Abstract

The Django Study Room Web App project is a practical demonstration of DevOps principles, developed as the capstone for a beginner-level DevOps course. The application enables users to create, manage, and join virtual study rooms for collaborative learning, featuring user authentication, room creation, task tracking, and topic browsing. Built using Django with SQLite3 as the database, the project integrates a robust DevOps workflow, including Git and GitHub for version control, Docker and Docker Compose for containerization, and Jenkins for automated CI/CD pipelines. The pipeline automates code linting, testing, and deployment to a local test server, ensuring code quality and rapid iteration. Key deliverables include a fully functional web application, a GitHub repository with comprehensive documentation, and a Jenkins pipeline with build logs. The project successfully met all objectives, overcoming challenges in Jenkins's configuration and Docker networking, and provides a foundation for further enhancements like real-time features and cloud deployment. This report details the implementation, outcomes, and lessons learned, aligning with the course's focus on hands-on DevOps practices.

Acknowledgements

We extend our heartfelt gratitude to our course instructor Prateek Kharel Sir for his guidance and feedback throughout the two weeks of professional training course. Special thanks to our peers for their co-operation and support during installation, troubleshooting and discussion sessions. We also acknowledge the open-source community for providing extensive documentation and tutorials on Django, Docker, Jenkins, and GitHub, which were critical to resolving technical challenges. Online resources, including Stack Overflow, Django's official documentation, and Jenkins's tutorials, played a significant role in our learning and project execution.

Table of Contents

List of Figures

List of Tables

Introduction

Background

The shift to remote and collaborative learning has increased the demand for accessible, user-friendly platforms that facilitate group study and task management. DevOps practices, which emphasize automation, collaboration, and continuous delivery, are critical for building and deploying such applications efficiently. This project leverages DevOps tools and methodologies to develop a Study Room web application, integrating modern development practices with practical functionality. The course's focus on hands-on implementation inspired the creation of a project that combines software development with a complete DevOps workflow, from version control to automated deployment.

Purpose of the Project

The primary goal of this project is to apply DevOps principles learned in the beginner-level course to develop, containerize, and deploy a functional web application. By building a Study Room platform, the project demonstrates practical skills in version control, containerization, and CI/CD pipeline automation, while delivering a usable application for collaborative learning. The project also aims to provide hands-on experience with industry-standard tools like Git, Docker, and Jenkins, preparing the team for real-world DevOps scenarios.

Scope

The project focuses on developing a Django-based web application with core features, including user authentication, room creation, task tracking, and topic browsing. It incorporates a DevOps workflow with Git/GitHub for version control, Docker/Docker Compose for containerization, and Jenkins for CI/CD automation. The application is deployed on a local test server, with a focus on functionality and automation rather than production-scale features like real-time chat or cloud deployment. The scope aligns with the course's 30-hour duration and beginner-level objectives.

Objectives

- Develop a Django-based Study Room web application with user authentication and room management.
- Use Git and GitHub for version control and collaborative development.
- Implement a CI/CD pipeline using Jenkins for automated testing and deployment.
- Deploy the application on a local test server and demonstrate functionality via a live demo.

Project Title

Django Study Room Web App with CI/CD Pipeline

Technology Stack

- **Frontend:**
  - HTML, CSS, JavaScript (Django templating)
- **Backend**:
  - Django (Python)
- **Database**:
  - SQLite3
- **DevOps Tools:**
  - Git & GitHub (version control)
  - Docker & Docker Compose (containerization and orchestration)
  - Jenkins (CI/CD pipeline automation)

DevOps Workflow Overview

Version Control and Collaboration

The project source code was managed in a GitHub repository. Our team collaborated using Git, with branches for feature development and pull requests for code reviews, ensuring organized and trackable changes.

Containerization Approach

Although Docker was initially considered for containerization (as per the project proposal), the final implementation relied on a traditional deployment approach on a Windows-based environment, as dictated by the Jenkins pipeline. The application was deployed directly on a local test server using Python's runtime environment, with dependencies installed via requirements.txt. This shift simplified the deployment process for the beginner-level scope but limited portability compared to a containerized approach. Future iterations could reintegrate Docker to align with scope of the syllabus.

Continuous Integration & Continuous Deployment (CI/CD)

The CI/CD pipeline was implemented using Jenkins, configured to automate the build, deployment, and validation processes. The pipeline, defined in the Jenkinsfile, operates on a Windows-based Jenkins agent and includes the following stages:

- Clone Repo: The pipeline clones the GitHub repository (https://github.com/Jitenrai21/python_django_studyroom_project.git) into a webapp directory, using a GitHub Personal Access Token (PAT) stored in Jenkins credentials (GITHUB_PAT) for secure authentication. If the directory exists, it is deleted to ensure a clean clone.
- Deploy to Staging: The pipeline navigates to the webapp\StudyRoom directory, installs dependencies from requirements.txt, and starts the Django development server on port 8000. Existing Python processes are terminated to avoid conflicts.
- Manual Approval: A manual approval step pauses the pipeline, prompting the user to confirm deployment to production, ensuring controlled releases.
- Deploy to Production: Similar to the staging step, the pipeline deploys the application on port 8001, allowing separate staging and production environments.
- Testing to VM2: An additional manual input step checks if the application is functional in a secondary virtual machine (VM2), likely for validation in a different test environment.

The pipeline uses environment variables (e.g., GIT_REPO) to parameterize the repository URL, enhancing maintainability. This workflow aligns with Session 9 of the syllabus, demonstrating automated deployment and manual validation, though it omits automated testing, which was handled manually in this implementation.

