SSW 590

by

Ivan Farfan, Johan Jaramillo, Ryan Davis

Stevens.edu

October 23, 2025

SSW 590

Ivan Farfan, Johan Jaramillo, Ryan Davis
Stevens.edu

This document provides the requirements and design details of the PROJECT. The following table (Table 1) should be updated by authors whenever major changes are made to the architecture design or new components are added. Add updates to the top of the table. Most recent changes to the document should be seen first and the oldest last.

Table 1: Document Update History

Date	**Updates**
10/16/2025	RD, IF, JJ • Added Overleaf and GitHub Integration chapter for Homework 6, documenting the connection between Overleaf and GitHub. • Updated Hosts table to include the domain and address of the local LaTeX instance.
10/08/2025	JJ: • Configured and deployed the Bugzilla container on the DigitalOcean droplet. • Added Bugzilla chapter documenting setup, troubleshooting, and verification. IF: • Deployed and verified the Overleaf Toolkit container on DigitalOcean. • Added Overleaf chapter detailing configuration, Docker installation, and web access confirmation. RD: • Updated Hosts table for DigitalOcean. • Updated Passwords table for DigitalOcean.
10/08/2025	RD, IF, JJ • Updated Hosts table for Digital Ocean • Updated Passwords table for Digital Ocean • Added Bugzilla chapter displaying Bugzilla docker container configurations for Bugzilla and Overleaf Dockers on Digital Ocean. • Added Overleaf chapter displaying Overleaf Docker container configurations for Bugzilla and Overleaf Dockers on Digital Ocean assignment.

Table 1: Document Update History

Date	Updates
10/01/2025	RD, IF, JJ • Added AWS deployment chapter for Docker on AWS homework and added LaTeX Docker.
09/24/2025	RD, IF, JJ • Added Project Proposal chapter for our project Figma2AWS
09/14/2025	RD, IF, JJ • Provided solutions for the Linux Problem Set in the Linux Commands assignment.
09/03/2025	RD, IF, JJ: • Created Kanban board on GitHub figures, and other labels.

Table of Contents

List of Tables

List of Figures

Chapter 1

Kanban Setup

– Ivan Farfan Diaz, Johan Jaramillo

In order to setup our Kanban board, we used Github Projects. The process consisted on:

1. **Creating a Project** – Going to our repository, clicking “Projects” tab, then “New project”
2. **Choosing a Template** – Selecting “Board” or “Kanban” template (or starting from scratch)
3. **Setting Up Columns** – Creating workflow columns like “To Do,” “In Progress,” “In Review,” “Done”
4. **Configuring Settings** – Naming our project, adding description, setting visibility (public/private)

Chapter 2

Passwords

– Ivan Farfan, Johan Jaramillo, Ryan Davis

Category	Password Rule
Server Rules	• We use a prefix for the site (which city the server is in), then a descriptive shorthand for what the server does. • Example: `XXFS01` would be a file server in location `XX`. • If there were a second file server in the same location, it would be named `XXFS02`.
User Rules	• Passwords must be changed every 60 days. • Users cannot reuse the last 4 passwords. • Accounts will be locked after 3 failed login attempts.
General Rules	• Passwords must have at least 10 characters. • Passwords must include at least one uppercase letter. • Passwords must include at least one lowercase letter. • Passwords must include at least one special character (e.g., !, @, #, $).
Custom Rule	• Include the abbreviation of the operating system you use. • Finish with a memorable number, such as the year of graduation or the current semester.

Category	**Password Rule**
Digital Ocean	• Follows the standardized password pattern: <coursetag><coursenumber><termLetter>@<serviceInitials> • The initials do represent DigitalOcean. • This ensures service-specific uniqueness across all deployed systems.

Hints

- User passwords often mix in course codes, semesters, or graduation years.
- General passwords follow the 10+ character, uppercase, lowercase, and special character requirements.
- Custom group passwords contain a Mac/OS reference combined with a number tied to student life.

Chapter 3

Hosts

– Ivan Farfan, Johan Jaramillo, Ryan Davis

This chapter lists the primary hosts used for the project deployment. Each host represents a service or container deployed within the shared DigitalOcean environment. IP addresses and configurations are specific to the development setup for Fall 2025.

