The International Society of Precision Agriculture presents the
16th International Conference on Precision Agriculture
21–24 July 2024 | Manhattan, Kansas USA

A Flexible Software Architecture for General Precision Agriculture Decision Support Systems

Walter Neils[1], Dustin Mommen[2]
[1]University of Idaho, 875 Perimeter Drive Moscow, ID, 83844, USA
[2]Laurel Grove Wine Farm, 2311 Laurel Grove Road Winchester, VA, 22602, USA

A paper from the Proceedings of the
16th International Conference on Precision Agriculture
21-24 July 2024
Manhattan, Kansas, United States

Abstract.
Agricultural data management is a complex problem. Both the data and the needs of the users are diverse. Given the complexity of the problem, it's easy to ascertain that a single solution will not be able to meet the needs of all users. This paper presents a software architecture designed to be extensible as well as flexible enough to provide agricultural management tools for a wide variety of users. The solution is based on a microservice architecture, which allows for the creation of new services to meet the unique needs which come with various sets of data without requiring the modification of existing services. The modularity provided by this system means it's possible for a farm to pick and choose what features they require from their agricultural data management software. Microservices are designed around the idea of fulfilling a single task with minimal dependencies on other services, if any at all. Each microservice is containerized, which allows for easy deployment, management, and scaling on the component level. Microservices can be written in whatever language is most appropriate for the task at hand, and the system is designed to be language-agnostic. Each microservice exposes its functionality through whatever method suits its use case best. This is primarily done through RESTful APIs, but other methods are possible, supported, and encouraged. These include WebSockets, gRPC, WebRTC, and more. In addition, the system is designed to be deployed on a variety of platforms, including cloud-based solutions, on-premise solutions, and hybrid solutions with indifference to any particular choice. The client-side of the system is implemented using React, which is built around the idea of reusable components, promoting system modularity. This is a good fit for the microservice architecture, as it allows functionality to be easily integrated into the user interface at various points in the application without requiring explicit knowledge of the microservice's implementation. This decision support system, part of the Sensor Collection and Remote Environment Care Reasoning Operations (SCARECRO) project, has been implemented at Laurel Grove Wine Farm in Winchester, Virginia since February 2023. The interface has curated the analysis of more than three hundred microclimate sensors, assisting the proprietors in planning the planting of the vineyard. The open-source release of the project code is planned for summer 2024.

Keywords.
Precision agriculture, dashboard, data visualization, architecture, software architecture, microservice architecture, containerization, web.

The authors are solely responsible for the content of this paper, which is not a refereed publication. Citation of this work should state that it is from the Proceedings of the 16th International Conference on Precision Agriculture. EXAMPLE: Last Name, A. B. & Coauthor, C. D. (2024). Title of paper. In Proceedings of the 16th International Conference on Precision Agriculture (unpaginated, online). Monticello, IL: International Society of Precision Agriculture.

Background

Precision agriculture technology has exploded in recent years. Despite the rapid growth of systems designed to manage and visualize data, adoption remains much lower than expected. The reasons for this are numerous. Farmers commonly have difficulty viewing precision agriculture technologies as useful and cost effective (Lindblom et al., P5). Additionally, these data management and visualization systems are often difficult to use and unwieldy. They commonly involve requirements such as a specific brand of sensors for data collection, a certain data storage backend that may not be compatible with existing data collection systems, a specific operating system which restricts usability to a certain platform (e.g. no mobile device access or no desktop access), and other restrictive requirements. (Lindblom et al., 5) This lack of standards and compatibility between data collection systems and their visualization counterparts results in a battery of options, all of which either fail to meet farmers feature requirements or require a sensor network unique to the visualization mechanism. Some systems attempt to correct compatibility issues by using commonly available system utilities, such as CSV formatted files accompanied by an XML schema describing the shape of the data (Tan et al., 2). While these approaches do improve compatibility between various data collection systems, they still fundamentally rely on the system parsing data in an expected format from a sole source. This requires the development of additional programs to load remote data, convert it to the desired format, and then place it where the analysis program can get to it, which is time-consuming and requires additional development time. An alternative method is to have a human manually export the data, which is also undesirable.

Significance

With these issues in mind, a system capable of presenting data in an easy, user-friendly manner must be developed. To avoid problems brought about by requiring a specific data collection backend, the system must not be designed around any specific data collection backend, and rather provide a set of data contracts for which one can build an adapter for any given data collection backend. The system must also be capable of being easily extended; farmers have extremely varied requirements for a data visualization system, and anything less than trivially extendible is unacceptable. Towards this end, a system which is modular, easily extensible, user friendly, and data source agnostic must be developed.