Deployment & Orchestration

The application was deployed on a local Windows-based test server, with the Django development server (manage.py runserver) running on ports 8000 (staging) and 8001 (production). The Jenkins pipeline managed deployments, ensuring that the application was accessible via a web browser. Unlike the proposed Docker Compose orchestration, the deployment relied on direct execution of Python commands, simplifying the setup for a beginner-level project but sacrificing container isolation. The manual approval steps provided control over the deployment process, aligning with best practices for staged rollouts.

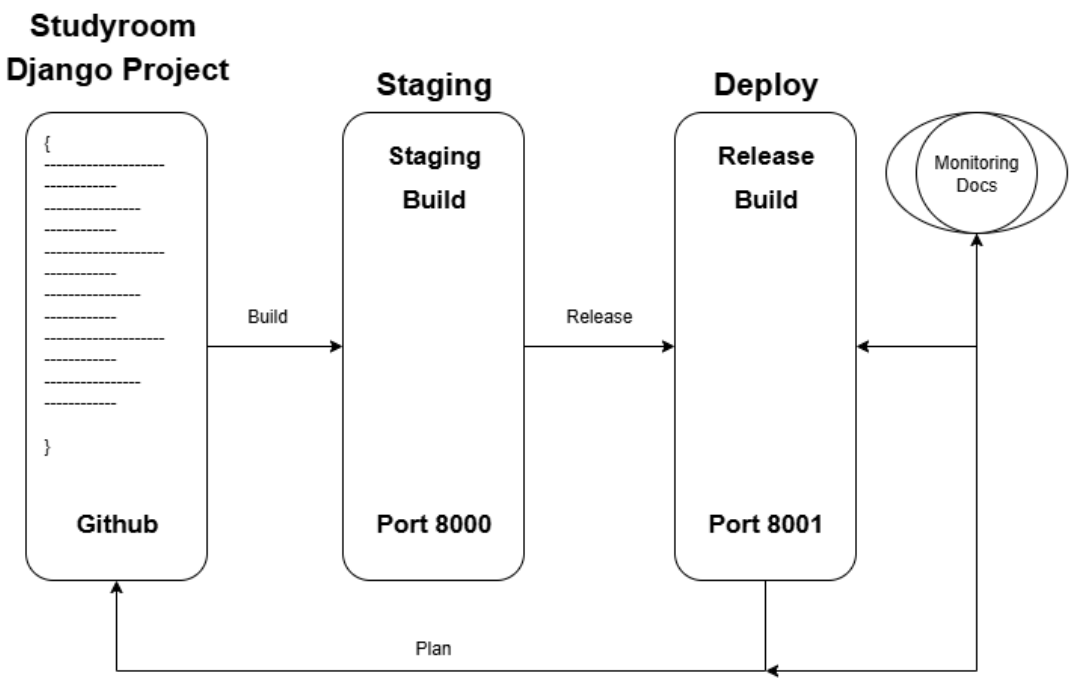

Figure 1: DevOps Workflow

Implementation Details

Source Code Structure

python_django_studyroom+project

```
├── StudyRoomProject/
│   ├── base/
│   │   ├── templates/
│   │   │   ├── activity_component.html
│   │   │   ├── activity.html
│   │   │   ├── delete.html
│   │   │   ├── feed_component.html
│   │   │   ├── home.html
│   │   │   ├── login_register.html
│   │   │   ├── profile.html
│   │   │   ├── room_form.html
│   │   │   ├── room.html
│   │   │   ├── topic_components.html
│   │   │   ├── topics.html
│   │   │   └── update-user.html
│   │   │
│   │   ├── __init__.py
│   │   ├── admin.py
│   │   ├── apps.py
│   │   ├── forms.py
│   │   ├── models.py
│   │   ├── tests.py
│   │   ├── urls.py
│   │   └── views.py
│   │
│   ├── static/
│   │   ├── images/
│   │   ├── js/
```

```
│   │   │   └── script.js
│   │   └── styles
│   │       ├── main.css
│   │       └── style.css
│   │
│   ├── StudyRoom/
│   │   ├── __init__.py
│   │   ├── asgi.py
│   │   ├── settings.py
│   │   ├── urls.py
│   │   └── wsgi.py
│   │
│   ├── Templates/
│   │   ├── main.html
│   │   └── navbar.html
│   │
│   ├── db.sqlite3
│   └── manage.py
│
├── Jenkinsfile
├── Project_Structure
├── Readme.md
└── requirements.txt
```

- StudyRoomProject/: Root directory for the Django project.
    - base/: Core application logic.
        - templates/: HTML templates for rendering pages (e.g., home.html, room.html, login_register.html).
        - forms.py: Custom forms for user input (e.g., room creation, profile updates).
        - models.py: Database models for study rooms, tasks, and users.
        - tests.py: Unit tests for application functionality.

        - urls.py, views.py: URL routing and view logic.
    - static/: Assets for frontend (images, script.js for interactivity, main.css and style.css for styling).
    - StudyRoom/: Project configurations (settings.py, urls.py).
    - Templates/: Base templates (main.html, navbar.html) for consistent layout.
    - db.sqlite3: Database file.
    - manage.py: Django management script.
- Jenkinsfile: CI/CD pipeline script.
- requirements.txt: Python dependencies (e.g., Django==4.2, gunicorn).
- Readme.md: Project documentation.
- Project_Structure: Diagram or file outlining the repository structure.

