2. **Pipeline Integration:** TruffleHog will also run in our CI/CD pipeline to scan each pull request or branch before merging into main. This ensures that no secrets are introduced even if local checks are bypassed.

3. **Periodic Scans:** Scheduled scans of the repository history will be conducted to ensure that no historical commits contain exposed secrets. If any are found, they will be invalidated, rotated, and removed using Git history rewriting tools.

9.21.5 Best Practices and Maintenance

- All secrets (AWS credentials, Django secret keys, database passwords) will be stored in environment variables, never directly in code.
- A .env file will be used locally and excluded from version control through the use of a .gitignore file.
- Compromised keys will be immediately rotated, and TruffleHog scan reports will be reviewed regularly to verify compliance.
- Access to secrets will be limited to authorized team members only, following the principle of least privilege.

9.21.6 Summary

By integrating TruffleHog into our development workflow, we create a proactive defense layer against one of the most common and dangerous security oversights—hard-coded secrets. This integration not only protects our AWS infrastructure and project data but also helps enforce professional software-security practices within the development team.

9.22 CI/CD Pipeline

The WADT project is hosted on GitHub. The team will make use of GitHub Actions to automate deployment, testing, and security scanning. The intended components are as follows:

- TruffleHog (Security Scanning) step. Checkout the main branch and use TruffleHog to check for potential password or data leaks.
- Backend test step. Use libraries such as `pytest` with Django to run test suite and report coverage.

- Frontend test step. Report coverage.
- Build step for frontend and backend. Install dependencies and upload build artifacts before deployment.
- Deployment to AWS step. Run DB migrations, issue system commands, and give status update on deployment.

9.23 Chosen API Tools

For API implementation, some of the biggest considerations are what will potential students need to have access too and what will make this project the easiest to utilize. Along with this we also had to ensure that interfacing with Docker would be seamless to ensure that we could control the container instances while minimizing the potential risks of security issues.

Due to this the project was decided to run on Python as Docker SDK for python was the most feature-rich and had the best integration with the Docker engine. With the integration this project will have much easier implementation of the core operations needed to control the containers i.e. start, stop, logs, restart, etc.

For the framework some of the biggest concerns were ensuring that authentication, admin control and security all had proper components available, which is why the final decision was to use Django as our framework for the API. One of the biggest motivators was that Django already has built in authentication features such as password hashing and session managment via HTTP only cookies. This allows for users to have secure registration, login, and allows the application to have session management features including user identification without having the need to build up the authorization system from scratch.

Django also offers a built-in admin panel along with object relational mapper will help associate users with the containers they are utilizing, and manage user data leading to simpler database operations. We will also be implementing distinct roles for the users of the program and the admins to help ensure that users can only run commands that they are authorized to run.

REST architecture is also being used as it is the industry standard for web API which will allow us to update and rework the frontend without having to change the backend API logic. Another benefit to this is that the API remains stateless in regards to the dockerized containers. This means that rather than relying on old data that is cached, it will instead query the Docker engine in real time for status updates.

This can help us prevent possible errors when pulling containers statuses and checking to see if containers are having any runtime issues or aren't connecting properly, as if we were looking at older cached data we might not be getting an accurate picture of what's going on.

9.24 API Considerations

Some other considerations were the usage of web-sockets for our communication protocols as they would provide continuous streaming of live logs or container stats. This was also considered due to the concern of ”heisenbugs”, race conditions and other overhead related issues such as our client initialization with each user who may be using the application.

After research the decision to not utilize web-sockets was that the REST API would be able to handle this much better with less complications of writing and implementing these systems ourselves. Rather than getting live updates instead we will be using client sided polling of the endpoints to get the container statuses and logs. The research also provided insight on Docker itself. The realization was that rather than initializing multiple clients when we have multiple users, instead initializing a single shared client would be the best option. This is because Docker already uses different request threads which will help us prevent race conditions or any ”heisenbugs” that were anticipated at first.

Using this Docker client management we also avoid high overhead costs as the clients are only created once the app boots up, it is inherently designed to be shared across threads while keeping user data separate, and prevents the need for database space being taken.

10 Database Research

When choosing our database we have a few options:

- MongoDB
- MySQL
- MariaDB
- PostgreSQL

Since AWS hosting provides a full Linux VM for the project, under current scope restrictions, each database would be installed locally and the bits and bytes of our data would exist on the same hosting as our site code and web server. The potential for growth of log entries beyond available storage space is not considered during the testing and development phase. A later migration to an independent database service with more storage capabilities may be completed if it is determined that a pileup log entries will grow beyond current limits under normal usage.

10.1 MongoDB

MongoDB is a document-oriented NoSQL database that stores data in flexible, JSON-like objects called BSON. It's great when you need to move fast and don't want to commit to a strict table structure — data can evolve as the project does. MongoDB supports ACID transactions and comes with solid security features like role-based access control and built-in encryption. It can also scale horizontally, letting data spread across multiple servers for speed and redundancy.

For our project, though, MongoDB's flexibility was more than we needed. We wanted something with stronger relational consistency and a clear schema, since our data follows predictable relationships. MongoDB shines when the data structure constantly changes; ours doesn't. So we passed on it.

10.2 MySQL

MySQL is the classic open-source SQL database — structured, reliable, and familiar. It organizes data into tables, rows, and columns, all defined by schemas that make relationships clear and enforce consistency. It's been the backbone of web apps for decades, and for good reason.

That said, MySQL starts to stumble when handling massive datasets or tons of concurrent users. Stored procedures are clunky, and scaling beyond a single node takes work. It's good, but not great for what we're building. We needed something with stronger concurrency handling and more flexibility under pressure.

10.3 MariaDB

MariaDB is basically MySQL's smarter, faster cousin. It was forked from MySQL after Oracle bought it out, keeping full compatibility but improving performance and adding more modern features. It handles queries faster, offers more storage engines, and supports parallel replication for better scaling. It's a solid choice for web-based apps that expect high traffic.

We considered it seriously — it performs better than MySQL across the board — but PostgreSQL still beat it out in terms of handling complex queries and maintaining data integrity. MariaDB is fast, but PostgreSQL is deep.

10.4 PostgreSQL

PostgreSQL ended up being our pick. It's open-source, built for reliability, and supports everything from basic relational data to advanced types like JSONB and arrays. It follows SQL standards closely but still gives room for flexibility. Transactions are fully ACID-compliant, and its concurrency control (MVCC) keeps things smooth even when multiple operations hit at once.

Performance-wise, PostgreSQL can scale well and handle complex queries without breaking a sweat. It's secure by default, supports role-based permissions, and encrypts traffic between the server and clients. For our project — managing user data, container metadata, and system events — PostgreSQL just makes sense. It gives us structure without limiting what we can do later.

10.5 Container Identity Management

In multi-container environments, especially those used for modern web app deployment, it's crucial to identify and reference containers accurately. Whether you're orchestrating services, debugging issues, or automating tasks, knowing which container you're interacting with helps avoid confusion and ensures your tooling behaves predictably. Docker provides two unique identifiers for each container: the container ID and the container name. Both are guaranteed to be unique within a host system, but they serve different roles depending on how you're working.

- Container ID – a long, unique string assigned when the container is created. It never changes during the container's lifetime and is ideal for scripting, logging, and low-level operations where precision matters. For example, if you're writing a monitoring script or inspecting container metadata, using the container ID ensures you're targeting the exact instance—even if multiple containers share similar configurations.
- Container Name - Container Name is a human-readable label, either auto-generated by Docker or explicitly set by the user or orchestration tool. Names are easier to remember and are especially useful for interactive debugging, Docker Compose setups, and service-level commands. When working with Compose, names often reflect the service role, making it easier to run commands

10.6 Docker Compose

Docker Compose requires a different Python client library than the one typically used for managing single Docker containers. The standard Docker SDK for Python (docker-py) is designed to interact with individual containers—allowing developers to start, stop, inspect, and configure them directly. However, Compose introduces a higher level of orchestration, where multiple services are defined together in a docker-compose.yml file and managed as a unified application stack. This difference means that the same library used for single-container control cannot fully handle the complexity of Compose projects.

When working with Compose, the Python tooling must be able to interpret and act on project-level definitions rather than just container-level commands. This involves parsing YAML files to understand service configurations, networks, and volumes, and then coordinating the lifecycle of all services in the correct order. For example, a database service may need to be initialized before a web server can connect to it, and Compose ensures this dependency is respected. Libraries or wrappers that integrate with Compose are therefore essential for managing these workflows programmatically.

In the context of WADT, this distinction is especially important because vulnerable web applications often rely on multi-service stacks. A single vulnerable app might include a web frontend, a backend API, a database, and perhaps a proxy or load balancer. Managing these components individually would be cumbersome and error-prone, but Compose allows them to be launched, networked, and reset together. To support this, WADT must incorporate a separate library or API layer that can issue Compose commands such as docker compose up and docker compose down, ensuring that vulnerable environments can be deployed and torn down consistently.

Ultimately, WADT needs to support two layers of Docker control: one for single-container operations using the standard Docker SDK, and another for Compose-based orchestration using a dedicated library. By recognizing and implementing this dual approach, WADT can seamlessly manage both simple and complex application deployments. This ensures that vulnerable web applications built with Compose can be tested, reset, and documented without manual intervention, while still maintaining compatibility with single-container setups.

10.6.1 Benefits for Vulnerable App Deployment

Docker Compose is pretty much valuable when deploying vulnerable applications consisting of multiple interconnected components, such as a web server, a database, and a proxy. Compose makes it easy to spin up and manage lots of web apps, which helps us quickly test and reset vulnerable setups.

- Service grouping – Docker Compose allows us to define multiple services—such as a web frontend, backend API, database, and proxy—as part of a single configuration file, enabling them to be launched and managed together as a unified stack. This grouping is especially useful for vulnerable application setups that require coordinated startup and inter-service communication. Instead of manually starting each container and ensuring dependencies are met, Compose handles the orchestration automatically, ensuring that services come online in the correct order and are properly networked. This makes it much easier to deploy complex environments for testing, exploitation, and teardown, especially when working with multi-component apps that simulate real-world attack surfaces.

- Network isolation – Each Docker Compose project creates its own dedicated virtual network, which isolates its services from the host system and from other Compose projects or standalone containers. This isolation is critical when deploying vulnerable applications, as it prevents unintended interactions between services and ensures that exploits or payloads remain contained within the test environment. By using Compose's built-in networking, we can simulate realistic service-to-service communication while maintaining strict boundaries that protect the host and other running containers. This controlled setup supports safer and more predictable exploit testing, especially when dealing with network-based vulnerabilities or lateral movement scenarios.

- Volume sharing – Compose makes it easy to define shared volumes between services, allowing them to persist and exchange data across container restarts. This is particularly valuable for vulnerable app deployments, where we often need to collect logs, exploit payloads, or forensic traces without modifying the container internals. By mounting volumes, we can store evidence of exploitation, track changes over time, and preserve artifacts for analysis or reporting. These volumes also support reproducibility, as they allow us to reset containers while retaining critical data, making it easier to rerun tests or compare outcomes across different attack attempts.