Table 3.1: Project Hosts and Roles

Hostname / Service	**IP Address**	**Operating System**	**Role / Purpose**
Overleaf Container	167.71.181.60	Ubuntu 25.04	Overleaf LaTeX Toolkit deployed via Docker; provides collaborative document editing capabilities.
Bugzilla Container	167.71.179.151	Ubuntu 25.04	Bugzilla issue tracking system deployed in a Docker container for project management and defect tracking.
DigitalOcean Droplet	N/A	Ubuntu 25.04	Cloud infrastructure host running both the Overleaf and Bugzilla containers within the same droplet environment.
Local Overleaf	ssw590f25.me	Nginx and Ubuntu 25.04	Public domain that runs our local overleaf version.

Chapter 4

Linux Commands

– Ivan Farfan, Johan Jaramillo, Ryan Davis

Environment Setup

The following bash commands were executed in a Linux terminal to create the test environment:

Chapter 5

Project Proposal

– Ivan Farfan, Johan Jaramillo, Ryan Davis

5.1 Figma2AWS

Project Description

We aim to create a plugin tool for the Figma UI/UX design tool, which allows users to turn their designs from the Figma software into a static and simple website with HTML and JavaScript.

Sample Task and DevSecOps Tools

Throughout our DevSecOps processes, we plan on using GitHub for source control. We'll create a pipeline consisting in transforming the Figma design into HTML, CSS and JS code by using the Figma Composio MCP Server.

Then, we will containerize the web application using Docker and deploy the Docker image to AWS by using the App Runner service. The user will obtain a default AWS domain where the website will be deployed, although they will have to provide an AWS key to use the plugin.

Chapter 6

AWS Deployment

– Ivan Farfan, Johan Jaramillo, Ryan Davis

6.1 Deployment Steps & Commands (Local → AWS)

This section summarizes the steps I used to containerize the two-buttons site locally and deploy it to AWS. The flow follows our prior Docker and AWS notes. :contentReference[oaicite:0]index=0

6.1.1 Local Build & Run (Docker)

```
# From the app directory containing Dockerfile (Express + /public)
docker build -t color-buttons-app .
docker run -p 8080:3000 color-buttons-app
# Visit http://localhost:8080
```

6.1.2 Authenticate & Push to ECR

```
# (Set once per shell; edit to your values)
export AWS_REGION=us-east-1
export ECR_REPO=color-buttons-app
export IMAGE_TAG=v1
export AWS_ACCOUNT_ID="$(aws sts get-caller-identity --query Account --output text)"

# Create (idempotent) repo if not present
aws ecr describe-repositories \
  --repository-names "$ECR_REPO" \
  --region "$AWS_REGION" >/dev/null 2>&1 || \
aws ecr create-repository \
  --repository-name "$ECR_REPO" \
  --image-scanning-configuration scanOnPush=true \
  --region "$AWS_REGION"

# Login Docker to ECR
```

```
aws ecr get-login-password --region "$AWS_REGION" \
| docker login --username AWS --password-stdin \
  "$AWS_ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com"

# Build for common runtime (x86) if on Apple Silicon
docker build --platform linux/amd64 -t "$ECR_REPO:$IMAGE_TAG" .

# Tag \& push to ECR
docker tag "$ECR_REPO:$IMAGE_TAG" \
  "$AWS_ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/$ECR_REPO:$IMAGE_TAG"

docker push "$AWS_ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/$ECR_REPO:$IMAGE_TAG"
```

6.1.3 Quick Deploy with AWS App Runner

```
export APP_NAME=color-buttons-apprunner
export CONTAINER_PORT=3000

aws apprunner create-service \
  --service-name "$APP_NAME" \
  --region "$AWS_REGION" \
  --source-configuration "{
    \"ImageRepository\": {
      \"ImageIdentifier\": \"$AWS_ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/$ECR_RE
      \"ImageRepositoryType\": \"ECR\",
      \"ImageConfiguration\": {\"Port\": \"$CONTAINER_PORT\"}
    },
    \"AutoDeploymentsEnabled\": true
  }" \
  --instance-configuration "{\"Cpu\":\"1 vCPU\",\"Memory\":\"2 GB\"}"
```

After creation, obtain the public URL:

```
aws apprunner list-services \
  --region "$AWS_REGION" \
  --query "ServiceSummaryList[?ServiceName=='$APP_NAME'].ServiceUrl" \
  --output text
```