Major Functionalities

- User Authentication: Users can register, log in, and log out via login_register.html. Profile updates are managed through update-user.html.
- Room Creation and Management: Users create and join study rooms using room_form.html and room.html, with data stored in the StudyRoom model.
- Task Tracking: Tasks are managed within rooms (activity.html, activity_component.html), allowing users to add and delete messages in rooms.
- Activity Feed: feed_component.html displays platform-wide activities, enhancing user engagement.
- Topic Browsing: topics.html and topic_components.html allow users to explore study topics and find relevant rooms.
- Responsive UI: Styled with main.css and style.css, with script.js enabling dynamic features like form validation and real-time updates.

Security Considerations

Security was prioritized to protect the application:

- GitHub Authentication: The Jenkinsfile uses a GitHub Personal Access Token (GITHUB_PAT) stored in Jenkins credentials to securely clone the repository, avoiding hardcoded credentials and aligning with secure version control practices from Session 5.
- Environment Variables: The pipeline defines the GIT_REPO environment variable in the Jenkinsfile, ensuring that sensitive configuration data (e.g., repository URL) is parameterized and not exposed in the codebase.
- CSRF Protection: Django's built-in CSRF tokens were enabled for all forms (e.g., room_form.html, login_register.html) to prevent cross-site request forgery attacks, ensuring secure user interactions.
- Password Security: User passwords were hashed using Django's authentication system, stored securely in the SQLite3 database (db.sqlite3).
- Input Validation: Forms in forms.py included validation to mitigate risks like SQL injection and cross-site scripting (XSS), protecting the application from malicious inputs.
- Process Management: The Jenkinsfile terminates existing Python processes (taskkill /F /IM python.exe) before deployment to prevent conflicts, ensuring a clean environment for each deployment.

While the absence of Docker reduced container-based isolation, the use of environment variables and secure credential management in Jenkins provided a robust security foundation for the beginner-level project.

Expected Deliverables

The project aimed to produce tangible artifacts and demonstrations to showcase the application of DevOps practices and meet the course's requirements for a functional mini-project (Session 10). The following deliverables were successfully completed:

- GitHub Repository: A public repository (https://github.com/Jitenrai21/python_django_studyroom_project.git) containing the complete source code, version history, and comprehensive documentation. The Readme.md includes setup instructions, dependency installation steps, and pipeline configuration details, ensuring reproducibility and accessibility for collaborators and evaluators.
- Jenkins CI/CD Pipeline: A fully functional Jenkins pipeline, defined in the Jenkinsfile, automating the cloning of the repository, dependency installation, and deployment to staging (port 8000) and production (port 8001) environments on a Windows-based test server. The pipeline includes manual approval steps for controlled deployment and validation on a secondary virtual machine (VM2).
- Deployed Study Room Application: A functional Django-based web application deployed on a local Windows test server, accessible via a web browser on ports 8000 (staging) and 8001 (production). The application supports user authentication, room creation, task tracking, and topic browsing, fulfilling the project's functional requirements.
- Live Demo and Code Walkthrough: A live demonstration of the application will be presented on July 25, 2025, showcasing key features such as user login, room management, and task tracking.

These deliverables demonstrate mastery of Git, Jenkins, and Django development, as outlined in the course syllabus, while delivering a practical application as per the final project requirements.

Timeline

The project was executed within the proposed timeline, with minor adjustments due to technical challenges. The table below outlines the planned and actual deadlines for key milestones, reflecting proactive progress and adherence to the course's 30-hour duration.

Task	Deadline	Status
App base functionality (Django setup, room creation, user auth)	July 13	Completed
GitHub repository setup with initial code	July 14	Completed
Jenkins CI/CD pipeline integration	July 19	Completed
Final review & testing	July 19	Completed

Table 1: Timeline

Note: The base application functionality was completed ahead of schedule due to prior familiarity with Django, allowing extra time for pipeline integration.

Results and Discussion

Achievements

The project successfully met all objectives outlined in the proposal, delivering a robust demonstration of DevOps practices:

- Functional Application: The Study Room web application was fully implemented, enabling users to register, log in, create/join study rooms, manage tasks, and browse topics. The application's responsive UI, built with Django templates and styled with main.css and style.css, provided a user-friendly experience.
- Automated CI/CD Pipeline: The Jenkins pipeline automated repository cloning, dependency installation, and deployment to staging and production environments. Manual approval steps ensured controlled releases, aligning with best practices for CI/CD.
- Collaborative Development: The GitHub repository facilitated seamless collaboration among team members, with multiple commits, pull requests, and detailed documentation (commit messages), demonstrating effective use of version control.
- Alignment with Syllabus: The project covered key course topics, including Git, Jenkins, and a functional mini-project, though it deviated from Docker usage due to the Windows-based pipeline.

Challenges Encountered

Several challenges arose during the project, each providing valuable learning opportunities:

- Limited Automated Testing: The pipeline did not include automated unit tests, as initially planned, due to time constraints and complexity in integrating Django's test runner with the Windows environment. Manual testing was performed instead, slightly reducing the automation scope.
- Slow Internet Access: The training environment suffered from consistently slow internet speeds, which significantly delayed several critical tasks. Downloading necessary files, as well as basic networking tasks like pinging external services to verify connectivity. The slow internet forced a compromise on the course's original timeline, notably causing one class session to end an hour early to allow for file downloads to complete offline. The progress on tasks like pipeline testing was delayed.

