

# CI/CD and Static Code Analysis: A Framework for Automated and Secure Development Practice

Alexander Mamani Yucra  
*Cloud DevOpsSec*  
Master of Science in Cloud Computing  
December 2024

**Abstract**—The growing need for software by small and large companies has changed the way it is developed compared in past days. The software development cycle has evolved from traditional methodologies to agile methodologies, in which new and small software features must be implemented in a short period of time, it is called iterations. In each iteration, different software development processes are carried out, including testing, building, performing static code analysis, releasing a new version of the software, and deploying it to production.

Automating all of these processes makes it easier to test and release new features quickly, allowing you to create quality software. This set of automated processes is known as continuous integration, and continuous delivery and deployment (CI/CD). Another important aspect of CI/CD is that it allows to perform static code analysis in an automated way for each new feature of the software. This helps identify possible errors made by developers and potential security vulnerabilities.

This project carries out the entire CI/CD process and static code analysis, taking as an object of study a web application described in section I Architectural Design Aspects of the application. Likewise, it address topics such as deployment architecture and security aspects to consider during deployment, in section II-A. The CI/CD workflow and pipelines is also covered in sections II-B and II-C; performing pipelines in GitHub Actions.

In section III Static Code Analysis, the application is exposed to be scanned by a static analysis code system (SonarQube), highlighting security vulnerabilities and ways to fix them. Finally, in section IV conclusions we present the improvements that the application has after implementing Static Code Analysis and CI/CD. We also present challenges that we could not address and must be implemented to improve the CI/CD and Security aspects of the application.

## I. ARCHITECTURAL DESIGN ASPECTS OF THE APPLICATION

To implement CI/CD in any application, the first step is to understand its structure and design. Therefore, as cloud engineers, we can suggest cloud-based architectural designs for deployment and recommend suitable technologies to support them. In this section, we will cover various application aspects related to deployment and CI/CD implementation.

It is important to clarify that the primary focus of this project is CI/CD and static code analysis. Consequently, the application's functionality takes a secondary role.

### A. Functional Requirements of House Billing App

The application aims to transparently manage household bills, allowing users to easily track how much each housemate owes. Additionally, it enables admins to create new groups,

add expenses, and settle housemates' expenses. Figure 1 illustrates the most relevant application features, such as creating groups, adding expenses, and settling expenses.

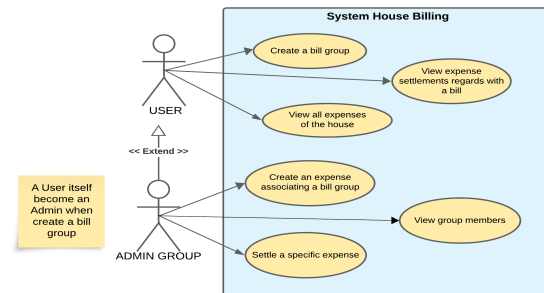


Fig. 1. Functional Requirements of House Billing App

The House Billing App is available in a public IP [Here](#) and the credentials are listed in the “Table. I”.

TABLE I  
HOUSE BILLING APP CREDENTIALS TABLE

House billing Credentials		
No	Email	Password
1	alex@user.com	user
2	beatriz@user.com	user
3	luis@user.com	user
4	clara@user.com	user

### B. Architectural Design

In terms of good architectural design practices, it is always beneficial to separate business logic from UI logic. This separation enables each logical layer of the project to leverage a wide range of technological ecosystems for implementation, testing, and deployment. While this practice may increase development complexity, it significantly improves project scalability, minimizes error propagation to specific logical domains, and facilitates faster debugging.

Figure 2 illustrates an architecture where the UI logic is separated from the business logic, with the database also isolated. This design allows each logical part to operate independently, enabling the creation of separate environments and the installation of package dependencies for each without conflicts.

Additionally, this architecture isolates the database, permitting only the business logic to access it, while restricting the UI logic from direct access. This enhances security, maintainability, and scalability.

