I-CANOPY: A SOFTWARE PLATFORM FOR IOT SENSOR INTEGRATION

Sheng(Jerry) Wang [a,b] Grayson Lottes[a] Spencer Holman [a] George Venzke[a]

[a]Electrical Computer Engineering, The University of Iowa, Iowa City, IA, USA [b]Iowa Technology Institute (ITI), The University of Iowa, Iowa City, IA, USA

Introduction and Hypothesis

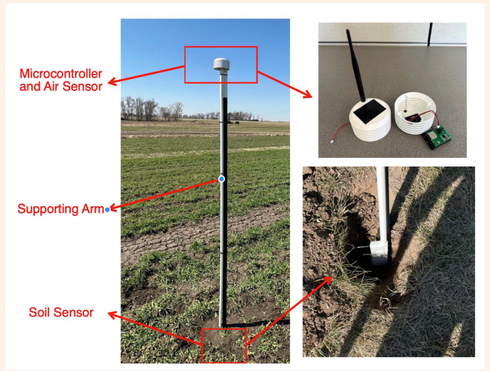


Figure 1. I-Canopy sensor on a local farm

In our previous projects, we developed air and soil sensors(shown in Fig. 1) aimed at citizen scientists and large-scale agricultural users, successfully deploying over 300 sensors across the U.S. in the past five years. However, we discovered through user feedback that key processes—such as ordering, sensor registration, and data access—required navigating websites, emails, or verbal instructions, often leading to long learning curves and communication challenges. To address these issues, we envisioned an integrated mobile platform that offers users an all-in-one solution, streamlining the journey from product browsing to real-time data monitoring.

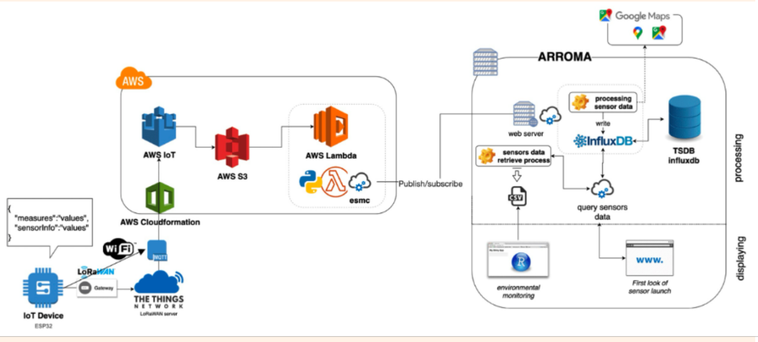


Figure 2. I-Canopy Sensor Data Processing Architecture

While we had already designed a complete IoT infrastructure (see Fig. 2), the individual features were fragmented and difficult for non-technical users to adopt. We hypothesize that developing an integrated mobile and desktop platform will significantly lower the learning barrier, improve communication efficiency, and enhance user satisfaction and engagement with IoT sensor data. By combining continuous user feedback with iterative software improvements, we aim to deliver a solution that meets both technical and user experience needs.

Process and Feature

We followed an agile process with iterative sprints and continuous user feedback. Early sprints refined Figma prototypes and delivered a stable, minimal interface; later sprints improved visuals and animations for consistency (Fig. 3). We focused on an intuitive UI with loading skeletons and clear feedback to simplify interactions and reduce the learning curve.

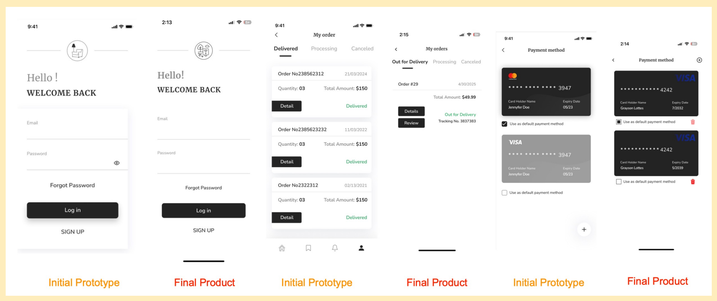


Figure 3. Figma prototype vs final application

Fig. 4 shows the tech stack used for the development, and Fig. 9 shows the entire system architecture for the mobile application.

Process and Feature

Initially, fetching 30-day sensor data via third-party APIs was slow and placed heavy load on the database. We introduced a Redis database to share the memory across all EC2 servers and a dedicated EC2 server running only Celery tasks (Fig. 5). This reduced database stress and boosted data retrieval speed by 5× (from 3.5s to 0.7s). Fig. 6 shows the time comparison for the entire data transfer.