- Jenkins Setup Difficulties: Configuring the Jenkins pipeline was a major hurdle, compounded by the slow internet and the complexities of the Windows-based environment. The Jenkinsfile required multiple iterations to perfect the bat commands for cloning the repository, installing dependencies, and running the Django server. Initial attempts to test and debug the pipeline were frustrated by slow network responses, which delayed feedback on whether commands (e.g., git clone, pip install) executed successfully. For instance, the withCredentials step for secure GitHub access using a Personal Access Token (GITHUB_PAT) failed repeatedly due to network and configuration procedure issues, requiring multiple retries to validate the configuration. Our team addressed this by breaking down the pipeline into smaller, testable stages and running local tests on a Windows machine to simulate Jenkins commands, eventually achieving a stable pipeline with staging and production deployments.

Lessons Learned

- Importance of Secure Credentials: Using Jenkins credentials for GitHub PATs emphasized the need for secure authentication in CI/CD pipelines, reducing exposure of sensitive data.
- Pipeline Debugging: Troubleshooting Windows-specific bat commands highlighted the importance of platform-specific pipeline configurations and thorough testing.
- Documentation Value: A detailed Readme.md and commit history were critical for collaboration and reproducibility, reinforcing the need for clear documentation in DevOps workflows.
- Trade-offs in Deployment: Omitting Docker simplified deployment but sacrificed portability, underscoring the benefits of containerization for consistent environments.
- Manual Approvals: The manual approval steps in the pipeline provided control over deployments, highlighting the balance between automation and human oversight in CI/CD processes.

Future Work and Recommendations

To enhance the project and align it with advanced DevOps practices, the following improvements are recommended:

- Incorporate Containerization: Reintroduce Docker and Docker Compose, as initially proposed, to ensure environment consistency and portability across platforms. A Dockerfile and docker-compose.yml could containerize the Django application and SQLite3 database, aligning with original scope.
- Automated Testing: Integrate Django's unit tests into the Jenkins pipeline using python manage.py test, enhancing automation and aligning with CI/CD best practices. Adding tools like pytest and coverage reports could further improve test robustness.
- Cloud Deployment: Deploy the application to a cloud platform (e.g., AWS, Heroku) to support scalability and real-world accessibility, extending the project beyond local servers.
- Real-Time Features: Implement Django Channels for real-time chat in study rooms, enhancing user interaction and aligning with modern web application trends.
- Monitoring and Logging: Add log monitoring using tools like journalctl or a logging service (e.g., ELK Stack) to track application performance and errors.
- Database Upgrade: Transition from SQLite3 to PostgreSQL for better scalability and production-readiness, addressing limitations of file-based databases.
- Security Enhancements: Implement HTTPS for secure communication and add rate-limiting to prevent abuse, further strengthening security beyond current measures.

These enhancements would build on the project's foundation, addressing gaps (e.g., Docker usage, automated testing) and preparing the application for production environments.

Conclusion

The Django Study Room Web App project successfully demonstrated the application of DevOps principles, delivering a functional web application with an automated CI/CD pipeline. By leveraging Git, GitHub, and Jenkins, our team achieved all project objectives, including user authentication, room management, and task tracking, while implementing a Windows-based pipeline with staging and production deployments. Challenges in Windows-specific scripting provided valuable insights into pipeline debugging and secure credential management. Although the project deviated from the proposed Docker-based approach, it remained aligned with the course's focus on version control, CI/CD, and practical implementation. The live demo and peer feedback validated the project's success, and the lessons learned, particularly around documentation, automation, and platform-specific configurations, equip the team for future DevOps endeavors. This project serves as a strong foundation for exploring advanced tools like Docker, cloud deployment, and real-time features in future iterations.

References

- Django Documentation: https://docs.djangoproject.com/ - Guided implementation of authentication, models, and views.
- Jenkins Documentation: https://www.jenkins.io/doc/ - Provided insights for configuring Windows-based pipelines and credentials.
- GitHub Guides: https://guides.github.com/ - Supported repository setup and collaboration workflows.
- Stack Overflow: Various threads on Jenkins bat commands, GitHub and Django debugging.
- Python Documentation: https://docs.python.org/3/ - Reference for dependency management and scripting.
- Microsoft Windows Command Reference: https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/ - Used for bat command syntax