- Environment reproducibility – The `docker-compose.yml` file acts as a portable blueprint for the entire application stack, capturing all service definitions, environment variables, ports, volumes, and dependencies in a single declarative format. This reproducibility is essential for vulnerable app testing, as it ensures that the same environment can be deployed consistently across different machines, operating systems, and CI/CD pipelines. Whether we're running tests locally, in a lab, or on a remote server, Compose guarantees that the setup behaves the same way every time. This consistency reduces debugging overhead, supports collaborative testing, and enables version-controlled infrastructure for security research and education.

- Simplified teardown and reset – One of the most practical advantages of Docker Compose is the ability to tear down and reset an entire environment with a single command. By running `docker compose down`, we can stop and remove all services, networks, and volumes associated with a project, restoring a clean slate for the next test run. This is especially helpful when working with vulnerable applications, where repeated exploitation and analysis require frequent resets. Compose's teardown capability ensures that no residual data or configuration persists between runs, allowing for reliable, repeatable testing and faster iteration during exploit development or forensic analysis.

10.6.2 Compose Workflow

Compose operates in three main stages:

- Definition – Services are declared in a YAML file, specifying images, ports, variables, volumes, and dependencies.

- Build and Up – The `docker compose up` command builds images and starts all services in the correct order.

- Teardown – The `docker compose down` command stops and removes containers, networks, and volumes, which allows to have a clean resets.

10.7 Docker Architecture

Docker gives a containerization framework that enables us to deploy vulnerable web applications with putting all their dependencies into isolated, secure environments. Most of the tools in Docker explains how we utilize them in our deployment pipeline.

10.7.1 Core Components

The Docker architecture consists of these key elements:

- Docker Engine – The runtime that builds and runs the containers.

- Images – Read-only files that contain everything needed to run an application. We create custom images using Dockerfiles and choose lightweight, secure base images to reduce risk.

- Containers – Containers are active instances of images. Each one runs in isolation using Linux namespaces or cgroups to safely execute and run vulnerable applications.

- Volumes – Volumes gives persistent storage for logs and payloads. We use them to examine attack traces without changing the container itself.

10.7.2 Benefits for Vulnerable App Deployment

The Docker architecture offers several advantages for deploying and managing intentionally vulnerable applications:

- Fast setup and teardown – Docker containers can be launched or removed in seconds, which dramatically speeds up the process of deploying and resetting vulnerable applications. This rapid lifecycle is ideal for exploit testing, where environments often need to be spun up, attacked, and torn down repeatedly. Instead of waiting for full system reboots or manual reconfiguration, testers can iterate quickly by deploying fresh instances, applying payloads, and resetting the state with minimal delay. This agility supports faster debugging, more efficient exploit development, and smoother transitions between test cases.

- Safe and isolated experimentation – Vulnerable applications deployed in Docker run inside sandboxed containers, which isolate them from the host system and other running services. This containment ensures that any malicious payloads, misconfigurations, or exploit attempts remain confined to the test environment, reducing the risk of unintended damage or data leakage. Researchers and students can safely explore attack vectors, privilege escalation paths, and post-exploitation techniques without compromising the integrity of their host machine or neighboring containers, making Docker a reliable platform for controlled security experimentation.

- Scalability for labs – Docker's lightweight architecture allows multiple instances of the same vulnerable application to be launched simultaneously, each with its own configuration, network, and volume. This scalability is especially useful in classroom or research lab settings, where parallel testing is needed across different scenarios or user groups. Instructors can deploy dozens of isolated environments for students, each tailored to specific learning objectives, while researchers can run comparative tests on different versions of an app or exploit. Docker's efficient resource usage makes this kind of horizontal scaling practical even on modest hardware.

- Cross-platform reproducibility – Docker abstracts away many platform-specific differences, ensuring that containerized applications behave consistently across Windows, macOS and Linux systems. This cross-platform reproducibility is essential for collaborative security testing, where team members may be working on different operating systems. By packaging vulnerable apps into containers, we eliminate environment drift and reduce platform-specific bugs, making it easier to share setups, reproduce results, and maintain infrastructure across diverse machines and deployment targets.

- Seamless CI/CD and cloud integration – Docker containers integrate smoothly with continuous integration and deployment pipelines, allowing vulnerable applications to be automatically built, tested, and deployed in cloud environments. This enables scalable exploit testing and automated regression analysis, where each code change or configuration tweak can trigger a fresh deployment and security check. Cloud platforms like AWS support container orchestration tools that make it easy to run large-scale vulnerability labs, simulate real-world attack surfaces, and collect telemetry across distributed environments.
- Resource control – Docker provides fine-grained controls over CPU, memory, and network usage, allowing testers to simulate constrained environments or prevent resource abuse during exploit runs. This is particularly useful when looking at denial-of-service vulnerabilities, sluggish performance, or privilege escalation techniques that rely on system exhaustion. By limiting container resources, we can mimic low-end devices, enforce realistic boundaries, and ensure that one test doesn't interfere with others running in parallel. These controls also help maintain stability and predictability in multi-user lab setups.

10.7.3 Security Isolation

Docker uses built-in Linux features to keep containers separate and safe. Namespaces and cgroups help in isolating processes and controlling resource usage. We also improve security by removing extra permissions and applying system call filters to block risky operations. This helps prevent containers from affecting the host system. Containers also run with their own network stack, which limits exposure to external threats unless explicitly configured.

10.7.4 Container Management Process

Our deployment pipeline manages every stage of container lifecycle with precision and reproducibility. It begins by building container images from well-structured Dockerfiles, which define the base environment, dependencies, and startup commands for each service. These images are then launched as containers using secure configurations—such as restricted network access, non-root users, and environment variables—to minimize exposure and support safe testing of vulnerable applications. To preserve critical data across test cycles, we mount volumes that retain logs, exploit payloads, and forensic traces, allowing for post-exploitation analysis without altering container internals. When a test run is complete, containers are stopped and removed cleanly, along with their associated networks and volumes, enabling fast resets and consistent re-deployment. This streamlined process ensures that each test begins with a fresh, isolated environment while maintaining traceability and control throughout the lifecycle.

11 Prototypes

11.1 ProtoWADT v0

The first prototype was created to explore controlling Docker containers programmatically and gain familiarity with Django. This prototype was timeboxed to a 1-day sprint to prohibit overanalysis and decision paralysis.

The scope was intentionally basic and focused on only programmatically controlling one container, instead of the usual 7-10. It ran local-only, depended on a specific vulnerable app (DVWA), and was intended to be thrown away.

ProtoWADT v0 was also a vital exploration of some of the invalid assumptions the team had made previously about our tech stack. Different libraries were depended on than originally predicted as a result of this exploration. This prototype also began the thinking which led to further design exploration and investigation of hidden complexity in the next prototype.

The prototype's definition of done required that at least the core states of Docker containers could be manipulated through API endpoints. Start, Restart, Logs, Stop, and Down were initially required.

A crucial oversight of this prototype was the lack of a database, which hid further problems with state and storage that were not discovered until the second prototype. The decision to store important data in memory for a local-only demo sidestepped issues of data corruption, handling asynchronous connections, and scaling that are discussed in more detail in 9.24 and 10.5.

11.1.1 Programmatic Control

The first goal of v0 was to programmatically control Docker containers with a suitable programming language. For Python, two main libraries exist that fit this purpose: python-on-whales and docker-py. Broadly, docker-py is intended for applications that run as a single container, and python-on-whales is intended for applications that run as a group of containers with Docker Compose as well as single-container applications. Since DVWA specifically uses Docker Compose to run multiple containers, python-on-whales was selected for the prototype.

First, a short demo was created that essentially listed and used the main parts of the python-on-whales API that the project would depend on. python-on-whales has essentially a one-to-one correspondence beteween its API and the Docker CLI command structure, meaning that finding these endpoints was often as easy as typing the command with Python syntax.

```
docker compose up --detach
```

Figure 8: `docker` command syntax

```
docker.compose.up(detach=True)
```

Figure 9: python-on-whales API

The key failures of this prototype related to the scope. A standard Python program has a synchronous top-down control flow, meaning that a given sequence of method calls to a single DockerClient object in memory will occur as expected. However, a web server aiming to provide the same service cannot use this model of execution.

Instead, the DockerClient object cannot be shared in memory between REST API endpoints without mechanisms for data persistence (serialization and storage in the session, cache, database, etc.) and data integrity (preventing race conditions or invalid states).

At a deeper level, HTTP is a stateless protocol. Specific to the project, shared state between requests needs to be managed by a separate system server-side. Sharing data between views as a global variable, such as providing a global DockerClient for all views to access, is not a sufficient solution due to the potential for data corruption.

A WSGI server typically assigns each request to a worker process or thread, meaning that sharing global variables in memory would break down upon deployment. While python-on-whales guarantees that their DockerClient object does not store any state that will affect program execution, docker-py does not make a similar guarantee. In the end, the potential for the "share global variables in memory" approach is bad design which may or may not work depending on the Docker library used.

11.1.2 Django

The second goal of v0 was to understand how Django is used to build webapps. A basic demo site was created with views to control the DVWA container. The completion criteria involved creating URLs for each Docker action and writing a view to achieve each. Completion was manually tested by monitoring container state with Docker Desktop while using the written views. Start, restart, stop, down, and logs views were required.

Each of these endpoints was simple due to the lack of complexity of the scope of the prototype. More complex requests for continuous information like container CPU usage were not considered. As a result, the prototype v0 didn't reveal much about how to write the views of our project, but did teach the team how to structure a Django project overall.

Gaps in the team's understanding, resulting from an insufficiently detailed prototype, were (partially) corrected by the second which was developed.

11.2 ProtoWADT v1

The next prototype iteration explored the use of Django REST Framework (DRF) to develop a REST API. Alternative models were experimented with (figure 10). The new, simplified model relies on an external Docker registry to pull images.

The simplified model incorporates insights gained from the first prototype and a better understanding of the divide in the project between persistent and temporary information.

11.2.1 Persistent Data

- A container's metadata (display name and description) are not likely to change for the duration of its existence on the WADT site.
- A container's registry URL is fixed for the duration of its lifespan. Changing the registry URL changes the container. Crucially, incorporating knowledge of an external registry into the database makes it easier for the Docker team to manage and debug images. Docker team need only concern themselves with modifying an image and updating it on an external registry, rather than pushing changes through idiosyncratic, project-specific means.

11.2.2 Temporary Data

- Docker container ID and Docker-provided name. These are auto-generated by the Docker Engine.

- State information. The containers must display status indications such as running or stopped. Since these values are relatively ephemeral, it doesn't make sense to store them in a database.
- Uptime. Container uptime is a continuously-changing value, and hence it doesn't make sense to store it in a database for the same reasons as state information.

```python
# Jimmy's Container
display_name = models.CharField(max_length=32)
# He wrote this in one day. #PHP #MySQL
description = models.CharField(max_length=256)

# 36fb332b39a7e05c666d46d7a04d392dcd3a2b880bd564225a4be27dbf092c3e
docker_id = models.CharField(max_length=64)
# jimmy-app
docker_name = models.CharField(max_length=64)
# docker.io/jimmy/jimmy-app
docker_url = models.CharField(max_length=256)
```

Figure 10: Alternative model for Container

WebSockets were first explored in this prototype as an option for providing a continuous stream of the temporary data to the frontend. Consumers for streaming log data and status information were created in the form of classes inheriting from Django Channels's provided AsyncWebsocketConsumer class. These consumers were able to implement a form of efficient, async-capable long polling that works for the project spec.

