UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL DE CALCULATOARE

PROIECT DE DIPLOMĂ

AcadNet.dev
Platformă online pentru rezolvarea problemelor de informatică

Dimitrie David

Coordonator științific:

Prof. Dr. Ing. Răzvan Victor Rughiniș

BUCUREȘTI

2023

UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

DIPLOMA PROJECT

AcadNet.dev
Online platform for solving computer science problems

Dimitrie David

Thesis advisor:

Prof. Dr. Ing. Răzvan Victor Rughiniș

BUCHAREST

2023

CONTENTS

SINOPSIS

Această lucrare prezintă AcadNet.dev, o platformă online pentru rezolvarea de probleme de informatică. Platforma oferă un mediu de lucru complet, care permite utilizatorilor să creeze probleme, să le rezolve și să le evalueze automat. Platforma oferă și un mediu de lucru online, care permite utilizatorilor să rezolve problemele direct în browser, fără a fi nevoie să configureze un mediu de dezvoltare local.

ABSTRACT

This paper presents AcadNet.dev, an online platform for solving programming problems. The platform offers a complete working environment, which allows users to create problems, solve them and automatically evaluate them. The platform also provides an online working environment, which allows users to solve problems directly in the browser, without the need to configure a local development environment.

1 INTRODUCTION

1.1 Motivation and problem statement

For the past half year, I was responsible for coordinating the development of problems for the Software Interoperability section of the [National Olympiad of Applied Informatics - Acadnet](). This section is different than the regular informatics olympiad, as it focuses more on the engineering side of informatics, rather than the theoretical side. The problems are more practical and require students to be more creative in order to solve them.

The problems are formulated as real life scenarios, where a code is given that should have a certain behavior, but it does not. The students have to find bugs in the code and fix them.

As of right now, there is no accessible methods for students to train for this olympiad. The only way to practice is to solve the problems from the previous years, by downloading their statement and original source. There are no tests to check if the solution is correct. The students have to compile and run the code themselves, and check if the output is correct.

The platform's goal is to create an environment where students can train and prepare for the olympiad in a more efficient way. Moreover, we want to create an online workspace for students, where they can solve problems directly in the browser. This allows us to get more creative with the engineering problems, as we can use more programming languages and configurations, without putting the students through the hassle of setting up a local development environment.

1.2 Objectives

During the last half year, I have been gathering insights on what the students and authors want from a platform like this. I have also been researching the available technologies, and I have been experimenting with different approaches. Based on this, the objectives of this project are defined as follows.

The objective of this project is to create a universal platform for solving engineering tasks. This should include the ability for authors to extend the platform and implement any new language that they want to write a problem in. In addition, the platform should give students the opportunity to solve the tasks directly in the browser, without requiring anything more than an account. In terms of functionality, the platform should allow authors to create problems, and students to solve them. The platform should also provide a way to automatically evaluate

the solutions, and give feedback to the students.

In terms of security, because the solutions will be evaluated by executing user-written code, the platform should be able to run the code in a sandboxed environment, and prevent malicious code from being executed. The platform should also prevent students from cheating, by not allowing them to see the test cases, other users submissions, or the source code of the solutions.

The platform should be easily maintainable, and should be able to scale to a large number of users. It should use containerization to allow for easy deployment and scaling.

1.3 Proposed solution and achieved results

The core of the platform is a web application, which acts as the interface between the users and the platform. The web application is responsible for managing the users, problems, submissions, and for evaluating the submissions. In addition to this, the web application will also be a proxy for online workspaces.

The web application is backed by a SQL database and an S3 file storage. The database is used to store the users, problems, submissions, and other metadata. The file storage is used to store the source code of the problems, and the submissions.

The web application is written in C# using the .NET Core framework. The database is a PostgreSQL database, and the file storage is an S3 compatible storage, provided by DigitalOcean. The web application is deployed using Helm Charts on a Kubernetes cluster, also provided by DigitalOcean. The checker is written in Python and it is more like an sandbox manager that spawns a new container for each submission, and runs the tests inside the container. Like the checker, the workspace manager is also written in Python, and it is responsible for spawning new workspaces. Workspaces are Docker images based on a Visual Studio Code fork, that allows us to run the code directly in the browser.

The working platform can be found at AcadNet.dev and the source code is open-source and available on Github inside the acadnet-dev organization.

2   SYSTEM DESIGN AND ARCHITECTURE

2.1   User journey
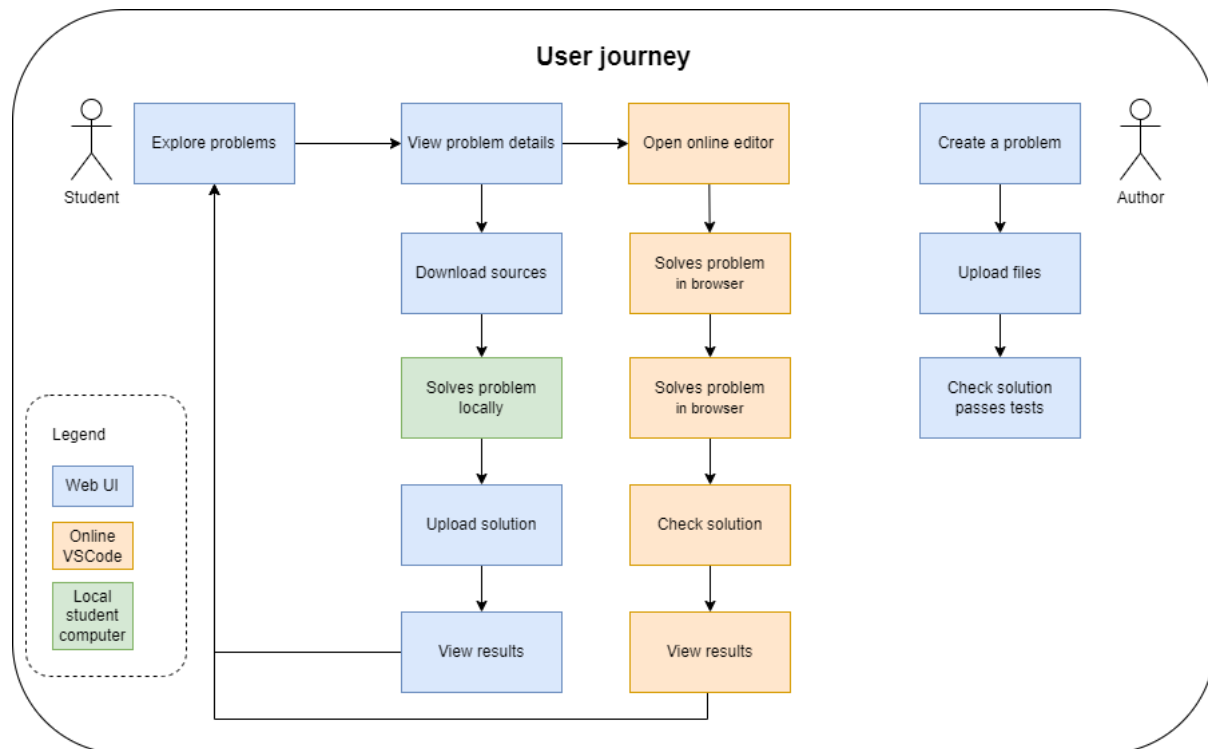
Figure 1: User journey

The platform's main actors are the users that want to learn and practice programming, by solving the problems available. The students can browse the problems, but in order to solve them, they have to create an account. After that, they can choose to solve the problems directly in the browser or they can download the source code and solve them locally. After they solve the problem, they can submit their solution for evaluation. The platform will automatically evaluate the solution, and give feedback to the them.

The other actors are the authors, who are responsible for creating the problems. The authors can create problems, and upload relevant files for the problem. These files include the statement, the source code of the problem, the test cases, and the solution.

