SUPPLYTOU IMPLEMENTATION DOCUMENTATION

1. Introduction

This chapter documents the actual implementation phase of the Supply2U platform. The system is designed to serve farmers, buyers, drivers, and administrators in a unified platform to manage farms, crops, weather conditions, soil health, orders, and transport logistics.

Following a requirements analysis and architectural design phase, this stage focused on translating design specifications into functional modules through iterative development. The project adhered to Agile methodology, with development divided into structured sprints that facilitated rapid prototyping, validation, and feedback integration.

Technologies employed included Django REST Framework for the backend, PostgreSQL + PostGIS for relational and geospatial data storage, Redis + Celery for background tasks, and Djoser + JWT for secure user authentication.

Tools such as Postman, GitHub, Docker, and Pytest/unittest were used for development, testing, and version control. All APIs were tested iteratively, and continuous feedback loops enabled quick resolution of bugs and alignment with evolving project requirements.

2. Development Environment

This section describes the setup used for developing, testing, and deploying the Supply2U platform. It outlines the programming languages, frameworks, tools, and services that support the application's development lifecycle, following Agile and DevOps best practices.

The platform is built using a modern technology stack to ensure scalability, real-time communication, and geospatial data handling. Version control, collaborative development, testing, and automated deployment processes are managed using cloud-based tools and continuous integration/continuous deployment (CI/CD) pipelines.

Component	**Technology / Tool**
Backend	Python 3.13, Django REST Framework

Geospatial Database	PostgreSQL 15 + PostGIS Extension
Authentication	Djoser + JWT with Secure Cookies
Asynchronous Tasks	Celery + Redis
Testing Tools	Django unittest, mock.patch
API Testing	Postman
Version Control	Git + GitHub
Integrated Development Environment (IDE)	Visual Studio Code (VS Code)
Task Automation	Custom management commands + crontab
Deployment	(Planned) Docker + Azure

3. Implementation Strategy (Agile & Sprint Breakdown)

The implementation of the platform followed the Agile software development methodology, focusing on iterative development, continuous feedback, and progressive system refinement. The system was implemented across five Agile sprints, each approximately one week long. Daily commits, incremental builds, and regular testing cycles ensured stability, quality, and alignment with project goals throughout development.

Each sprint had specific deliverables, aimed at building and integrating functional modules in a systematic and controlled manner.

Sprint	**Activities**	**Deliverables**

Sprint 1	Set up Django project, configure PostgreSQL + PostGIS, initialize GitHub repository	Working repository, initial models (User, Farm)
Sprint 2	Implement user authentication with JWT and cookie storage, define role-based access	User login/logout functionality and secure protected routes
Sprint 3	Create modules for Farm, Crop, Soil, and Weather	API endpoints for farm management, crop tracking, and soil records
Sprint 4	Implement orders, transport management, and signal-based evaluations	Dynamic order workflow integration, Celery background task for weather updates
Sprint 5	Conduct testing, debugging, validation, and documentation generation	Unit tests, final ER diagram, system-wide integration and optimization

4. Module-by-Module Implementation

This section provides a detailed breakdown of how the major components of the Supply2U platform were implemented. Each module was developed to address specific functional requirements, ensuring seamless integration with other system components while adhering to best practices in software engineering.

a) User Authentication Module

Purpose

The User Authentication Module ensures secure access to the platform by validating user credentials and enforcing role-based permissions. It integrates with the frontend and backend to provide a seamless login experience while safeguarding sensitive data.

Implementation Details

JWT-Based Login:

- Implemented using Djoser and SimpleJWT for token generation and validation.
- Tokens are stored in HttpOnly cookies to prevent XSS attacks.
- Secure and SameSite cookie policies are enforced to mitigate CSRF risks.

Role-Based Access Control (RBAC):

- Users are assigned roles (FARMER, BUYER, DRIVER, ADMIN) during registration.
- Custom permission classes (IsOwner, IsFarmerOrAdmin, IsAuthenticated) restrict access to specific endpoints based on user roles.
- Example: Only FARMERs can create or modify farms, while BUYERs can only view their orders.

Custom Authentication Class:

- Overrides Django REST Framework's default token handling to extract tokens from cookies instead of headers.
- Ensures compatibility with browser-based flows and secure session persistence.

Code Testing

- Unit tests validated token generation, cookie storage, and role-based permissions.
- Postman was used to manually test login/logout flows and endpoint access control.

b) Core Business Logic

Purpose

The Core Business Logic Module handles the platform's primary functionalities, including farm management, crop tracking, order processing, and transport logistics. It ensures data consistency and enforces business rules across all operations.

Implementation Details

Farm Management:

- Farmers can create, update, and delete farms.
- Geospatial data (farm boundaries, locations) is stored using PostGIS fields (PolygonField, PointField).
- Validation ensures only FARMERs can own farms (enforced via model clean() method).

Crop and Soil Tracking:

- Crops are linked to farms, with metadata like variety, water requirements, and market price.
- Soil samples are evaluated for suitability against crops, with recommendations generated dynamically.
- Post-save signals trigger soil evaluations when new crops or soil data are added.

Order and Transport Workflow:

- BUYERs place orders, which are processed and assigned to DRIVERs for delivery.
- Transport routes are tracked using LineStringField for real-time updates.
- Celery tasks handle background processes like order status updates and weather data fetching.

Testing

- Unit tests verified farm ownership rules, crop-soil evaluations, and order pricing logic.
- Integration tests validated the end-to-order workflow, including transport assignments.

c) API Implementation

Purpose

The API Implementation Module provides RESTful endpoints for frontend-backend communication, ensuring data exchange is efficient, secure, and well-documented.

Implementation Details

RESTful Design:

- Endpoints follow REST conventions (e.g., /farms/ for farm management, /orders/ for order processing).
- Serializers transform Django models into JSON responses and handle input validation.

Tools for Testing and Documentation:

- Postman was used for manual API testing and debugging.
- Swagger/OpenAPI documentation was generated for key endpoints to aid frontend integration.

Key Endpoints:

- POST /auth/login/: Handles user authentication and token generation.
- GET /farms/: Lists farms owned by the authenticated FARMER.
- POST /orders/: Allows BUYERs to place new orders.
- PATCH /transport/<id>/: Updates transport status (used by DRIVERs).

Testing

- Django's unittest framework tested serializer validation and endpoint responses.
- Mocking (unittest.mock.patch) simulated external API calls (e.g., weather data) during tests.

Summary

The module-by-module implementation ensured that each component of the Supply2U platform was developed with clarity, scalability, and security in mind. By adhering to Agile methodologies and leveraging modern tools, the team delivered a robust system ready for real-world use.

Key Achievements:

- Secure authentication with JWT and RBAC.
- Geospatial data handling via PostGIS.
- Seamless integration between frontend and backend APIs.
- Comprehensive testing and validation at every stage.

5. Frontend Development

The frontend of Supply2U is the main touchpoint for users. It is designed to provide a clean, responsive, and intuitive experience for each person that interacts with it. Built with close attention to detail, it ensures smooth navigation, effortless data entry, and real-time interaction with backend services. Each user role i.e farmers, retailers, transporters, and admins, has a dedicated interface tailored to their specific tasks, helping them work more efficiently while minimizing errors. By putting user needs at the center of the design, the frontend makes day-to-day operations simpler, faster, and more satisfying.

Tools and Technologies

To ensure that development was efficient and scalable, the following tools were used:

- React.js
  This served as the core library for building dynamic and reusable user interface components. Its component-based architecture ensures modularity, easier maintenance, and testability.
- Vite
  This was selected as the build tool to enable rapid development and optimized production builds.
- React Router
  Facilitated routing and navigation, enabling movement between pages while maintaining state across views.
- TailwindCSS
  Using Tailwind ensured responsiveness and rapid prototyping through predefined utility classes, reducing development overhead.
- Material-UI (MUI)
  Integrated selectively for form components and layout elements, providing an accessible and consistent design system.
- Axios
  Used for handling HTTP requests to backend APIs, ensuring reliable data exchange and error handling.
- React Context API & Custom Hooks

This was used for state management across the application.

- Email JS
  This was integrated to facilitate email-based interactions such as handling user inquiries and sending notifications.

Pages and Features

Each page was designed with clear objectives and user-centric functionality:

Landing Page

The Landing Page serves as the introduction to the system to users. For logged-in users, the page acts as a gateway to their role-specific dashboards, while unauthenticated users are encouraged to log in or sign up.

Dashboard Pages

Each user role is provided with a dashboard tailored to their specific needs. Farmers can manage their farm listings, transporters can update transportation schedules, and admins can oversee the entire system through tools for user management and data analysis. These dashboards include interactive widgets that display summarized data, such as pending orders or upcoming deliveries, enabling users to access critical information at a glance.

Form Pages

These pages allow users to perform essential tasks such as registration, login, farm creation, farm updating, transport status updates, and inquiries. Each form is designed with dynamic input fields, dropdowns, and validation logic to ensure proper data entry. For example, registration forms check for valid email addresses and secure passwords.

Protected Routes

Secure access was implemented using React Router guards. These routes ensure that only authenticated users can access sensitive pages, and role-based access controls further restrict users to the specific functionalities relevant to their roles.

Error Handling

Custom error pages were added for scenarios like unauthorized access, session expiration, or unresponsive endpoints. A 404 page present includes navigation suggestions for better user experience.

Responsive Design

The use of TailwindCSS utilities allowed developers to implement responsive layouts with ease, while Material-UI’s responsive components provided additional scalability for grids, typography, and other interactive elements. On mobile devices, the interface includes touch-friendly buttons and collapsible menus, while tablet layouts feature intermediate two-column designs for improved usability. Desktop layouts, on the other hand, maximize workspace with full-width views and detailed dashboards. This adaptive design guarantees a consistent and engaging user experience across all devices. Dynamic scaling ensures navigation bars, form inputs, tables, and dashboards adjust intelligently based on screen size.

Other technical features that also enhanced Supply2U’s functionality and user experience include:

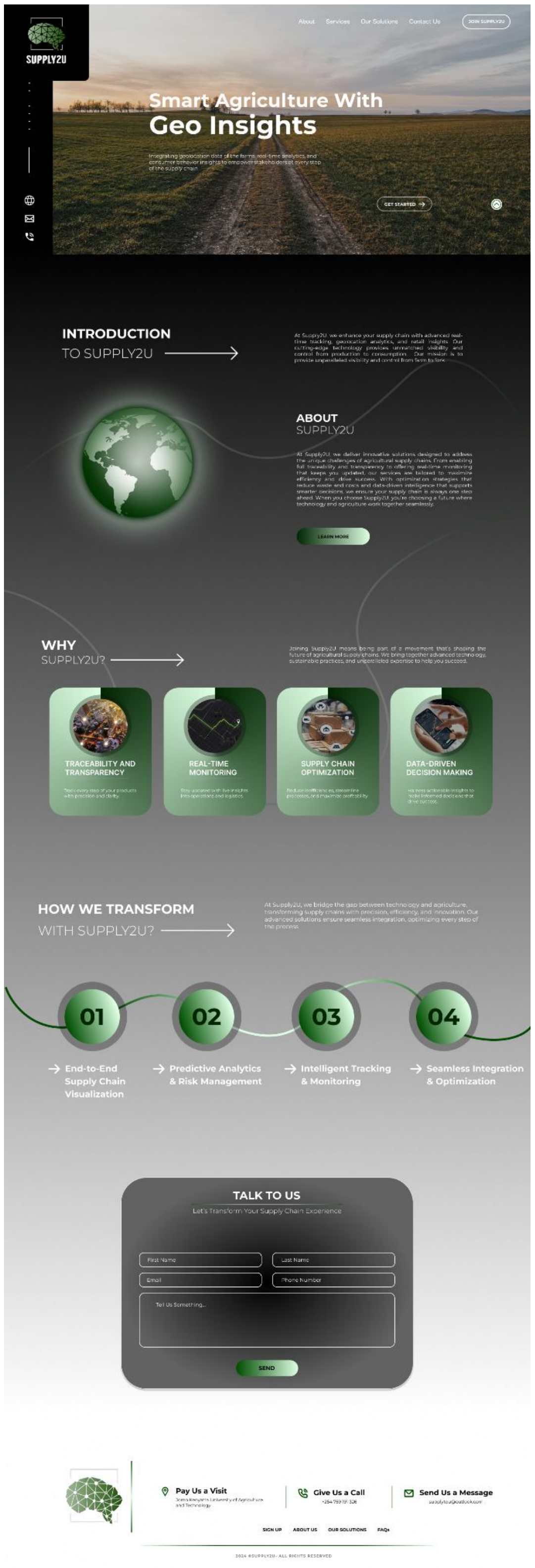
1. Lazy Loading
   This was implemented with React.lazy and Suspense, ensuring that heavy components and pages load asynchronously, reducing initial load times and improving overall performance.
2. Dynamic Imports
   This split code into smaller bundles using dynamic imports (e.g. import './Header') for faster rendering of individual components.
3. Pagination
   Lists for farms and orders leverage pagination, dividing large datasets into manageable chunks for better user experience and performance.
4. Authentication & Role Management
   Secured authentication was implemented using HttpOnly cookies for session management. Role-based dashboards enforce access control, displaying only role-specific information.
5. Environment Variables
   API endpoints, sensitive keys, and other configurations are stored securely in .env files to minimize exposure.