The primary design flaw of this prototype was a lack of clean templates. No React components were incorporated, and therefore the frontend code to handle WebSockets is bundled with standard Django templates in a vanilla JS script.

From this prototype and its results, the team established goals to find libraries to make frontend use of WebSockets easier and more convenient, enforce a sensible separation between DRF-provided viewsets and standard Django function-based views, and document and test the written API.

This prototype also revealed issues of container identity management, Compose interoperability, and the inclusion of source code for target vulnerable webapps. These design issues were clarified in a meeting with our sponsor, further refining the MVP and appropriately modifying its scope.

12 Frontend Research

12.1 Frontend Framework

12.1.1 React

The front-end of our application is built using React. React employs a virtual Document Object Model, or DOM, to more efficiently manage updates to the UI. Only the components that have changed are updated, rather than needing to re-render the entire page to make changes. This makes the application faster and more responsive, which is an important quality for any web application.

React provides several benefits that we found attractive [3]:

- Reusability: Components can be reused across different parts of the application, improving consistency and helping us avoid the DRY principle as much as possible. This will also make it easier for future development teams to extend or modify the project.
- Flexibility and Maintainability: Individual components can be updated or replaced without affecting the rest of the system, allowing for simpler debugging, testing, and any future changes.
- Organization and Readability: React promotes self-contained components, leading to more structured and readable code. This will make it much easier for future us to understand code if we need to refer to it at some point in the future.

12.1.2 Drawbacks of React

While this React offers many benefits, it does present some challenges that we have to consider. According to our research, large applications can become quite messy if the different components are not managed and organized carefully. However, given the relatively limited scope of our project and our attention to good design practices, we do not expect this to be a significant concern. Also, the emphasis on reusability can sometimes limit customization, but for our needs, we find the benefits of React far outweigh these potential drawbacks.

12.1.3 React-Bootstrap

In order to speed up and simplify the design and process, we incorporated React-Bootstrap, which is a library built on top of Bootstrap 5. Our sponsor is already familiar with Bootstrap, so adopting React-Bootstrap also gives us an additional source of guidance and expertise if needed during development.

12.1.4 JavaScript vs. Typescript with React

React primarily uses JavaScript, but it also supports TypeScript, which we chose for this project. TypeScript extends JavaScript by adding some additional support which offers several advantages.

- Compile time error checking. Catches errors at compile time rather than runtime, speeding up the process of finding and fixing bugs.
- Improved readability: Type annotations make the code easier to understand for both our team and others in the future.
- Easier maintainability: As the project grows, TypeScript will help maintain consistency across components and reduce the risk of introducing hard to find errors.

12.2 CSS Libraries

When it comes to frontend work, part of the problem is making sure that a website looks appealing enough to attract users, providing a more enjoyable coding experience for the developer and ensure pleasurable user experience.

With this in mind, we wanted to choose a professional CSS library that provides the user with good readability and accessibility. There were a lot of options for CSS libraries, but we narrowed the options down to three: Bootstrap, Tailwind and Chakra UI.

12.2.1 Bootstrap

This one was our first consideration, as bootstrap is a very popular CSS library. Bootstrap and REACT have their own merged CSS library that makes coding the project smoother called 'REACT-Bootstrap'. REACT-Bootstrap utilizes the bootstrap visuals with REACT type programming to ensure a coherent coding experience.

Bootstrap's reviews are overall positive, with extensive praise on the use of child themes. Its REACT-Bootstrap also gets pretty high reviews, being praised for it's developer friendly nature. The only major complaint on Bootstrap is it's lack of flexibility, requiring the developer to edit the CSS code themselves to achieve the desired look.

There are also custom themes for bootstrap to use that can be found online.

One example is 'Bootswatch', which provides free themes for bootstrap that can be easily implemented into the project. Utilizing these themes can help us achieve a more unique look for our project, and possibly implement different themes for the user to choose from.

12.2.2 Tailwind

Tailwind is another popular CSS library containing many good options for CSS design. Through research, we found that tailwind operated very differently from bootstrap, as it is a utility-first CSS library. This means that instead of having pre-designed components like bootstrap, tailwind gives a lot of freedom to the developer to design their own components using CSS.

Tailwind reviews are good, with 63% of the reviews on their website being 5 star. A complaint that we have noticed when it came to tailwind is that while it is versatile and useful, there is a learning curve. It suggested that those who don't have extensive knowledge on CSS will struggle a little to utilize tailwind's components.

Unlike Bootstrap which has it's own REACT library, tailwind does not have one. Instead, tailwind has it's own third party library called Tailwind UI that integrates tailwind with REACT. However, this third party library is locked behind a paywall, requiring the developer to pay to use it. Our research concludes that tailwind is extremely versatile for CSS design, however requires a learning curve.

12.2.3 Chakra UI

Chakra UI also came across our radar as a popular CSS library. On Chakra's website, there was a page that showcased the professional websites made using Chakra UI. This was a nice way to look at what a fully developed website using Chakra UI can do.

Chakra UI's reviews are mostly positive, with emphasis on good developer experience. A downside to Chakra UI is a lack of certain components (one example is the search bar component), as well as good ways to change the themes of the components.

Chakra UI is also developer friendly with Figma with their Chakra UI + Figma Kit. We found it interesting that there was a Figma page that showed us exactly what to use for certain components. This was a nice touch that made us consider Chakra UI more seriously since we used Figma for our design mockups.

12.2.4 What we chose

After weighing all considerations, our team selected **REACT-Bootstrap** as our primary CSS framework. It offers a familiar structure, excellent documentation, and seamless integration with React, enabling us to focus on functionality without getting bogged down in complex setup processes.

Additionally, having a developer with prior Bootstrap experience allowed us to accelerate onboarding and maintain consistency across our components. While Tailwind and Chakra

UI both provide unique advantages — creative freedom and modern theming, respectively — React-Bootstrap strikes the most effective balance between stability, usability, and developer productivity.

12.3 Themes

Since we decided to use React-Bootswatch for our CSS library, the goal was to find themes that actually fit our project's personality. We wanted a setup that not only looked clean but gave users the option to tweak the website's vibe to their liking. We ended up utilizing bootswatch, a theme library made for bootstrap and can be utilized with react-bootstrap. Bootswatch made that easy—it's built for Bootstrap, works smoothly with React-Bootstrap, and comes loaded with theme options we could build around.

Our design direction aimed to capture a sleek, "hacker-style" aesthetic—something that felt sharp and modern without overcomplicating the visuals. The frontend team tested several themes that balanced readability and contrast while keeping the interface lightweight.

Here are some theme designs that we decided on (note subject to change until due date)

- Bootswatch Cyborg (11):
- Bootswatch Lux (12):
- Bootswatch Zephyr (13):
- Bootswatch Pulse (14):

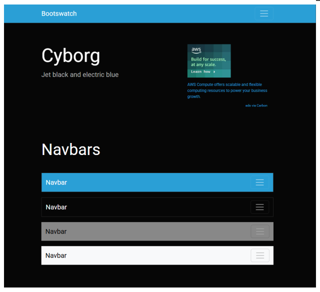


Figure 11:

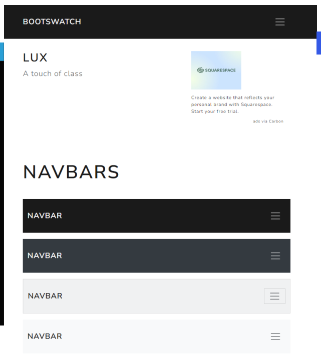


Figure 12:

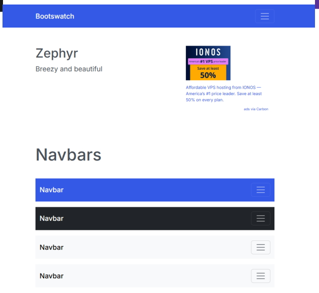


Figure 13:

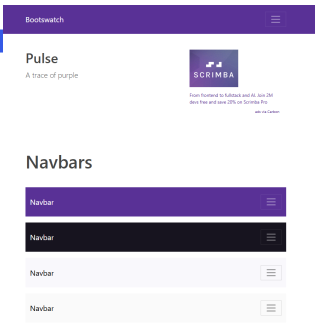


Figure 14:

Since Bootstrap isn't exactly known for wild creativity, we relied on Bootswatch to inject variety while keeping the codebase clean. When a specific design wasn't available, we customized the Bootswatch SCSS files to match our desired color palette and maintain a consistent style across components.

12.4 Wireframing

Wireframing is an essential component of front-end development; it serves as the blueprint from which all design and implementation decisions follow. Like all blueprints, its primary goal is to serve as a medium of communication. Communication is the foundation of any collaborative effort, and wireframing provides the first stage of communication for the front-end team.

It is important that the frontend team uses a tool that will allow us to work together on designs to ensure we are on the same page before implementation. It is much easier to tackle a large problem with a clear objective in mind prior to implementation. This will improve efficiency and lead to a better overall result. It is also important that we have the flexibility to make changes and add to designs as we progress further into the project and gain a better understanding of our end goals.

While we are the first group to work on this project, it is entirely possible that future Senior Design teams will continue our work. Therefore, documenting and wireframing all of our designs serves as an important organizational tool, not only for our current team but also for any future teams to draw inspiration from or build upon our designs.

12.4.1 Decision Criteria

There are many wireframing tools available, so it was important to establish clear decision criteria to guide our final decision. Since this project focuses on building an educational web application, practicality and clarity were our main priorities rather than highly complex or flashy design features. The major factors we considered included:

- Ease of Use. In order to begin implementation as soon as possible, we prioritized researching tools that are intuitive, beginner-friendly and have a well-established resource for working with the tool.

- Collaboration Features. The ability to work together both in real time and asynchronously through sharing feedback easily was essential. Effective collaboration ensures smooth communication across the team and aligns with the overall goal of efficient group work.

- Cost. As students, we aimed to minimize additional expenses and preferred tools that offered strong free versions or educational discounts

- Flexibility. We wanted tools that could support our needs without requiring excessive plug-ins or extraneous tools to work sufficiently.

12.4.2 Figma

Figma was the first technology we considered, as it is one of the most widely used and well-supported wireframing and UI design tools available today. A notable feature of Figma is its emphasis on real-time collaboration. Multiple team members can work on or view the same design simultaneously, which allows for instant feedback and streamlined communication. Figma's commenting feature also makes it easy to communicate asynchronously, which is very helpful as scheduling meetings at the same time can be difficult throughout the week.

Sharing designs is also extremely easy with Figma, as links can be shared with anyone, including non-front-end team members, for feedback or review of designs. Figma also offers a large collection of community resources, like templates, plug-ins, and tutorials, which help users get acclimated to the software quickly. While Figma is capable of supporting more high-level designs, its basic functionality is straightforward and practical for our project.