Tech Stack

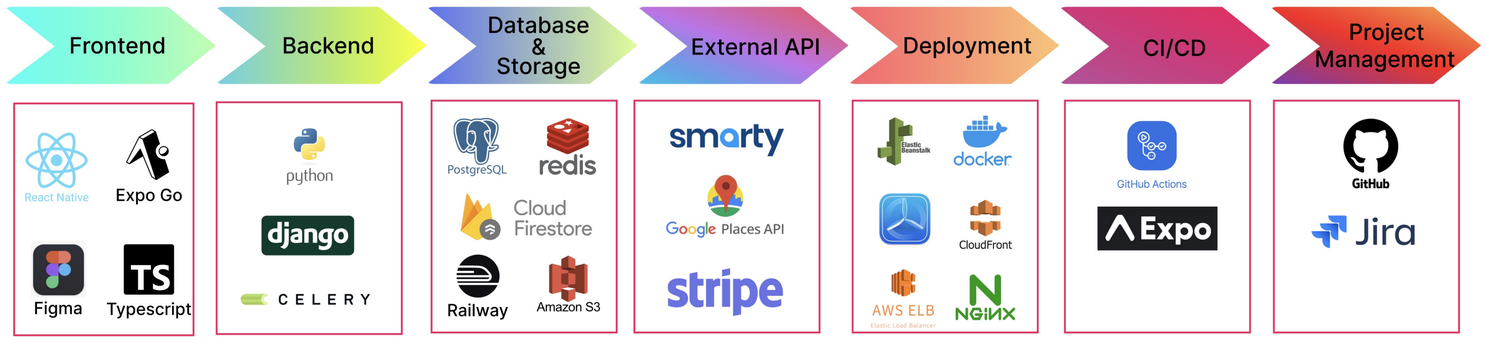
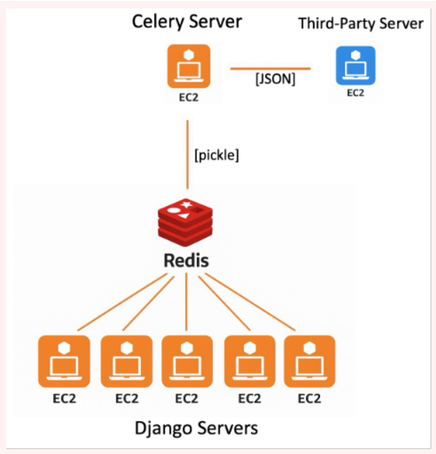


Figure 4. Tech stack used for the development

Data Caching and Aggregation



Testing #	Average(second)
Third-party API	3.8234
Caching	1.3318
Caching with serializers	1.0406

Figure 6. Total time taken for different caching strategies

Figure 5. Caching Architecture

Querying multi-year historical data was inefficient due to a large unoptimized table. We redesigned the system using Celery to update a pre-aggregated summary table in O(1) time during data ingestion. This reduced data size by 400× and improved query performance without compromising the charting detail, as visualized in Fig. 7.

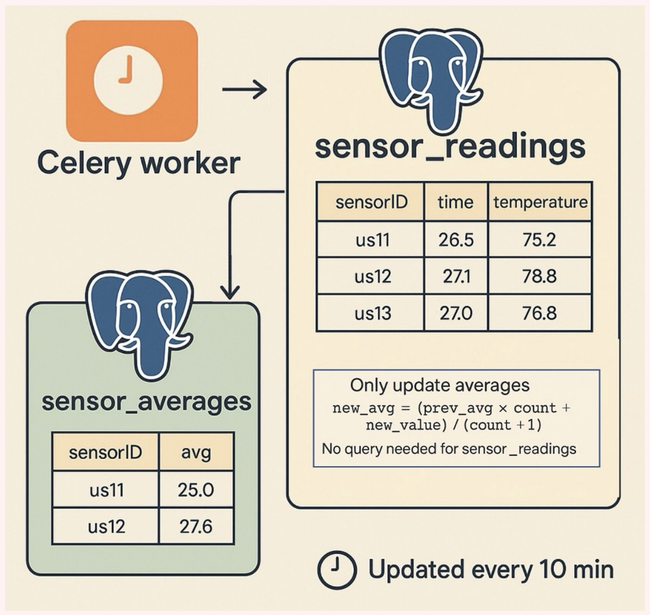


Figure 7. Aggregation workflow

Continuous Integration/Deployment

Over four months of development, we used continuous integration and deployment (CI/CD) with GitHub Actions to test every pull request on both the React Native frontend and Django backend. We maintained a dedicated deployment branch, which automatically pushed builds to TestFlight. As a result, syncing branches and updating the app on our devices became a seamless, 30-minute process. The entire system architecture is shown in Fig. 9.

External Tools and APIs

- **Stripe API** — secure payment processing and default card support.
- **Google Places API** — real-time address autocomplete.
- **Smarty** — validated and geocoded shipping addresses.