Visuals and Wireframes

The following visuals and wireframes provide an in-depth perspective on how the frontend of Supply2U was structured, designed, and optimized to meet the needs of its diverse user roles. By showcasing key interfaces and design elements, they highlight the functionality, responsiveness, and user-centric approach that guided the development process.

1. **Landing Page**

   The landing page features options for login, signup, and redirects to role-specific dashboards once authenticated.



Smart Agriculture With
Geo Insights

Integrating geolocation data of the farms, real-time analytics, and consumer behavior insights to empower stakeholders at every step of the supply chain

INTRODUCTION
TO SUPPLY2U

At Supply2U, we enhance your supply chain with advanced real-time tracking, geolocation analytics, and retail insights. Our cutting-edge technology provides unmatched visibility and control from production to consumption. Our mission is to provide unparalleled visibility and control from farm to fork.

ABOUT
SUPPLY2U

At Supply2U, we deliver innovative solutions designed to address the unique challenges of agricultural supply chains. From enabling full traceability and transparency to offering real-time monitoring that keeps you updated, our services are tailored to maximize efficiency and drive success. With optimization strategies that reduce waste and costs and data-driven intelligence that supports smarter decisions, we ensure your supply chain is always one step ahead. When you choose Supply2U, you're choosing a future where technology and agriculture work together seamlessly.

WHY
SUPPLY2U?

Joining Supply2U means being part of a movement that's shaping the future of agricultural supply chains. We bring together advanced technology, sustainable practices, and unparalleled expertise to help you succeed.

TRACEABILITY AND TRANSPARENCY
Track every step of your products with precision and clarity.

REAL-TIME MONITORING
Stay updated with live insights into operations and logistics.

SUPPLY CHAIN OPTIMIZATION
Reduce inefficiencies, streamline processes, and maximize profitability.

DATA-DRIVEN DECISION MAKING
Harness actionable insights to make informed decisions that drive success.

HOW WE TRANSFORM
WITH SUPPLY2U?

At Supply2U, we bridge the gap between technology and agriculture, transforming supply chains with precision, efficiency, and innovation. Our advanced solutions ensure seamless integration, optimizing every step of the process.

01 → End-to-End Supply Chain Visualization
02 → Predictive Analytics & Risk Management
03 → Intelligent Tracking & Monitoring
04 → Seamless Integration & Optimization

TALK TO US
Let's Transform Your Supply Chain Experience

First Name
Last Name
Email
Phone Number
Tell Us Something...

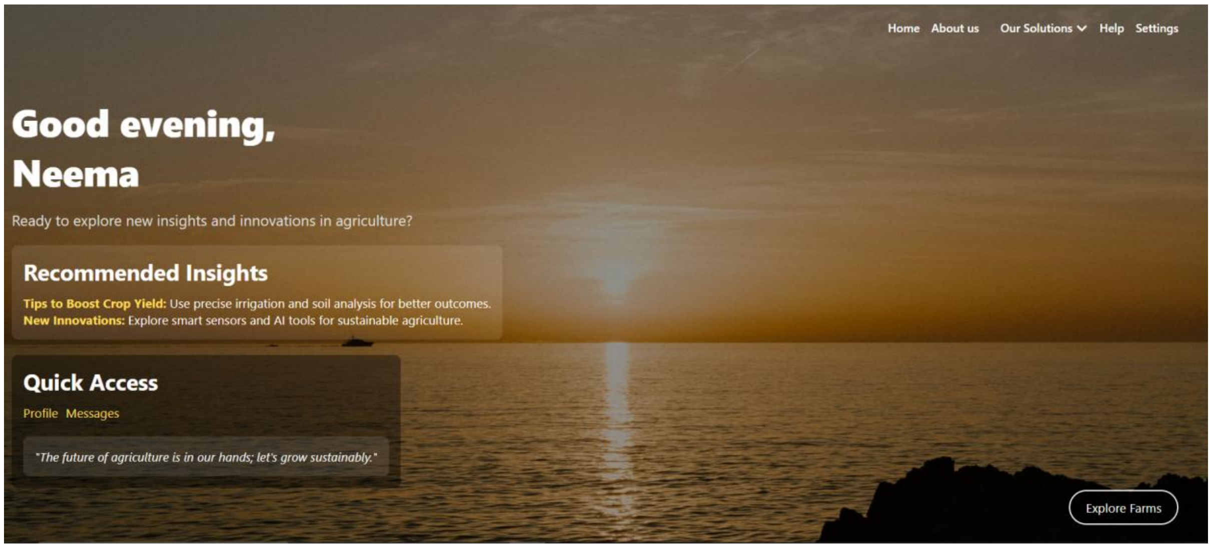
Pay Us a Visit
Jomo Kenyatta University of Agriculture and Technology

Give Us a Call
+254 759 191 326

Send Us a Message
supplytou@outlook.com

2024 ©SUPPLY2U- ALL RIGHTS RESERVED

2. **Role-Specific Dashboards**



Good evening, Neema

Ready to explore new insights and innovations in agriculture?

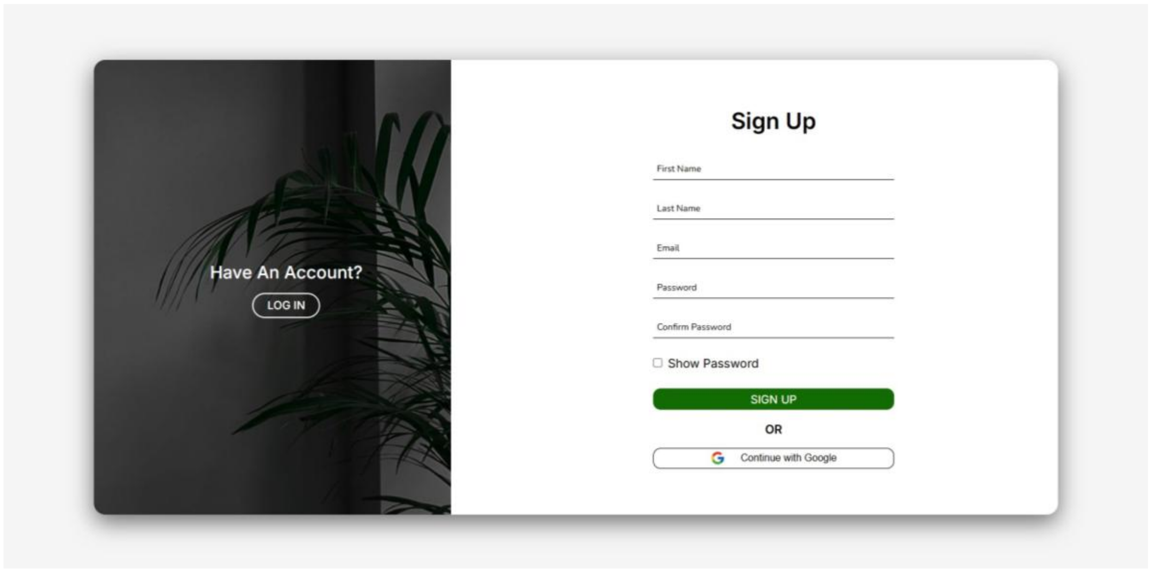
Recommended Insights

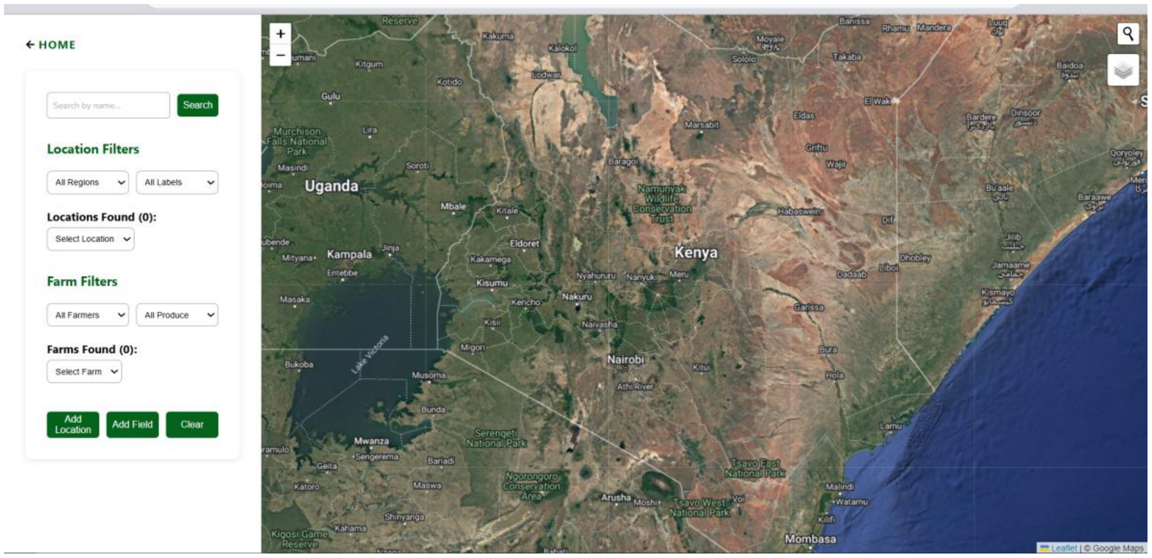
Tips to Boost Crop Yield: Use precise irrigation and soil analysis for better outcomes.
New Innovations: Explore smart sensors and AI tools for sustainable agriculture.

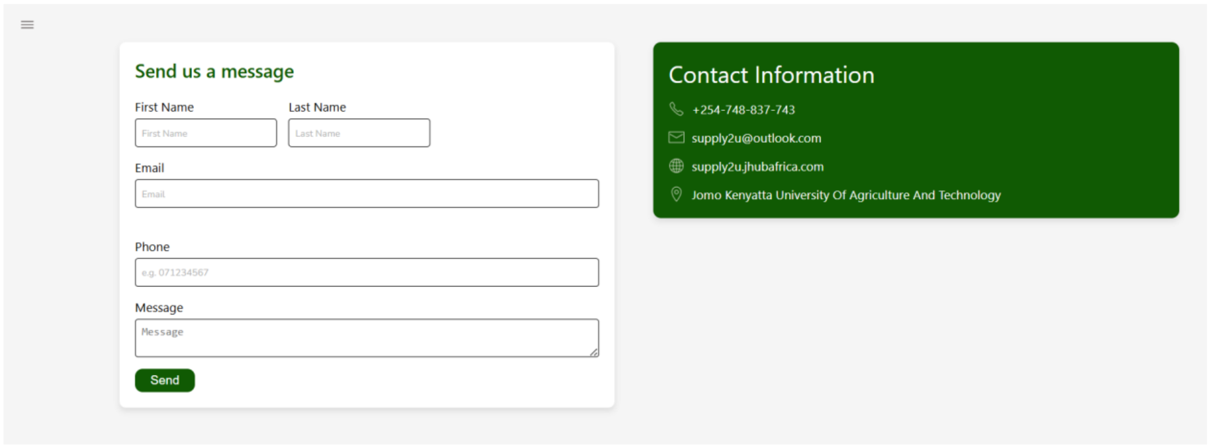
Quick Access

"The future of agriculture is in our hands; let's grow sustainably."