2.2 Architecture overview

The architecture is composed of 3 main components: the web application, the checker, and the VSCode workspaces. The web application is the main component, and it is responsible for managing the users, problems, submissions, and for evaluating the submissions. The checker is responsible for evaluating the submissions, and the VSCode workspaces are used to allow users to solve the problems directly in the browser. The block diagram of the architecture can be seen in the figure below.
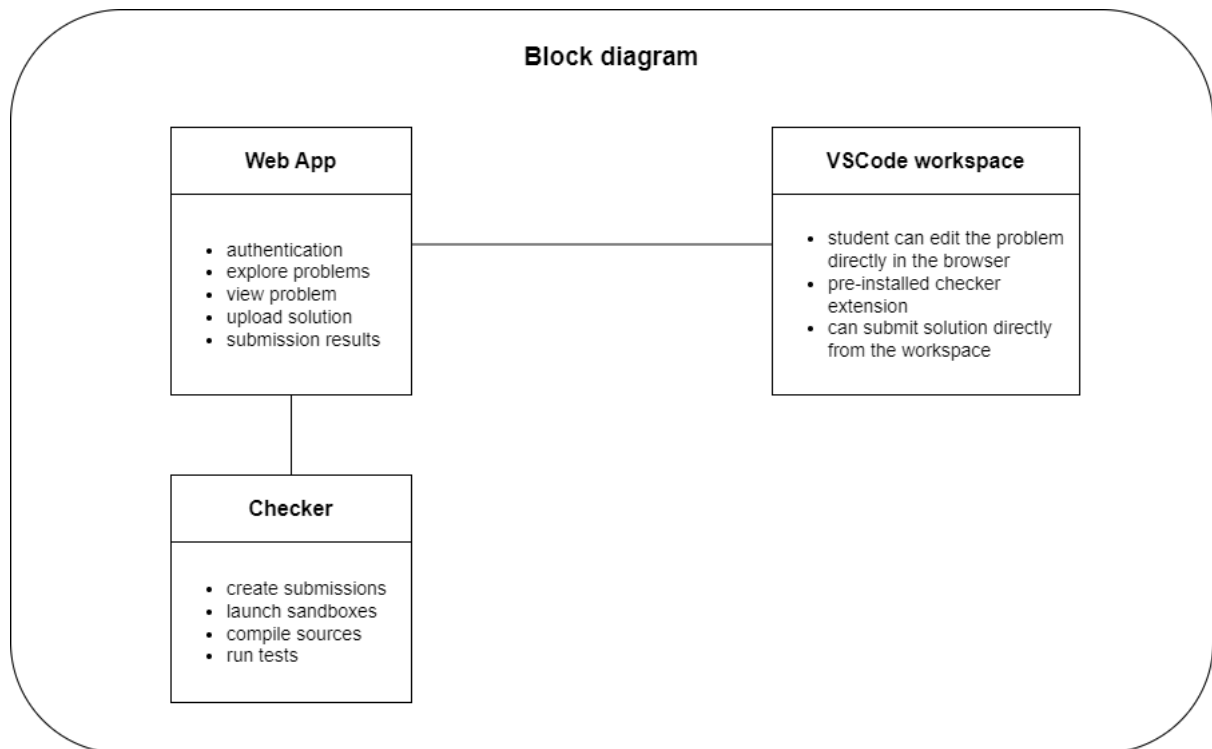
Figure 2: Software block diagram

Table 1: Component's responsibilities

Component	Responsibilities
Web App	Manage users, problems and submissions; send submissions to the checker and present results; manage VSCode workspaces
Checker	Test submissions by spawning a new sandbox in the K8s cluster and compare reference outputs with actual outputs
VSCode workspace	Allow users to solve problems directly in the browser

2.3 System components and interactions

The web app is the brain of the platform. It is interacting with all the other components. With regards to the checker, the web app is responsible for receiving the submissions from the user or from the online workspace and to send them to the checker. While the checker is evaluating the submission, the web app constantly polls the checker for the updates and results. When the checker is done, the web app will receive the results and will update the submission with the status. On the other hand, when it comes to the interaction with the online workspaces, the web app is responsible for spawning new workspaces via the workspace manager and it acts as a HTTP and websocket proxy between the user's browser and the workspace.
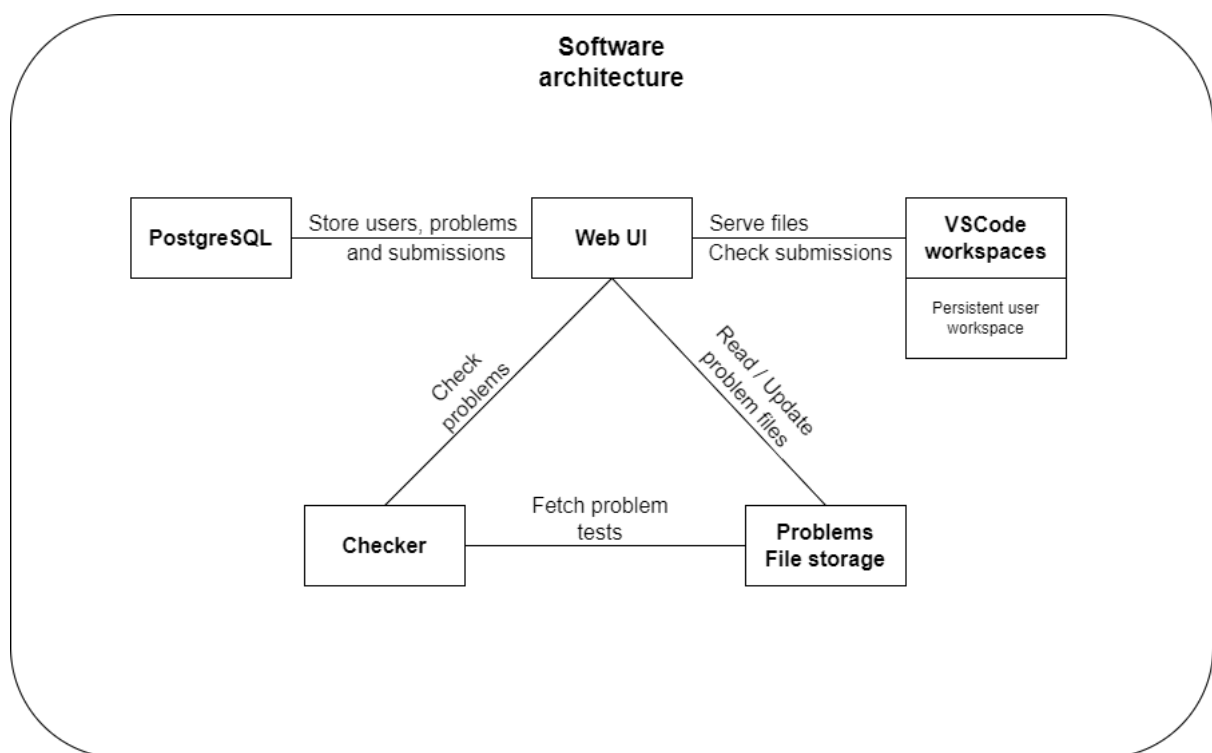
Figure 3: System components and their interactions

2.3.1 SQL and file storage

Other than the 3 main components, the architecture also includes a SQL database and an S3 file storage. The database is used to store the users, problems, submissions, and other metadata. The file storage is used to store the source code of the problems, and the submissions. The interactions between the components can be seen in the figure below. The web application is the only component that interacts with the database. With the file storage, the web application and the checker interact with it. The web application uses it to store the files uploaded by the authors, and the checker uses it to pull the test cases for the problem.

2.4   Database design

The database is designed using an ORM (Object Relational Mapper) called Entity Framework. This allows us to define the database schema using C# classes, and because we are using a "code first" approach, the database tables and relations will be created automatically, using migrations. The database schema can be seen in the figure below.
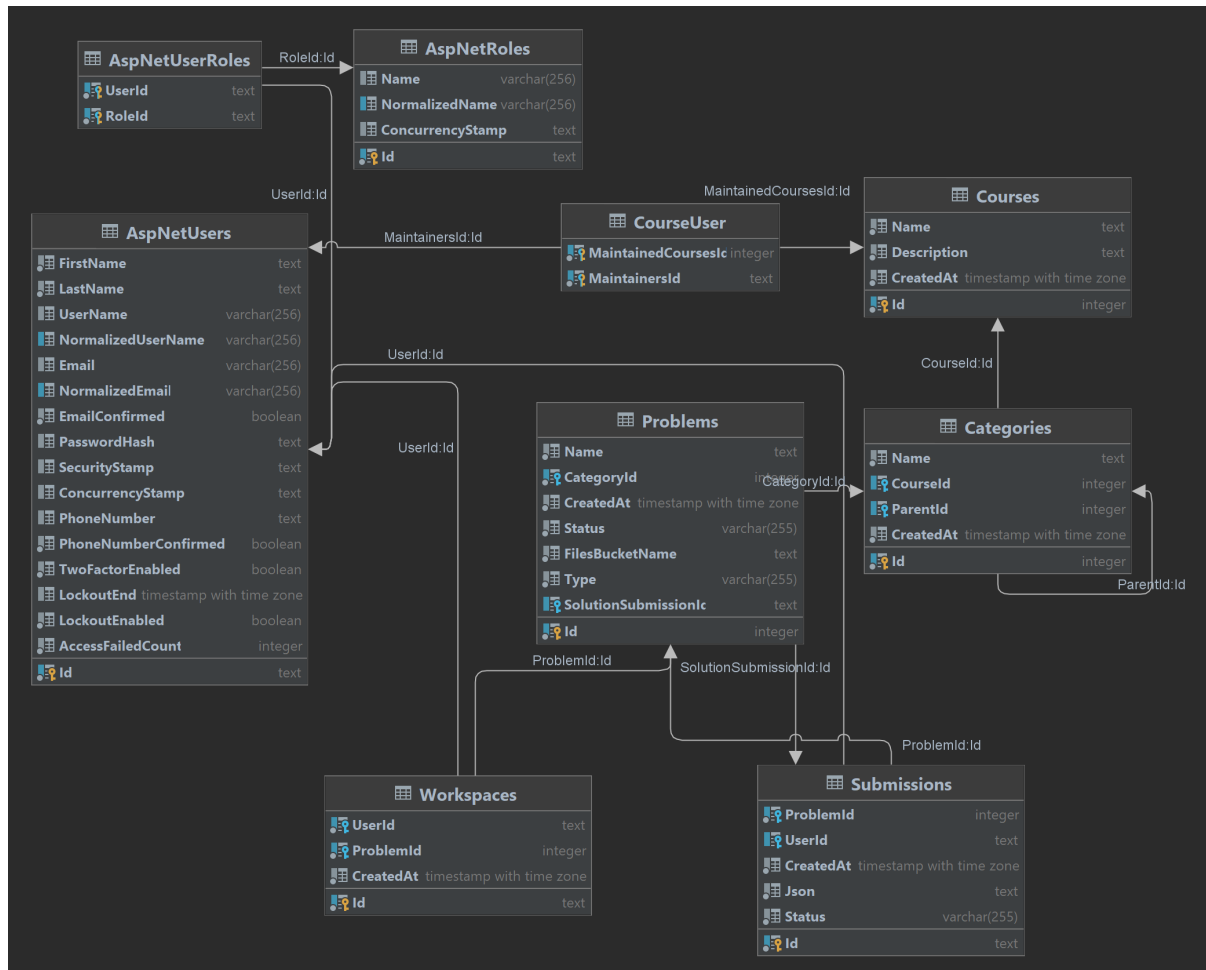
Figure 4: Database schema diagram