6.1.4 Live Links

AWS App Runner URL:
https://6ibmrtgk5s.us-east-2.awsapprunner.com/

6.2 Updated Design Class Diagram

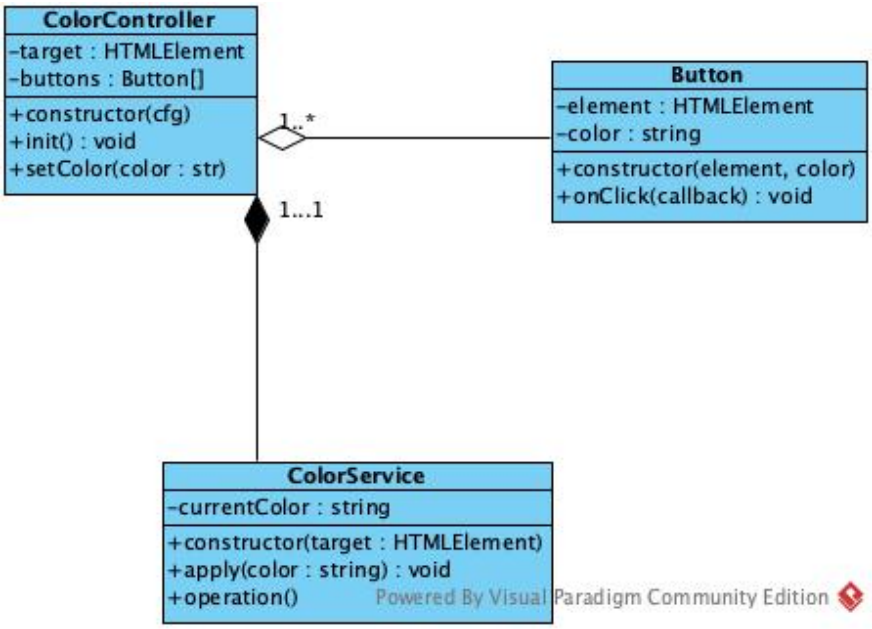


Figure 6.1: UML Class Diagram for the Refactored Website

Chapter 7

LaTeX Docker

– Ivan Farfan, Johan Jaramillo, Ryan Davis

7.1 Project Layout

Create a folder with these files:

```
.Dockerfile
main.tex
```

7.2 Dockerfile

This Dockerfile installs TeX Live and sets up a container to compile LaTeX.

```
FROM debian:bookworm-slim
RUN apt-get update && apt-get install -y \
    texlive-latex-base \
    texlive-latex-recommended \
    texlive-latex-extra \
    texlive-fonts-recommended \
    latexmk \
    python3-pygments \
 && apt-get clean && rm -rf /var/lib/apt/lists/*
WORKDIR /data
CMD ["latexmk", "-pdf", "-shell-escape", "main.tex"]
```

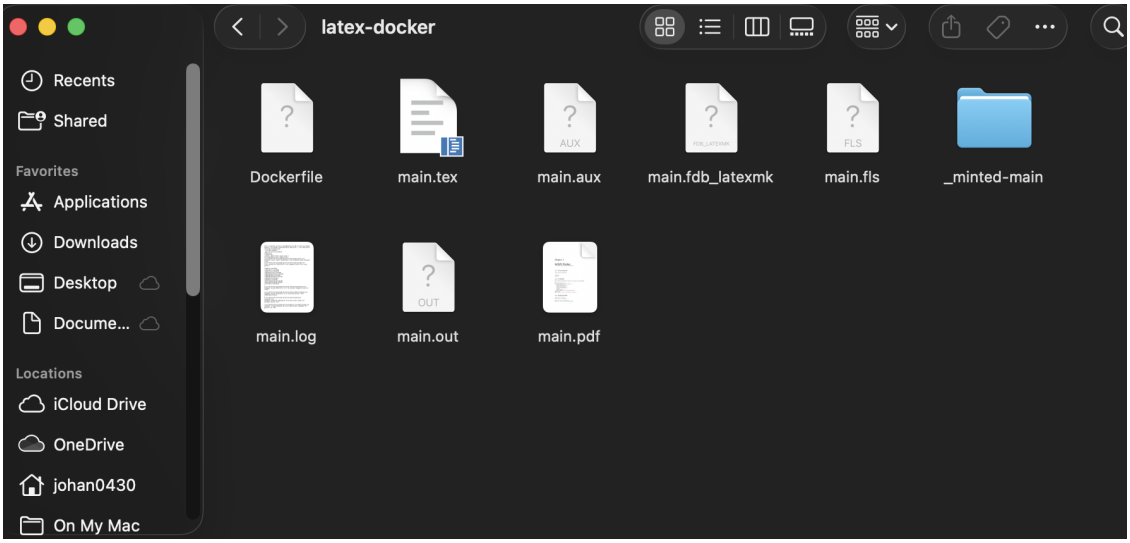
7.3 Build and Run

```
# Build the Docker image
docker build -t latex-docker .

# Run the container (Linux/macOS)
docker run --rm -v $(pwd):/data latex-docker
```

```
# Run the container (Windows PowerShell)
docker run --rm -v ${PWD}:/data latex-docker

# Run the container (Windows CMD)
docker run --rm -v %cd%:/data latex-docker
```