3. **Form Pages**







Send us a message

First Name | Last Name

Email

Phone

Message

Send

Contact Information

+254-748-837-743

supply2u@outlook.com

supply2u.jhubafrica.com

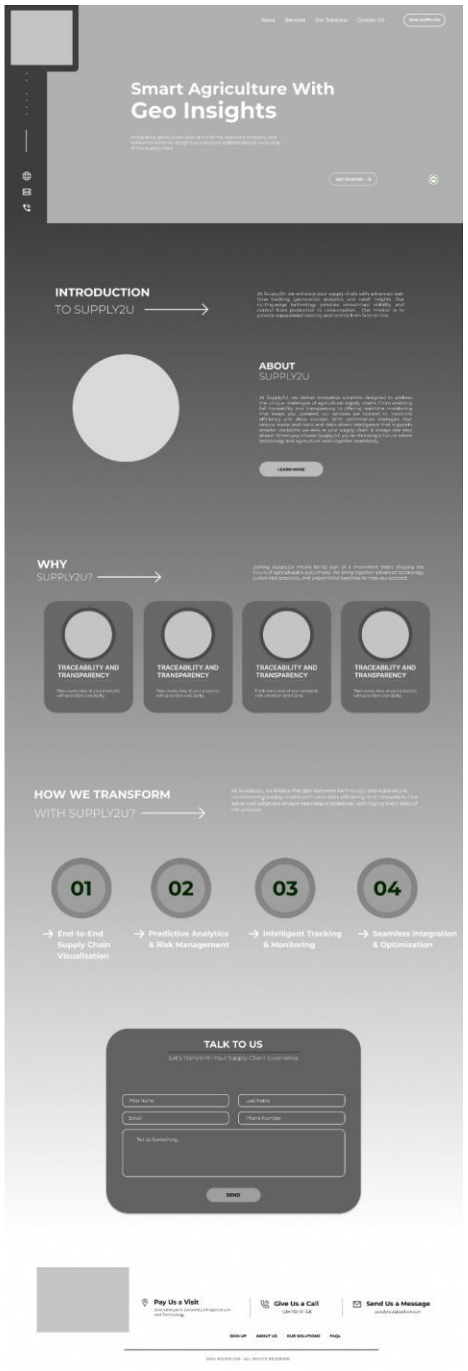
Jomo Kenyatta University Of Agriculture And Technology

4. Responsive Design on Mobile Devices

The system's responsive design ensures usability across all devices, featuring touch-friendly interfaces and layout adjustments for varying screen sizes. Mobile interfaces include collapsible sidebars, optimized buttons, and vertical stacking. Tablet views adapt to display information in two-column layouts for better clarity.

5. Wireframe Designs

Wireframes from the design phase, created using Figma, guided the development process and ensured alignment with user needs.


About Services Our Solutions Contact Us
Smart Agriculture With
Geo Insights
GET STARTED
INTRODUCTION
TO SUPPLY2U
ABOUT
SUPPLY2U
LEARN MORE
WHY
SUPPLY2U?
TRACEABILITY AND TRANSPARENCY
TRACEABILITY AND TRANSPARENCY
TRACEABILITY AND TRANSPARENCY
TRACEABILITY AND TRANSPARENCY
HOW WE TRANSFORM
WITH SUPPLY2U?
01
02
03
04
End-to-End Supply Chain Visualization
Predictive Analytics & Risk Management
Intelligent Tracking & Monitoring
Seamless Integration & Optimization
TALK TO US
Let's Transform Your Supply Chain Experience
First Name
Last Name
Email
Phone Number
Tell Us Something...
SEND
Pay Us a Visit
Give Us a Call
Send Us a Message
SIGN UP ABOUT US OUR SOLUTIONS FAQs

The frontend of Supply2U was built to be modular, scalable, and highly responsive. Every feature—from role-specific dashboards to secure authentication and responsive layouts—was designed to mainly optimize the user experience across all devices. With a focus on usability, performance, and maintainability, the frontend plays a key role in ensuring that Supply2U remains efficient, secure, and user-friendly even as the system grows.

6. Database Implementation

Purpose

The database was designed to support a smart agriculture system, enabling geospatial tracking of farms and transportation routes, dynamic soil evaluation, and management of crop production, weather data, and e-commerce transactions. The objective was to ensure data consistency, relational integrity, and scalable performance while accommodating domain-specific needs such as spatial queries and dynamic user roles.

Database Type

- **Relational SQL Database:** The system uses PostgreSQL due to its strong ACID compliance, relational capabilities, and community support.
- **Geospatial Support:** Integration with PostGIS allows for advanced geospatial operations, including farm boundary mapping (PolygonField), farm location (PointField), and transport routing (LineStringField).

Key Tables and Relationships

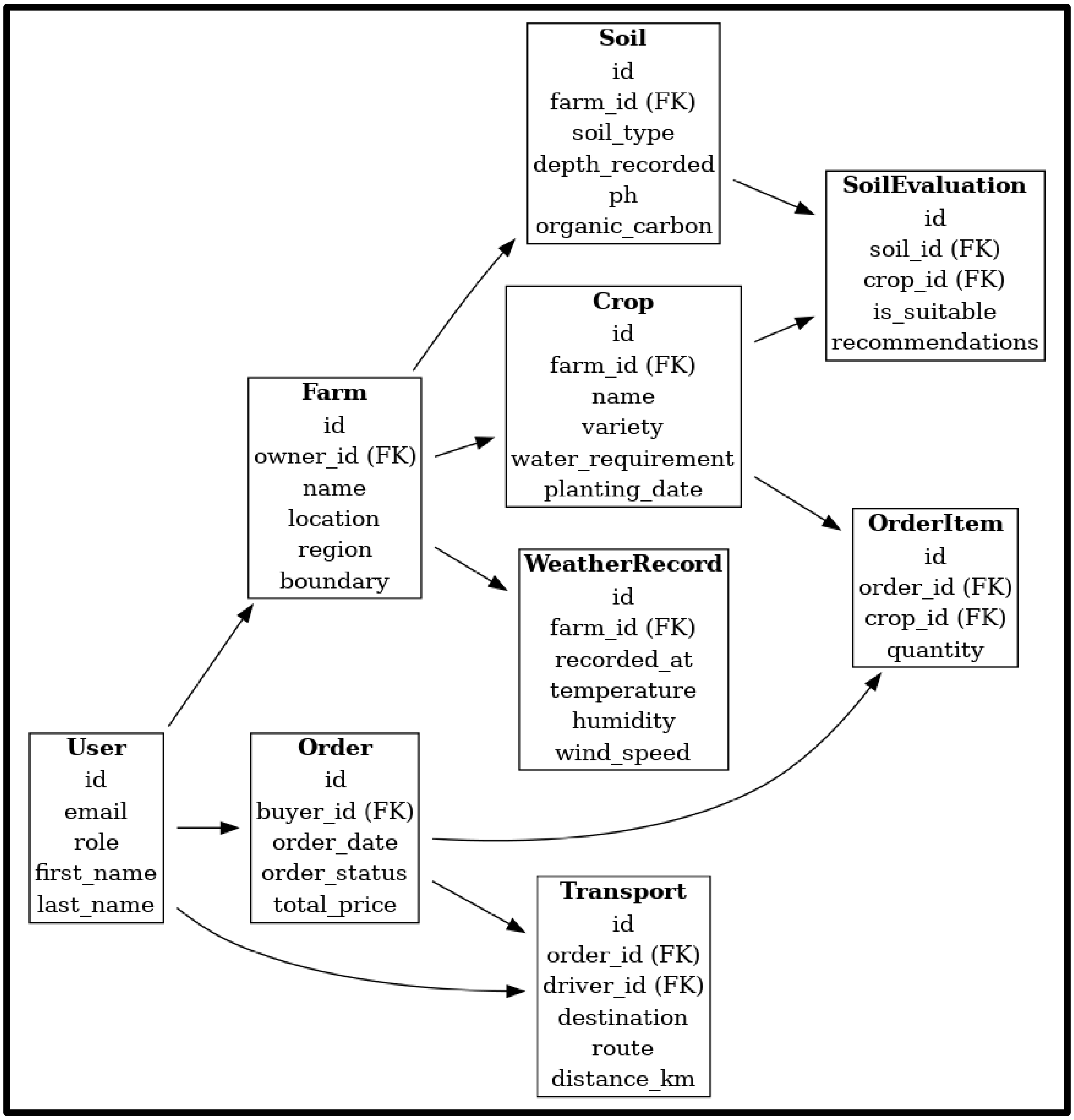
The schema is composed of modular apps in Django, each corresponding to a functional area of the system:

Table Name	Description
users_useraccount	Stores user credentials and roles (FARMER, DRIVER, ADMIN).
farmmanagement_farm	Represents individual farms, includes spatial fields for location and boundary.
crops_crop	Contains crop metadata such as variety, water requirements, and market price.
soil_soil	Records soil test results for each farm, linked to crops through evaluations.
soil_soilevaluation	Holds dynamic suitability and recommendations per crop-soil combination.
orders_order	Represents buyer orders with status tracking and pricing logic.

orders_orderitem	Each line item in an order referencing a crop and quantity ordered.
transportation_transport	Tracks delivery information for an order, including route, driver, and duration.
weather_weatherrecord	Captures periodic farm-level weather data fetched from OpenWeather API.

Schema Relationships

- **User → Farm:** One-to-many relationship. A FARMER can own multiple farms.
- **Farm → Crop:** One-to-many relationship. Each farm has multiple crops.
- **Farm → Soil:** One-to-many relationship. Soil samples are tied to a specific farm.
- **Soil ↔ Crop:** Many-to-many relationship via SoilEvaluation. Every soil sample can be evaluated against multiple crops.
- **Order → OrderItem:** One-to-many relationship. An order has multiple order items.
- **Order ↔ Transport:** One-to-one relationship. A single transport record is associated with each order.
- **Farm → WeatherRecord:** One-to-many relationship. A farm can have multiple recorded weather snapshots.



The full schema including the table structures, constraints, and relationships is available online: View Full DB Schema

7. Testing and Debugging

Purpose

To ensure that the backend logic behaves as expected under different conditions and roles, and that all API endpoints respond with correct data while enforcing access control.

Backend Testing Strategy

Framework

The project uses Django's unittest framework for model, view, and serializer testing.

Unit Testing

Model logic:

Validation for Farm.clean() to restrict ownership to FARMERs only.
Price computation logic in Order.calculate_total() tested for consistency.

Serializer validation:

Field-level checks to reject unauthorized data assignments, e.g., forcing OWNER role during farm creation.

Integration Testing

JWT-based authentication logic through custom views:

CustomTokenObtainPairView, CustomTokenRefreshView, and CustomTokenVerifyView were tested for cookie handling and token refresh.

Post-save signals:

Soil and crop creation automatically triggers evaluations, validated with test cases to confirm SoilEvaluation entries were correctly generated.

Manual Testing

Tools used: Postman, Django Admin Panel
Use cases tested:

- Role-based access to farm, order, and transport APIs
- Transport visibility for buyers vs. farmers
- Soil evaluation logic when a new crop is added

Mocking

The Hugging Face model integration (call_huggingface_model) is mocked using unittest.mock.patch to simulate responses during test runs without triggering real API calls.

Bug Tracking and Resolution

- Token errors: Refreshed tokens failing due to missing cookies were fixed by overriding TokenRefreshView to inject refresh from cookies.
- Permission errors: Errors where non-farmers could submit farms were fixed with explicit permission classes and custom clean() methods.
- Soil evaluation: Infinite loops from signals were mitigated using creation flags and instance state checks.