Methods

Designing a system to handle the exceedingly disparate problems faced by farmers required the development / adoption of an architecture designed for extensibility, as well as a high-performance, system-agnostic user interface for using the system in the varied environments farmers find themselves in. In order to support all possible display devices, we decided to create a web application, as browsers are ubiquitous and thus extremely accessible to farmers. Modern browsers are extremely powerful; they present interfaces through which a developer can cache data locally, render 3D models in real-time, create socket connections, and a host of other features. For the development of the actual user interface, rather than the driving software that runs locally on the client's device, we decided to use React, a high-performance user interface library for the web. React is an industry standard for developing web interfaces, which leads to an enormous collection of available support libraries. The choice of React is also largely in part due to the fact that its common use yields itself towards future development by other developers without the need to learn a new library. The browser environment is geared towards JavaScript, which is a weakly typed interpreted language. This unfortunately makes simple logic and structure errors extremely easy to make. In order to amend this, we decided the implementation would be

written in TypeScript, which is a superset of JavaScript with strong typing. TypeScript transpiles to JavaScript, which means the resulting output runs natively in the browser even while developers are able to use advanced language features to create stronger compile-time guarantees and a more comprehensible codebase. With the tech stack for the frontend out of the way, we started working on the tech stack for the backend. An extensible system engineered to be capable of the extremely varied needs of each farmer requires a backend architecture which supports modular components. During research, our team came down to two possible architectures: monolithic, or microservice. A monolithic architecture depicts a system in which all logic runs in the same service. This has some advantages over its competitor; namely, certain data is shared across client endpoint accesses, specifically authentication data. A monolithic architecture ensures that data is close and easily accessible. However, it is not as extensible. Developers are restricted to writing new features in the same language that the original implementation was written in, and you can't separate more resource hungry endpoints onto their own servers as all instances of the service will expose the same endpoints, therefore requiring the same resources. It's in this case where the alternative candidate architecture shines. A microservice architecture is one in which each unique responsibility is handled by its own service. This separation of responsibilities into their own services provides quite a few benefits. Primarily, problems can be solved in the language best suited to handle them. A high-performance HTTP(S) REST API that interfaces with a database frequently isn't best implemented in Python. A machine learning based system designed to classify plant health is almost guaranteed to be implemented in Python. In a monolithic architecture, you're required to pick one language over the other and deal with the shortcomings that come with it. In a microservice architecture, the implementation is written in whichever language best suits the problem on a case-by-case basis. Other microservices don't care that a microservice is written in one language or the other so long as it adheres to the API it exposes. In addition, a microservice architecture yields itself to short, succinct codebases specialized to solve a specific problem. This makes maintenance much easier, as it's possible to digest the contents of the codebase rapidly without needing to concern oneself with the inner workings of other features. This allows developers to more rapidly integrate into the project and fix issues, as they're always working on a subset of the codebase and have less to worry about. Developers are also able to add new features to the backend without regard to the rest of the codebase due to the fact that new features necessitate a new service, which will be implemented in line with the ideas and framework a developer is comfortable with.