As we can see in Figure 4, the database is composed of multiple tables. Some of them are automatically generated by the .NET Identity Framework that handles authentication.

Below is a description of each table and its purpose.

2.4.1 Users and roles

- **AspNetUsers** - stores the users that have an account on the platform. The users can be either students or authors.
- **AspNetRoles** - stores the roles available on the platform. Currentyl, there are only 2 roles: student and author.
- **AspNetUserRoles** - because the realtion between users and roles is many to many, this table is used to store what roles each user has.

2.4.2 Problem hierarchy

The problems are organized in a tree-like structure. As the root there are the courses that have a name and a description. Each course have multiple categories, and each category can have as children other categories or problems. An example of the hierarchycal structure is shown in Figure 5.
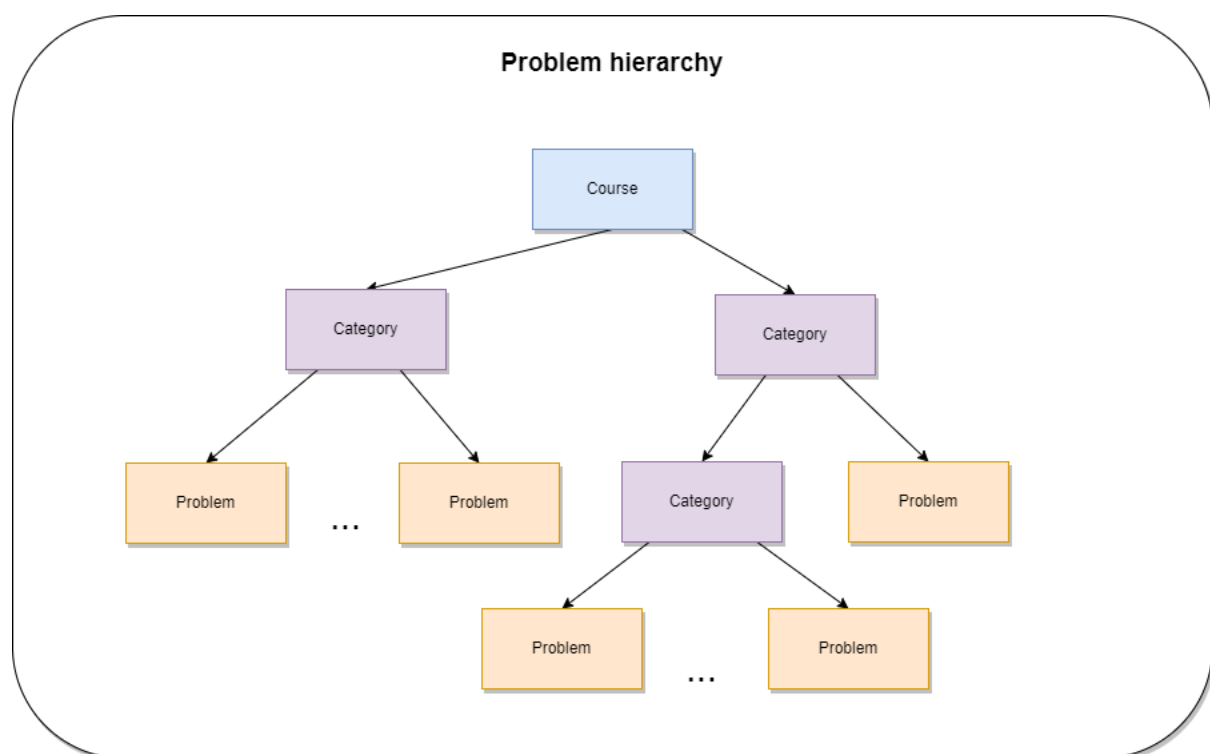
Figure 5: Problem hierarchy

- **Courses** - stores the courses available on the platform.
- **CourseUser** - a course can have maintainers. This table stores the relation between the courses and the users that are maintainers.
- **Categories** - stores the categories available on the platform. Each category have a course and may have a parent category.

- **Problems** - stores the problems available on the platform. Each problem have a category. Also, the problem stores the bucket id of the file storage where the problem's files are stored.

2.4.3   Submissions and workspaces

- **Submissions** - stores the submissions made by the users. Each submission have a problem and a user.
- **Workspaces** - stores the workspaces available on the platform. Each workspace have a problem and a user.

3 IMPLEMENTATION DETAILS

All the code for the platform and all its components is available on Github inside the acadnet-dev organization.

3.1 Web application ⭕ **Repository**

The web application is the main component of the platform. It is responsible for managing the users, problems, submissions, and for evaluating the submissions. In addition to this, the web application acts as a proxy for the online workspaces. The web application is written in C# using the .NET Core framework. It is using the MVC (Model-View-Controller) pattern, and it is using an ORM (Object Relational Mapper) called Entity Framework to interact with the database. The web application is deployed as a Docker image using multi-stage builds. The Docker image is deployed on a Kubernetes cluster using Helm Charts. The web application is composed of multiple components, which are described below.

3.1.1 Controllers

The controllers are the components that handle the HTTP requests. They receive a request from the user, process it by calling different services and interacting with other components so that it returns a response. The platform has multiple controller, each acting in its own domain. The controllers are described below.

AuthController It handles all the actions related to user authentication and authorization. It is responsible for registering new users, logging in existing users, and managing the user's roles.

Registering users and logging in users is done via external authentication providers. Currently, the platform only supports Google authentication, but it can be easily extended to support other providers. Support for other popular external providers is already implemented in the .NET framework.

CoursesController It handles all the actions related to courses like creating new courses. This action can be done only by users that have the author role. It also handles creating categories and listing the hierarchycal structure of the courses so that users can browse them.

ProblemsController This controllers controls all the actions related to problems. It handles creating new problems, showing the problem's statement, and evaluating the submissions.

Problems can only be created if the user created the respective course and is a maintainer of the course.

Submitting a solution is done by uploading a file with the solution's source code. The file is sent to the checker. There is also a 'GET' endpoint that returns the submission's status. This endpoint is called periodically by the frontend to check if the submission is done.

WorkspaceController This controller is responsible for managing the online workspaces. It handles creating new workspaces, and sets the environment for the proxy to connect to the workspace. More details about the proxy and how the workspaces are accessed by the user's browser can be found in Section 3.1.4.

3.1.2 UI

The UI is built using the Razor Pages framework. It is created based on a template from Theme Forest. In terms of technologies, the UI uses SASS, jQuery, and Bootstrap. The UI is responsive and it works on both desktop and mobile devices.

Some of the interesting elements that the UI has are described below.

Problem statement The problem author's are required to upload a Markdown file with the problem's statement. The UI is using the Markdig library to convert the Markdown file to HTML. The HTML is then rendered in the browser using custom CSS styles for the different elements in Markdown.