8. Security Implementation

Purpose

To safeguard sensitive agricultural and user data from unauthorized access, ensure secure transmission of data, and protect against standard web vulnerabilities.

Authentication and Session Security

JWT with Cookie Integration

Secure login and session persistence is implemented using JWT (JSON Web Tokens) stored in:

- HttpOnly cookies to prevent JavaScript access (XSS protection)
- Secure cookies to ensure transmission only over HTTPS
- SameSite policy to prevent CSRF attacks via cross-domain requests

Custom Authentication Class

A CustomJWTAuthentication class was implemented to extract tokens from cookies instead of headers, supporting secure browser-based login flows.

```
from rest_framework_simplejwt.authentication import JWTAuthentication
from django.conf import settings
```

```
class CustomJWTAuthentication(JWTAuthentication):
    def authenticate(self, request):
        try:
            header = self.get_header(request)
            if header is None:
                raw_token = request.COOKIES.get(settings.AUTH_COOKIE)
            else:
                raw_token = self.get_raw_token(header)

            if raw_token is None:
                return None

            validated_token = self.get_validated_token(raw_token)

            return self.get_user(validated_token), validated_token
        except:
            return None
```

Role-Based Access Control (RBAC)

Granular access logic ensures:

- FARMERs can only manage their own farms, crops, and soil data.
- BUYERs can only see their orders and associated transports.
- DRIVERs cannot access unrelated transport assignments.
- Admins have full system access.
- Key permission classes: IsOwner, IsFarmerOrAdmin, IsAuthenticated

Validation and Model Constraints

Models and serializers enforce role logic at the validation layer e.g., Farm.owner must have role FARMER, Transport.driver must have role DRIVER

```
    def clean(self):
        '''Ensure only a FARMER role can own a farm'''
        if not hasattr(self, 'owner') or self.owner is None:
            raise ValidationError({'owner': 'Farm must have an owner.'})
        if self.owner.role != 'FARMER':
            raise ValidationError('Only Farmers can own farms')
        super().clean()

    def __str__(self):
        return f'{self.name} ({self.owner.email})'
    def save(self, *args, **kwargs):
        if self.boundary:
            self.location = self.boundary.centroid
        self.clean()
        super().save(*args, **kwargs)
```

API Hardening

SQL Injection: Prevented by using Django’s ORM (no raw SQL used)
XSS Protection:

- Input fields (e.g., recommendations) are rendered in safe formats
- Expected frontend implementation to escape and sanitize HTML inputs

HTTPS Enforcement:

- Secure=True and SESSION_COOKIE_SECURE=True are enforced in production
- Assumes use of HTTPS through a reverse proxy or deployment service

FrontendTesting Strategy

Unit Tests:

- Unit and integration tests partially complete.
- .test.js files exist for key components but overall coverage is still in progress.

Manual Testing:

- Responsive design was manually verified across devices.
- Layouts adapt using a hybrid styling approach (TailwindCSS + Material-UI components).

9. Performance Optimization

Purpose

To ensure that the smart agriculture platform remains responsive and scalable under real-world workloads, several backend performance optimizations were applied. These enhancements focused on reducing latency, improving query efficiency, and managing resource usage effectively during API calls and background tasks.

Backend Optimizations

Efficient Querying with ORM Enhancements

- Leveraged select_related() for foreign key relationships and prefetch_related() for many-to-many relationships to minimize redundant database queries.
- QuerySets were filtered at the database level (e.g., filter(farm__owner=user)) to ensure only authorized and relevant data is loaded into memory.

Pagination for List Views

- All list views such as farms, crops, soil evaluations, orders, and weather records implement pagination using Django REST Framework's pagination classes.
- This prevents excessive memory consumption and network load when retrieving large datasets.

Task Offloading with Celery

- Periodic weather data collection is offloaded from synchronous HTTP requests using Celery and Redis.
- This asynchronous execution ensures that heavy external API calls (OpenWeather) do not delay user requests or degrade system responsiveness.

Data Integrity via Signals

- Post-save signals were used to automate soil evaluation creation and prevent duplicate evaluations.
- Triggers are lightweight and scoped, ensuring they only execute under valid creation conditions (e.g., using created=True).

Frontend Optimizations

Lazy Loading:

React.lazy and Suspense used to defer loading of heavy pages, especially maps and detailed dashboards.

Dynamic Imports:

Critical but non-core components (e.g., Footer) are dynamically imported using import("./Footer") style syntax.

Code Splitting:

Vite automatically code-splits bundles during build time, separating vendor libraries, route-level chunks, etc.

Tree Shaking:

Unused code is purged during builds to ensure the final JS bundles are lean.

Pagination:

Frontend fetches paginated lists (farms, orders) and implements infinite scroll or page-by-page fetch to avoid bloated UI renders.

State Management Optimization:

Global states handled via React.Context and custom hooks — avoiding unnecessary props drilling and redundant re-renders.

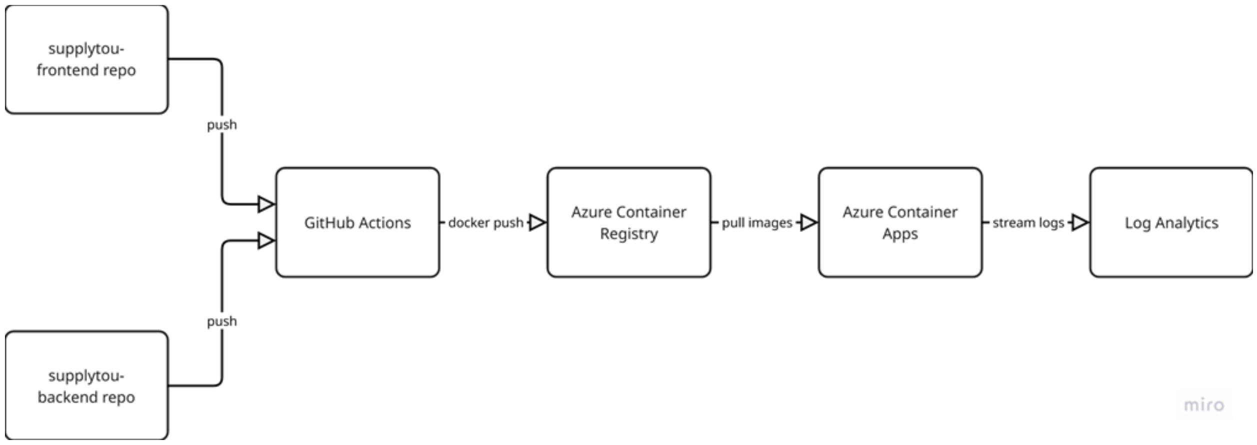
Memoization Techniques:

React.memo applied to heavy components.
useMemo and useCallback systematically used to prevent the re-creation of functions and re-renders on dependency-stable parts.

10. CI/CD and Deployment

Purpose & Overview

Our Continuous Integration (CI) and Continuous Deployment (CD) streamlines how code changes move from our local machine into the production environment. Rather than manually building, testing, and deploying, every commit runs through a standardized pipeline, catching errors early and guaranteeing consistency.



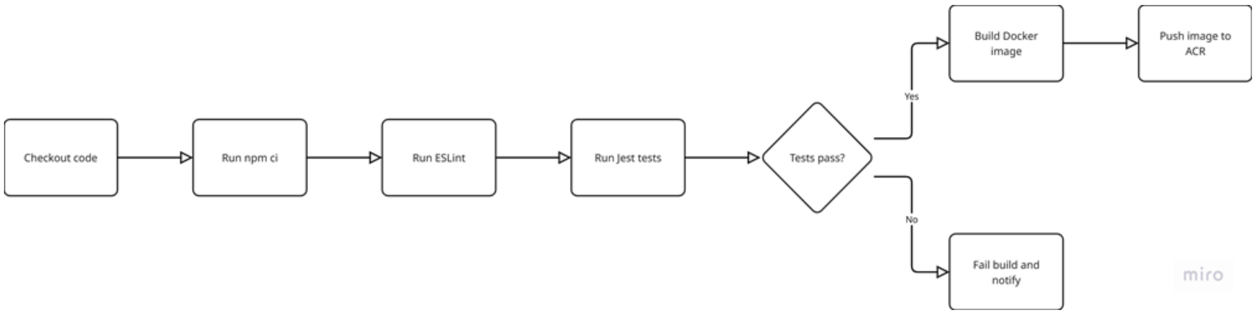
Continuous Integration (CI) Pipelines

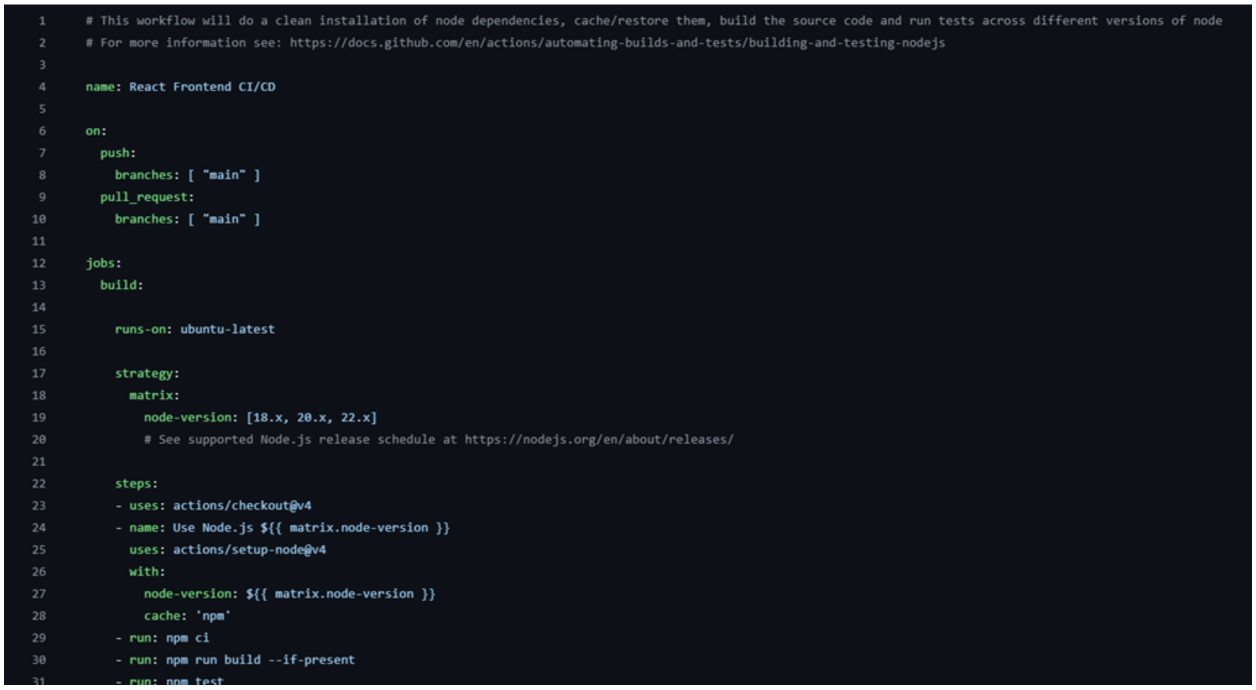
Frontend CI

Its goal is to ensure your React code always builds cleanly, passes lint rules, and has no failing tests before packaging.

1. **Trigger & Branch Policy**
   - Fires on push to main and any PR targeting main.
   - main is protected—cannot merge if CI fails.
2. **Install & Lint**
   - npm ci installs exact versions from package-lock.json.
   - npm run lint (ESLint) enforces your code style and catches common JS bugs.
3. **Unit & Snapshot Tests**
   - npm test -- --coverage runs Jest, ensures components render as expected.
   - Coverage gate: must stay above 80%.
4. **Docker Build**
   - Builds the production-ready static files.
   - Multi-stage Dockerfile ensures only the compiled output ends up in the final image.