7.4 Screenshot

The following screenshot shows the result of compiling our LaTeX document inside the Docker container.



7.5 Conclusion

In this chapter we showed how to use Docker to build a container that can compile LaTeX documents with TeX Live. The steps included writing a Dockerfile, setting up the folder structure, and running basic commands to build and run the container. This setup is similar to how Overleaf works behind the scenes. Even though our example was simple, it demonstrates how Docker can make LaTeX compilation consistent and portable across different systems.

Chapter 8

Bugzilla

– Ivan Farfan, Johan Jaramillo, Ryan Davis

8.1 Overview

For this part of the assignment, Bugzilla, a web-based bug tracking system, was deployed on a DigitalOcean Droplet using Docker containers. Instead of using a prebuilt image with MariaDB included, we configured two separate containers: one running Bugzilla and another running MySQL as the database backend. This approach ensured greater flexibility and easier debugging when dealing with database connection issues.
The Bugzilla image used was the official image available on Docker Hub:

https://hub.docker.com/r/bugzilla/bugzilla-dev

8.2 Configuration Steps

8.2.1 Droplet Setup

- A new DigitalOcean Droplet was created running `Ubuntu 25.04`.
- Docker was installed using the standard package manager:

```
apt update
apt install docker.io -y
```

- Verified that Docker was running correctly with:

```
systemctl status docker
```

8.2.2 MySQL Container Configuration

- Pulled the MySQL 5.7 image and started the container:

```
docker run -d \
  --name bugzilla-mysql \
  -e MYSQL_ROOT_PASSWORD=root \
  -e MYSQL_DATABASE=bugs \
  -e MYSQL_USER=bugs \
  -e MYSQL_PASSWORD=bugspass \
  mysql:5.7
```

- This container served as the database backend for Bugzilla.
- Once running, the container could be verified using:

```
docker ps
```

8.2.3 Bugzilla Container Setup

- Pulled and started the Bugzilla container:

```
docker run -d \
  --name bugzilla \
  -p 8080:80 \
  --link bugzilla-mysql:mysql \
  bugzilla/bugzilla-dev
```

- Accessed the Bugzilla container shell:

```
docker exec -it bugzilla bash
cd /var/www/html/bugzilla
```

- Edited the `localconfig` file to match the MySQL settings:

```
$db_driver = 'mysql';
$db_name   = 'bugs';
$db_user   = 'bugs';
$db_pass   = 'bugspass';
$db_host   = 'bugzilla-mysql';
```

- Ran the initial setup:

```
./checksetup.pl
```

- Created the Bugzilla administrator account when prompted.

8.2.4 Troubleshooting and Resource Limitations

During deployment, the initial droplet used was one of the smallest and most affordable DigitalOcean options, which lacked sufficient memory to run both containers simultaneously. This caused the MySQL container to fail with the following error:

```
Can't connect to local MySQL server through socket '/var/lib/mysql/mysql.sock'
```

To resolve this issue:

- The droplet was temporarily powered off to modify its plan.
- It was upgraded to the following configuration:

Plan Type	Basic Premium Intel
vCPUs	2
Memory	2 GB
Disk	25 GB
Bandwidth	3 TB
Cost	$24/month ($0.036/hr)

- After resizing, both containers were able to start successfully without memory-related errors.