File upload The file upload input is a custom component that is using the FilePond library. This library is responsible for creating a responsive and dynamic file upload input. It lets authors upload multiple files at once and remove already uploaded files. It also has a drag and drop feature, and it can be easily extended to support other features like file validation, file preview, and image editing.
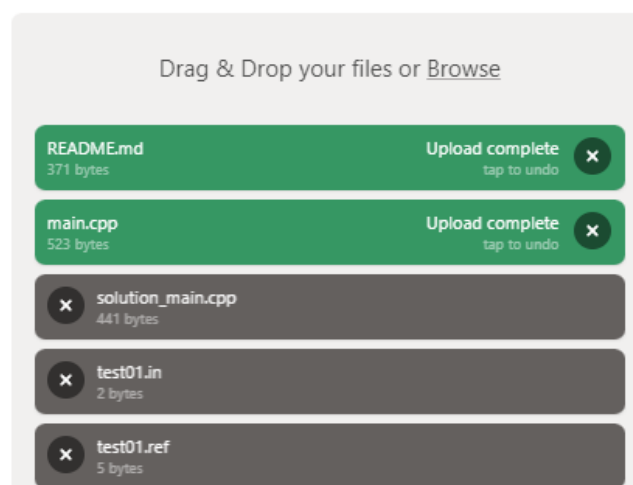
Figure 6: File upload component

10

3.1.3   Services

The services are the components that handle the business logic of the platform. They are used by the controllers to process the requests. They are responsible for interacting with the database and the file storage. Services are a pair made of an interface describing the available methods, and a class implementing the interface. Services are using a dependency injection mechanism to be injected into the controllers and other services. The functionality of each service is described below.

FileService This service is responsible for interacting with the file storage. It is used to create buckets[1] and upload and download files from the file storage.

ProblemService This service handles all the actions regarding problems. It is implemented using the Factory design pattern, because the problems can be written in different languages and have different methods of evaluation and initialization. Currently, the platform implements only one type of problem with one file source code of C++ and tests that are based on finding differences between the reference output and the actual output. But it can be easily extended to support other types of problems, written in other languages, and with other methods of evaluation like asserting the execution time or the memory usage.

Some of the methods that needs to be implemented for each type of problem are described below.

- creating a problem
- getting a problem by id
- getting a problem's statement and converting it from Markdown to HTML
- get the sources of a problem to be downloaded by the user
- get the test cases of a problem so that they can be sent to the checker
- creating and updating a submission
- getting the necessary files to initialize a workspace

CheckerService This service is responsible for interacting with the checker. It is used to create submissions and to get the status of a submission.

WorkspaceService This service is responsible for interacting with the workspace manager. It is used to create workspaces and to get the url of a workspace for the proxy redirect.

3.1.4   Workspaces Proxy

The workspaces proxy is a component that acts as an intermediate between the user's browser and the workspace. It is responsible for receiving user requests for the workspace and redirect-

---

[1]A bucket is a collection of files in an S3 compatible storage

ing them to the appropriate workspace container. It also handles the websocket connections between the user's browser and the workspace container.

The proxy is written as a middleware in .NET Core and it is part of the web application. Because of this, it is very easy to check if the user is authenticated and authorized to access the workspace, because the authentication cookies are already in the user's browser and are transmitted with each request.

3.2   SQL database

The SQL database is a PostgreSQL database. It is deployed as a managed service on DigitalOcean. The database is used to store the users, problems, submissions, and other metadata. The database schema can be seen in Figure 4. The web application is the only component that interacts with the database. It is controlled using Entity Framework, which is an ORM (Object Relational Mapper) that allows us to define the database schema using C# classes. The database tables and relations are defined in a "code-first" approach, which means that the tables and relations are created by the ORM by translating the C# classes and relationship between them to tables and keys.

Each modification to the database schema is handled by migrations. Migrations are C# classes that describe the changes that need to be made to the database schema. The migrations are created automatically by the ORM, and they are applied automatically when the application starts. This allows for easy modifications on the database schema without having to manually create the tables and relations.

3.3   File storage

The platform needs a distributed storage method for the files uploaded by the authors and the submissions made by the users. An S3 compatible storage is a good solution for this, because it is easy to use and it is scalable. For this project, when developing we are running a local S3 compatible storage using MinIO. For production, we are using a managed S3 compatible storage provided by DigitalOcean called Spaces.

The storage is accessed using Amazon's S3 C# SDK. The SDK is used to create buckets, upload and download files, and list files in a bucket. The buckets are created automatically by the web application when a new problem is created. The bucket's name is a UUID. The files are uploaded with specific names, defined by the problem type.

For example, for the C++ problems, the files are uploaded with the following names.

- **main.cpp** - the source code of the problem
- **soltion_main.cpp** - the source code of the solution
- **README.md** - the statement
- **test0.in** - the first test case
- **test0.ref** - the first test case's reference output
- **test1.in** - the second test case
- **test1.ref** - the second test case's reference output
- **...**

3.4 Checker Repository

The checker is the component of the platform that handles the evaluation of the submissions. It is written in Python and it is exposed as a simple REST API. The checker is deployed as a Docker image on the Kubernetes cluster. The checker is composed of multiple components, which are described below.

3.4.1 API

The API is built using the FastAPI framework. It is a simple REST API that exposes 2 endpoints: one for creating a new submission, and one for getting the status of a submission. The API is responsible for receiving the submission from the web application, and for sending the results back to the web application.

The API endpoints are described below.

POST /submission/create

Query parameters

- **type** - the type of the problem
- **bucket** - the bucket id where the problem's files are stored

Request body

- **file** - the file with the source code of the submission as a multipart form data

Response body

- **submission_id** - the id of the newly created submission

GET /submission/status

Query parameters

- **submission_id** - the id of the submission

Response body

- **status** - the status of the submission

An example of a submission status for a C++ problem can be seen in the appendix A.1. Important things to be noted are the presence of the compilation status, the expected and actual results for each test case and all the logs from the sandbox.

3.4.2   Sandbox adapter

The purpose of the sandbox adapter is to communicate with the Kubernetes cluster and with the Sandbox itself, for things like creating the sandbox, uploading files to the sandbox, running commands inside the sandbox or purging the sandbox after the submission is evaluated.

When creating a submission, the actual thing that happens is that a new sandbox is launched so that the code can be executed inside it. The sandbox is a Docker container that is launched on the Kubernetes cluster. The sandbox is controlled with the Python Kubernetes client. The launched sandbox depends on the type of problem that is being evaluated. This way we can have different types of problems, written in different languages, and with different methods of evaluation and the sandbox will have all the resources needed to evaluate the submission.

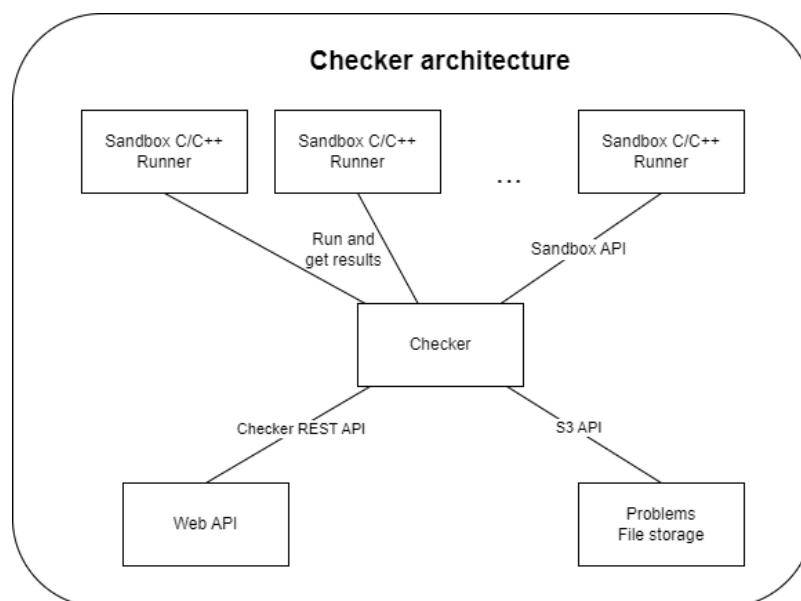The interactions between the checker and other components can be seen in Figure 7.

Figure 7: Checker interactions

14

3.4.3   File manager

All the files that come to the checker, being them received from the web application or downloaded from the file storage, are stored in a temporary directory. The file manager is responsible for managing the files in this directory. It is responsible for creating the directory, uploading files to the directory, and deleting the directory when the submission is evaluated.

3.4.4   Submission factory

The submission factory is responsible for creating the submission object, based on the problem type. The implementation of the submission object is different for each type of problem. It needs to know stuff like what commands to run to build the code, how do the tests look, how to run them and how to compare the results. Currently, the platform only supports one type of problem, but it can be easily extended to support other types of problems.