5. **Push to ACR**
    - Tagged frontend:latest or semver tag (v1.2.3).
    - Stored in Azure Container Registry under your registry namespace.



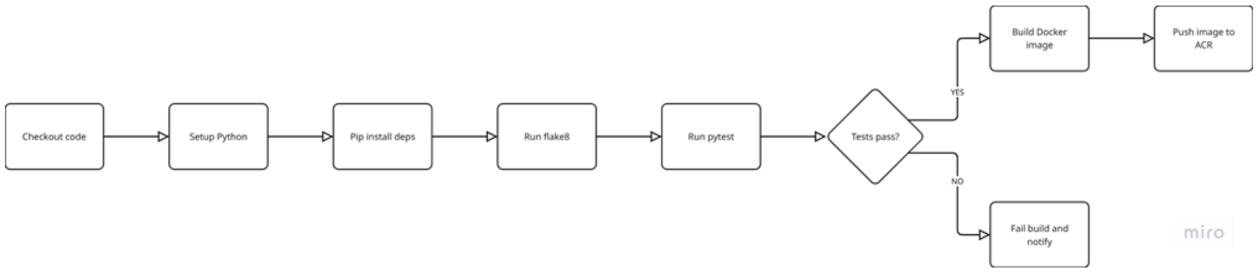


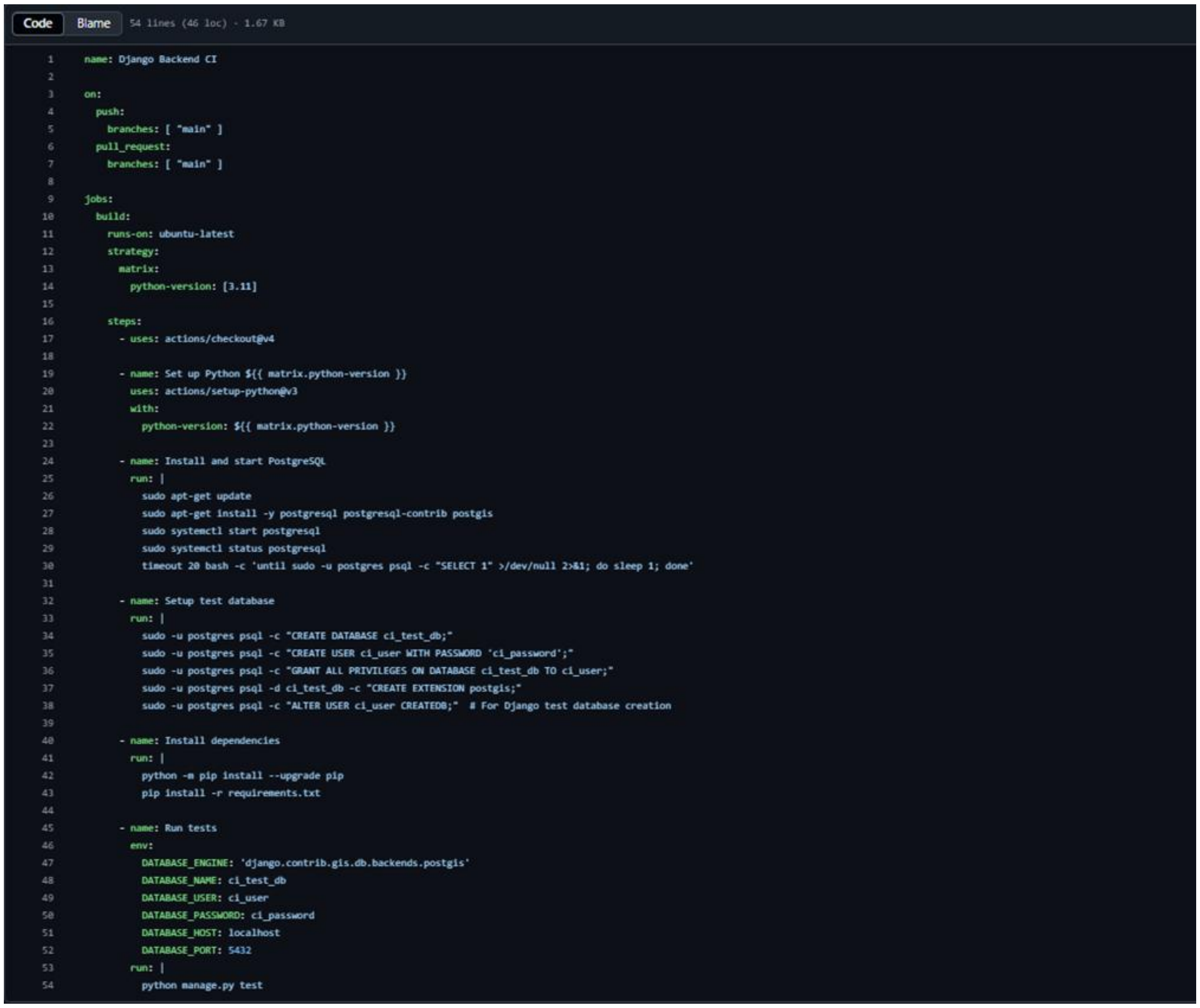
```yaml
# This workflow will do a clean installation of node dependencies, cache/restore them, build the source code and run tests across different versions of node
# For more information see: https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-nodejs

name: React Frontend CI/CD

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:

    runs-on: ubuntu-latest

    strategy:
      matrix:
        node-version: [18.x, 20.x, 22.x]
        # See supported Node.js release schedule at https://nodejs.org/en/about/releases/

    steps:
    - uses: actions/checkout@v4
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v4
      with:
        node-version: ${{ matrix.node-version }}
        cache: 'npm'
    - run: npm ci
    - run: npm run build --if-present
    - run: npm test
```

Backend CI

It verifies that your Django app installs dependencies, passes lint, and runs all unit tests before containerization.

1. **Trigger & Protection**
   - Runs on both direct pushes to main and on PRs.
   - PRs cannot merge unless CI passes.
2. **Environment Setup**
   - Uses actions/setup-python@v4 to install Python 3.10.
   - pip install -r requirements.txt installs pinned versions.
3. **Lint & Testing**
   - flake8 checks for style and potential errors.
   - pytest (or manage.py test) runs unit tests—models, views, serializers.
4. **Docker Build & Push**
   - Builds a Gunicorn-ready image.
   - Pushes to ACR under backend:latest or a release tag.





```yaml
name: Django Backend CI

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.11]

    steps:
      - uses: actions/checkout@v4

      - name: Set up Python ${{ matrix.python-version }}
        uses: actions/setup-python@v3
        with:
          python-version: ${{ matrix.python-version }}

      - name: Install and start PostgreSQL
        run: |
          sudo apt-get update
          sudo apt-get install -y postgresql postgresql-contrib postgis
          sudo systemctl start postgresql
          sudo systemctl status postgresql
          timeout 20 bash -c 'until sudo -u postgres psql -c "SELECT 1" >/dev/null 2>&1; do sleep 1; done'

      - name: Setup test database
        run: |
          sudo -u postgres psql -c "CREATE DATABASE ci_test_db;"
          sudo -u postgres psql -c "CREATE USER ci_user WITH PASSWORD 'ci_password';"
          sudo -u postgres psql -c "GRANT ALL PRIVILEGES ON DATABASE ci_test_db TO ci_user;"
          sudo -u postgres psql -d ci_test_db -c "CREATE EXTENSION postgis;"
          sudo -u postgres psql -c "ALTER USER ci_user CREATEDB;"  # For Django test database creation

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt

      - name: Run tests
        env:
          DATABASE_ENGINE: 'django.contrib.gis.db.backends.postgis'
          DATABASE_NAME: ci_test_db
          DATABASE_USER: ci_user
          DATABASE_PASSWORD: ci_password
          DATABASE_HOST: localhost
          DATABASE_PORT: 5432
        run: |
          python manage.py test
```

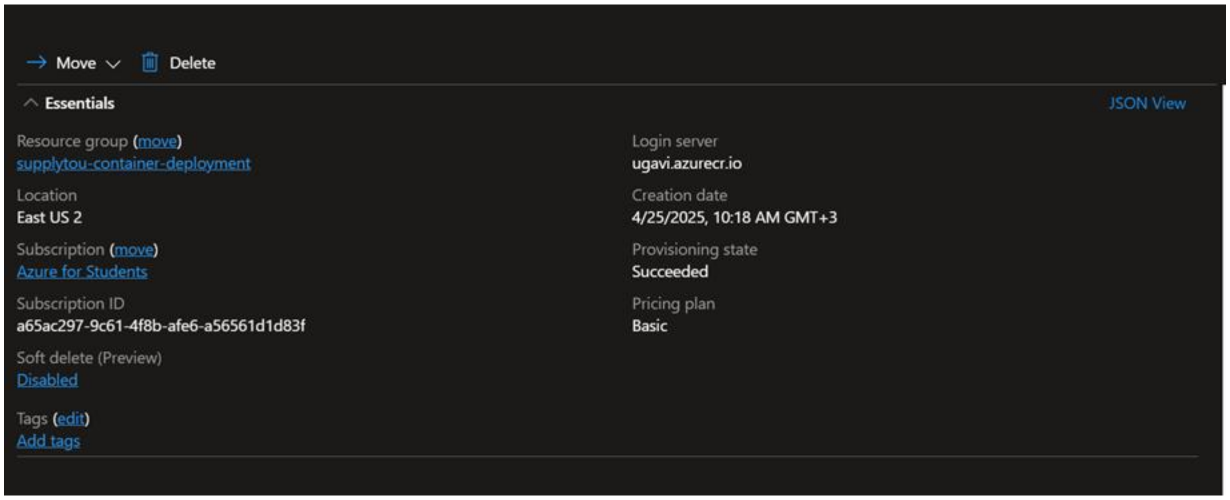
Container Registry (ACR) Configuration

Azure Container Registry serves as your private Docker registry.

- **Provisioning**

```
az acr create \
--name supplytouACR \
--resource-group supplytou-rg \
--sku Standard \
--location eastus
```

- **Repository Layout**
  - supplytouacr.azurecr.io/frontend:latest
  - supplytouacr.azurecr.io/backend:latest
- **Security**
  - Grant a Service Principal AcrPull/AcrPush.
  - Store ACR_USERNAME, ACR_PASSWORD, and ACR_LOGIN_SERVER as GitHub Secrets.



Resource group (move)
supplytou-container-deployment

Location
East US 2

Subscription (move)
Azure for Students

Subscription ID
a65ac297-9c61-4f8b-afe6-a56561d1d83f

Soft delete (Preview)
Disabled

Tags (edit)
Add tags

Login server
ugavi.azurecr.io

Creation date
4/25/2025, 10:18 AM GMT+3

Provisioning state
Succeeded

Pricing plan
Basic

Deployment to Azure Container Apps

Container Apps gives us managed containers with auto-scale and Dapr integration, without the full complexity of Kubernetes.