8.2.5 Apache Restart and Verification

- Restarted Apache within the Bugzilla container:

```
apachectl restart
```

- Verified that the Bugzilla web interface was accessible through:

```
http://167.71.179.151:8080/bugzilla/
```

8.3 Result

After adjusting the droplet resources and reconfiguring the database settings, Bugzilla successfully initialized and connected to the MySQL backend. The administrator account was created, and the web interface became fully functional and accessible through the public IP and port 8080.

8.4 Container Web Access

To confirm that the Bugzilla instance was properly running and publicly accessible, the container was tested by visiting the droplet's IP address on port 8080. The Bugzilla login interface loaded successfully, verifying that both the Apache server and MySQL backend were operational.

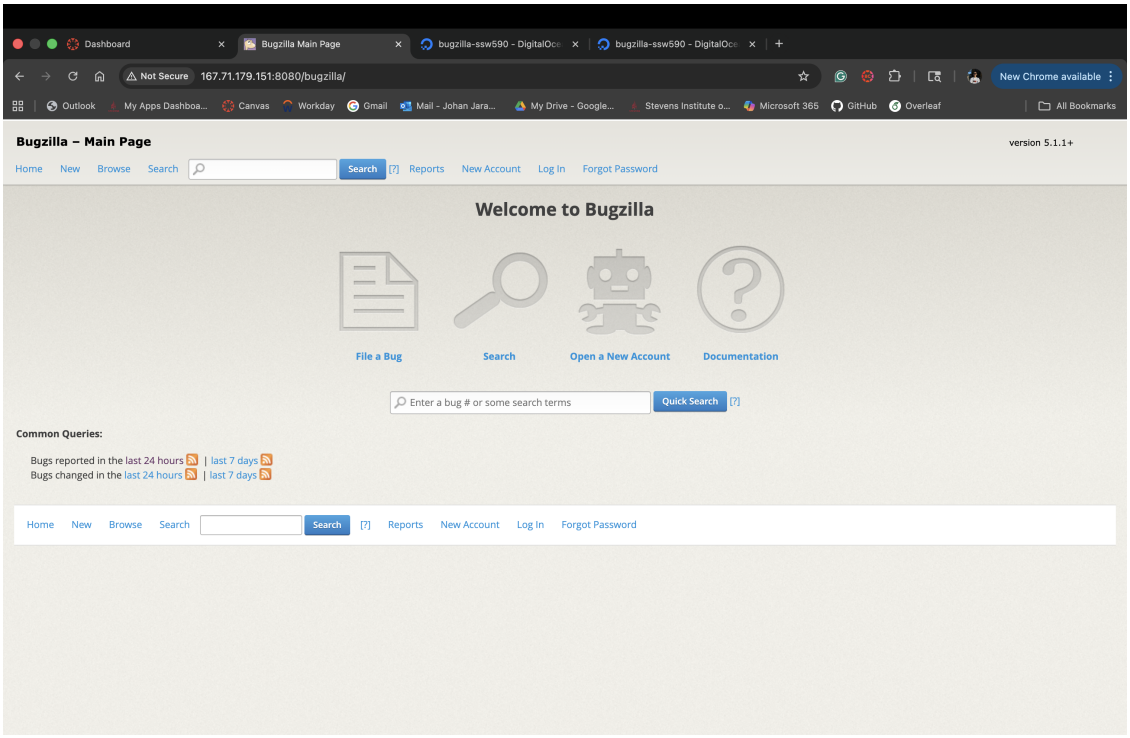


Figure 8.1: Bugzilla web interface confirming successful container deployment on port 8080.

Chapter 9

Overleaf

– Ivan Farfan, Johan Jaramillo, Ryan Davis

9.1 Overview

For this part of the assignment, Overleaf, a collaborative LaTeX writing platform, was deployed on a DigitalOcean Droplet using Docker containers. The setup required installing Docker and the Docker Compose plugin, verifying the Docker service, and configuring the Overleaf Toolkit container. This approach allowed Overleaf to run directly on port 80, making it accessible through the droplet's public IP.
The Overleaf Toolkit used was obtained from the official GitHub repository:

https://github.com/overleaf/toolkit

9.2 Configuration Steps

9.2.1 Droplet Setup

- A new DigitalOcean Droplet was created running `Ubuntu 25.04`.
- Docker was installed using the following command:

  ```
  sudo apt install docker.io
  ```