In Figure 2, the application flow is also illustrated. It begins with a client (web browser - Step 1) sending an HTTP request to the UI interface instance (Step 2), which is served by a web server (Nginx). If the UI interface needs to retrieve information from the business logic, it interacts with the REST API service instance (Step 3). The REST API service instance then retrieves the required data from the database instance (Step 4).

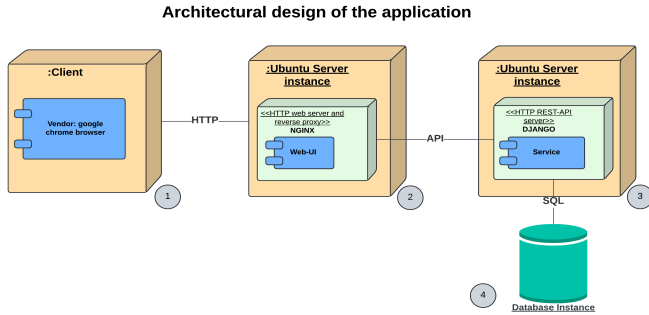


Fig. 2. The architectural design of the application illustrates the application flow and how requests are handled by users

## II. CONTINUOUS INTEGRATION, CONTINUOUS DELIVERY, AND DEPLOYMENT

In this section, we cover various aspects related to CI/CD, including the deployment architecture and security measures to protect our instances, as well as tools that help automate these tasks. We also explore the CI/CD workflow, detailing how each continuous stage is performed and the tools involved in each stage. Additionally, we examine the pipelines that make up the CI/CD workflow and the jobs carried out within each pipeline.

### A. Deployment Architecture and Security Aspects

Before starting with the CI/CD workflow and pipelines, it is important to set up the deployment infrastructure we will use. As cloud architects, it is our duty to analyse the application's architectural design (see Figure 2) and, based on that, propose a suitable cloud-based deployment architecture.

1) *Network Security Aspect:* In terms of security, as cloud architects, we must always identify which services are critical and which can be exposed to the internet, and then define the domain of each. Networking plays a key role in achieving this.

For this project, the network was divided into two public subnets and two private subnets to maintain availability zones across all instances. Figure 3 illustrates the network block, along with the private and public subnet blocks, and shows which instances are located in each one.

In this project, three types of instances were identified. One type only needs traffic on the private network, such as the database. Another type only needs to access the internet for

package updates and should only allow traffic from specific instances in the private network, like the REST-API service. The third type needs to be exposed to the public network to serve content, such as the NGINX instances. By placing the first two types of instances in the private network and configuring a NAT gateway, we can allow some instances to access the internet while ensuring that no instances on the internet can reach them.

Network and subnets

No	House billing network and subnets	Block	Instances
1	Network (VPC)	10.0.0.0/16	
2	Private subnet 1	10.0.1.0/24	RDS REST-API Service
3	Private subnet 2	10.0.2.0/24	
4	Public subnet 1	10.0.3.0/24	SonarQube NGINX proxy NAT instance
5	Public subnet 2	10.0.4.0/24	

Fig. 3. Network block and subnet blocks with their instances on each subnet

By configuring specific firewall rules, we can also restrict traffic to only the private network and prevent internet access for the database. See the firewall rules in Figure 4. This approach allows us to establish low-level access security between instances in the project.

Firewall rules

No	Group Security	Ingress			Egress		
		Ports	Protocol	Block	Ports	Protocol	Block
1	NAT instance	*	*	10.0.0.0/16(VPC)	*	*	0.0.0.0/0
2	RDS	3306	TCP	IT_ADMIN_IP REST-API service	*	*	10.0.0.0/16
3	REST-API service	*	*	10.0.3.0/24 10.0.4.0/24	*	*	0.0.0.0/0 via NAT instance
4	NGINX proxy	80 22	HTTP SSH	0.0.0.0/0 IT_ADMIN_IP	*	*	0.0.0.0/0
5	SonarQube	9000 22	HTTP SSH	0.0.0.0/0 IT_ADMIN_IP	*	*	0.0.0.0/0

Fig. 4. Firewall rules of all instances within the deployment infrastructure

2) *AWS Infrastructure Aspects:* To deploy all services onto instances, we chose to use Infrastructure as a Service (IaaS) in AWS. In Figure 5 (Deployment Architecture), we can see that we have two availability zones, ensuring the application's availability in case one of the networks fails. We also identified the following services in the application and placed them on separate instances to ensure isolation for each. Figure 4 (Firewall Rules) illustrates how the firewall rules add a security layer to prevent unauthorized access between instances within the deployment infrastructure.