1. Create Environment & Logging

```
az containerapp env create \

  --name supplytou-env \

  --resource-group supplytou-rg \

  --logs-workspace-id $LOG_ANALYTICS_ID
```

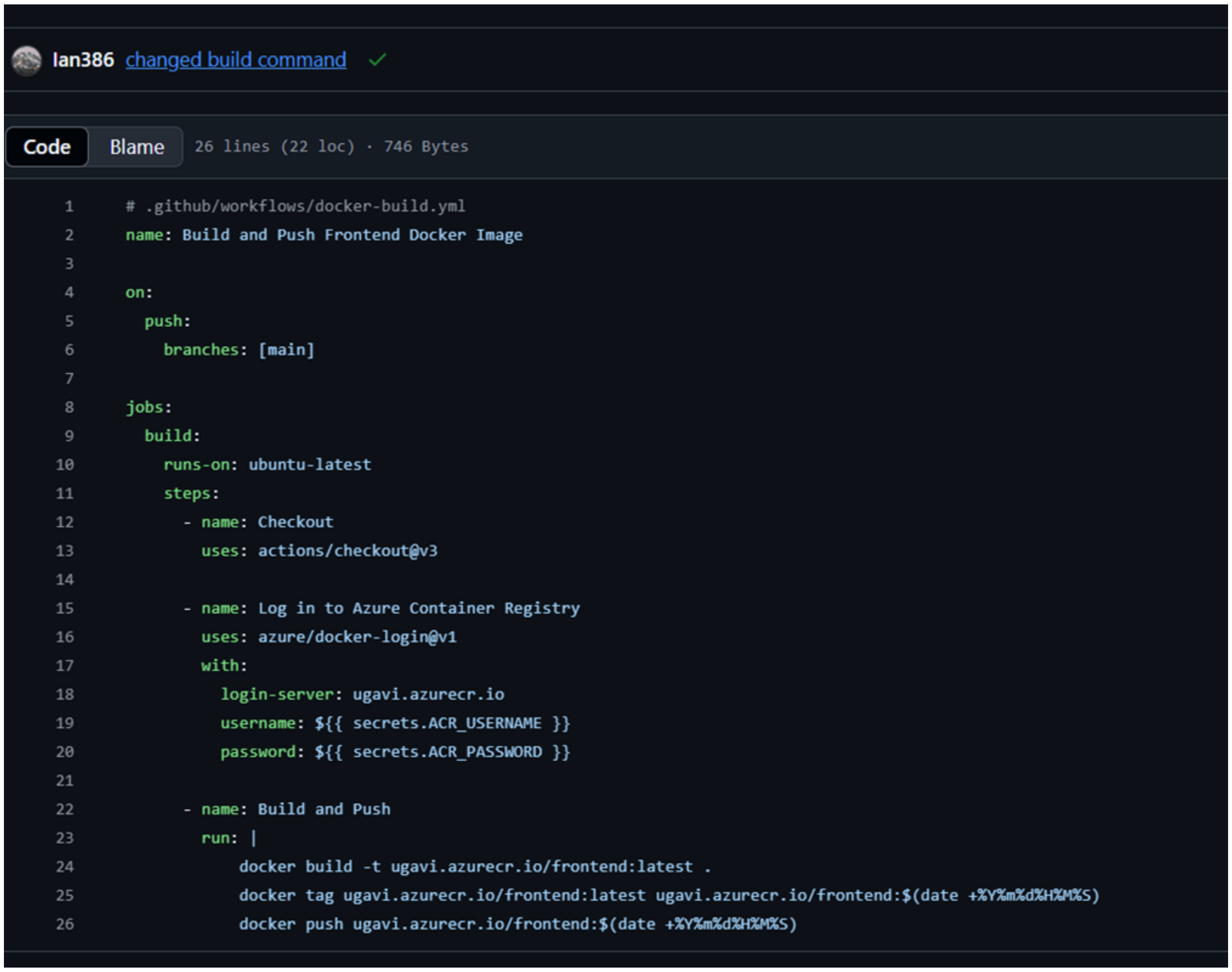
2. Deploy Frontend

```
az containerapp create \

  --name supplytou-frontend \

  --resource-group supplytou-rg \
```

--environment supplytou-env \

--image supplytouacr.azurecr.io/frontend:latest \

--ingress external --target-port 80



Ian386 changed build command ✓

Code Blame 26 lines (22 loc) · 746 Bytes

```yaml
# .github/workflows/docker-build.yml
name: Build and Push Frontend Docker Image

on:
  push:
    branches: [main]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Log in to Azure Container Registry
        uses: azure/docker-login@v1
        with:
          login-server: ugavi.azurecr.io
          username: ${{ secrets.ACR_USERNAME }}
          password: ${{ secrets.ACR_PASSWORD }}

      - name: Build and Push
        run: |
            docker build -t ugavi.azurecr.io/frontend:latest .
            docker tag ugavi.azurecr.io/frontend:latest ugavi.azurecr.io/frontend:$(date +%Y%m%d%H%M%S)
            docker push ugavi.azurecr.io/frontend:$(date +%Y%m%d%H%M%S)
```

3. Deploy Backend

az containerapp create \

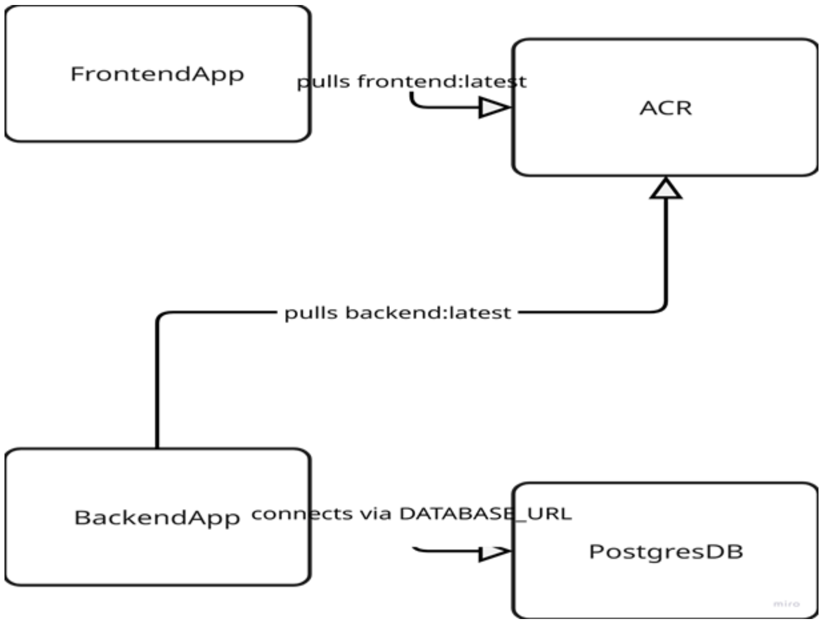
--name supplytou-backend \

--resource-group supplytou-rg \

--environment supplytou-env \

```
--image supplytouacr.azurecr.io/backend:latest \
--ingress internal --target-port 8000 \
--env-vars \
  DATABASE_URL=$DATABASE_URL \
  DJANGO_SETTINGS_MODULE=backend.settings.production
```

4. **Networking & Secrets**
   - Frontend's BACKEND_API_URL env var points to internal FQDN.
   - Database credentials stored securely in Azure Key Vault or Container App secrets.



Rollback & Versioning Strategy

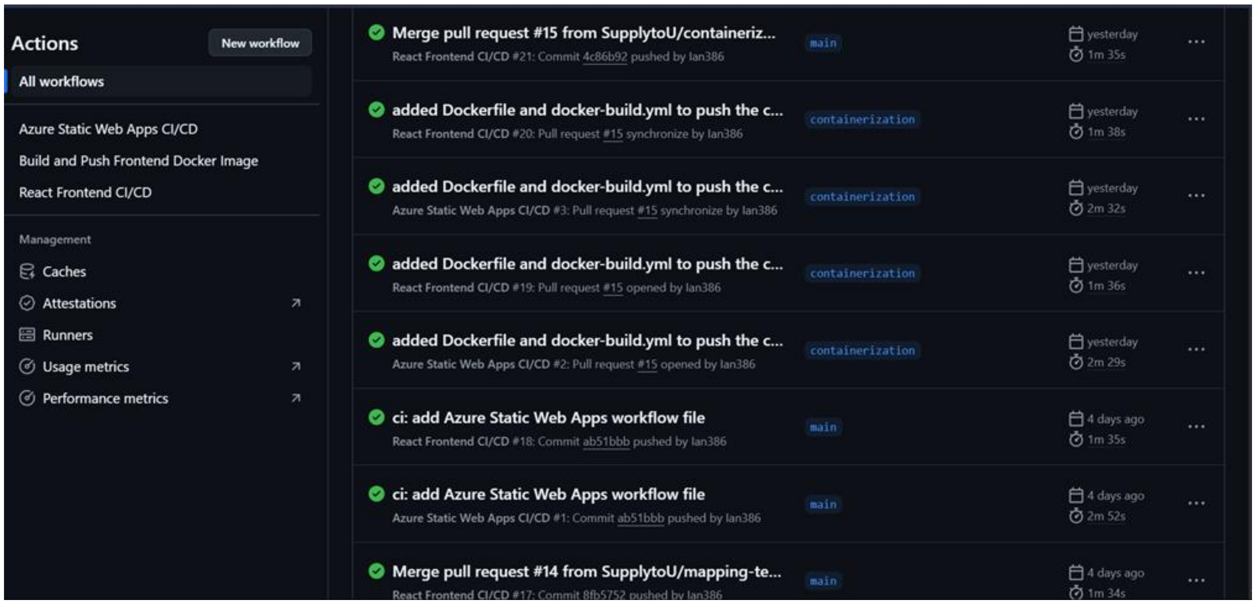
- **Semantic Versioning:** Tag images v1.0.0, v1.1.0, etc.
- **Rollback Process:**

```
az containerapp revision list \
 --name supplytou-frontend \
--resource-group supplytou-rg


az containerapp revision set-active \
--name supplytou-frontend \
--resource-group supplytou-rg \
--revision <previousRevision>
```

- **Health Probes:** HTTP readiness probes catch failed deployments before traffic shifts.

Notifications & Monitoring

- **CI Notifications:** GitHub Actions ➔ email and GitHub Mobile on failure.
- **Container Insights:** CPU/memory metrics, log queries in Log Analytics.
- **Alerts:**
  - CPU > 80%
  - 5 HTTP 5xx errors per minute



Merge pull request #15 from SupplytoU/containeriz... main yesterday 1m 35s
React Frontend CI/CD #21: Commit 4c86b92 pushed by Ian386

added Dockerfile and docker-build.yml to push the c... containerization yesterday 1m 38s
React Frontend CI/CD #20: Pull request #15 synchronize by Ian386

added Dockerfile and docker-build.yml to push the c... containerization yesterday 2m 32s
Azure Static Web Apps CI/CD #3: Pull request #15 synchronize by Ian386

added Dockerfile and docker-build.yml to push the c... containerization yesterday 1m 36s
React Frontend CI/CD #19: Pull request #15 opened by Ian386

added Dockerfile and docker-build.yml to push the c... containerization yesterday 2m 29s
Azure Static Web Apps CI/CD #2: Pull request #15 opened by Ian386

ci: add Azure Static Web Apps workflow file main 4 days ago 1m 35s
React Frontend CI/CD #18: Commit ab51bbb pushed by Ian386

ci: add Azure Static Web Apps workflow file main 4 days ago 2m 52s
Azure Static Web Apps CI/CD #1: Commit ab51bbb pushed by Ian386

Merge pull request #14 from SupplytoU/mapping-te... main 4 days ago 1m 34s
React Frontend CI/CD #17: Commit 8fb5752 pushed by Ian386

11. Collaboration and Workflow

Purpose & Overview

Effective collaboration and a clear workflow are essential for delivering a complex GIS platform on time and with high quality. In this section, we describe how our team structured task tracking, communication, version control, and continuous improvement to stay aligned across frontend, backend, DevOps, and QA.