- Enabled Docker to automatically start on boot:

  ```
  sudo systemctl enable docker
  ```

9.2.2 Docker Compose Plugin Installation

- Installed required certificates and tools, then added Docker's official GPG key:

  ```
  sudo apt-get update
  sudo apt-get install ca-certificates curl
  sudo install -m 0755 -d /etc/apt/keyrings
  ```

```
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
↪  /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc
```

- Added Docker's repository to the Apt sources:

```
echo \
  "deb [arch=$(dpkg --print-architecture)
  ↪  signed-by=/etc/apt/keyrings/docker.asc] \
  https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "${UBUNTU_CODENAME:-$VERSION_CODENAME}")
  ↪  stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

- Installed Docker Engine, CLI, and Compose plugin:

```
sudo apt-get install docker-ce docker-ce-cli containerd.io \
docker-buildx-plugin docker-compose-plugin
```

9.2.3 Verifying Docker Status

- Checked that Docker was active and running:

```
sudo systemctl status docker
```

- The response confirmed a successful installation:

```
docker.service - Docker Application Container Engine
     Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled)
     Active: active (running) since Wed 2025-10-08 19:22:46 UTC
     Main PID: 55151 (dockerd)
     Tasks: 8
     Memory: 43.1M
     CPU: 627ms
```

9.3 Overleaf Toolkit Installation

9.3.1 Preparing the Environment

- Created a directory to hold the container environment:

```
mkdir Overleaf_container
cd Overleaf_container
```

- Cloned the official Overleaf Toolkit repository:

```
git clone https://github.com/overleaf/toolkit.git
cd toolkit
```

9.3.2 Configuration

- Initialized the configuration files:

  `bin/init`

- Edited the Overleaf configuration to listen on all network interfaces:

  `nano config/overleaf.rc`

- Updated the following line to allow access from the droplet's public IP:

  `OVERLEAF_LISTEN_IP=0.0.0.0`

9.4 Starting Overleaf

- Launched the Overleaf container stack using the toolkit command:

  `bin/up`

- This downloaded all required images (MongoDB, Redis, and the Overleaf web app) and automatically started the containers.

- Once setup completed, Overleaf was accessible on port 80 of the droplet.

9.5 Result

After installation, Overleaf successfully deployed and became accessible through the droplet's public IP address. All Docker services were active, and the platform functioned correctly for collaborative LaTeX editing.

9.6 Container Web Access

To verify that the Overleaf container was successfully deployed and reachable via the assigned port, the application was accessed through the droplet's public IP address using a web browser. The interface loaded correctly on port 80, confirming that the Docker service and network configuration were functioning as expected.

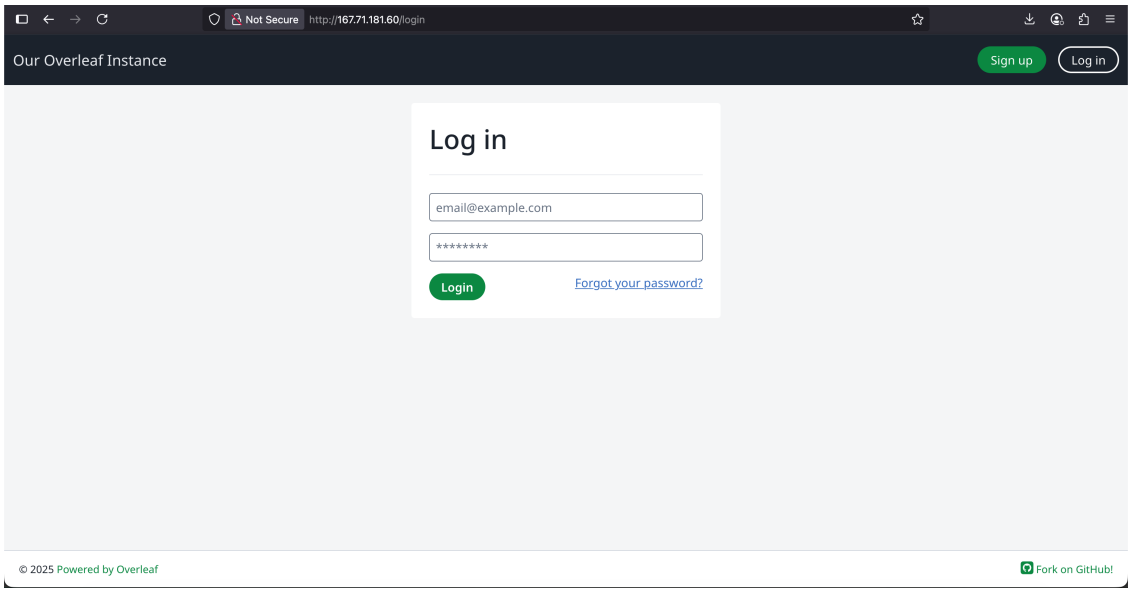


Figure 9.1: Overleaf web interface accessible through port 80 on the droplet.

Chapter 10

Overleaf and Github Integration

– Ivan Farfan, Johan Jaramillo, Ryan Davis

10.1 Overview

For this assignment, we performed the following tasks: getting a domain via the GitHub Student Developer Pack, getting an SSL certificate for our overleaf domain, configure Latex / Overleaf to support all packages, connecting Overleaf to Github for project version control, compiling TEX projects from the command line.

10.2 Domain Name

We decided to register the domain `ssw590f25.me` using the GitHub Student Developer Pack, which provided a free one-year domain through Namecheap. This allowed us to set up a professional web address for our project with minimal cost and effort. We had to add an A record using the Overleaf container droplet IP as host so that all traffic to the domain gets redirected to the local version of overleaf we're hosting.

10.3 SSL Configuration

To enable HTTPS on our Overleaf container, we installed and configured SSL certificates from Let's Encrypt using `certbot`. This ensures secure encrypted communication between users and the server.