- The NAT instance allows outbound traffic from the private subnet to the internet via an Internet Gateway, but restricts inbound traffic from the internet to the private subnet. This enables instances like the REST-API instance to update their packages and pull new features from repositories or registries. The NAT instance is located in the public subnet because it serves as an

interface to access the internet. See Figure 3 (Network Deployment) for a list of all subnets in the deployment infrastructure.

- The RDS instance is critical due to its role in storing the database. Therefore, we opted to delegate the administration of this instance to the AWS Support team via RDS. It is placed in the private subnet to limit access to only the REST-API instance and other instances within the private subnet. See Figure 4 for the firewall rules associated with this instance.
- The REST-API instance handles the business logic of the application, so access to this instance must be restricted to specific instances, such as the NGINX instance, which serves the UI interface and depends on it. It also requires outbound traffic to the internet for package updates. This is particularly important when performing the pipelines.
- The NGINX instance acts as a content server (serving the Angular UI interface) and a reverse proxy (for the REST-API service), redirecting all requests made by the Angular UI interface to the REST-API instance. In this way, if the NGINX instance is compromised by a hacker attack, they would be unable to access either the REST-API instance or the database instance.
- The SonarQube instance serves as the Static Code Analysis system, scanning for security vulnerabilities in the application code. Therefore, it needs to be in the public subnet to be accessible by the pipelines (e.g., GitHub Actions).

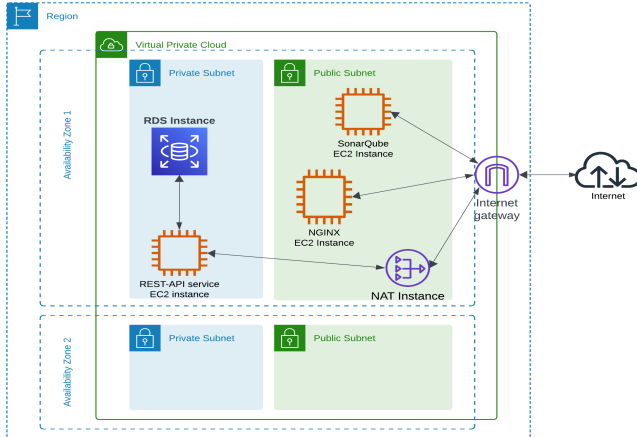


Fig. 5. AWS Deployment Architecture

3) *Infrastructure as a Code*: Infrastructure as Code was key in the deployment infrastructure because it allowed us to manage and automate the entire process of creating the deployment infrastructure. Figure 6 illustrates the Terraform outputs, showing the IP addresses of each service described in Section II-A2 (AWS Infrastructure Aspects).