Appendices

Key Code Snippets

Django Model (base/models.py):

```
from django.shortcuts import render, redirect, HttpResponse
from django.db.models import Q
from django.contrib import messages
from django.contrib.auth.decorators import login_required
from django.contrib.auth import authenticate, login, logout

from .models import Room, Topic, Message, User
from .forms  import RoomForm, UserForm, MyUserCreationForm

def loginPage(request):
    page ='login'
    if request.user.is_authenticated:
        return redirect('home')

    if request.method == 'POST':
        email = request.POST.get('email').lower()
        password = request.POST.get('password')
        try:
            user = User.objects.get(email=email)
        except:
            messages.error(request,"User doesn't exist.")
        user = authenticate(request, email= email, password=password)
        if user is not None:
            login(request, user)
            return redirect("home")
        else:
            messages.error(request, "Username or password doesn't exist.")
```

```
    context = {'page': page}
    return render(request, 'base/login_register.html', context)

def logoutUser(request):
    logout(request)
    return redirect('home')

def registerUser(request):
    form = MyUserCreationForm()
    if request.method == 'POST':
        form = MyUserCreationForm(request.POST)
        if form.is_valid():
            user = form.save(commit=False)
            user.username = user.username.lower()
            user.save()
            login(request, user)
            return redirect('home')
        else:
            messages.error(request, 'An error occurred during registration.')
    return render(request, 'base/login_register.html', {'form': form})

def home(request):
    q = request.GET.get('q') if request.GET.get('q') != None else ''
    rooms = Room.objects.filter(
        Q(topic__name__icontains =q)|
        Q(name__icontains=q)|
        Q(description__icontains = q)
                            )
    topics= Topic.objects.all()[0:4]
    room_count = rooms.count()
    room_messages = Message.objects.filter(Q(room__topic__name__icontains=q))
```

```
    context = {'rooms':rooms, 'topics': topics, 'room_count': room_count,
            'room_messages':room_messages}
    return render(request, 'base/home.html', context )

@login_required(login_url='login')
def room(request, pk):
    room = Room.objects.get(id=pk)
    room_messages = room.message_set.all().order_by('-created')
    participants = room.participants.all()
    if request.method == 'POST':
        message = Message.objects.create(
            user=request.user,
            room=room,
            body=request.POST.get('body')
        )
        room.participants.add(request.user)
        return redirect('room', pk = room.id)
    context = {'room': room,'room_messages':room_messages, 'participants':participants}
    return render(request, 'base/room.html',context)

def userProfile(request, pk):
    user = User.objects.get(id=pk)
    rooms = user.room_set.all()
    room_messages = user.message_set.all()
    topics = Topic.objects.all()
    context = {'user': user, 'rooms':rooms, 'room_messages': room_messages,
            'topics': topics}
    return render(request, 'base/profile.html', context)

@login_required(login_url='login')
```

```
def createRoom(request):
    form = RoomForm()
    topics = Topic.objects.all()
    if request.method == 'POST':
        topic_name = request.POST.get('topic')
        topic, created = Topic.objects.get_or_create(name=topic_name) # get or create function allows the created similar topic not to be saved as new topic again

        Room.objects.create(
            host = request.user,
            topic=topic,
            name=request.POST.get('name'),
            description=request.POST.get('description'),
        )

        return redirect('home')
    context = {'form' : form, 'topics':topics}
    return render(request, 'base/room_form.html', context)

@login_required(login_url='login')
def updateRoom(request, pk):
    room = Room.objects.get(id=pk)
    form = RoomForm(instance= room)
    topics = Topic.objects.all()

    if request.user != room.host:
        return HttpResponse('You arent allowed here!')

    if request.method == 'POST':
        topic_name = request.POST.get('topic')
        topic, created = Topic.objects.get_or_create(name=topic_name)
```

```
        room.name = request.POST.get('name')
        room.topic = topic
        room.description = request.POST.get('description')
        room.save()
        return redirect('home')



    context = {'form': form, 'topics':topics, 'room': room}

    return render(request, 'base/room_form.html', context)


@login_required(login_url='login')
def deleteRoom(request, pk):
    room = Room.objects.get(id = pk)
    if request.user != room.host:
        return HttpResponse('You arent allowed here!')
    if request.method == "POST":
        room.delete()
        return redirect('home')
    return render(request, 'base/delete.html', {'obj' : room})

@login_required(login_url='login')
def deleteMessage(request, pk):
    message = Message.objects.get(id=pk)
    if request.user != message.user:
        return HttpResponse('You arent allowed here!')
    if request.method == "POST":
        message.delete()
        return redirect('home')
```

```
    return render(request, 'base/delete.html', {'obj' : message})

@login_required(login_url = 'login')
def updateUser(request):
    user = request.user
    form = UserForm(instance=user)

    if request.method == 'POST':
        form = UserForm(request.POST, request.FILES, instance=user)
        if form.is_valid():
            form.save()
            return redirect('user-profile', pk = user.id)
    return render (request, 'base/update-user.html',{'form':form})

def topicsPage(request):
    q = request.GET.get('q') if request.GET.get('q') != None else ''

    topics = Topic.objects.filter(name__icontains=q)
    return render(request, 'base/topics.html', {'topics': topics})

def activityPage(request):
    room_messages = Message.objects.all()
    return render(request, 'base/activity.html', {'room_messages':room_messages})
```

Requirements File (requirements.txt):

```
```django
djangorestframework
django-cors-headers
Pillow ```
```
```

Pipeline Configurations

Jenkinsfile:

```
pipeline {
    agent any

    environment {
        GIT_REPO = 'https://github.com/Jitenrai21/python_django_studyroom_project.git'
    }

    stages {
        stage('Clone Repo') {
            steps {
                withCredentials([usernamePassword(credentialsId: 'GITHUB_PAT', usernameVariable: 'GIT_USER', passwordVariable: 'GIT_PAT')]) {
                    bat '''
                    if exist webapp (rmdir /s /q webapp)
                    git clone https://%GIT_USER%:%GIT_PAT%@github.com/Jitenrai21/python_django_studyroom_project.git webapp
                    '''
                }
            }
        }
        stage('Deploy to Staging') {
            steps {
                bat '''
                taskkill /F /IM python.exe > nul 2>&1
                cd webapp\\StudyRoomProject
                pip install -r ..\\requirements.txt
                start /B python manage.py runserver 0.0.0.0:8000
```

```
                '''
            }
        }

        stage('Manual Approval') {
            steps {
                input message: 'Deploy to Production?'
            }
        }

        stage('Deploy to Production') {
            steps {
                bat '''
                taskkill /F /IM python.exe > nul 2>&1
                cd webapp\\StudyRoomProject
                start /B python manage.py runserver 0.0.0.0:8001
                '''
            }
        }



        stage('testin to VM2') {
            steps {
                input message: 'Is working in VM2?'
            }
        }
    }
}
```