```
apt install python3-certbot-nginx nginx
sudo certbot --nginx certonly
```

Once the certificate was obtained, we configured Nginx to serve as a reverse proxy — forwarding client requests to the Overleaf service while handling SSL termination.

```
server {
    listen 80;
```

```
    server_name overleaf.ssw590f25.me;
    return 301 https://$server_name$request_uri;
}

server {
    listen 443 ssl;
    server_name overleaf.ssw590f25.me;

    ssl_certificate /etc/letsencrypt/live/overleaf.ssw590f25.me/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/overleaf.ssw590f25.me/privkey.pem;

    location / {
        proxy_pass http://localhost:8877;
        proxy_set_header X-Forwarded-For $remote_addr;
    }
}
```

Configuring Nginx in this way is crucial because it acts as a reverse proxy—managing secure HTTPS traffic, terminating SSL connections, and forwarding requests to the internal Overleaf container.

10.4 Configuring LaTeX Packages in Overleaf (Container)

To enable full LaTeX package support inside the Overleaf container, we used the following commands:

Steps

1) Update `tlmgr` (TeX Live Manager).

```
docker exec sharelatex tlmgr update --self
```

Why: Brings the package manager itself up-to-date so subsequent installs use the latest metadata.

2) Install the full TeX Live package set.

```
docker exec sharelatex tlmgr install scheme-full
```

Why: `scheme-full` pulls essentially all TeX Live packages, covering most class/style/font dependencies you'll encounter. **Note:** This took a long time and it's not permanent, it will reset to the basic Latex version if the container is rebuilt.

10.5 Connecting Overleaf to GitHub

We connected our Overleaf instance at `overleaf.ssw590f25.me` to GitHub to automatically sync compiled project files.

Generating SSH Keys and Configuring Access

```
ssh-keygen -t ed25519 -f /root/.ssh/id_ed25519_exports -N "" -C "overleaf-exports"
ssh-keyscan github.com >> /root/.ssh/known_hosts
chmod 600 /root/.ssh/known_hosts
cat /root/.ssh/id_ed25519_exports.pub
ssh -i /root/.ssh/id_ed25519_exports -T git@github.com
```

Response:

```
Hi IvanFarfan08/Overleaf_Files_SSW590F! You've successfully authenticated,
but GitHub does not provide shell access.
```

Configured Git to always use this key:

```
cat >> /root/.ssh/config <<'EOF'
Host github.com
  User git
  HostName github.com
  IdentityFile ~/.ssh/id_ed25519_exports
  IdentitiesOnly yes
EOF
chmod 600 /root/.ssh/config
```

The public key was then added as a deployment key in the GitHub repository.

Creating Export Service