Our main concern with Figma was its potential cost. While there is a free version, we were unsure if the free version would be overly limiting. Fortunately, we found that the free plan provided all the core features we needed for the project. The paid version, priced at around 15 dollars per month, could be considered later if premium features became necessary [1]. The month-to-month payments would be manageable if deemed necessary. Figma also works well on all major operating systems, so it would be easily accessible for everyone on the team. Overall, Figma met nearly all our criteria as it is collaborative, user-friendly, and cost-effective given our current scope.

12.4.3 Canva

Another tool we considered was Canva. Although Canva is not a traditional UI/UX design platform, it can be used effectively for wireframing, especially for projects with simpler design requirements, such as this. Canva's biggest strength is its ease of use. It has an easy-to-use drag-and-drop interface. It also has a large variety of templates, icons, and pre-made design elements, making it extremely approachable for beginners. This allows team members to quickly create clear layouts without needing extensive design experience with the tool.

Canva also supports collaborative editing, where multiple users can work on a design at the same time and leave comments similar to Figma. Files are stored in the cloud, making them easily accessible and shareable across different devices, giving it a great amount of flexibility. Canva works on all major operating systems through most web browsers, and it also offers desktop and mobile apps for added flexibility.

In terms of cost, Canva offers a free version that includes most core features, while the premium version costs 13 dollars per month, making it the cheapest of the options considered, if premium features were deemed necessary. Canva also provides educational access for free to students, which could make it even more accessible for our team.

However, Canva does have a major limitation in terms of flexibility. Its heavy use of pre-made templates, while convenient, can be restrictive when trying to create or experiment with more complex designs. This could become a drawback if future teams wish to implement more advanced UI layouts in the future. Despite this, Canva's simplicity and focus on clarity make it a strong option for an educational web application like ours, where functional, easy-to-understand designs are the top priority.

12.4.4 What we chose

After evaluating the most practical tools for our needs, our team ultimately selected Figma as our primary technology for design prototyping. It offered the most balanced combination of collaboration features, accessibility, and functionality. Allowing our front-end team to efficiently plan, iterate, and communicate interface designs.

Figma's collaboration tools, and ease of sharing designs and information made it especially effective for the needs of this project. Its easy-to-use interface lowered the learning curve for members who were new to working with the tool. Additionally, its extensive library of community resources and templates enabled us to quickly build and refine wireframes that align with our project's goals and communicate the front-end vision with the rest of the team.

12.5 Figma Designs

Below are several of the mock-up pages designed in Figma along with a discussion of their purpose within the application. While these designs are subject to change as implementation of the project progresses, the goals and requirements for each page will likely remain the same.

12.5.1 Login Page

A login page (figure 15)is necessary as both students and instructor users will need different levels of access and capabilities. User accounts will also allow students to save their work and return to it later. Although containers will automatically stop after 24 hours, this approach will provide students the flexibility to work in intervals rather than needing to complete everything in a single session.

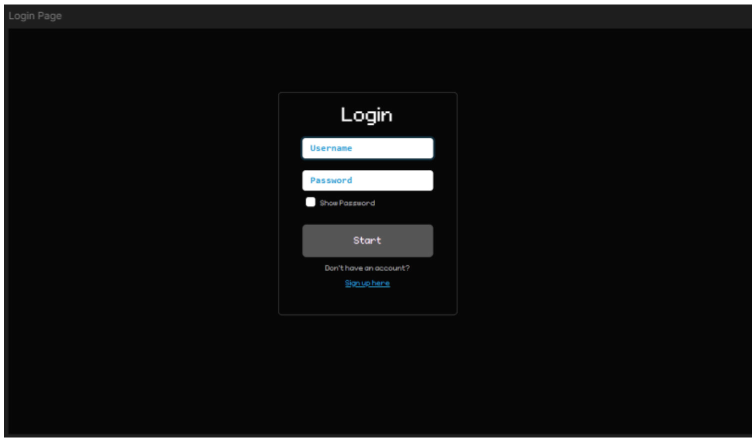


Figure 15: Example login page design

12.5.2 Vulnerable Applications Dashboard

Students need a clear and easily accessible way to work with vulnerable applications. Therefore, the webapp will have a dashboard (figure 16) which will contain all available vulnerable applications. Each listed application will include a description of the application alongside the relevant skills that are being tested when attempting to patch any vulnerabilities. The front-end team will work with the sponsor and docker team to rank the difficulty as well.

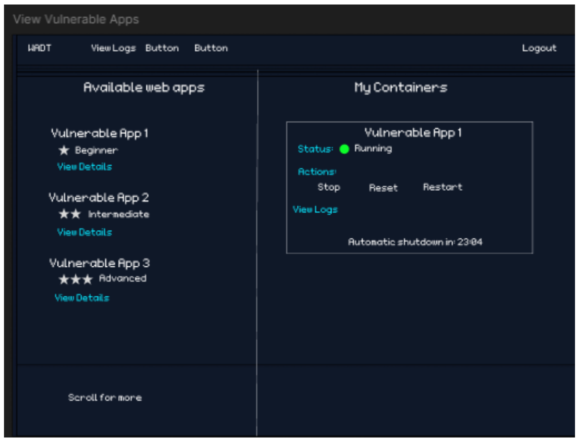
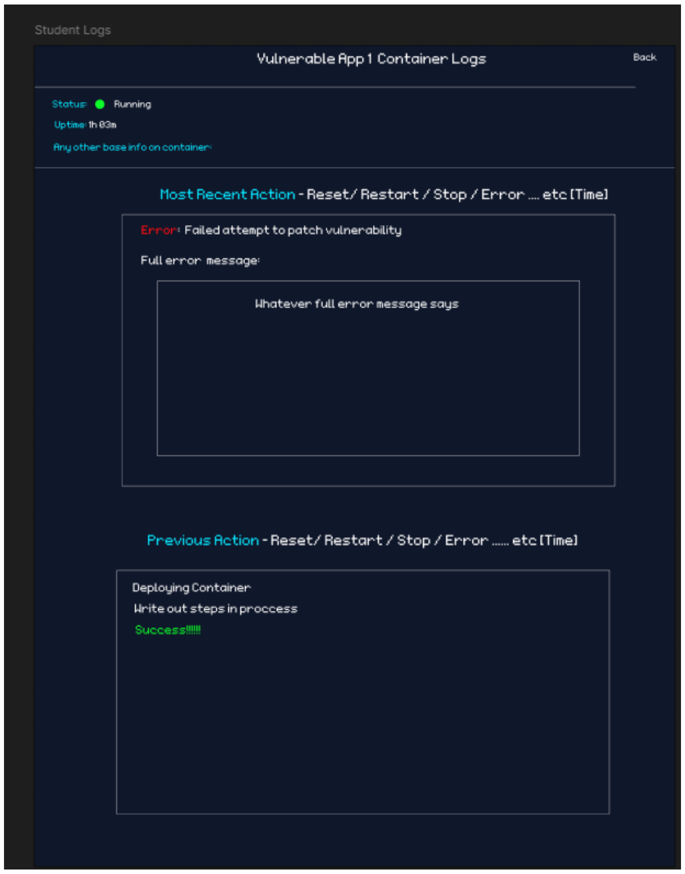


Figure 16: Example dashboard design

12.5.3 Container Logs

The container logs page (figure 17) is crucial, as it will provide all of the information needed to resolve any issues related to any relevant container actions or events.



Vulnerable App 1 Container Logs

Status: ● Running

Uptime: 1h 03m

Any other base info on container:

Most Recent Action - Reset/ Restart / Stop / Error .... etc [Time]

Error: Failed attempt to patch vulnerability

Full error message:

Whatever full error message says

Previous Action - Reset/ Restart / Stop / Error ...... etc [Time]

Deploying Container
Write out steps in proccess
Success!!!!!!

Figure 17: Example Container Logs

Logs will be maintained for the entire lifespan of each container until it is stopped manually or automatically. Since feedback is critical for an educational tool, the logs page must present information clearly and simply. Only relevant troubleshooting information will be included to avoid overwhelming the user.

The designs shown above represent only some of the pages created throughout the planning of this project. Over the course of development, many of our original ideas have been adjusted, or expanded as we learned more about the requirements and how each page would function within the application. Figma has been especially helpful in this process, as it allows us to quickly draft new ideas and revise our designs without needing to commit to coding right away. While not every design can be included in this document, the mock-ups we have shown highlight the core pages necessary for the application and reflect the overall direction of the project.

13 Testing Plans

13.1 Unit & Integration Testing

Component	Scenario	Expected Outcome
Scheduler	Mock container creation time > 24 hours.	`stop()` command issued for expired ID.
RBAC	Student role attempts Admin action.	`403 Forbidden`.
Views	POST request with invalid JSON.	`400 Bad Request`.
Docker Client	Mock DockerException	`503 Service Unavailable`.

Table 1: Backend Logic Testing

Component	Scenario	Expected Outcome
Login Form	User submits empty form.	validation error ”Fields cannot be empty.” appears; API is NOT called.
Dashboard	API returns 500 Error.	”System Offline” banner is displayed to user.
Deploy Button	User clicks ”Deploy”.	Button becomes disabled; Spinner icon appears.
Status Badge	Container status is ”stopped”.	Badge color renders Red (not Green).

Table 2: Frontend Testing

14 Ethical Concerns and Considerations

WADT will host intentionally vulnerable applications, which come with its own unique ethical and security challenges. We have a responsibility to protect our users, and systems from foul play and misuse. The most prominent ethical consideration for this project revolves around limiting potential harm to protect our users in order to promote responsible cybersecurity education.

14.1 Data Privacy

WADT serves as an educational tool for students, rather than as commercial application, therefore it is unnecessary to store significant amounts of personal information. The only personal information needed will be a username and password. Other information such as

Components	Scenario	Pass Criteria
API + Docker	Full Container Lifecycle	1. Deploy triggers start. 2. List returns correct container ID. 3. Stop command halts container.
API + DB	User-Container Association	User ID matches corresponding container ID.
API + Docker	Container Isolation	Two users deploying same app results in two distinct Container IDs.
API + Network	Vulnerability Reachability	HTTP request to mapped port returns `200 OK`.

Table 3: End to End Testing

Workflow	Steps	Pass Criteria
Authentication	Navigate to login page and enter valid credentials.	User is redirected to /dashboard; JWT token stored in memory/cookie.
Deployment	User selects chosen container, clicks ”Deploy” and waits 5 seconds.	Dashboard list updates to show new container with ”Running” status.
Log Viewer	Click ”View Logs”.	Log opens with text not being empty.
Route Guarding	Logout and manually attempt to enter /dashboard URL.	User is forcibly redirected back to login page.

Table 4: Frontend End-to-End Testing

Category	Metric Name	Target Value	Tool
Performance	API Latency	< 200ms (excluding Docker)	Postman
Reliability	Deployment Success	99% success rate	Scripts
Efficiency	Container Cleanup	100% expired apps removed	Logs
Accessibility	Lighthouse Score	100% on Lighthouse Report	Lighthouse

Table 5: Performance and Reliability Metrics

credit card information, address, or other personal contact information will never be stored by WADT. Additionally, logs will exclude any identifiable information beyond what is required for container management system audits. We will store minimal information to ensure that if an account is compromised for any reason, there will be minimal impact for the user.