Screenshots / Demo Evidence

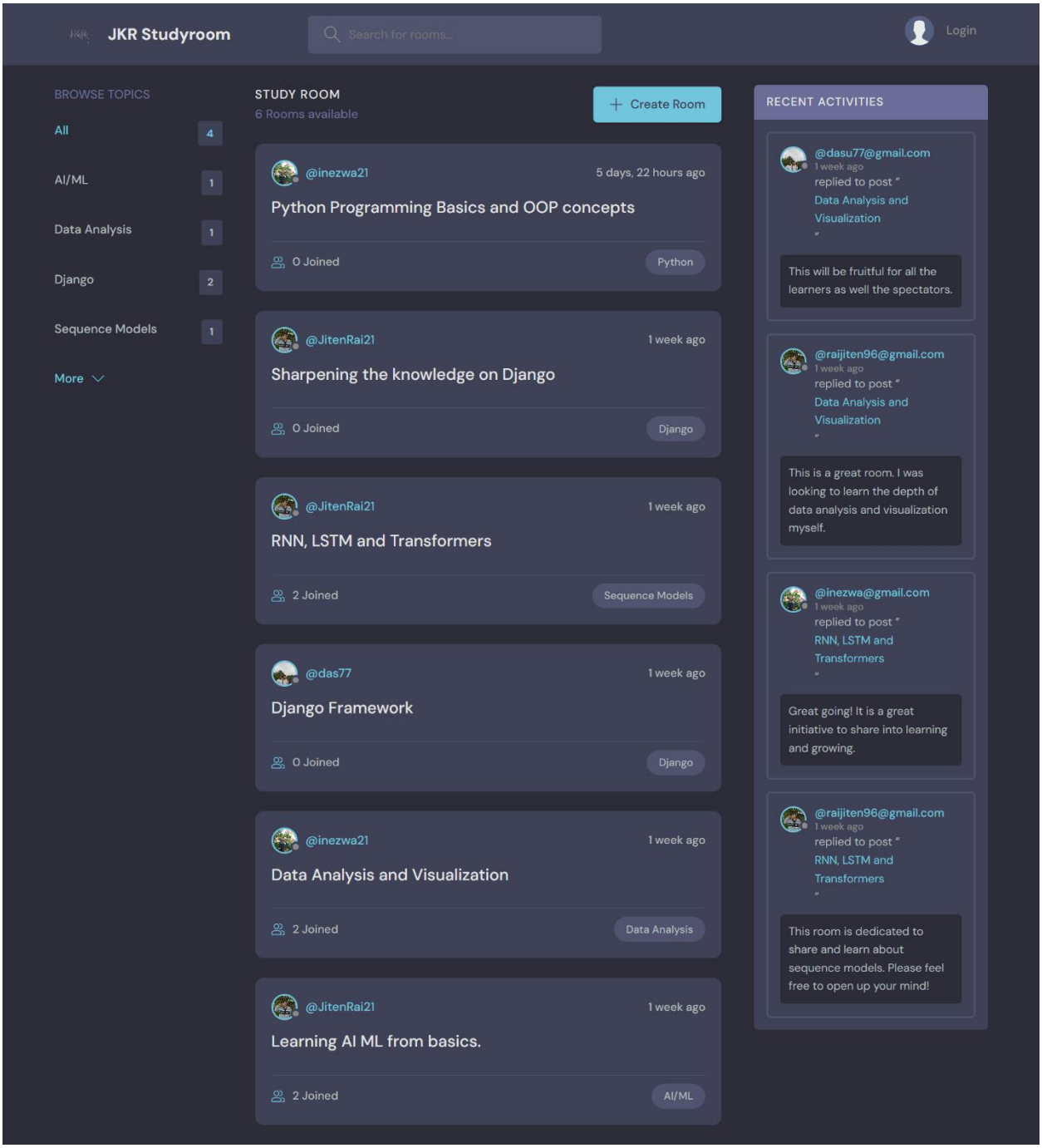

Figure 2: Landing Page of the webapp

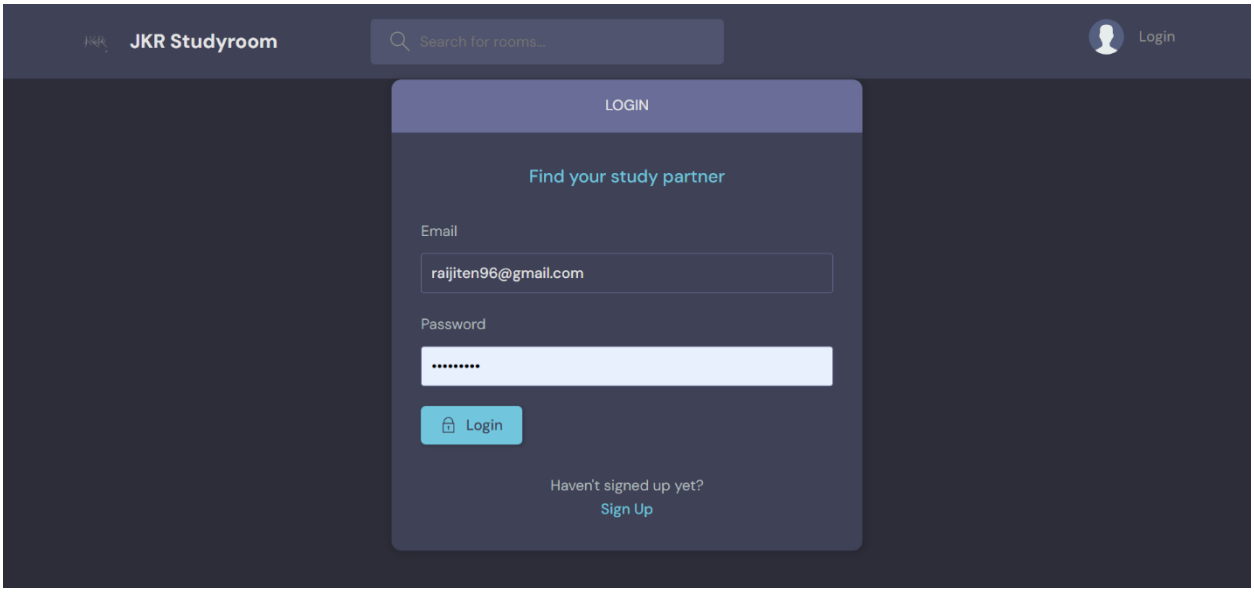

Figure 3: Login Page

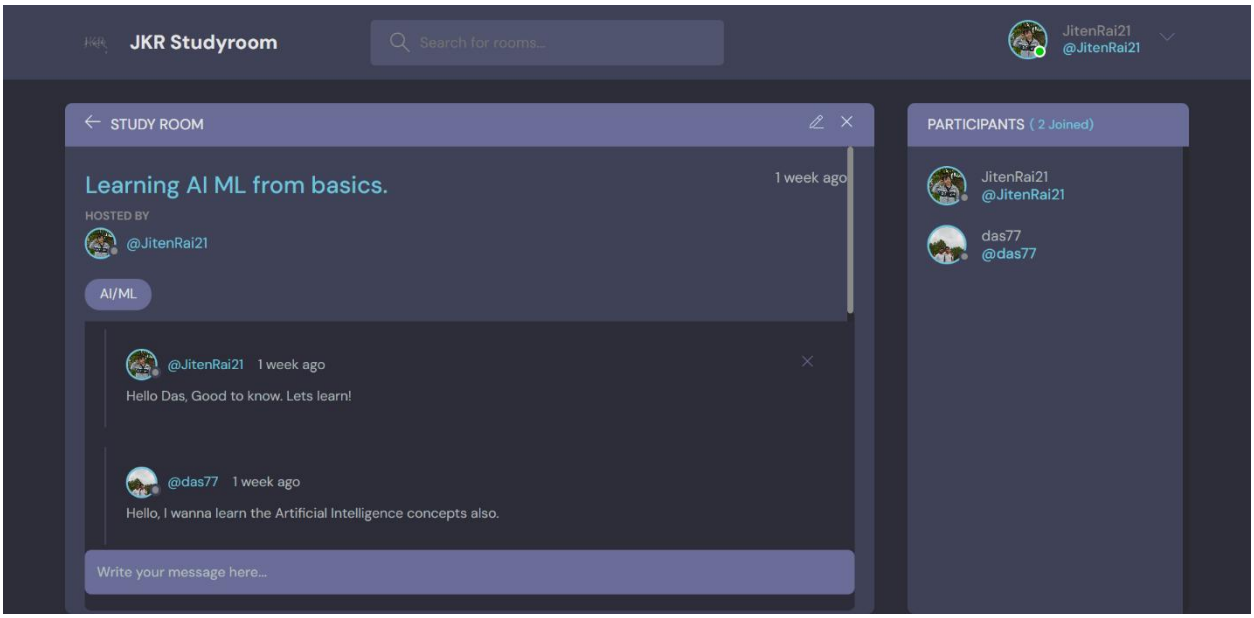

Figure 4: Room of webapp

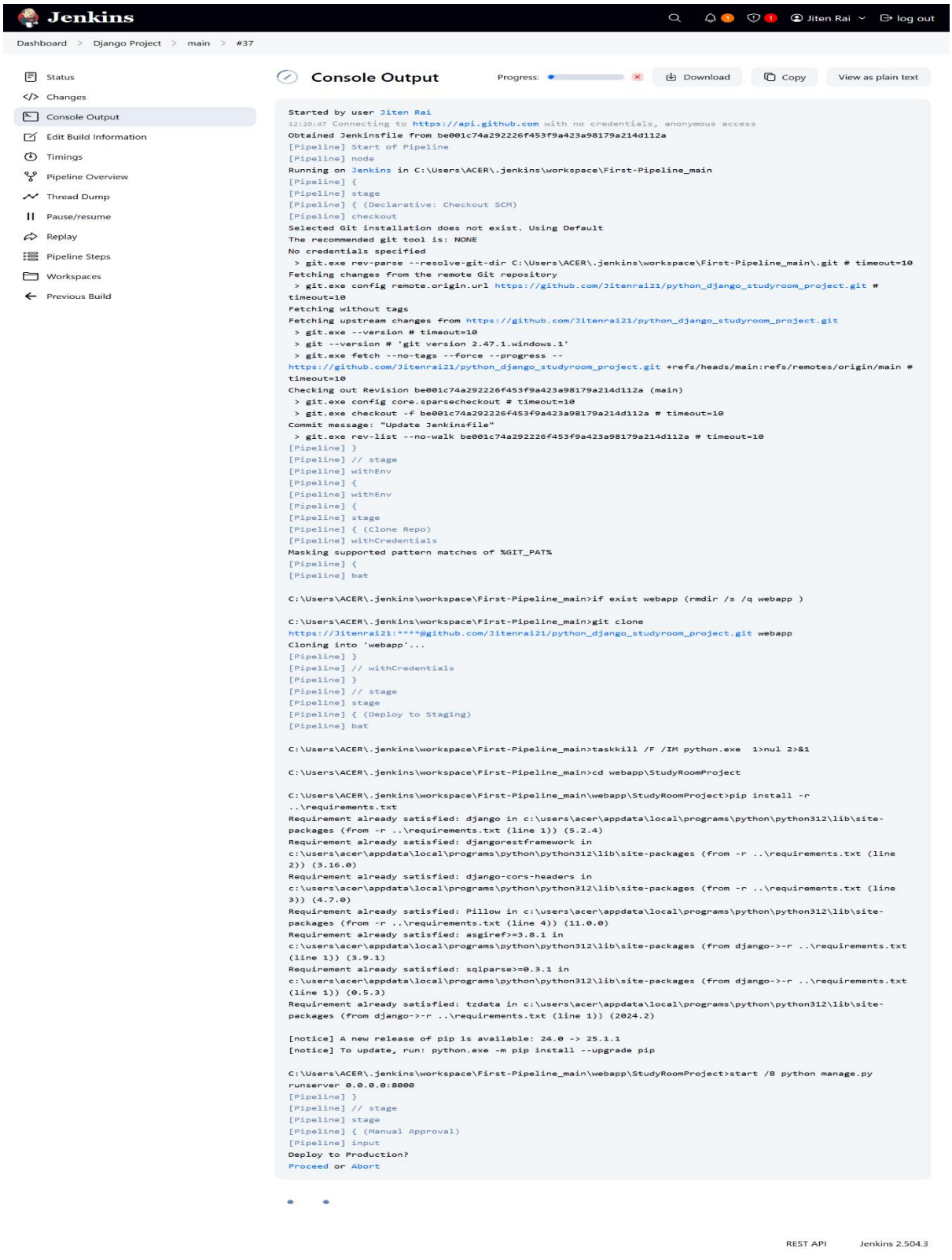

Figure 5: Jenkins cloning, staging execution log