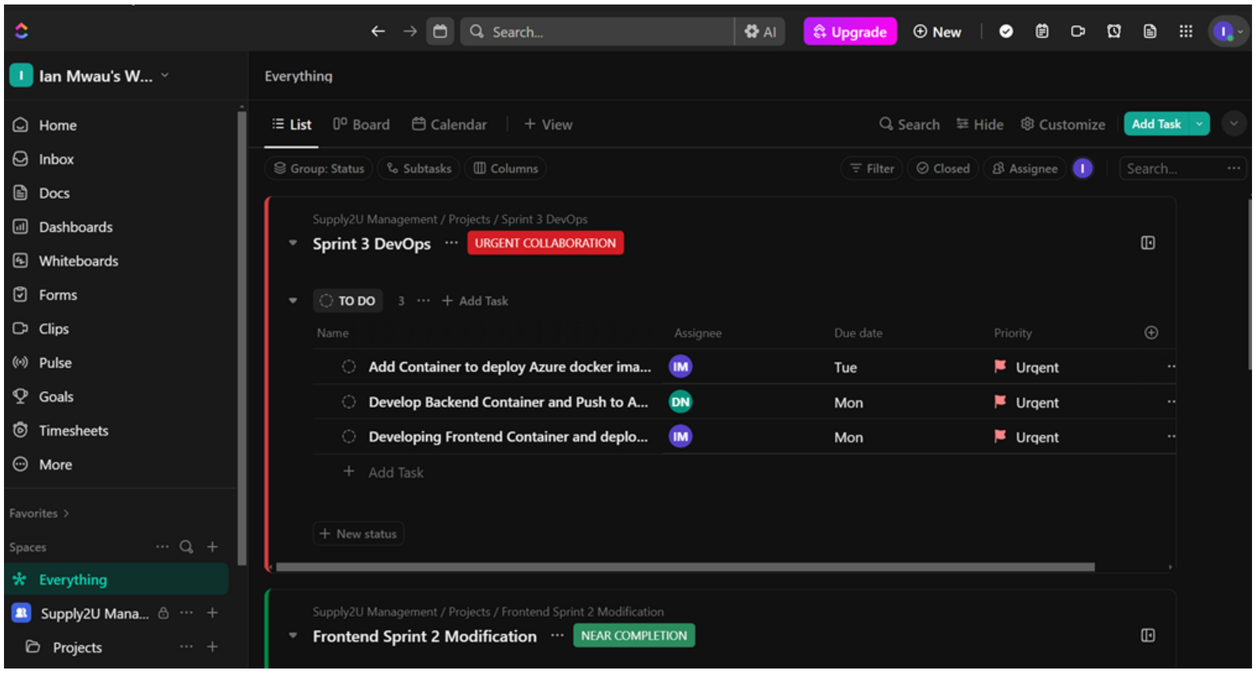
Project Management & Task Tracking

We chose ClickUp as our primary tool to capture requirements, plan sprints, and track progress. For development-focused boards, we mirrored high-priority tasks into GitHub Projects, ensuring that code work and project tasks stayed in sync.

- **ClickUp Setup**
  - **Spaces & Folders:**
    - **"Frontend Changes"** for all UI feature requests and bugs.
    - **"CI/CD"** folder containing continuous development sprint.
  - **Custom Fields:**
    - **Priority** (Urgent, High, Medium, Low)
    - **Status** (To Do, In Progress, QA, Done)
  - **Task Lifecycle:**

1. **To Do:** New items await triage in grooming sessions.
2. **In Progress:** Developer picks up and moves card.
3. **QA:** After passing unit tests, moved for manual/integration testing.
4. **Done:** Deployed and verified.

- **GitHub Projects**
  - **Automated Sync:** ClickUp tasks tagged dev-ready are copied to GitHub as issues, assigned to team members.
  - **Release Planning:** Milestones align with ClickUp Sprint Lists.



Supply2U Management / Projects / Sprint 3 DevOps

Sprint 3 DevOps — URGENT COLLABORATION

TO DO 3 + Add Task

Name	Assignee	Due date	Priority
Add Container to deploy Azure docker ima...	IM	Tue	Urgent
Develop Backend Container and Push to A...	DN	Mon	Urgent
Developing Frontend Container and deplo...	IM	Mon	Urgent

+ Add Task

+ New status

Supply2U Management / Projects / Frontend Sprint 2 Modification

Frontend Sprint 2 Modification — NEAR COMPLETION

Communication Channels & Meeting Rhythms

To keep everyone connected—whether remote or in the labs—we used a mix of synchronous and asynchronous channels:

1. **Synchronous Meetings (Google Meet)**
   - **Sprint Planning (Mondays, 1 hr):** Define sprint goals, estimate effort, assign tasks.
   - **Mid-Sprint Review (Wednesdays, 30 min):** Quick health check on progress and blockers.
   - **Sprint Demo & Retrospective (Fridays, 1.5 hr):**
     - **Demo:** Each developer shows working features.
     - **Retro:** "What went well," "What can improve," and action items logged in ClickUp.
2. **Asynchronous Updates (WhatsApp)**
   - **Group Chat:** Quick questions, urgent bug alerts, coordination outside meeting hours.
   - **Automated Notifications:** ClickUp pushes due-date reminders and status changes to the WhatsApp group via webhook.
3. **Email Summaries**
   - Weekly summary of completed tasks and upcoming priorities emailed to stakeholders.

Version Control & Branching Strategy

All code lives in GitHub, with a strict branching and pull-request process to maintain stability:

- **Branch Model:**
  - main – Always deployable. Protected: no direct pushes.
  - develop – Integration branch for features (optional, adopted if multiple features overlap).
  - feature/<ticket-number>-short-desc – One per task; branches merged back into develop or main.
  - hotfix/<issue-number> – Urgent fixes branched from main, merged back into both main and develop.
- **Pull Request (PR) Process:**

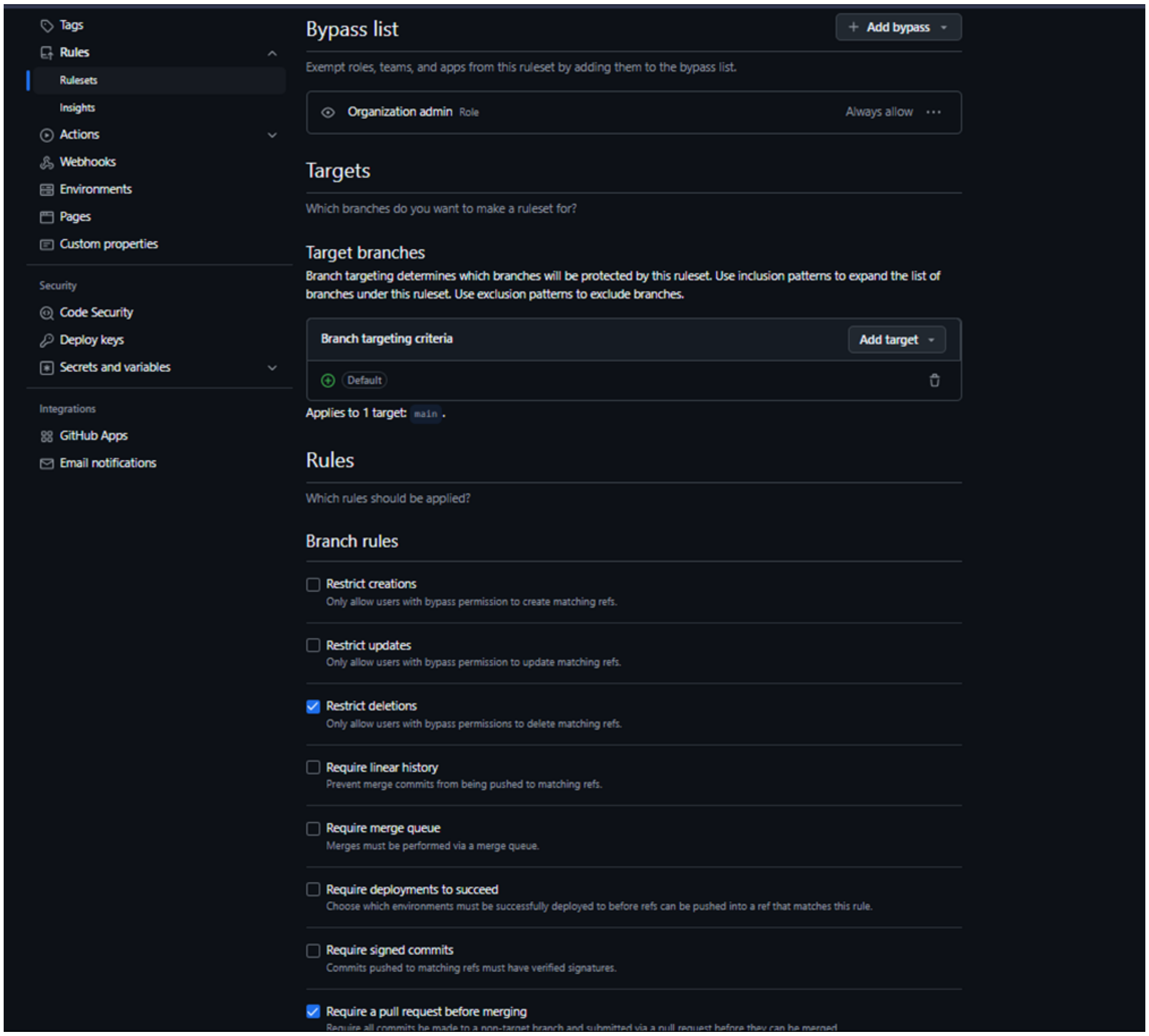
1. **Open PR** against develop (or main).

2. **Automated Checks:** CI pipeline must pass (lint, tests, build).

3. **Reviewers:** At least two team members, including one from backend or frontend as relevant.

4. **Merge Strategy:** Squash-and-merge, with a concise commit message summarizing the feature or fix.

5. **Release Tagging:** On merge into main, a GitHub Action tags the commit semantically (e.g., v1.2.0) and triggers the CD pipeline.



Bypass list

Exempt roles, teams, and apps from this ruleset by adding them to the bypass list.

Organization admin Role — Always allow

Targets

Which branches do you want to make a ruleset for?

Target branches

Branch targeting determines which branches will be protected by this ruleset. Use inclusion patterns to expand the list of branches under this ruleset. Use exclusion patterns to exclude branches.

Branch targeting criteria

Default

Applies to 1 target: `main`.

Rules

Which rules should be applied?

Branch rules

- [ ] Restrict creations — Only allow users with bypass permission to create matching refs.
- [ ] Restrict updates — Only allow users with bypass permission to update matching refs.
- [✓] Restrict deletions — Only allow users with bypass permissions to delete matching refs.
- [ ] Require linear history — Prevent merge commits from being pushed to matching refs.
- [ ] Require merge queue — Merges must be performed via a merge queue.
- [ ] Require deployments to succeed — Choose which environments must be successfully deployed to before refs can be pushed into a ref that matches this rule.
- [ ] Require signed commits — Commits pushed to matching refs must have verified signatures.
- [✓] Require a pull request before merging — Require all commits be made to a non-target branch and submitted via a pull request before they can be merged

Roles, Responsibilities & Ownership

Clear ownership prevents tasks from slipping through the cracks. We defined roles, assigned primary and secondary owners, and documented responsibilities.