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:
  house_billing_nat_instance = "3.235.233.134"
  house_billing_nginx_proxy_instance = "18.207.98.99"
  house_billing_sonarqube_instance = "35.153.156.39"
  house_billing_web_rest_api_instance = "10.0.1.168"
  rds_endpoint = "house-billing-db-instance.cizdihoh0eru.us-east-1.rds.amazonaws.com:3306"
  (venv) [aleksey@fedora-desktop ec2_RDS_MATInstance]$
```

Fig. 6. Terraform outputs that list all IP instance addresses within the AWS Deployment Infrastructure

## B. CI/CD Workflow

According to Redhat [1], Continuous Integration (CI) involves automating tasks such as building application components and testing their integration with each other. It also includes testing to ensure that each component functions properly without conflicts when new features are added to the software. Continuous Delivery (CD) is the automation of tasks related to releasing a new version of the software and publishing it to a repository or registry. Finally, Continuous Deployment (CD) automates tasks like deploying the new software features to the deployment infrastructure. In order to deploy these new features, the software must first pass through stages such as Continuous Integration and Delivery.

This subsection covers the entire CI/CD workflow and the tools used to implement it. Figure 7 illustrates the complete workflow and the tools used in each stage. It will be referenced in the following sections to provide more detailed explanations.

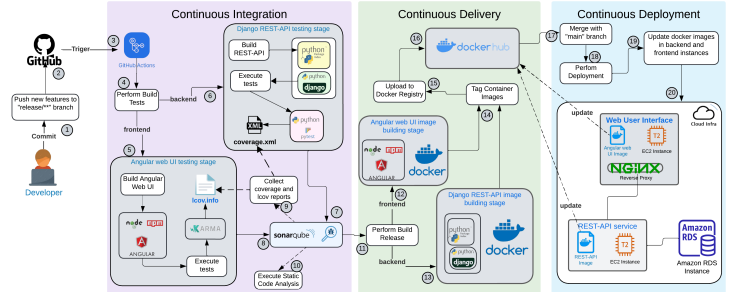


Fig. 7. CI/CD Workflow and tools used for House Billing App

1) *Continuous Integration Workflow*: This stage begins when a new commit is pushed to the release/\*\* branch on GitHub. On GitHub Actions, any commits to branches such as release/1.1.0, release/1.2.0, etc., trigger the "Perform Build" stage. During this stage, the application is built, and test cases for the Angular Web UI and Django REST-API are executed as a result of running these test cases, Figure 7 in steps 5 and 6 illustrates all the frameworks and tools used to execute the tests.

Files such as `coverage.xml` and `lcov.info` are generated. These files are important and will be referenced in the Static Code Analysis section (Section III). In step 7, SonarQube collects the `coverage.xml` and `lcov.info`

files, then executes static code analysis and publishes the results to the AWS SonarQube instance. See Section II-A2.

2) *Tools for Continuous Integration Workflow:* According to [2], Karma is a JavaScript tool for Single Page Applications that allows running tests in a testing environment simulating a web browser. This tool was important because it not only allows running the tests but also generates the `lcov.info` file, which is used by SonarQube. Figure 8 illustrates the performance of the Angular tests and the generation of the `lcov.info` file.

```

AUTHORS      dist      eze      node_modules package-lock.json reports  src
[alekey@fedora-desktop house-billing-web-user-interface]$ ng test --watch=false --code-coverage
✓ Browser application bundle generation complete.
06 12 2024 14:27:15.017:INFO [karma-server]: Karma v6.4.4 server started at http://localhost:9876/
06 12 2024 14:27:15.021:INFO [launcher]: Launching browsers ChromeHeadless with concurrency unlimited
06 12 2024 14:27:15.031:INFO [launcher]: Starting browser ChromeHeadless
06 12 2024 14:27:17.150:INFO [Chrome Headless 131.0.0.0 (Linux x86_64)]: Connected on socket xUIvWSynx
LOG: 'Petición Saliendo'
LOG: 'Se agregara HEADER AUTH'
Chrome Headless 131.0.0.0 (Linux x86_64): Executed 18 of 18 SUCCESS (0.1 secs / 0.078 secs)
TOTAL: 18 SUCCESS
[alekey@fedora-desktop house-billing-web-user-interface]$

```

Fig. 8. Performance of Angular tests with Karma and generation of `lcov.info` file

PyPi [3] and Pytest [4] were also fundamental in building and testing the Django REST-API service, allowing the execution of test cases and the generation of the `coverage.xml` file, which is used by SonarQube. Figure 9 shows the successful execution of all Django tests with Pytest and the generation of the `coverage.xml` file.