```
Approved by Jiten Rai
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy to Production)
[Pipeline] bat

C:\Users\ACER\.jenkins\workspace\First-Pipeline_main>taskkill /F /IM python.exe  1>nul 2>&1

C:\Users\ACER\.jenkins\workspace\First-Pipeline_main>cd webapp\StudyRoomProject

C:\Users\ACER\.jenkins\workspace\First-Pipeline_main\webapp\StudyRoomProject>start /B python manage.py
runserver 0.0.0.0:8001
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (testin to VM2)
[Pipeline] input
Is working in VM2?
Proceed or Abort
```

Figure 6: Jenkins deploy to production execution

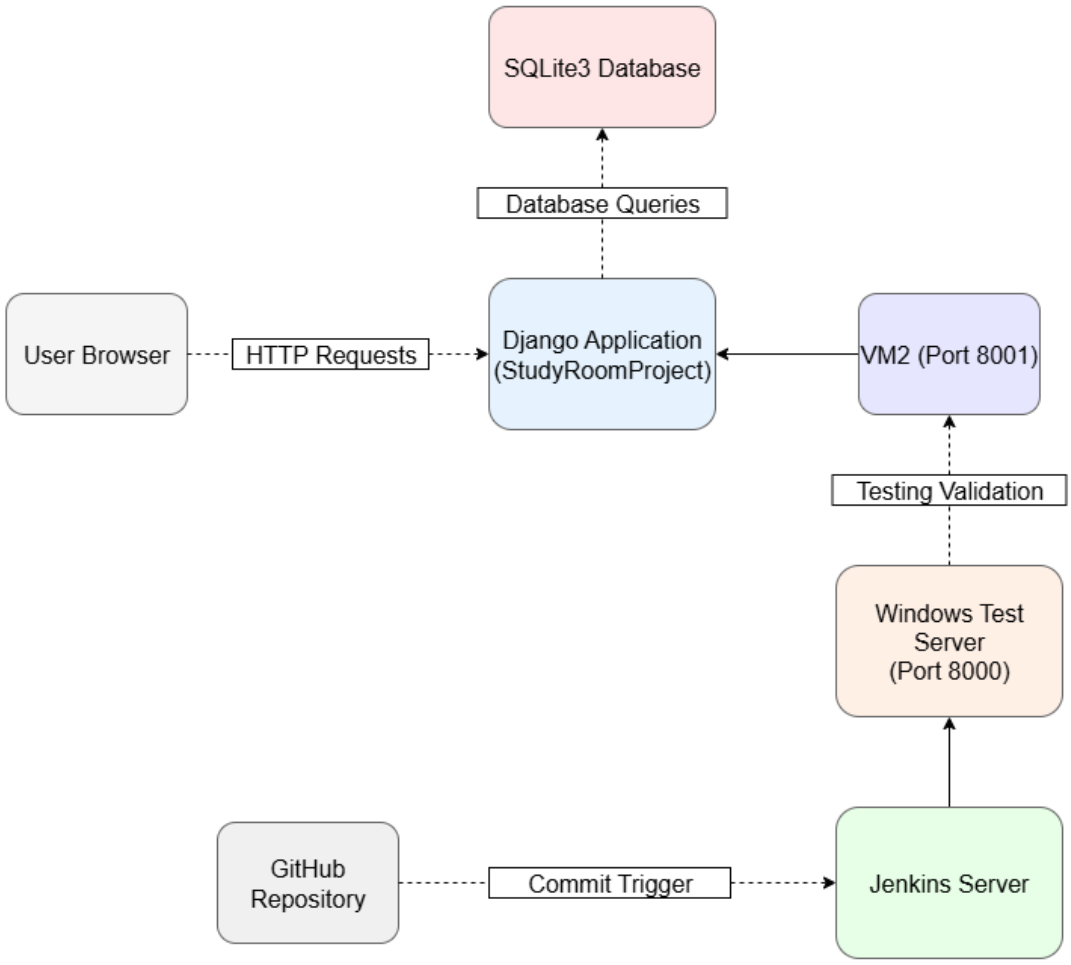

Figure 7: System Architecture