14.2 Responsible Handling of Vulnerable Applications

We will host vulnerable applications with exploitable vulnerabilities by design. Given this, we have an ethical responsibility to ensure these vulnerabilities cannot be exploited and used for harm. These applications will be used only for educational purposes within a controlled environment. In order to prevent unethical use of our systems, we will have strict controls to mitigate any risks that come with such hosting. Our AWS instance will not be publicly accessible and will only accept or send out traffic to a small number of white-listed IP addresses for the intended students and instructors. All vulnerable applications will be deployed using OCI-compliant container tooling and images to further reduce risk of exploitation of a contained vulnerability. We will also implement automated shutdowns of containers to reduce exposure and mitigate the ability of vulnerable containers to interact with external environments.

14.3 Information Logging

Our application will maintain logs of container actions for both students and instructors such as deployment, stoppage, and resetting. These logs will be limited only to container relevant events or actions. These logs will serve two primary functions.

- Feedback. Provide clear feedback to students on their interactions with containers.
- Oversight. Provide visibility to instructors into container usage and potential misuse.

Our logs will only capture information required for necessary functions and will exclude excessive information such as private user information, or unrelated system information. Logging will only be used as a tool for debugging, learning, or ensuring proper usage

15 The “WADTtastic” Future In Store

The WADT project is the team’s opportunity to support cybersecurity education and make the greatest of all makeable things, a useful tool. We want to see the project revolutionize the workflow of our target market and make jumping into a lab as quick and easy as one-click shipping.

Tedium is the worst thing that smart people can be subjected to, and WADT gets rid of some of it for students and the instructor. We hope that by making it easier to get started, students can retain their motivation for the problems they're really after.

We also hope that in creating a free and open-source tool, our same users will become our contributors as they envision new changes to WADT that we couldn't have imagined ourselves. We have strived to create a maintainable codebase with low technical debt, so that future Senior Design groups or users may pick up our project again. Ideally, others may use what they have learned to make something neat, as we are doing.

And rounding up our hopes for WADT, the most important vision has come from our project sponsor. We know that he will put our MVP in the hands of real users and appreciate the objective which he has given to the group. We have built up a great deal of confidence in our sponsor as a long-term project owner who can carry the project forward to future teams of developers, or develop it further into a more available webapp that can serve more users.

16 System Architecture

16.1 Entity Relationship Diagram (ERD)

The database requires two tables: Container and LogEntry. Container encapsulates the state information required to manage containers, while LogEntry tracks the information which is required for logs of each container's individual behavior.

As visible in figure 18, each container has a name and description. These are user-provided metadata which will display on the site.

Each container also requires an id, which is needed for making requests through the Docker API.

Each log entry has its contents, a timestamp, and a foreign key which links back to the container the log is sourced from.

Other portions of the ERD are automatically generated by Django.

Examples of each field type can be seen in table 6.

16.2 Use Case Diagram

The use case diagram (figure 19) illustrates the interactions between the two primary actors, students and instructors, inside the web application system. Students can browse available vulnerable web applications, deploy containers with automatic 24-hour timeouts, and manage their containers through stop, restart, and reset operations. Students also have access to

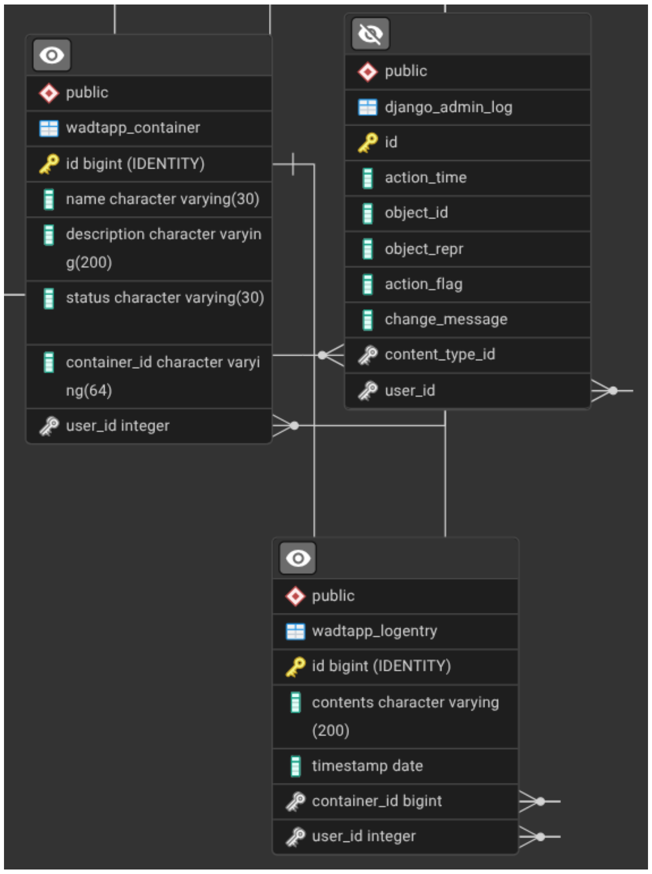


Figure 18: WADT ERD

Table	**Field**	**Type**	**Example**
Container	id	bigint (PK)	42
	name	character varying(30)	DVWA
	description	character varying(200)	Damn Vulnerable Web Application with SQL injection and XSS challenges
	status	character varying(30)	running
	container_id	character varying(64)	a3f7b9c2d1e8...
	user_id	integer (FK)	15
LogEntry	id	bigint (PK)	1337
	contents	character varying(200)	Container started successfully on port 8080
	timestamp	date	2025-11-23 14:32:17
	container_id	bigint (FK)	42
	user_id	integer (FK)	15

Table 6: Container and LogEntry Tables

timestamped logs specific to their container activities. Instructors inherit all student capabilities, but have additional privileges which include, access to more comprehensive instructor logs, the ability to view statuses across all student containers, disabling containers to restrict student access, and overriding student actions with race condition prevention.

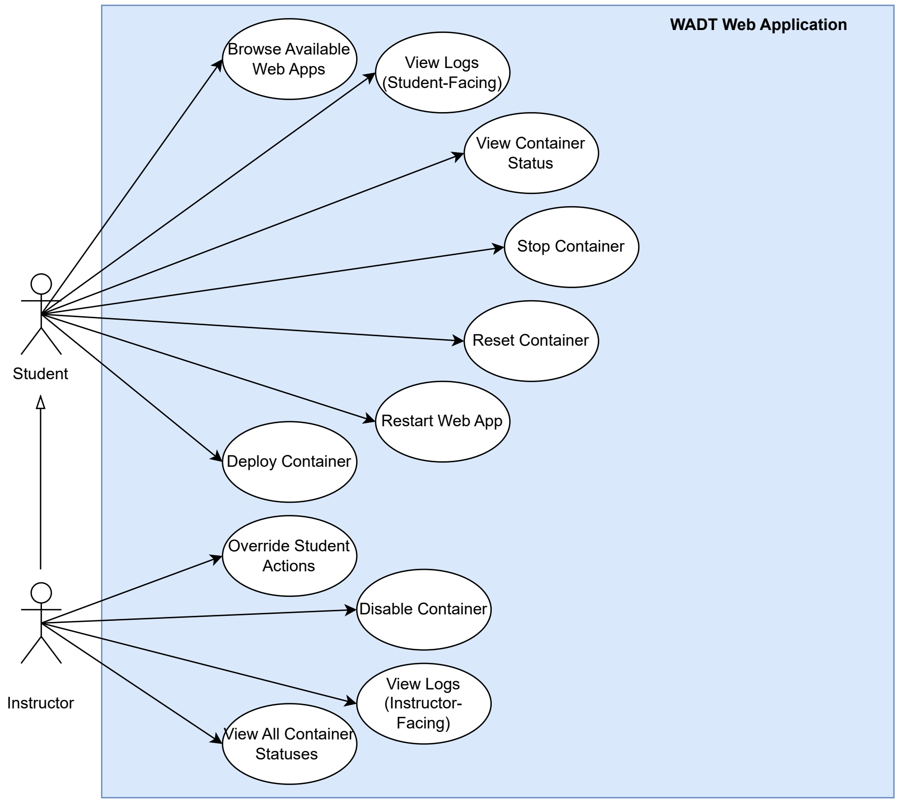


Figure 19: WADT Web Application Use Case Diagram

16.3 Deploying Containers

The activity diagram (figure 20) details the process behind deployment of a container, from catalog selection through backend processing.

The backend applies security configurations automatically during deployment, ensuring each container runs in an isolated network with enforced lifetime limits. If an image is unavailable, the system returns an error rather than proceeding with a partial deployment.

16.4 Interactions with Containers

The figure 21 illustrates the management actions available to users during an active session.

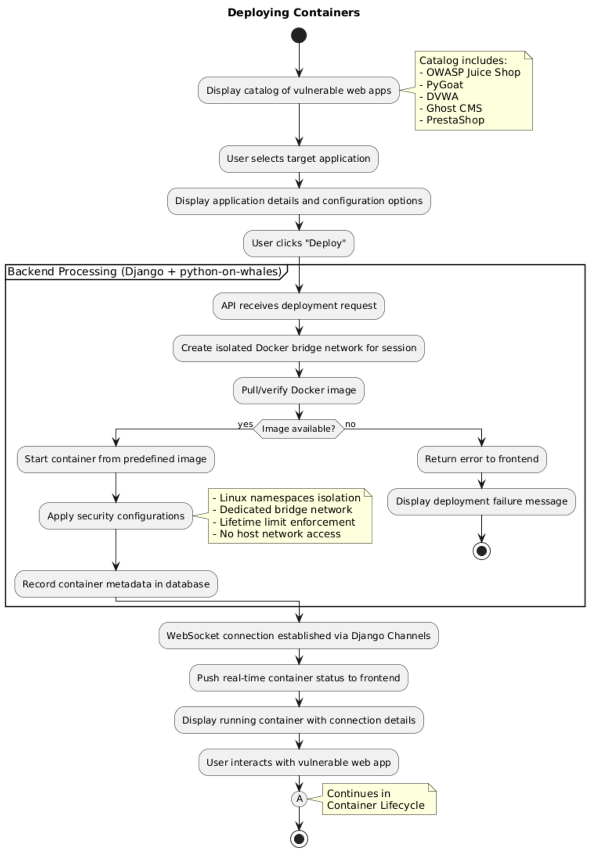

Deploying Containers
Display catalog of vulnerable web apps
Catalog includes:
- OWASP Juice Shop
- PyGoat
- DVWA
- Ghost CMS
- PrestaShop
User selects target application
Display application details and configuration options
User clicks "Deploy"
Backend Processing (Django + python-on-whales)
API receives deployment request
Create isolated Docker bridge network for session
Pull/verify Docker image
Image available?
yes
no
Start container from predefined image
Apply security configurations
- Linux namespaces isolation
- Dedicated bridge network
- Lifetime limit enforcement
- No host network access
Record container metadata in database
Return error to frontend
Display deployment failure message
WebSocket connection established via Django Channels
Push real-time container status to frontend
Display running container with connection details
User interacts with vulnerable web app
A
Continues in
Container Lifecycle