```

(venv) [alekey@fedora-desktop house-billing-rest-service]$ pytest --cov --cov-report=xml
===== test session starts =====
platform linux -- Python 3.12.6, pytest-8.3.4, pluggy-1.5.0
django: version: 4.2.14, settings: billbuddy.settings.testing (from env)
rootdir: /home/alekey/trabajos_de_cursos/NCI/DevOpsSec/house-billing-rest-service
configfile: pytest.ini
plugins: cov-6.0.0, Faker-33.1.0, factoryboy-2.7.0, django-4.9.0
collected 30 items

tests/authentication_tests.py ...
tests/expense_models_tests.py ...
tests/expense_serializers_tests.py ..
tests/expense_tests.py ...
tests/group_models_tests.py ...
tests/group_serializers_tests.py ..
tests/group_tests.py ...
tests/profile_tests.py .....
tests/user_models_tests.py ...
tests/user_serializers_tests.py ..

----- coverage: platform linux, python 3.12.6-final-0 -----
Coverage XML written to file coverage.xml

===== 30 passed in 8.88s =====

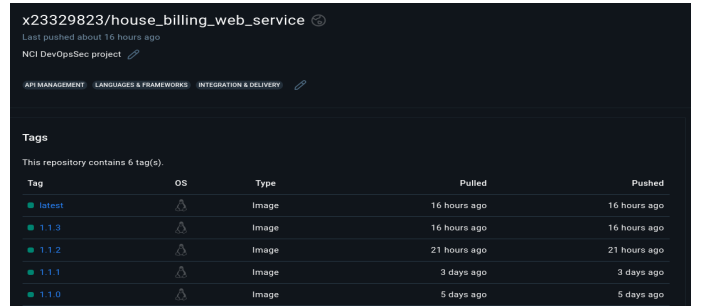
```

Fig. 9. Performance of Django tests with Pytest and generation of `coverage.xml` file

3) *Continuous Delivery Workflow:* This stage continues in step 11, Perform Build Release when all tests have been executed successfully and SonarQube has already submitted the status of the Static Code Analysis report. In step 12, the Angular Web UI build stage is performed, installing all dependencies required by the Angular framework through Node Package Manager (NPM). The result of this step is a Docker image containing all dependencies and new features of the Angular Web UI, ready to be deployed. The Django REST-API is also built following the same process, installing all dependencies required by Django through PyPi and creating a Docker image with the new features. Once both images are built, they are tagged with the `release/**` branch name and uploaded to DockerHub, ready for deployment.

4) *Tools for Continuous Delivery Workflow:* According to [5], a container is a standard unit of software that packages up code and all its dependencies, allowing the application to run quickly and reliably across different computing environments. This technology is suitable for creating executable components of the project. In Continuous Delivery, containers were key in packaging technologies such as NodeJS, NPM, Angular, and Webpack within the Angular Web UI image, and technologies such as Python and Django within the Django REST-API image.

Another crucial part of the Continuous Delivery stage was pushing every new version of the images to DockerHub. This allows the AWS instances to be updated and also serves as a versioning registry. Figure 10 illustrates the Angular Web UI image and its various versions.



Tag	OS	Type	Pulled	Pushed
latest	linux	Image	16 hours ago	16 hours ago
1.1.3	linux	Image	16 hours ago	16 hours ago
1.1.2	linux	Image	21 hours ago	21 hours ago
1.1.1	linux	Image	3 days ago	3 days ago
1.1.0	linux	Image	5 days ago	5 days ago

Fig. 10. List of all versions of Angular Web UI docker image pushed on DockerHub

5) *Continuous Deployment Workflow:* Once a new version of the software is released in subsection II-B3, the next step is to merge the new features into the main branch. This branch performs tasks such as pulling the new Docker images from Docker Hub and then deploying them into the AWS deployment infrastructure.

### C. CI/CD Pipelines

To understand how the pipelines work, it is important to know how the project is divided into repositories and what branches exist in each one.