Role	**Person**	**Responsibilities**
Product Lead	Ian	Defines priorities, writes user stories, stakeholders liaison
Frontend Lead	Neema	Oversees React code, reviews UI/UX implementations

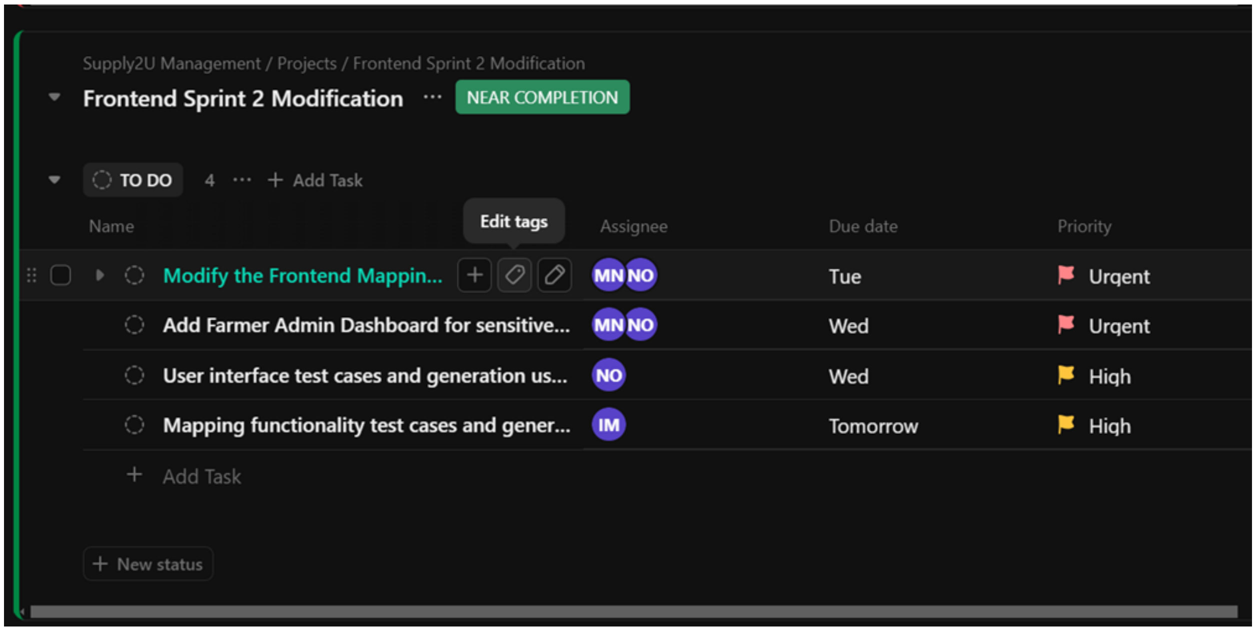
Backend Lead	David	Manages Django architecture, database schema, API design
DevOps Engineer	David/Ian	CI/CD pipelines, Azure infra, monitoring & alerts
QA Engineer	Ian/Neema	Creates test plans, runs manual/e2e tests, reports bugs

- **Task Assignment:**
  - ClickUp automatically assigns tasks based on “Assignee” field and notifies via WhatsApp.
  - GitHub issues mirror the same assignment.
- **Escalation:**
  - Critical production issues: Ping DevOps Engineer on-call; if no ack in 15 min, escalate to Product Lead

Retrospectives & Process Improvement

At the end of each sprint, we held a structured retrospective to capture lessons learned and turn them into actionable improvements.

1. **Retro Format:**
   - **What Went Well:** Celebrate successes (e.g., “API docs improved onboarding”).
   - **What Didn’t:** Surface pain points (e.g., “Merge conflicts slowed us down”).
   - **Action Items:** Create ClickUp tasks tagged retro-action with owners and due dates.
2. **Follow-Up:**
   - Review previous retro actions at the next sprint planning.
   - Track “Did it ship?” in ClickUp until closed.
3. **Continuous Improvement:**
   - After three sprints, evaluate cumulative actions and adjust team norms or tooling.



Supply2U Management / Projects / Frontend Sprint 2 Modification

Frontend Sprint 2 Modification ··· NEAR COMPLETION

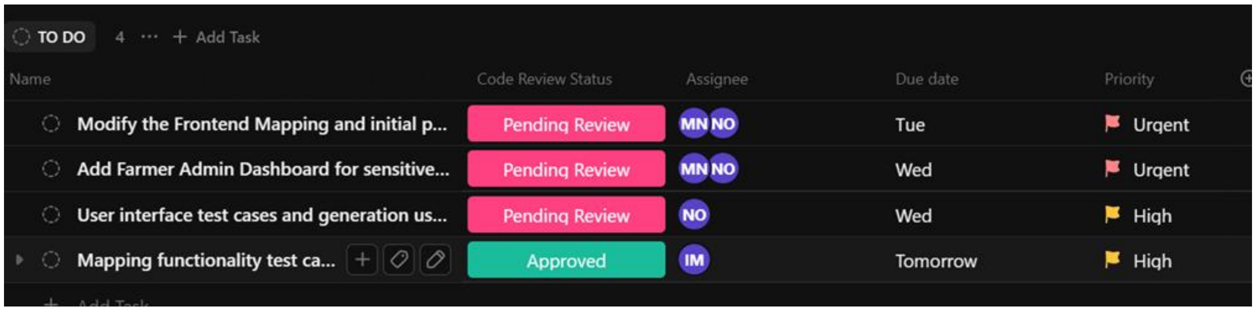
TO DO 4 ··· + Add Task

Name	Assignee	Due date	Priority
Modify the Frontend Mappin...	MN NO	Tue	Urgent
Add Farmer Admin Dashboard for sensitive...	MN NO	Wed	Urgent
User interface test cases and generation us...	NO	Wed	High
Mapping functionality test cases and gener...	IM	Tomorrow	High

Best Practices & Lessons Learned

Over the project, we refined our process into a set of guidelines:

- **Break Down Tasks Early:** Vertical slices that include frontend, backend, and DB changes prevent mid-sprint surprises.
- **Automate Everything:** From linting to notifications, removing manual steps reduced errors by ~40%.
- **Use Templates:** Issue and PR templates ensured consistent metadata (e.g., environment, steps to reproduce).
- **Stale-PR Reminders:** A GitHub App automatically reminds authors of open PRs older than 48 hours.
- **"QA Ready" Label:** Added a distinct label in both ClickUp and GitHub to signal when a feature passes unit tests and is ready for manual testing—cut hand-off wait times by 30%.



TO DO 4 ··· + Add Task

Name	Code Review Status	Assignee	Due date	Priority
Modify the Frontend Mapping and initial p...	Pending Review	MN NO	Tue	Urgent
Add Farmer Admin Dashboard for sensitive...	Pending Review	MN NO	Wed	Urgent
User interface test cases and generation us...	Pending Review	NO	Wed	High
Mapping functionality test ca...	Approved	IM	Tomorrow	High

12. Challenges and Solutions

Purpose & Overview

In any non-trivial software project, obstacles arise that test your team's processes, tools, and skills. This section candidly reflects on the key challenges we encountered while building the Supplytou GIS platform—and how we overcame them.

Development Delays

Challenge: Early in Sprint 2, integrating the React frontend's map-rendering component with the Django backend's geospatial API took far longer than estimated. We underestimated the complexity of serializing GeoJSON features and passing them through our REST endpoints.

- **Impact:** Two user-story cards slipped by one entire sprint, causing knock-on effects on demo readiness.
- **Solution:**
  1. **Spike Task:** We created a dedicated "Map Data Spike" task to prototype and benchmark GeoJSON serialization in isolation.
  2. **Incremental Delivery:** Broke the work into smaller PRs—first sending a single static polygon, then adding dynamic layers—so reviewers could merge partial functionality quickly.
  3. **Pair Programming:** Frontend and backend leads paired on one afternoon to resolve CORS and serialization mismatches in real time.
- **Lesson Learned:** Always schedule "spike" or research tasks when touching new libraries—factoring them into sprint capacity prevents over-commitment.

Merge Conflicts

Challenge: As multiple developers worked on overlapping areas of the Django models and API serializers, frequent merge conflicts in models.py and serializers.py became painful.

- **Impact:** Each conflict cost an average of 15–20 minutes to resolve, interrupting flow and causing context-switch overhead.
- **Solution:**
    1. **Smaller PRs:** Enforced a hard limit of 200 lines changed per PR to minimize conflict surface.
    2. **Branch Naming & Ownership:** Assigned "owners" to key files (e.g., Carol owns models, Eve owns serializers). Any incoming PR touching owned files required at least one approval from that owner.
    3. **Automated Merge Tools:** Adopted GitHub's "merge queue" app to serialize merges—only one PR could land to main at a time, reducing simultaneous edits.
- **Lesson Learned:** Clear file ownership and merge queues go a long way toward conflict reduction in rapidly evolving codebases.

Bugs & Debugging

Challenge: During our testing Sprint, we discovered an intermittent bug in the asset-tracking feature: under high load, the real-time truck locations would occasionally drop out for 10–15 seconds.

- **Impact:** This undermined stakeholder confidence in the "live tracking" demo.
- **Investigation:**
    - Added detailed logging around the WebSocket reconnection logic in Django Channels.
    - Simulated 50 concurrent connections locally using a custom Python script.
    - Identified that our heart-beat interval (30 s) was too long, causing some connections to be GC'd by Azure.
- **Solution:**

1. **Heartbeat Tuning:** Reduced heartbeat interval to 10 seconds and enabled automatic reconnection retries with exponential backoff.

2. **Resource Scaling:** Updated Container App autoscale rule to spin up additional replicas when CPU > 60% for 2 minutes.

3. **Monitoring Alerts:** Configured Log Analytics query to alert if more than 5 WebSocket disconnects occur in a 1-minute window.

- **Lesson Learned:** Always include production-like load testing in a sandbox environment before demos—small configuration tweaks can have big impacts under stress.

Time Management & Scope Creep

Challenge: Mid-project, stakeholders requested additional filters on the map (e.g., show only "organic" farms), which was outside our original scope.

- **Impact:** Without control, these requests risked ballooning the sprint backlog and missing deadlines.
- **Solution:**
    1. **Formal Change Requests:** Any new feature must now go through our ClickUp "Change Control" list with impact analysis and re-estimation.
    2. **Buffer Capacity:** Reserved 10% of each sprint's capacity for unplanned "high-priority" work, so urgent requests don't derail planned stories.
    3. **Stakeholder Demos:** We began doing mid-sprint demos of partial functionality, so stakeholders could see work-in-progress and refine requirements earlier.
- **Lesson Learned:** Building intentional slack and requiring formal change requests keeps the team focused and delivery predictable.

Infrastructure & Environment Challenges

Challenge: Setting up consistent Docker environments for local, CI, and production revealed subtle differences: a Debian vs. Alpine package mismatch caused the Django image to fail at runtime in Azure.

- **Impact:** CI builds passed locally but failed on Container Apps, delaying deployment by a day.
- **Solution:**
    1. **Unified Base Images:** Standardized both frontend and backend to use Alpine-based images (node:18-alpine, python:3.10-alpine).
    2. **Environment Parity:** Added a docker-compose.ci.yml that mirrored Azure environment variables, so CI ran in the exact same setup as production.
    3. **Automated Smoke Tests:** Extended our CI pipeline to spin up the Container Apps image in docker compose and run a basic health-check script before pushing to ACR.
- **Lesson Learned:** Strive for 100% parity between local, CI, and production container builds to catch platform-specific issues early.

Data Accuracy & GIS-Specific Issues

Challenge: When mapping farm boundaries, small geometry errors (self-intersecting polygons) caused Leaflet to render blank tiles or crash.

- **Impact:** Users saw missing farm areas or JavaScript exceptions.
- **Solution:**

1. **Geometry Validation:** Integrated django-gis's clean() method on model save to reject invalid polygons.
2. **Automated Tests:** Wrote pytest fixtures with sample invalid and valid GeoJSON files to ensure our API rejects bad data.
3. **User Feedback:** On the frontend, catch geometry errors and surface friendly messages ("Please redraw boundary; invalid shape detected.").

- **Lesson Learned:** In geospatial apps, validate all geometry server-side and provide immediate feedback client-side to prevent corrupt data ingestion.

13. Summary

The implementation of the Supplytou GIS platform has been a journey of careful planning, rigorous engineering, and continuous refinement.

- **Modular Architecture Delivered:**
  We built a clean separation between React-based frontend, Django-driven backend, and PostgreSQL database—each packaged in its own container and orchestrated via Azure Container Apps.
- **Robust CI/CD Pipeline:**
  Every commit now runs through automated linting, unit tests, multi-stage builds, and image pushes to Azure Container Registry, with zero-touch deployment upon merge to main.
- **Streamlined Collaboration:**
  ClickUp, GitHub Projects, WhatsApp, and Google Meet kept our team tightly aligned—enabling rapid task hand-offs, clear code ownership, and an evolving set of best practices.
- **Resilient Production Readiness:**
  We've tuned autoscaling rules, health-check probes, and alerting; established rollback procedures; and implemented smoke tests that mirror production environments.

Ready for What's Next:

- Our **staging environment** mirrors production and is armed with end-to-end tests.
- **Monitoring dashboards** and **alert rules** are in place to catch issues early.
- A **structured feedback loop** (retros, user surveys, bug triage) ensures that every real-world insight drives the next iteration.

With the core implementation complete and operations fully automated, Supplytou stands poised for comprehensive testing, stakeholder demos, and iterative enhancement based on direct user feedback.