```
SRC="/root/Overleaf_container/toolkit/data/overleaf/data/compiles"
DST="/root/overleaf_exports"
REMOTE="git@github.com:IvanFarfan08/Overleaf_Files_SSW590F.git"

sudo apt update
sudo apt install -y git inotify-tools rsync
```

Initialized the staging repository:

```
sudo mkdir -p "$DST"
cd "$DST"
git init -b main
git config user.name  "Overleaf Export Bot"
git config user.email "export@overleaf.local"

cat > .gitignore <<'EOF'
*.aux
*.log
*.out
```

```
*.synctex.gz
EOF

git add -A
git commit -m "Initial staging repo"

ssh-keyscan github.com >> /root/.ssh/known_hosts
chmod 600 /root/.ssh/known_hosts
GIT_SSH_COMMAND="ssh -i $KEY -o StrictHostKeyChecking=yes" git push -u origin main
```

Watcher Script

```
sudo tee /usr/local/bin/compiles-export-push.sh >/dev/null <<'EOF'
#!/usr/bin/env bash
set -euo pipefail

SRC="/root/Overleaf_container/toolkit/data/overleaf/data/compiles"
DST="/root/overleaf_exports"
BRANCH="main"
KEY="/root/.ssh/id_ed25519_exports"

export GIT_SSH_COMMAND="ssh -i $KEY -o StrictHostKeyChecking=yes"

mkdir -p "$DST"

copy_and_commit() {
  local top="$1"
  rsync -a --delete "$SRC/$top/" "$DST/$top/"
  cd "$DST"
  git add "$top"
  if ! git diff --cached --quiet; then
    msg="export: $top @ $(date -u +'%Y-%m-%d %H:%M:%S %Z')"
    git commit -m "$msg"
    git push origin "$BRANCH"
    echo "Pushed: $msg"
  fi
}

DEBOUNCE=5
declare -A last_at

inotifywait -m -r \
  -e close_write,modify,attrib,create,delete,move \
  --format '%w%f' "$SRC" | while read -r path; do
    rel="${path#"$SRC/"}"
```

```
    top="${rel%%/*}"
    [ -n "$top" ] || continue

    now=$(date +%s)
    last=${last_at[$top]:-0}
    if (( now - last >= DEBOUNCE )); then
      last_at[$top]=$now
      echo "Detected change in $top"
      copy_and_commit "$top" || echo "Commit/push failed for $top"
    fi
  done
EOF

sudo chmod +x /usr/local/bin/compiles-export-push.sh
```

This script continuously monitors the Overleaf compile directory for any changes using `inotifywait`. Whenever a project is recompiled, it automatically syncs the updated subfolder to the local Git staging directory using `rsync`, commits the changes, and pushes them to the GitHub repository. The `DEBOUNCE` variable ensures that each subfolder is only pushed once every few seconds, preventing redundant commits during rapid file writes.

Systemd Service

```
sudo tee /etc/systemd/system/compiles-export-push.service >/dev/null <<'EOF'
[Unit]
Description=Export changed Overleaf compiles subfolders to root-owned repo and push
After=network-online.target
Wants=network-online.target

[Service]
Type=simple
User=root
ExecStart=/usr/local/bin/compiles-export-push.sh
Restart=always
RestartSec=2

[Install]
WantedBy=multi-user.target
EOF

sudo systemctl daemon-reload
sudo systemctl enable --now compiles-export-push.service
```

The Systemd service runs the watcher script automatically in the background. It starts on boot, restarts on failure, and ensures continuous syncing between Overleaf and GitHub. By enabling it

with `systemctl enable --now`, the service remains active and constantly monitors the compile directory without manual intervention. The service keeps updating projects (including this one) and is available at this Github repo.

10.6 Compiling Overleaf Projects from the Command Line

We found that compiled versions of Overleaf projects are stored in toolkit/data/overleaf/data/compiles.

Each folder in this directory is mapped to a unique `projectID_userID` pair.

This means that one can enter the Docker image created in Chapter 7 and use the instructions there to compile the project files. However, Overleaf automatically generates compiled PDF files for each project, and these can already be found in the corresponding folder under the `compiles` directory.

Bibliography

Index