3.4.5   Workflow for C++ problems

1. The checker receives a new submission from the web application via the API.
2. The checker stores the submitted file in a temporary directory.
3. Based on the problem type, the checker creates a new submission object.
4. The submission object fetches all the necessary files from the file storage.
5. The sandbox adapter creates a new sandbox on the Kubernetes cluster.
6. The sandbox adapter uploads all the necessary files to the sandbox.
7. The sandbox adapter runs the build command inside the sandbox.
8. The sandbox adapter iterates through all the test cases and runs them inside the sandbox.
9. The sandbox adapter compares the actual output with the reference output and stores the results.
10. The sandbox adapter deletes the sandbox.

3.4.6   Global status object

All the submissions are run in a background thread. The status is a global object that is updated by the sandbox adapter as it progresses through the workflow. When the API receives a request for the status of a submission, it returns the status object.

3.5   Sandbox image ⦿ **Repository**

All the sandbox images are running a Python FastAPI server that is used by the sandbox adapter to communicate with the sandbox. Each sandbox image is created specifically for each problem type. Currently, the only sandbox image available is for C++ problems. The image is based on the GCC Docker image and has Python installed and all the dependencies for the FastAPI server. Then the image is pushed to the Docker registry and it is deployed on the Kubernetes cluster via the sandbox adapter.

The endpoints of the FastAPI server are described below.

POST /upload_file

Request body

- **file** - the file to be uploaded. all the files are uploaded in the same directory

Response status

- **200** - the file was uploaded successfully
- **other** - an error occurred

POST /run
This endpoint is used to run commands on the sandbox and get the output. For example, it is used to run the build command and to run the tests.

Request body

- **cmd** - the command to be run inside the sandbox

Response body

- **stdout** - the standard output of the command
- **stderr** - the standard error of the command
- **returncode** - the return code of the command

3.6 VSCode workspaces manager Repository

The VSCode workspaces manager is a component that is responsible for spawning new workspaces. It is written in Python and it is exposed as a simple REST API. The workspaces manager is deployed as a Docker image on the Kubernetes cluster. This manager is very similar to the checker, but it is much simpler. A workspace is unique for each user and problem.

It is responsible for receiving the request for a new workspace from the web application, and for spawning a new workspace container on the Kubernetes cluster. The workspace container is a Docker image based on a Visual Studio Code fork that allows us to run VS Code directly in the browser.

The workflow of creating a new workspace is described below.

1. The users requests a new workspace from the web application by clicking a button.
2. The web application creates the workspace object in the database and sends a request to the workspaces manager.
3. The workspaces manager checks if the workspace already exists. If it does, it returns the internal IP of the workspace pod.
4. If the workspace does not exist, the workspaces manager creates a new workspace pod on the Kubernetes cluster.
5. The workspaces manager copies the necessary files to the workspace pod.
6. The workspaces manager returns the internal IP of the workspace pod.
7. The web application proxies the user's browser to the workspace pod.

The endpoints of the workspaces manager are described below.

POST /workspace/create

Request body

- **id** - the id of the workspace
- **problem_name** - the name of the problem
- **files** - the files that need to be copied to the workspace

Response body

- **endpoint** - the internal IP of the workspace pod

POST /workspace/get

Request body

- **id** - the id of the workspace

Response body

- **endpoint** - the internal IP of the workspace pod

3.7 VSCode online image ⭘ Repository

As mentioned above, the workspace image is a web server that serves Visual Studio Code in the browser via HTTP requests and web sockets. The image is based on a fork of Visual Studio Code called Gitpod OpenVSCode Server that allows us to run VS Code in the browser. The image is pushed to the Docker registry and it is deployed on the Kubernetes cluster via the workspaces manager.

The platform has its own fork of the Gitpod OpenVSCode Server, because we needed to make some changes to the code to make it work with our platform, like adding the checker extension. The fork can be seen here acadnet-dev/vscode-server.
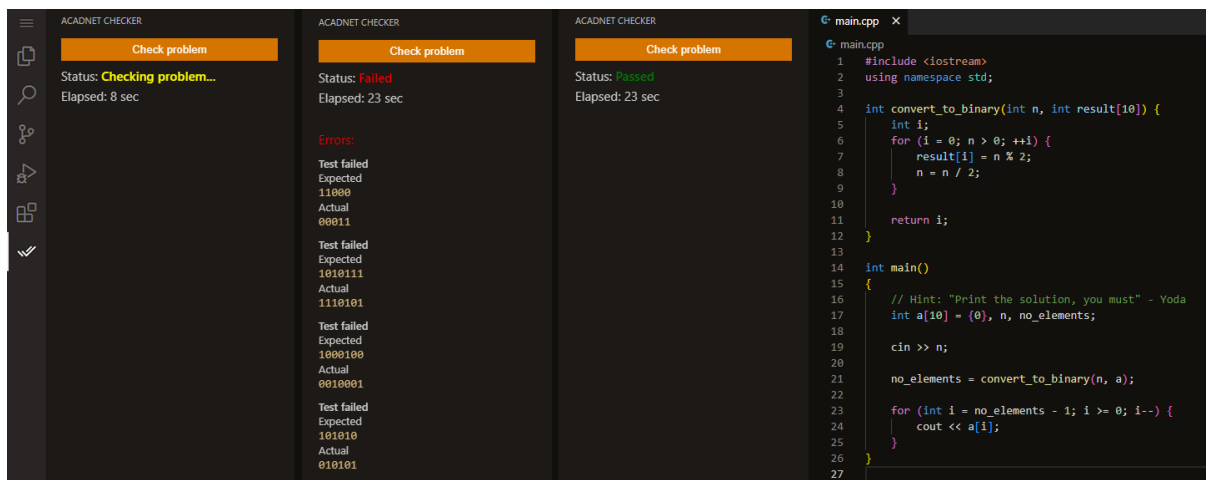
3.8 VSCode checker extension ⭘ Repository

Figure 8: Checker extension example

The chceker extension is a custom made extension for Visual Studio Code that is created specifically to be run in the browser. It is written in TypeScript and it is composed of a frontend and a backend. The backend is responsible for communicating with the checker and the frontend is responsible for displaying the results to the user. The frontend is a sidebar panel that is displayed when the user clicks the extension's icon. The extension is written using the VS Code Extension API. The backend uses fetch API to communicate with the web app for submitting solutions for the problem and polling for the results. The frontend uses plain HTML, JS and CSS to display the results.

The extension is installed compiled as a '.vsix' file and served as a Github Release in the repository. The workspace image pulls the extension and installs it. The extension is automatically enabled when the workspace is created.

4 FEATURES AND FUNCTIONALITIES

4.1 Main page

The main page is the first page that the user sees when accessing the platform. It is a simple page that contains an incentive for the user to create an account and a list of the available courses.
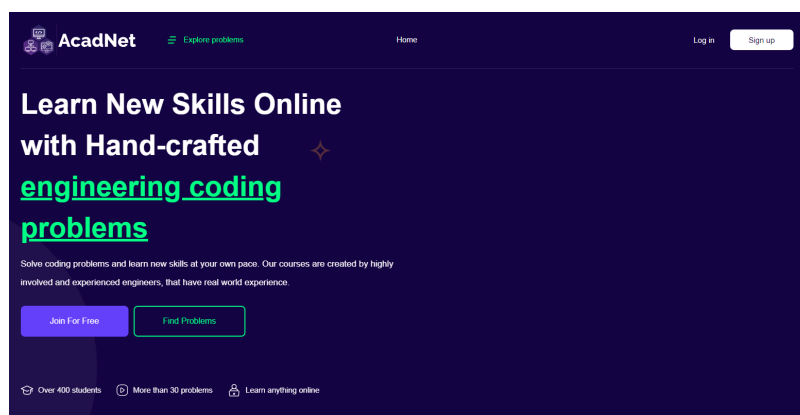

Figure 9: Main page

4.2 User authentication

The users can register and log in using their Google account. This is done via the Google OAuth2 authentication provider.
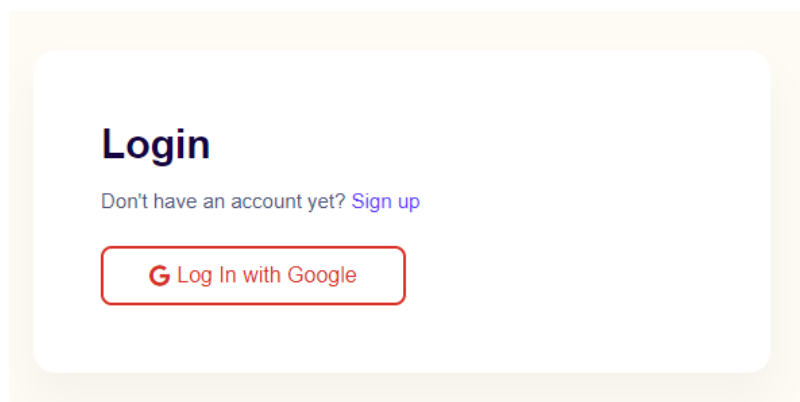

Figure 10: Login page

4.3   Problem browsing

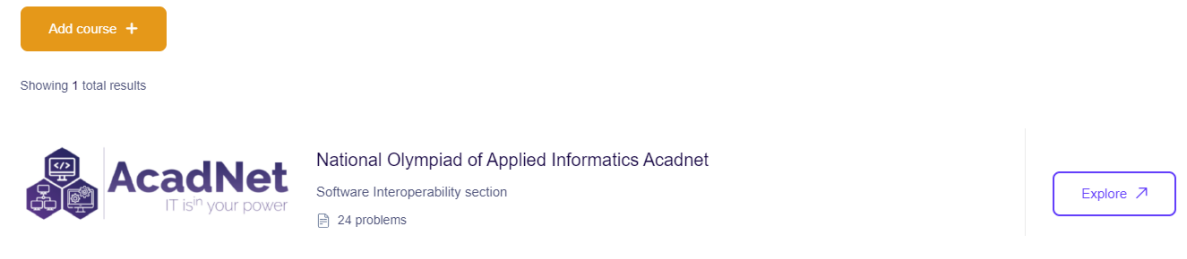As a student, the user can browse the problems by course and category.