Figure 20: Container Deployment Activity Diagram

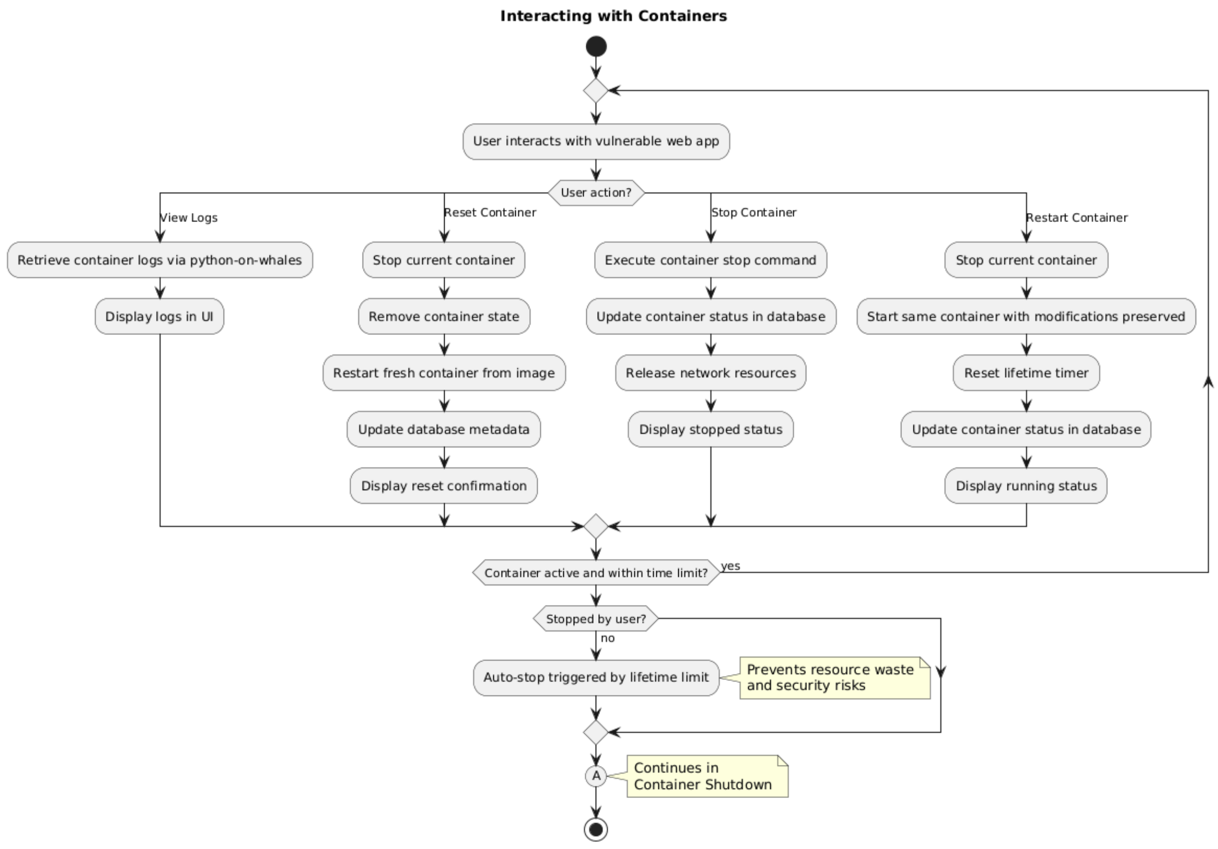

Interacting with Containers
User interacts with vulnerable web app
User action?
View Logs
Reset Container
Stop Container
Restart Container
Retrieve container logs via python-on-whales
Display logs in UI
Stop current container
Remove container state
Restart fresh container from image
Update database metadata
Display reset confirmation
Execute container stop command
Update container status in database
Release network resources
Display stopped status
Stop current container
Start same container with modifications preserved
Reset lifetime timer
Update container status in database
Display running status
Container active and within time limit?
yes
Stopped by user?
no
Auto-stop triggered by lifetime limit
Prevents resource waste and security risks
A
Continues in Container Shutdown

Figure 21: Interacting with Containers

During an active session, users can view logs, reset to a clean state, restart with any changes made preserved, or stop containers entirely. The system monitors lifetime limits in the background and triggers automatic shutdown if a container exceeds its allocated time.

16.5 Container Cleanup Process

The figure 22 describes the cleanup process that occurs when a container session ends.

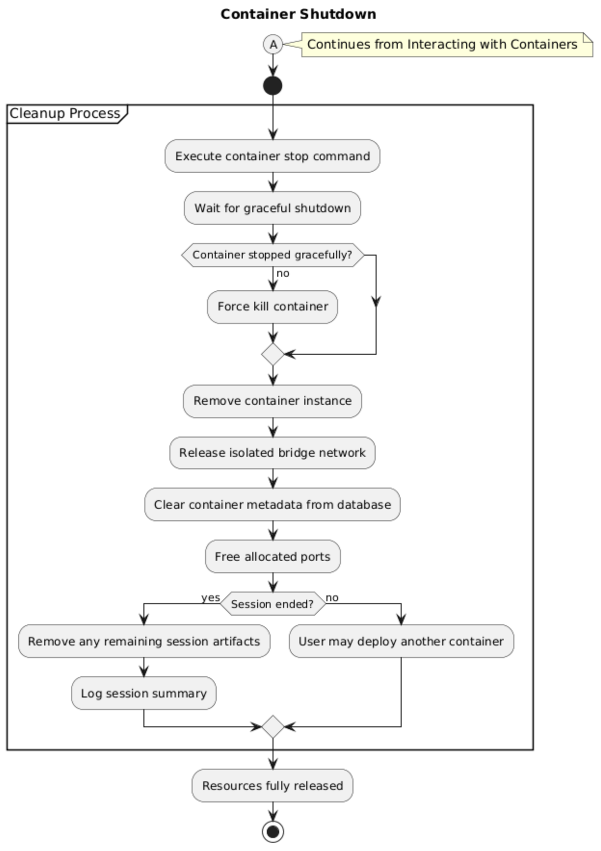

Container Shutdown
A
Continues from Interacting with Containers
Cleanup Process
Execute container stop command
Wait for graceful shutdown
Container stopped gracefully?
no
Force kill container
Remove container instance
Release isolated bridge network
Clear container metadata from database
Free allocated ports
yes
Session ended?
no
Remove any remaining session artifacts
User may deploy another container
Log session summary
Resources fully released

Figure 22: Container Cleanup

The system attempts a graceful shutdown before forcing termination if necessary. All associated resources, network bridges, database metadata, and allocated ports are released to maintain system integrity for subsequent deployments.

17 Design Summary

17.1 Framework Selection: Django and DRF

Reasons for choosing Django + DRF:

- Built-in ORM and admin panel, make managing database easier with migrations
- Django Channels for WebSockets
- Function-based views + viewsets make a powerful set of abstractions available

We chose a REST API architecture with React instead of Django MVT alone, for these reasons:

- Web-browsable HTML version of API
- Viewsets inheriting common CRUD operations
- Serializer classes enforce separation of concerns
- Backend and frontend are more independent

17.2 Programming Language: Python

Python was chosen for its Docker SDK integration and compatibility with Django. We opted not to use Go because its Docker SDKs were low-level and detail-heavy in comparison, beyond what was needed to implement WADT.

17.3 Containerization Tool: Docker over Podman

We chose Docker over Podman for the following reasons:

- More extensive Python SDK
- Better ecosystem and documentation
- Highly available public registries (DockerHub and others)
- No potential incompatibilities that Podman might include

17.4 Docker SDK: python-on-whales over docker-py

python-on-whales is a better fit for WADT, as was experimentally verified through the two prototypes. The following advantages make python-on-whales a more attractive choice:

- Thread safety. Guaranteed thread-safe client means views are simpler to write
- Compose support. Built-in Docker Compose support extends range of available vulnerable apps
- Clear API. python-on-whales mirrors the Docker CLI syntax almost perfectly.

17.5 Database: PostgreSQL

The team evaluated MongoDB, MySQL, MariaDB, and PostgreSQL, ultimately choosing PostgreSQL for its support and documentation. The database runs locally on the AWS EC2 instance. We may migrate to cloud storage if log entries result in a larger database, requiring a dedicated storage solution.

17.6 Async Support: Django Channels over Celery

For real-time status information from containers, we chose Django Channels instead of Celery:

- Celery's capacity exceeded project needs
- Channels has fewer dependencies

17.7 Frontend: React and TypeScript

React was selected for its extensive selection of useful components. The reasons for choosing React include:

- Efficiency
- Component reusability
- Enforce division between frontend and backend work, leading to a cleaner codebase
- High-quality documentation

TypeScript over JavaScript provided compile-time error detection and type annotations.

17.8 React-Bootstrap

The team chose React-Bootstrap (Bootstrap 5 integration) over Tailwind and Chakra UI:

- Team familiarity and sponsor familiarity
- Already available components
- Bootswatch themes (Cyborg, Lux, Zephyr, Pulse) available

17.9 Hosting: AWS

We selected our hosting for the following reasons:

- sponsor-provided hosting option was AWS EC2
- Improved familiarity
- Needed packages compatible with Ubuntu and available in standard repos

17.10 Security Measures

- IP restrictions (deny-by-default)
- Limited network access for containers
- Resource limits and uptime limits
- Docker configuration best practices

17.11 2FA with pyotp + pass-otp

The authentication system will use TOTP:

- Users must have SSH access to the AWS instance to generate auth codes
- `pyotp` generates one-time codes on the instance
- Users store secrets using `pass` with `pass-otp` extension on the instance
- Therefore, users must have SSH access as a prerequisite to access WADT

17.11.1 Secrets Management

- Pre-commit hooks
- Scan every commit
- Measures in place for rotation and emergencies relating to leaks

17.12 Vulnerable Application Catalog

17.12.1 Intentional Vulnerabilities:

- OWASP Juice Shop
- PyGoat
- DVWA
- vAPI
- NIVA
- Vulnerableapp-facade
- BodgeIt Store
- Damn Vulnerable RESTaurant

17.12.2 Unintentional CVEs:

- Ghost CMS (CVE-2023-32235)
- PrestaShop (CVE-2023-3882)
- Ghost Kit WordPress Plugin (CVE-2025-9992)

17.13 Containerization

Each application includes:

- Dockerfiles
- Resource limits
- Maximum uptime (24hrs)

17.14 CI/CD Pipeline

1. TruffleHog scan
2. Backend tests
3. Frontend tests

4. Build artifacts
5. AWS deployment

18 Timeline and Milestones

18.1 Project Timeline

The following timeline outlines the weekly tasks and objectives throughout Senior Design I and II.