The project is divided into two repositories: the Django REST API repository and the Angular Web UI repository. Each repository includes two key branches: `release/**` and `main`. `release/**` branch is used to push new features of the project and manage release versions, such as `release/1.2.0`, `release/1.2.1`, `release/1.2.2`, and so on. On the other hand, the `main` branch is used to push new tested versions of the repository.

1) *release/\*\* branch pipeline:* This pipeline allows the execution of tasks related to Continuous Integration and Continuous Delivery, such as running tests, performing Static Code Analysis, building a publishing a new Docker release, and creating a pull request to the `main` branch. See Sections II-B1 and II-B3.

Figure 11 illustrates all the jobs that the `release/**` branch has



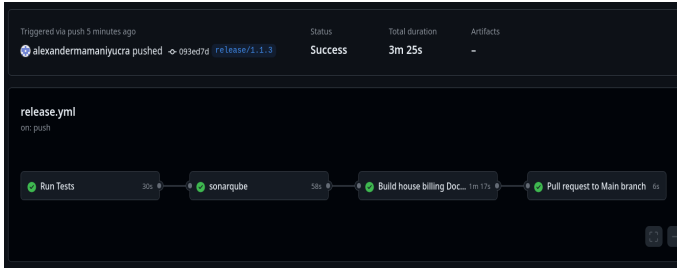


Fig. 11. Execution of release/\*\* branch pipeline jobs

2) *main branch pipeline*: This pipeline performs tasks such as connecting to the AWS instance, stopping the Docker containers, pulling the new Docker image released in the previous pipeline, and finally restarting the Docker container. Figure 12 illustrates this pipeline in the Django REST-API repository.

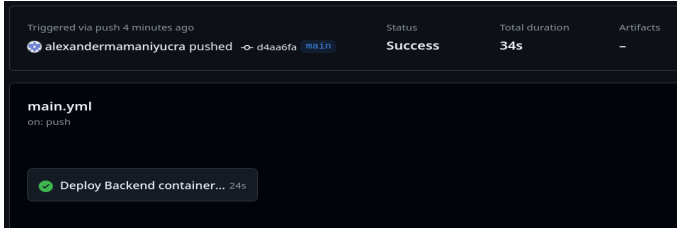


Fig. 12. main branch pipeline jobs

### III. STATIC CODE ANALYSIS

According to [6], SonarQube Community is an open platform that helps inspect and improve code quality. It provides comprehensive code analysis by identifying potential bugs, security vulnerabilities, code smells, and duplications in codebases. SonarQube supports multiple programming languages and integrates seamlessly into CI/CD pipelines.

Figure 13 shows the current status of the project in terms of code quality, highlighting vulnerabilities such as 0% of the code not covered by tests, multiple vulnerable hotspots, code smells, duplicated code lines, and others. The following sections dive into more detail about these issues and how they were fixed in the project.

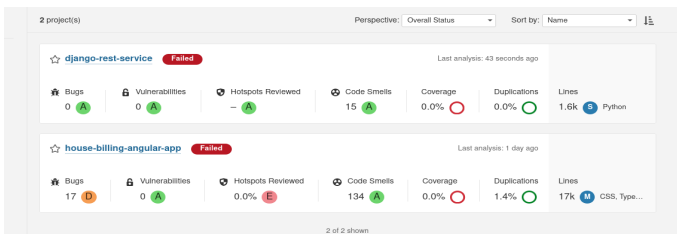


Fig. 13. Initial status of Static Code Analysis

The SonarQube system is available in a public IP [Here](#) and the credentials are listed in the “Table. II”.

TABLE II  
SONARQUBE CREDENTIALS TABLE

SonarQube Credentials		
No	User	Password
1	admin	admin23329823

#### A. Test coverage reports

Testing execution is key to ensuring code quality, SonarQube lets us generate test coverage reports, which show the percentage of our code covered by test cases. It also tells us which tests have been run and their results.

According to the SonarQube Quality Gate, 80% of the project code must be covered by tests to be considered a reliable and production-ready project. Figure 14 shows that only the Django REST-API repository satisfies this requirement after executing its tests. See Figure 9 for the performance of Django REST-API tests.

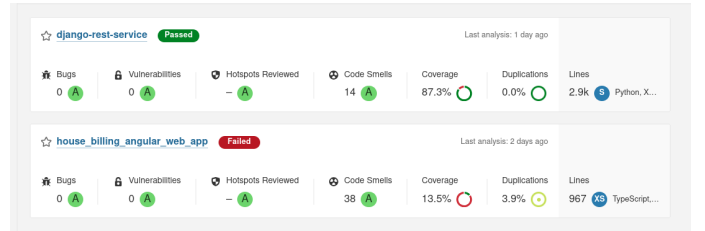


Fig. 14. Status of coverage after implementation testing

#### B. Security Hotspots

A security hotspot is a piece of code that needs to be reviewed and fixed because it might become a security vulnerability. It helps create secure code by avoiding potential security bugs [7].

Figure 15 highlights that this piece of code is exposing sensitive information. This error occurs when we attempt to hard-code user credentials to test JWT.



Fig. 15. Hostpot that alerts you that sensitive information is being exposed in code

The solution for this hotspot is to create fake credentials, preventing the exposure of information that could be used by hackers. By faking the credentials, we delegate this task to the Pytest library, which generates the credentials dynamically. Figure 16 illustrates the solution for this hotspot.