Figure 11: Course list

After selecting a course, the user can see the categories and problems in each category. In the figure below, the problems in category '9-10' are shown.
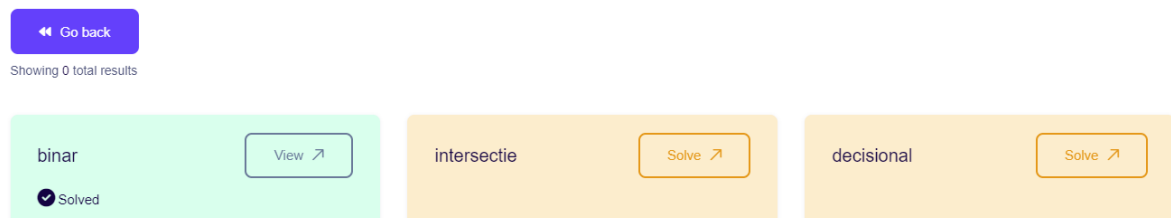

Figure 12: Problem list

4.4 Problem creation

A problem can be created only by course maintainers. After the problem is created, its status is 'Incomplete' until the author uploads the necessary files and the solution passes all the tests. After that, the problem's status is 'Ready' and it can be solved by the students.

Figure 13: Problem editing

This is how a maintainer sees the problems in a category. In the figure below, we can see that the problem 'intersectie' is 'Ready' and the problem 'decisional' is 'Incomplete' in the category '9-10'.

Figure 14: Maintainer problem list

4.5 Problem solving

When a user chooses a problem to solve, he is shown the page in Figure 15. The page contains the problem's statement, a button to download the sources, a file uploader for the solution and a button to luanch the online workspace.

Figure 15: Problem solving

4.6 Submission evaluation

The user can evaluate the submission directly in the problem solving page. He can upload the solution and see the status of the submission. In the figure below, we can see the process of submitting a solution.

Figure 16: Submission status

4.7 Online workspaces

The user can launch an online workspace directly from the problem solving page. The workspace is a Visual Studio Code instance that is running in the browser. The user can edit the source code of the solution and run it directly in the browser. The workspace is preconfigured with the necessary files and extensions for the problem.
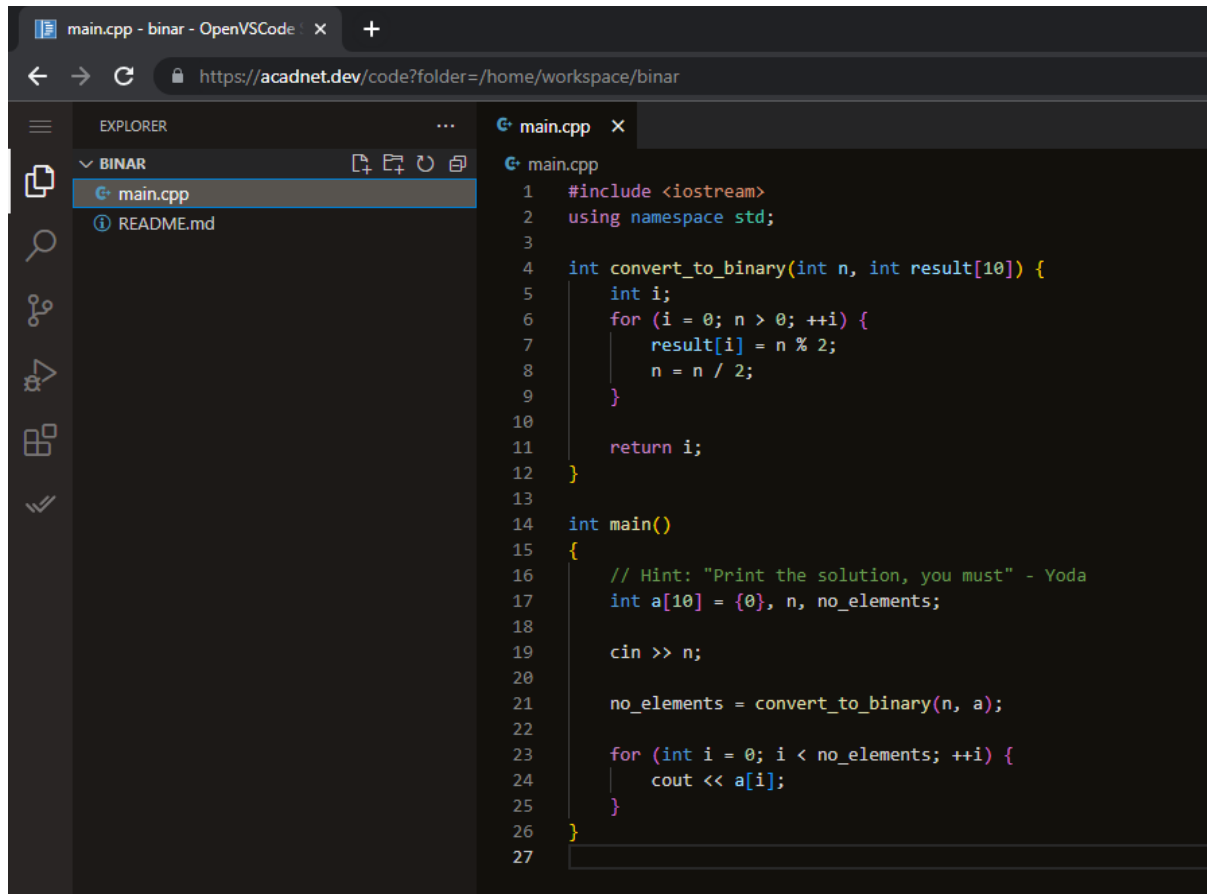

Figure 17: Online workspace

5 DEPLOYMENT AND MAINTENANCE

5.1 Infrastructure

5.1.1 Kubernetes cluster

The entire platform was created with the idea of being scalable and responsive. The platform is deployed on a Kubernetes cluster managed by DigitalOcean. The cluster is composed of multiple Droplets[1]. DigitalOcean provides automatic horizontal scaling for the cluster, which means that the cluster can automatically add or remove nodes based on the load. The platform is configured to have a pool of a minimum of 2 Droplets at any time and a maximum of 5. The cluster is also configured to automatically restart the pods if they crash. This way, the platform is always available and responsive.

A droplet has 2 vCPUs, 2GB RAM and 60GB SSD. The cluster is configured to automatically add new nodes if the CPU usage is above 70% for more than 5 minutes.

On the cluster, the following components are deployed.

- **Web application** - the web application is deployed as a Docker image using multi-stage builds. The Docker image is deployed on the cluster using Helm Charts in an ArgoCD application.
- **Checker** - the checker is deployed as a Docker image. The Docker image is deployed on the cluster using Helm Charts in an ArgoCD application.
- **Workspaces manager** - the workspaces manager is deployed as a Docker image. The Docker image is deployed on the cluster using Helm Charts in an ArgoCD application.
- **Checker Sandboxes** - dynamic Pods that are created by the checker to evaluate the submissions. The Pods are created using the Kubernetes Python client.
- **VSCode Server** - dynamic Pods that are created by the workspaces manager to create the online workspaces. The Pods are created using the Kubernetes Python client.

5.1.2 PostgreSQL

The PostgreSQL database is deployed as a managed service on DigitalOcean Databases. The database is configured to automatically restart if it crashes and to automatically backup the data. The database is configured on a Digitalocean VPC (Virtual Private Cloud) so that it

---

[1]A Droplet is a virtual machine on DigitalOcean

can be accessed only from the Kubernetes cluster. This way, the database is not exposed to the internet and it is more secure.

5.1.3   File storage

The storage used for the files is a managed service provided by DigitalOcean called Spaces. This S3 compatible storage organizes the files in buckets as objects. Like the database, the storage is configured on a Digitalocean VPC (Virtual Private Cloud) so that it can be accessed only from the Kubernetes cluster. This way, the storage is not exposed to the internet and it is more secure.