Senior Design I

- **Week 1 (Sept 9 – Sept 15):** Meet team, contact sponsor, set up communication channels
- **Week 2 (Sept 16 – Sept 22):** Meet sponsor, write Team Contract, assign roles, and gather requirements
- **Week 3 (Sept 23 – Sept 29):** Set up Jira, conduct research on Docker integration, database, and security concerns
- **Week 4 (Sept 30 – Oct 6):** Write Early Status Report and present research to sponsor to set up AWS instance
- **Week 5 (Oct 7 – Oct 13):** Set up GitHub, create initial UI design mockups, and start Preliminary Design Document
- **Week 6 (Oct 14 – Oct 20):** Continue working on Preliminary Design Document, start initial coding and setting up installations, packages, and organizing the codebase
- **Weeks 7–8 (Oct 21 – Nov 3):** Finish Preliminary Design Document, create additional UI mockups for the rest of the pages, and get initial API endpoints working
- **Weeks 9–10 (Nov 4 – Nov 17):** Implement UI for base website, finish APIs for base website, and work on Final Design Document
- **Week 11 (Nov 18 – Nov 24):** Final Design Document and plan work that needs to be done between semesters

Senior Design II

- **Week 1 (Jan 12 – Jan 18):** Reconvene with sponsor and review SD1 feedback

- **Weeks 2–3 (Jan 19 – Feb 1):** Begin implementation of container management features (deploy, stop, restart, reset) and establish stable frontend-backend integration
- **Weeks 4–5 (Feb 2 – Feb 15):** Deploy and test first vulnerable web application, validate UI interactions and backend functionality
- **Weeks 6–8 (Feb 16 – Mar 8):** Expand system to include all planned web applications, begin integration testing and security validation
- **Weeks 9–10 (Mar 9 – Mar 22):** Performance testing, get feedback from sponsor, implement stretch goals
- **Weeks 11–13 (Mar 23 – Apr 12):** Implement instructor features, final touches, and work on Final Design Document
- **Weeks 14-15 (Apr 13- Apr 27):** Final presentation and Design Document

18.2 Project Milestones

The following milestones represent the major deliverables and functional achievements of the project. Reaching milestones will help us ensure we reach our Definition of Done and have all of the required features and functions.

Senior Design I

- **Oct 3:** Submit Early Status Report
- **Oct 31:** Submit Preliminary Design Document
- **Nov 17:** Build functional web app with basic features implemented
- **Nov 24:** Submit Final Design Document

Senior Design II

- **Feb 8:** Container management system fully implemented with fully working deploy, stop, restart, and reset functions, container resource limits enforced and accurate status information
- **Mar 1:** First vulnerable web application deployed and fully functional
- **Mar 22:** All 7 web applications integrated
- **Apr 26:** Instructor features implemented and system testing
- **Apr 27:** Final presentation and Final Design Document delivered

19 Budget and Financing

The WADT project's primary cost is hosting and domain registration. The group was able to secure a free domain name for 1 year, extending beyond the timeline of SD1 and SD2.

Our sponsor has agreed to pay for hosting (see figure 23) through the end of SD2. The instance size and capabilities are flexible and may scale up to meet the project's usage requirements. Since inbound data transfer from the internet is free, the team doesn't need to budget Docker image pulls from registries.

However, since outbound data transfer is tracked and charged, the team needs to carefully monitor outbound data transfer to prevent excessive usage that will result in excessive bills for our sponsor. Based on current Amazon AWS pricing, outbound data transfer costs begin to accumulate after the first 100GB sent in a month.

The project is not likely to reach the scale of genuinely requiring 100+GB of outbound traffic by the end of SD2. If these costs were to accumulate, it is far more likely due to a security vulnerability within WADT that allows users to send malicious outbound requests.

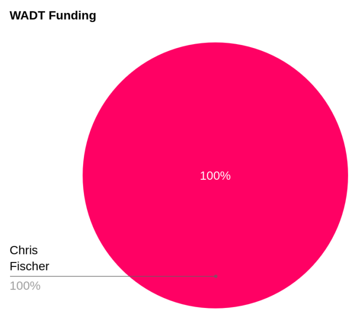


Figure 23: WADT Funding Breakdown

20 What's Next?

Our group is the first to work on the Web App Deployment Tool, and while the project has a relatively small set of core features, there is a lot of room for additional features. Once we begin testing the application and receiving feedback, the relevance of additional features will become clearer. By the end of Senior Design II, we will have a functional website with all of the core features, but the project is designed so that future groups can continue building on what we have created. As implementation progresses, we plan to add extra features whenever possible, based on feedback and any additional ideas that emerge as the application becomes more complete. This will create a strong foundation for later groups to extend.

Cybersecurity education evolves constantly, especially as new vulnerabilities, technologies, and industry practices emerge. With that in mind, there is a lot of potential to expand the catalog of vulnerable applications beyond the initial seven we will include. Future groups will be able to update the tool to introduce new applications, new vulnerability types, and new skills for students to practice. Our team is essentially starting the process and establishing the structure, leaving plenty of room for refinement as more students and instructors interact with the application. While we will make updates based on early feedback, the most meaningful insights will come from long-term use in educational settings. Over time, future teams together with the sponsor will be able to use that feedback to set new goals and evolve the project further.

There are also additional features outside our core requirements that later teams could explore. One major possibility is adding automated checks to determine whether a vulnerable application has been successfully patched. Even if this isn't part of our team minimum viable product, another group could create a companion tool to evaluate student work more thoroughly. Our focus is on giving students the environment they need to work on vulnerabilities, but a complementary system that provides automated feedback without requiring instructor involvement would make the tool even more useful. The overarching goal of this project is to make the learning process more efficient, and giving students fast, and thorough feedback would help them learn more independently. We are supplying the core environment, but there is significant room to elevate the project by expanding the amount of guidance, assessment, and feedback available to students.

21 Project Management (Agile)

21.1 Product Backlog with User Stories

- arbitrary command execution - Medium - To Do
- admin controls/advanced - Medium - To Do
- edit network settings per-container - Medium - To Do
- admin controls/basic - Medium - To Do - container resource usage limits and auto-down after 24hrs
- colorschemes switching - Medium - To Do

- test dockerized TVWAs - Highest - To Do - ”As a backend developer, I want to ensure that each containerized Target Vulnerable WebApp is functioning properly so that immediate problems can be diagnosed.”
- Docker Compose compatibility - Medium - To Do - ”As a user, I want WADT to work for both single-container and Compose projects, so that I can practice while ignoring Docker-specific details”
- accessibility testing - Medium - To Do - ”As a user, I want a fully accessible site so that I can use WADT under a diverse set of circumstances”
- performance testing - Medium - To Do - ”As a sponsor providing hosting, I want a site that has minimal overhead to minimize bandwidth charges.”
- domain name - Highest - In Review - ”As a user, I want a convenient domain to use to access WADT so that I can easily remember it and start practicing.”
- 2-factor auth - Lowest - To Do - ”As a site administrator, I need 2-factor authentication for user logon to prevent unauthorized access from bad actors.”
- write API spec - High - To Do - ”As a frontend developer, I require an API spec in order to develop frontend UI components for the project.”
- initial deployment - Highest - To Do - ”As a user, I want a real website to access so that I don’t have to install software on my computer.”
- set up CI/CD pipeline - Highest - To Do - ”As a developer, I want a CI/CD pipeline to format, lint, test, and scan for vulnerabilities my code before deploying it automatically, so that I don’t have to hurt my wrists typing the same things over and over.”
- deploy database - Highest - To Do - ”As a developer, I need a database on a server to store data so that my app works.”
- dockerize unintentionally vulnerable webapps 2 - Medium - Done - ”As a user, I want WADT to include modern webapps with unintentional vulnerabilities which are published CVEs, so that I can practice in a realistic environment.”
- dockerize unintentionally vulnerable webapps 1 - Medium - Done - ”As a user, I want WADT to include modern webapps with unintentional vulnerabilities which are published CVEs, so that I can practice in a realistic environment.”
- Dockerize second 5 Web Apps - High - Done

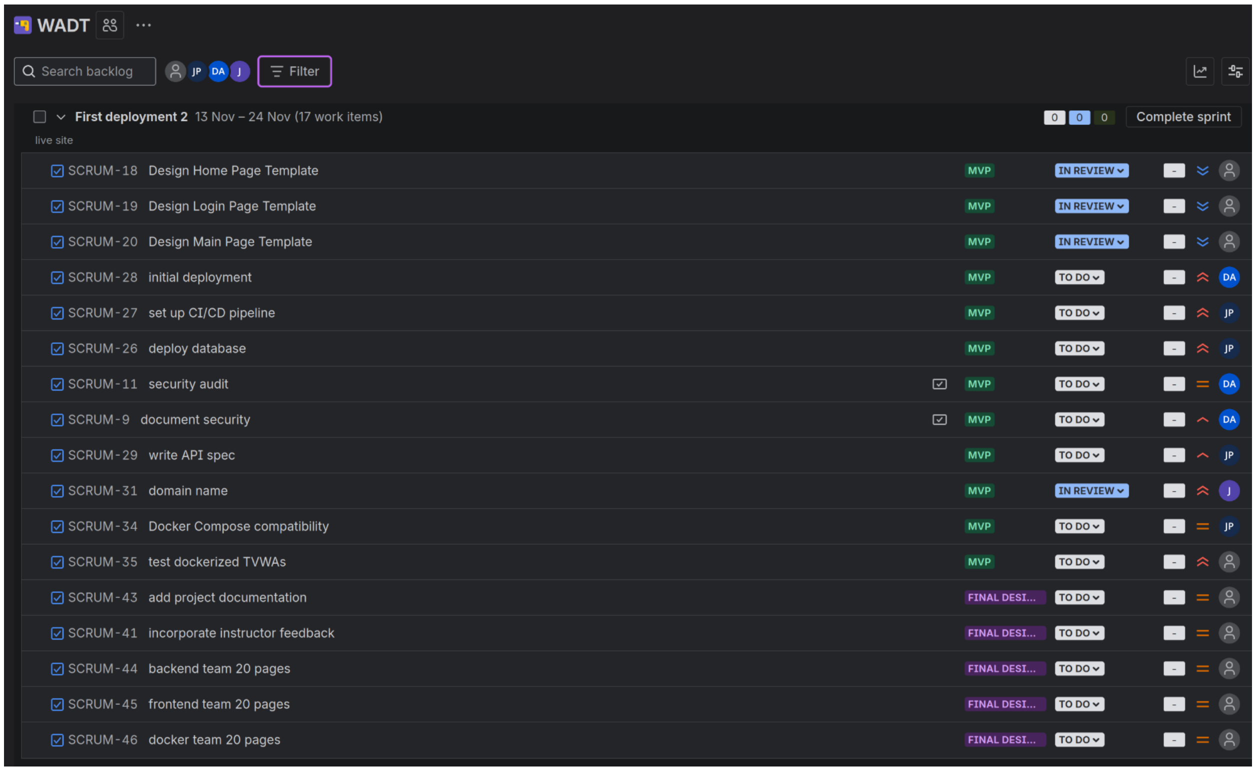
- Dockerize first 5 Web Apps - High - Done
- Create repo - Highest - Done - ”As a developer, I want a repository to work on that will automatically deploy to the instance so that code changes are tracked”
- Design Main Page Template - Lowest - In Review - ”Communicate with Backend to determine what is necessary for a main page (where the user can access the docker containers). Then use Figma to design a wireframe that matches the theme of the Home page.”
- Design Login Page Template - Lowest - In Review - Design a wireframe login page utilizing the themes from the home page on figma
- Design Home Page Template - Lowest - In Review - ”Design a wireframe home page using Figma to determine: * Website Color Theme * Website Format * Other UI components”
- Determine essential interactions - High - Done - Answer the question: what do our users want to do with Docker containers? The answer determines which API endpoints the backend must expose and which controls the UI team must anticipate.
- Integrate with backend team - Medium - To Do
- ”create reports for each image (file size - potential problems - etc.)” - Medium - To Do
- write Dockerfiles/compose files - Medium - To Do
- Select 7 apps - Medium - To Do
- choose 10 vulnerable web apps and propose to sponsor - Lowest - Done - Select and containerize 7 appropriate vulnerable apps
- security audit - Medium - To Do - Entire team verifies the security practices used to set up the AWS instance. Each member documents potential issues with the setup.
- docker seminar from sponsor - High - Done - Attend meeting with sponsor where he will show us how to identify and dockerize vulnerable apps.
- document security - High - To Do - ”Based on security decisions made for sandboxing containers, create documents to explain choices and setup to sponsor, our future selves - potential users, and the Senior Design Committee.”
- Prereqs for hosting - Highest - Done