```

def setup(self):
    self.admin_email = fake.email()
    self.user_email = fake.email()
    self.user_password = fake.password()
    self.admin_password = fake.password()
    # ...
    self.url = reverse("token_obtain_pair")
    # create superuser
    self.superuser = User.objects.create_superuser(email=self.admin_email, password=self.admin_password)
    self.superuser.save()
    # create user
    self.user = User.objects.create_user(email=self.user_email, password=self.user_password)
    self.user.save()

```

Fig. 16. Piece of code that fixes that hotspot, faking user credentials

### C. Duplication code

Based on the SonarQube analysis, see Figure 17, the project had a duplication rate of 17.2%, with significant duplication across files such as `modal-create-expense.component.ts` (68.3%), `modal-settle-expense-up.component.ts` (66.1%), `modal-list-settle-up.component.ts` (56.7%), and `modal-create-group.component.ts` (47.3%). These high duplication percentages indicate repeated code, which can lead to maintenance challenges.

Duplicated Lines (%)	Duplicated Lines (%)	Duplicated Lines
68.3%	66.1%	56.7%
47.3%		

Fig. 17. List of files that have duplicate lines

We implemented the extract-superclass refactoring in these classes as a solution by moving the repeated code to a superclass (`ModalOpenParent` class) and making all these classes extend from it. Figure 18 illustrates the code where we implemented the extract-superclass refactoring.

```

export class ModalCreateExpenseComponent extends ModalOpenParent implements OnInit {
    @Input() groupId: string;
    public members_and_admins_of_group: any = {};
    form: FormGroup;

    constructor(private fb: FormBuilder, private modalService: NgbModal, private router: Router, private expenseService: ExpenseService, private userService: UserService) {
        super();
    }

    ngOnInit() {
        // ...
    }

    create_expense() {
        // ...
    }
}

```

Fig. 18. Implementation of extract-superclass on `ModalCreateExpenseComponent` as a solution of duplication code

### D. Code Smells

The implementation of a for loop with index-based iteration is considered a code smell because it introduces unnecessary complexity for simple array-like objects. Figure 19 shows that SonarQube flagged the use of a for loop as a code smell, suggesting the use of a for-of loop for better readability and simplicity.

```

// ...
for (let i = 0; i < membersnode.length; i++) {
    let id_user = membersnode[i].id;
    let amount_user = membersnode[i].getElementsByTagName("input")[0].value;
    this.addItem(id_user, amount_user);
}
// ...

```

Fig. 19. For loop bad smell with an index-based iteration that adds complexity to object list iteration

Figure 20 illustrates how the code was refactored to use a for-of loop instead. This change simplifies the iteration logic by directly accessing each element in the `membersnode` collection without requiring an explicit index. This fix makes the code more concise and less error-prone.