5.2   Continuous integration

All the components described in Chapter 3 are built and tested using Github Actions. All the components, are built as Docker images and pushed to Github Container Registry. The Docker images are tagged with the commit hash and with the latest tag.

The building process is mostly the same for all the components. The only difference is the 'Dockerfile' used to build the image, because each component runs on its own environment. The building process is described below.

1. Checkout the repository
2. Login to Github Container Registry
3. Replace variables in JSON configuration files with environment variables stored in Github Secrets
4. Build the Docker image with layer caching
5. Push the Docker image to Github Container Registry

The Github Action is ran on every push to the main branch and on every pull request to the main branch.
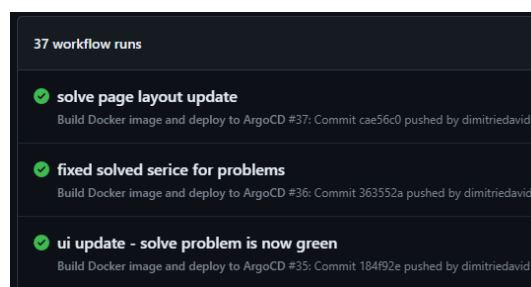
Figure 18: Example of Github Actions workflows

5.3 ArgoCD

ArgoCD is a tool that allows us to deploy Kubernetes manifests from a Git repository. It is used to deploy the Helm Charts for the web application, the checker, and the workspaces manager. The manifests are stored in the acadnet-dev/infra repository. The manifests are deployed automatically when a new commit is pushed to the repository, because ArgoCD is polling the repository for changes every 3 minutes.

5.4 Continuous deployment

All the components described in Chapter 3 are deployed on the Kubernetes cluster using Helm Charts. The Helm Charts are deployed using ArgoCD. The deployment is done automatically at the end of the Github Actions build process. The deployment process is done by updating the image version in the ArgoCD deployment using the ArgoCD API.
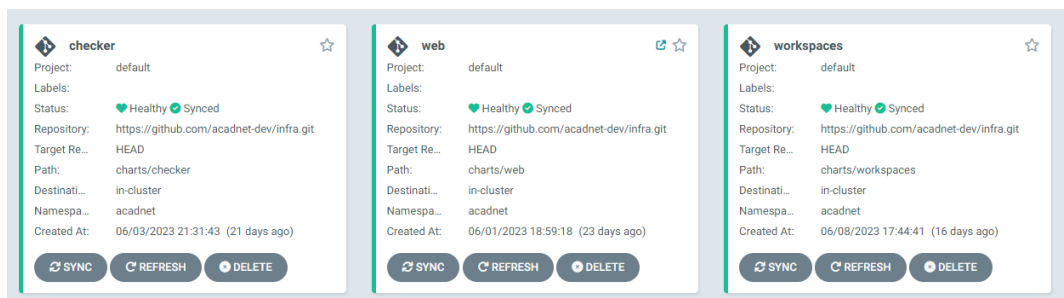
Figure 19: Applications deployed with ArgoCD
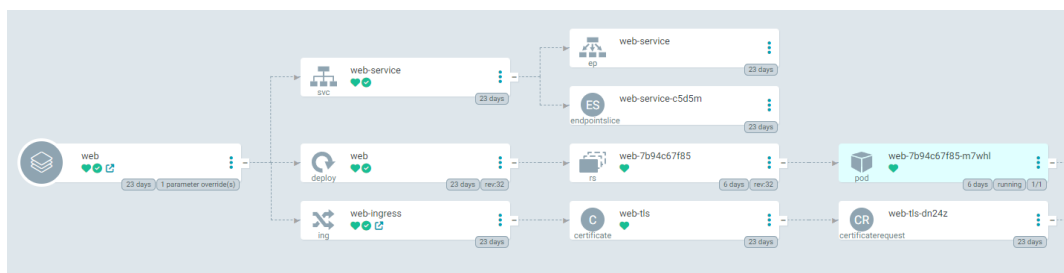
Figure 20: Web application deployment

5.5   Monitoring

Like any software application, monitoring is a very important aspect of the platform. Because of the complexity of the platform, it is difficult to monitor all applications individually. As just a little effort was put into monitoring, the platform is currently monitored only at a high level. When looking at the performance and resource usage, the monitoring is done using DigitalOcean's monitoring service. This service provides a dashboard with the CPU, memory and disk usage of the Droplets. The dashboard can be seen in Figure 21. In terms of logging, the logs can be seen in a centralized place in ArgoCD. The logs are collected from the Kubernetes cluster and displayed in each application's dashboard like in Figure 22.
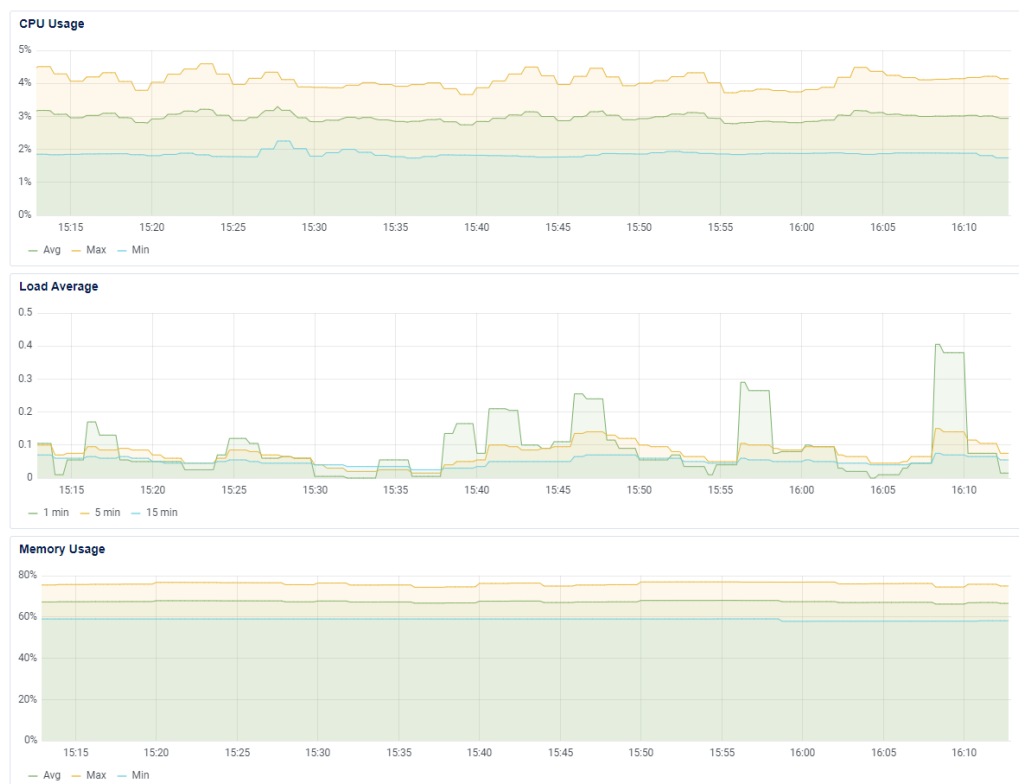

Figure 21: Monitoring dashboard

```
INFO:     10.244.2.106:58658 - "GET /submission/status/?submission_id=49640e47-87ec-42af-93bd-09e515b40676 HTTP/1.1" 200 OK
INFO:     10.244.2.106:58658 - "GET /submission/status?submission_id=49640e47-87ec-42af-93bd-09e515b40676 HTTP/1.1" 307 Temporary Redirect
INFO:     10.244.2.106:58658 - "GET /submission/status/?submission_id=49640e47-87ec-42af-93bd-09e515b40676 HTTP/1.1" 200 OK
Stopped sandbox with id sandbox-cpp-8913166b-6c25-4f3d-969c-07225e5c88d8
INFO:     10.244.2.106:58658 - "GET /submission/status?submission_id=49640e47-87ec-42af-93bd-09e515b40676 HTTP/1.1" 307 Temporary Redirect
INFO:     10.244.2.106:58658 - "GET /submission/status/?submission_id=49640e47-87ec-42af-93bd-09e515b40676 HTTP/1.1" 200 OK
INFO:     10.244.2.106:46024 - "POST /submission/create?type=SimpleAcadnetIS&bucket=9bd6d6ea-2f0e-494d-932a-ea9efec2bcaa HTTP/1.1" 307 Temporary Redirect
INFO:     10.244.2.106:46024 - "POST /submission/create/?type=SimpleAcadnetIS&bucket=9bd6d6ea-2f0e-494d-932a-ea9efec2bcaa HTTP/1.1" 200 OK
Pod created with name sandbox-cpp-2d729696-a192-4c9c-b402-99134443c999
INFO:     10.244.2.106:46024 - "GET /submission/status?submission_id=093a1c9f-2677-4e22-8b41-6c5227788da2 HTTP/1.1" 307 Temporary Redirect
INFO:     10.244.2.106:46024 - "GET /submission/status/?submission_id=093a1c9f-2677-4e22-8b41-6c5227788da2 HTTP/1.1" 200 OK
INFO:     10.244.2.106:46024 - "GET /submission/status?submission_id=093a1c9f-2677-4e22-8b41-6c5227788da2 HTTP/1.1" 307 Temporary Redirect
```

Figure 22: Logs for the checker

27

6 TESTING AND VALIDATION

6.1 Submission evaluation fairness

An important aspect of submission evaluation is fairness. The platform needs to be fair to all the users. This means that the submission evaluation performance and results should be the same for all the users. This thing should be achieved under any load, for any hardware and for any number of users. This is a very difficult thing to achieve, because the platform is running on a Kubernetes cluster, which means that the pods are running on different nodes and the nodes have different hardware. The Kubernetes cluster is also configured to automatically scale the number of nodes based on the load, which means that the pods can be moved from one node to another at any time. This means that the pods can run on different hardware at any time.