- learn to dockerize - Medium - Done - Research the process of Dockerizing a sample web app and perform this task on a demo example.
- create UI mockups - Low - Done - create mockups/wireframes for the UI components of each page
- setup database - Medium - Done - ”Select the database for the project. Create a schema or model of the data. Project scope is one user at a time, limited enough to allow for not putting much thought into the database. However, an approach that at least allows for the possibility of multiple users eliminates technical debt which may haunt future SD teams.”
- aws instance setup - High - Done - Ensure all members can SSH into the AWS instance. Ensure instance is secure.
- sandboxing - Highest - Done - Determine how to sandbox vulnerable applications for safe hosting on an AWS instance. Must protect instance from privilege escalation attacks and prevent internet-connected containers from being used in botnets or similar.

21.2 Sprint Backlog with User Stories

- test dockerized TVWAs,”As a backend developer, I want to ensure that each containerized Target Vulnerable WebApp is functioning properly so that immediate problems can be diagnosed.”
- Docker Compose compatibility,”As a user, I want WADT to work for both single-container and Compose projects, so that I can practice while ignoring Docker-specific details”
- domain name,”As a user, I want a convenient domain to use to access WADT so that I can easily remember it and start practicing.”
- write API spec,”As a frontend developer, I require an API spec in order to develop frontend UI components for the project.”
- initial deployment,”As a user, I want a real website to access so that I don’t have to install software on my computer.”
- set up CI/CD pipeline,”As a developer, I want a CI/CD pipeline to format, lint, test, and scan for vulnerabilities my code before deploying it automatically, so that I don’t have to hurt my wrists typing the same things over and over.”

- deploy database,”As a developer, I need a database on a server to store data so that my app works.”
- Design Main Page Template,”Communicate with Backend to determine what is necessary for a main page (where the user can access the docker containers) Then use Figma to design a wireframe that matches the theme of the Home page.”
- Design Login Page Template,Design a wireframe login page utilizing the themes from the home page on figma
- Design Home Page Template,”Design a wireframe home page using Figma to determine: * Website Color Theme * Website Format * Other UI components”
- security audit,Entire team verifies the security practices used to set up the AWS instance. Each member documents potential issues with the setup.
- document security,”Based on security decisions made for sandboxing containers, create documents to explain choices and setup to sponsor, our future selves, potential users, and the Senior Design Committee.”



First deployment 2 13 Nov – 24 Nov (17 work items) 0 0 0 Complete sprint

live site

SCRUM-18	Design Home Page Template	MVP	IN REVIEW	
SCRUM-19	Design Login Page Template	MVP	IN REVIEW	
SCRUM-20	Design Main Page Template	MVP	IN REVIEW	
SCRUM-28	initial deployment	MVP	TO DO	DA
SCRUM-27	set up CI/CD pipeline	MVP	TO DO	JP
SCRUM-26	deploy database	MVP	TO DO	JP
SCRUM-11	security audit	MVP	TO DO	DA
SCRUM-9	document security	MVP	TO DO	DA
SCRUM-29	write API spec	MVP	TO DO	JP
SCRUM-31	domain name	MVP	IN REVIEW	J
SCRUM-34	Docker Compose compatibility	MVP	TO DO	JP
SCRUM-35	test dockerized TVWAs	MVP	TO DO	
SCRUM-43	add project documentation	FINAL DESI...	TO DO	
SCRUM-41	incorporate instructor feedback	FINAL DESI...	TO DO	
SCRUM-44	backend team 20 pages	FINAL DESI...	TO DO	
SCRUM-45	frontend team 20 pages	FINAL DESI...	TO DO	
SCRUM-46	docker team 20 pages	FINAL DESI...	TO DO	

Figure 24: Current Sprint Backlog with User Stories

21.3 Milestone Chart

21.3.1 Current Milestone Chart

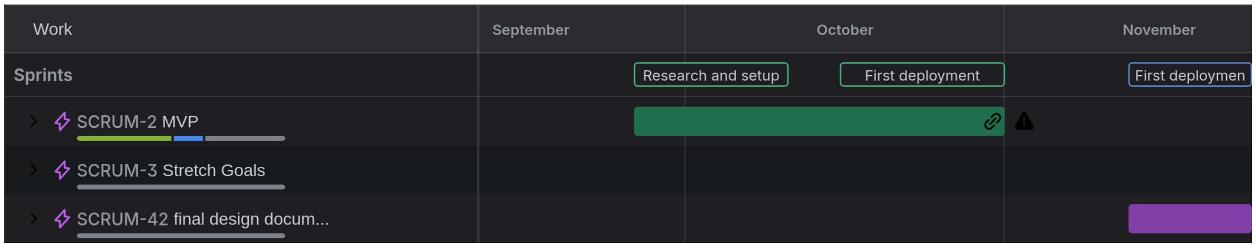


Figure 25: Current Milestone Chart (End of SD1)

21.3.2 Projected Milestone Chart

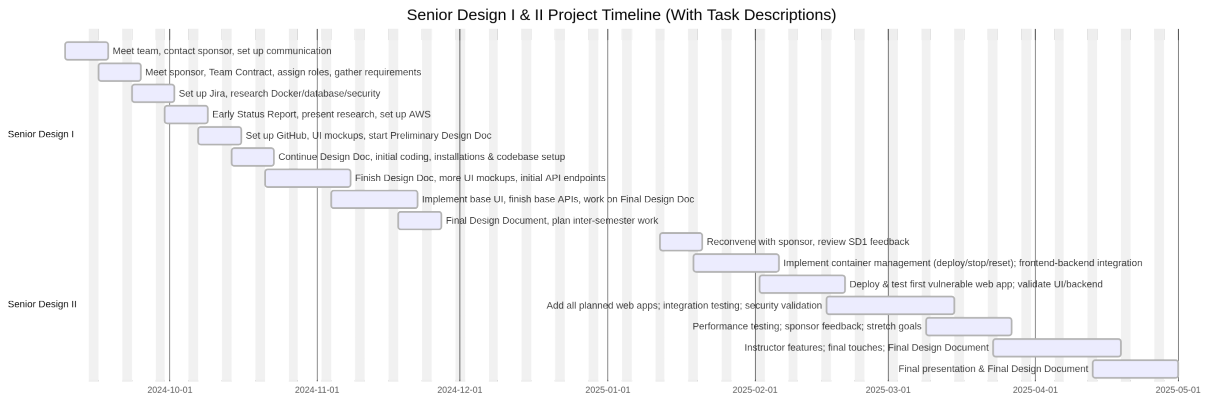


Figure 26: Projected Milestone Chart (End of SD2)

21.4 Setbacks

- Security research - didn't fit into original timebox. Team fell behind on first spike. Adapt through simplifying requirements and communicating with sponsor.
- Deployment - delayed due to issues with team members accessing hosting. Slight delay on first sprint. Adapt through creating team-internal documentation.
- Early Security vulnerability - due to an error in planning early sprints, repository created before security measures fully in place. Django secret keys accidentally leaked due to human error. No delay, but would be problematic if site active at time of leak. Adapt through committing to spike of research on security scanning tools and CI/CD pipeline.

21.4.1 Lessons Learned

Early rapid prototyping and constant communication have been vital for the team. As we uncover oversights in our original plans, we are able to quickly adapt and modify our sprints. We are fully equipped to continue the development cycle through SD2 while fulfillng our MVP.

22 Acknowledgements

The members of the WADT project wish to thank our sponsor Chris Fischer for providing hosting, guidance, and valuable advice and insight. Fischer was able to impart his relevant development experience to the team and assist us with beginning our study of the project technologies. He contributed to our design by giving loose guidance on system architecture.

We are grateful for the opportunity to create WADT in an excellent environment for it to thrive and see real usage. Fischer was able to connect us to UCF alums with cybersecurity experience and ensure our MVP conformed to the needs of a realistic lab setting. The multi-disciplinary user base (students and airmen) which we are thereby connected to is invaluable for guiding our user stories and design objectives.

We would like to acknowledge our sponsor for providing flexible, highly-available hosting to meet the project's needs. Since funding comes from our sponsor and is independent of any particular research, military, or government agency, this has enabled the team to take *our* chosen direction with the project and make technical decisions as we see fit to achieve the MVP agreed to by our sponsor. We are grateful for the opportunity to have free reign with the technical details, all the while accessing dependable hosting at no cost to any member of the team.

Permissions for Graphics

- Logo: ”Please feel free to use any of it you would like. I would appreciate a link back.”

- Podman diagram (figure: 4, source: [6]): ”To the extent the Site, portions of the Site, or Red Hat Content does not designate a license, Red Hat hereby grants you a copyright license to copy such material for your own personal or internal business purposes."

References

[1] *Figma Plans and Pricing*. URL: `https://www.figma.com/pricing/`. (accessed: 10.31.2025).

[2] *Getting Started*. URL: `https://docs.hoppscotch.io/documentation/getting-started/introduction`. (accessed: 10.29.2025).

[3] *Getting Started React (Legacy)*. URL: `https://legacy.reactjs.org/docs/getting-started.html`. (accessed: 10.27.2025).

[4] *Introduction to Celery*. URL: `https://docs.celeryq.dev/en/stable/getting-started/introduction.html`. (accessed: 10.31.2025).

[5] Gabriel de Marmiesse. *Python on Whales*. URL: `https://gabrieldemarmiesse.github.io/python-on-whales/`. (accessed: 10.31.2025).

[6] Podman: Managing pods and containers in a local container runtime. *Brent Baude*. URL: `https://developers.redhat.com/blog/2019/01/15/podman-managing-containers-pods#`. (accessed: 10.31.2025).

[7] *Postman documentation overview*. URL: `https://learning.postman.com/docs/introduction/overview/`. (accessed: 10.29.2025).

[8] *Postman Plans and Pricing*. URL: `https://www.postman.com/pricing/`. (accessed: 10.29.2025).

[9] Daniel Stenberg. *Everything curl*. URL: `https://everything.curl.dev/`. (accessed: 10.31.2025).

[10] *Swagger Explore*. URL: `https://swagger.io/product/explore/`. (accessed: 10.29.2025).

[11] *Swagger Pricing*. URL: `https://swagger.io/product/pricing/`. (accessed: 10.29.2025).