Coding Standards

We adopted consistent coding conventions across the React Native frontend and Django backend. Testing was enforced using **Jest** for frontend components and **django-pytest** for backend views and models.

Fig 8 shows a sample code coverage summary, which we used to help monitor our test completeness and identify untested paths.

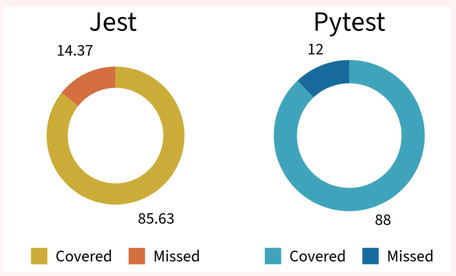


Figure 8. Sample code coverage report

Conclusion

This project sharpened our user-centered problem solving and continuous software refinement skills, as we practiced agile teamwork and self-learned new technologies throughout development.

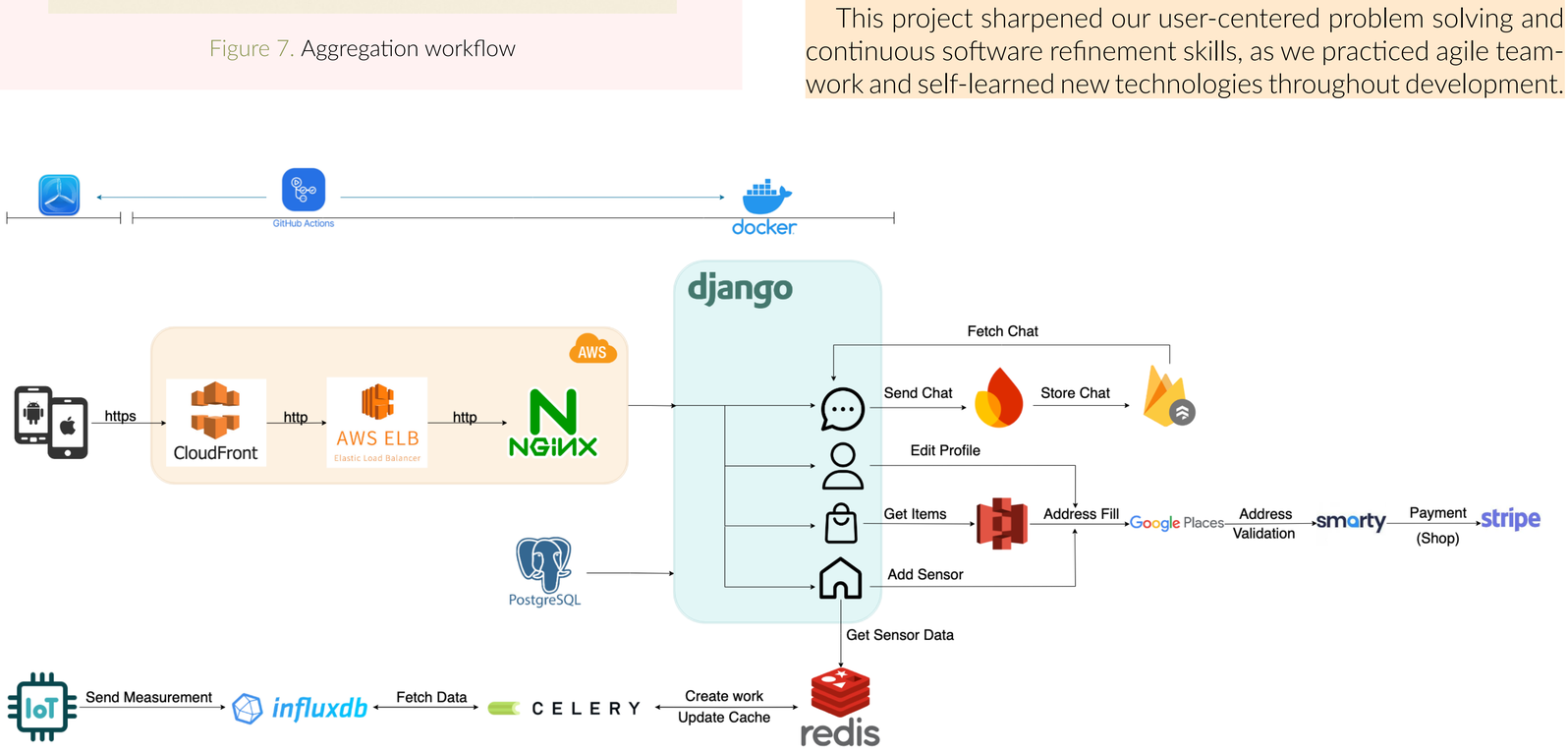


Figure 9. System architecture