To address this, the platform implements the following things.

Same hardware for all the nodes All the nodes in the cluster are configured to have the same hardware. This means that all the pods will run on the same hardware, even if it is virutalized.

Resource limits for sandbox deployments The sandbox deployments are configured to have resource limits and requests. This means that the pods will have the same amount of CPU and memory available, or the submission will not start and the cluster will need to scale up.

Submissions ran separately The submissions are ran separately, which means that the pods will not interfere with each other. This means that the pods will not have to share the CPU and memory with other pods.

Hand-crafted tests Authors are instructed to create tests that are not easily influenced by hardware resource fluctuations. For example, by limiting submission execution time and memory usage, we do not test the performance of the code, but the complexity of the algorithm. This way, the difference in performance between different complexities will be so big that it will not be influenced by the unpredictability of the hardware.

6.2 Performance and scalability

The platform is designed to be scalable and responsive. This means that the platform should be able to handle a large number of users and submissions. Being deployed on a Kubernetes

cluster in a cloud environment, the platform can automatically scale up by spinning up new nodes. This comes with the downside of having unpredictable costs, because the number of nodes can change at any time.

The only problem is that spinning a new node takes a few minutes, which means that the user will have to wait a few minutes for the submission to start or for the workspace to open. Because of this, the platform is configured to have at least a free node (buffer node) at any time. This way, the user will not have to wait for the node to spin up. When this buffer node is used, the cluster will automatically spin up a new node.

6.3   Security

Networking The platform is deployed on a DigitalOcean VPC (Virtual Private Cloud). This means that the cluster is not exposed to the internet and it is more secure. The only thing that is exposed to the internet is the web application, which is exposed via a load balancer. The load balancer is configured to only allow traffic on port 80 and 443. The applications are talking between them using their internal IP addresses and ports.

Certificates All the exposed endpoints are secured with SSL certificates. The certificates are automatically generated and renewed by Let's Encrypt. The certificates are automatically renewed every 3 months.

Authentication The users can authenticate using their Google account. This is done via the Google OAuth2 authentication provider. This way, the platform does not store any passwords and does not manage forgotten passwords.

Authorization The platform has authorization based on user roles. The two roles are for authors and students. All the access is based on the user's journey. Each role's journey is described in Chapter 2.1.

6.4   Platform testing

For testing purposes, the platform has already 6 problems uploaded. These problems are from the 2023 edition of the National Olympiad of Applied Informatics Acadnet. The problems statements are in Romanian and they are implemented using the only developed problem type, C++.

The platform was tested by multiple users. The users were asked to create an account and solve a problem, by uploading the solution and by solving directly in the online workspace. The users were asked to give feedback about the platform. The feedback was mostly positive. The users liked the platform and they found it easy to use. The users also found some bugs and suggested some improvements. The bugs were fixed and the improvements were implemented.

7 CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

The platform is a success as it is easy to use and it is scalable. All the planned objectives were achieved. AcadNet.dev will be hosted with the main purpose of being used by high-school students to learn programming and to prepare for the National Olympiad of Applied Informatics.

On the techincal side, there were multiple challanges, starting from selecting which technologies to use and ending with the deployment and maintenance of the platform. Interconnecting all the components was also a challange, because the platform is composed of multiple components written in different technologies. One more thing that was a challange was debugging the Kubernetes pod manager that the checker and the workspace manager use to launch pods on the cluster, because it could be only tested in the production environment.

7.2 Future enhancements

More problem types The platform can be extended to support more problem types. Currently, the only supported problem type is C++. The platform can be extended to support other languages like Java, Python, C#, etc. As it is open-source and it is easy to extend, the platform can be extended by any author that wants to create problems for the platform.

Automatical removal of workspaces The workspaces are currently removed manually by the maintainer. The platform can be extended to automatically remove the workspaces after a period of inactivity. This way resources will be saved and the costs will be reduced.

Persistent workspaces The workspaces are currently not persistent. This means that the user will lose all the changes made to the workspace after the workspace is shut down. The platform can be extended to support persistent workspaces. This way, the user will be able to save the workspace and continue working on it later.

Contests The platform can be extended to support contests. This way, the platform can be used to organize contests for highschool students. The contests can be organized by the teachers and the students can participate in the contests. This includes a contest component that allows the teachers to create contests and to add existing problems to the contests. The contests can be open or private. The private contests can be accessed only by the students that are enrolled in the course. There should be a leaderboard and a monitoring component

that allows the teachers to see the progress of the students during the contest.

Private test results Currently the test results are public. This means that the students can see the test results. Maybe the students should not see the test results, because they can use the test results to cheat. The platform can be extended to support private test results.

Execution time and memory usage limits Currently, the submissions are not limited in terms of execution time and memory usage. This means that the submissions can run for as long as they want and they can use as much memory as they want. This can be a problem because the submissions can run for a long time and they can use a lot of memory. The platform can be extended to support execution time and memory usage limits. Also, this limit can be used by authors to create tests that check the complexity of the algorithm.

7.3   Lessons learned

Developing a complex project like this one requires a lot of planning. Unfortunately, I only created a bird's eye view of the project and I did not plan the implementation in detail. This led to a lot of research and to a lot of time spent before actually starting the implementation. This also led to a lot of changes in the implementation, because I did not know exactly what I wanted to implement.

Another thing that I learned is that it is very important to have a clear vision of the project. This way, the project can be implemented in a consistent way. This is very important when working with other people, because it is very important to have a consistent code style and to have a consistent architecture. This way, the code is easier to read and to understand by other people.

The last thing that I think had a big impact on writing this thesis is the during the implementation, I created multiple diagrams and I wrote a lot of documentation du. This documentation can be found in the acadnet-dev/docs repository, together with the diagrams. This documentation helped me a lot when writing this thesis, because I did not have to remember why I implemented the things the way I did. I just had to read the documentation and the diagrams and I remembered everything.

Overall, I think that this project was a success and I learned a lot of things from it. I learned a lot of things about the technologies used, how to design the architecture and how to automate the deployment.

APPENDICES

A CODE SNIPPETS

A.1 Submission status JSON example

```json
1  {
2          "submission_id": "330f946d-95d0-45da-bffd-a45996045ba1",
3          "status": "finished",
4          "build_status": "success",
5          "test_results": [{
6                          "test_name": "test01.in",
7                          "passed": false,
8                          "status": "failed",
9                          "exec_result": {
10                                 "actual": "00011",
11                                 "ref": "11000"
12                         }
13                 }
14         ],
15         "status_history": [{
16                         "status": "created",
17                         "timestamp": 1686422831.5370035
18                 },{
19                         "status": "creating sandbox",
20                         "timestamp": 1686422831.5370176
21                 },{
22                         "status": "pod created with name sandbox-cpp-5d8720e1-c042-43ea-
                                a95a-e8e268c6f0be",
23                         "timestamp": 1686422831.570527
24                 },{
25                         "status": "waiting for sandbox to start - pod status: Pending",
26                         "timestamp": 1686422836.5834553
27                 },{
28                         "status": "sandbox launched",
29                         "timestamp": 1686422836.6700737
30                 },{
31                         "status": "uploading submission file",
32                         "timestamp": 1686422836.6700778
33                 },{
34                         "status": "compiling submission",
35                         "timestamp": 1686422836.685021
36                 },{
37                         "status": "uploading test test01.in",
38                         "timestamp": 1686422840.5165641
39                 },{
40                         "status": "running test test01.in",
41                         "timestamp": 1686422840.5247858
42                 },{
43                         "status": "comparing output for test test01.in",
44                         "timestamp": 1686422840.5364146
45                 },{
46                         "status": "finished",
47                         "timestamp": 1686422852.6135993
48                 }]
49  }
```