```

// ...
for (const memberNode of membersnode) {
    let id_user = memberNode.id;
    let amount_user = memberNode.getElementsByTagName("input")[0].value;
    this.addItem(id_user, amount_user);
}
// ...

```

Fig. 20. For-of loop implementation as a solution of for loop bad smell instead

## IV. CONCLUSIONS

After implementing static code analysis, I noticed an increase in the application's security, as it forces developers to correct errors and vulnerabilities. Although only one of the applications passed this filter, the other application is now more secure than its version prior to implementing static code analysis. See Figure 21.

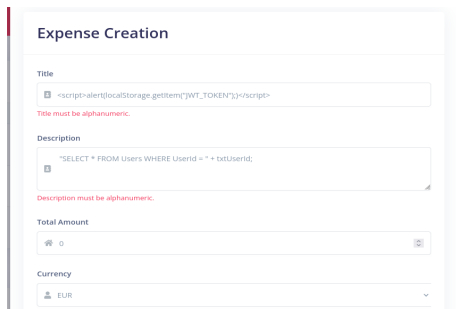
Regarding CI/CD, there was also an improvement in the execution of tests, integration, and rapid deployment, preventing new features from conflicting with existing ones. This also facilitated the agile deployment of the application, achieving a performance of just a few minutes for the release and deployment of a new software version.

django-rest-service	Passed	Last analysis: 40 minutes ago
Bugs	0	2.9k Python X...
Vulnerabilities	0	
Hotspots Reviewed	14	
Code Smells	87.3%	
Coverage	0.0%	
Duplications	0	
Lines	2.9k	

Fig. 21. Current status of Application on SonarQube

It was also concluded that security should not only come from systems like static code analysis but should also be implemented from the graphical interface by validating and limiting what the user enters into the system. This can prevent

vulnerabilities such as SQL injection. Figure 22 illustrates the validation of these fields.



The screenshot shows a web form titled "Expense Creation". It has four input fields: "Title", "Description", "Total Amount", and "Currency". The "Title" field contains the text: `<script>alert(localStorage.getItem('JWT_TOKEN'))</script>`. Below it, a red error message says "Title must be alphanumeric.". The "Description" field contains the text: `"SELECT * FROM Users WHERE UserId = " + txtUserId;`. Below it, a red error message says "Description must be alphanumeric.". The "Total Amount" field is empty. The "Currency" field has a dropdown menu with "EUR" selected.

Fig. 22. SQL injection attempt through a form field in the application

As a next step, it is necessary to implement different stages of software execution, such as testing and staging. While this involves an additional cost to the project due to the need to replicate the entire infrastructure for each stage, it will allow for more thorough testing of the project, enabling qualified personnel to perform black-box testing focused on functionality, as white-box testing is carried out by the pipelines.

In terms of infrastructure security, the SonarQube instance needs to be moved from the public subnet to the private subnet, and the infrastructure for a VPN must be set up. This will ensure that the SonarQube instance can only be accessed by authorised personnel. Similarly, the NGINX reverse proxy must be configured so that the Django admin panel can only be accessed via this VPN.

## REFERENCES

- [1] Redhat, "What is CI/CD?," Redhat.com, Dec. 12, 2023. [Online]. Available: <https://www.redhat.com/en/topics/devops/what-is-ci-cd>
- [2] KarmaJs, "Karma - How It Works," Github.io, Dec. 07, 2024. [Online]. Available <http://karma-runner.github.io/6.4/intro/how-it-works.html>
- [3] PyPi, "The Python Package Index (PyPI) — documentación de Python - 3.6.15," Python.org, 2021. Dec. 07, 2024. [Online]. <https://docs.python.org/es/3.6/distutils/packageindex.html> (accessed Dec. 07, 2024).
- [4] PyTests, "pytest: helps you write better programs — pytest documentation," docs.pytest.org. Dec. 07, 2024. [Online]. <https://docs.pytest.org/en/stable/>
- [5] Docker, "What is a Container?," Docker, 2023. Dec. 07, 2024. [Online]. <https://www.docker.com/resources/what-container/>
- [6] SonarQube, "SonarQube Documentation" 2023. Dec. 07, 2024. [Online]. <https://docs.sonarsource.com/sonarqube-server/9.9/>
- [7] SonarQube, "SonarQube Security Hotspots" 2023. Dec. 07, 2024. [Online]. <https://docs.sonarsource.com/sonarqube-server/10.6/user-guide/security-hotspots/>
- [8] Guru, "Extract Superclass," refactoring.guru. Dec. 07, 2024. [Online]. <https://refactoring.guru/extract-superclass>