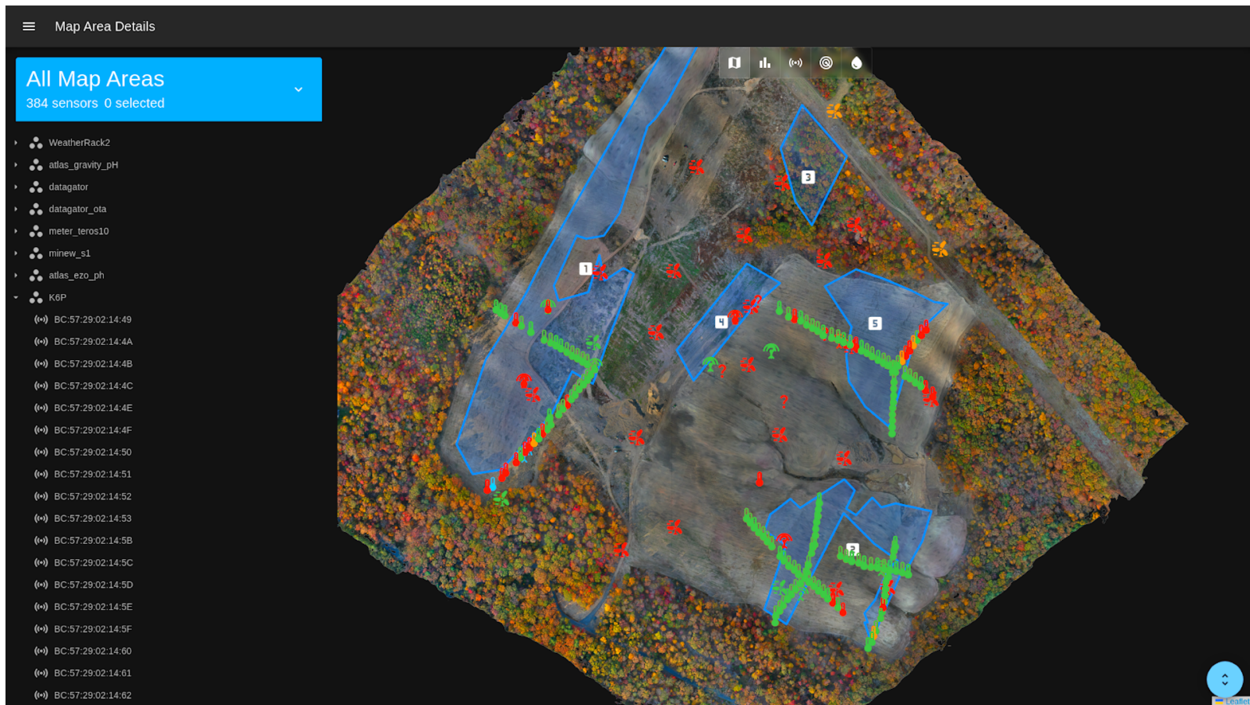
Another advantage of microservice architecture implementations stems from the fact that all failures are implicitly transient. In a monolithic service, certain categories of bugs (specifically those which include signals such as SIGSEGV) will immediately shut down the entire service, which leads to a total outage of all features while the service restarts. In this manner, faulty code can prevent all other features from functioning due to failures in one module. Microservice implementations don't suffer from this; each responsibility is handled in its own separate service which means a failure in one component is isolated to that specific component. A service failure can only affect the failing service and its dependents, if any. Having determined that a microservice architecture implementation would be best for our use case, research began on a system by which to orchestrate the individual services.

Previous work experience pointed the researchers to Docker, a containerization technology which packages an executable and the environment which it needs to run into a single, shareable image. Docker containers are effectively lightweight Virtual Machines (VMs); they're given their own isolated view of the host system, which prevents them from seeing other processes, accessing shared memory, reading the host filesystem, determining hardware capabilities, and a host of other attributes and functions. This is, of course, configurable; some processes may require hardware accelerators (e.g., NPU or GPU units) and it will be necessary to expose those resources to the containers which need them. This containerization system provides multiple benefits. Primarily, the environment in which a process runs is part of the container image, which means transferring an image between host systems will not lead to missing library errors or anything of the sort. Another benefit comes from the isolation inherent to containerization; a server which has a security vulnerability will not immediately allow attackers access to the host system.

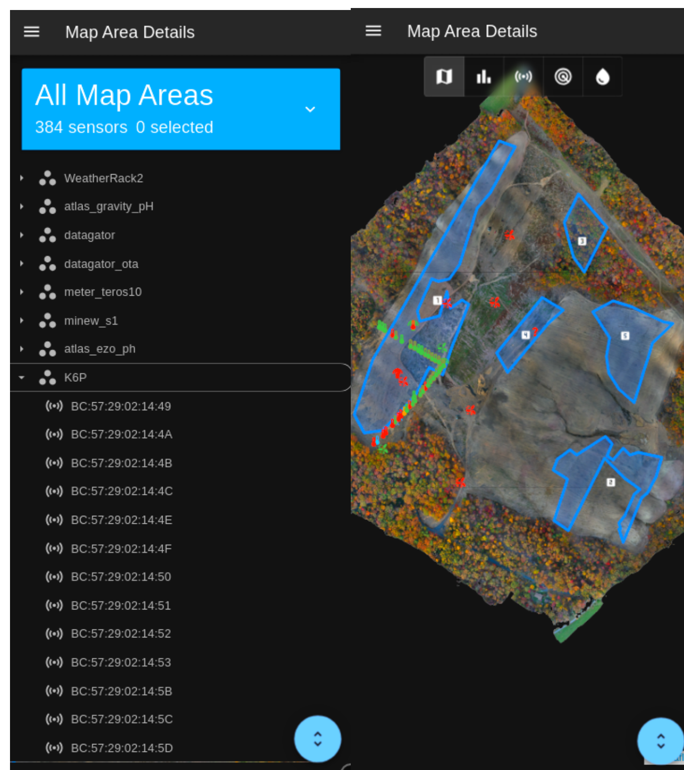
While not a surefire way to contain breaches, isolation is certainly a step in the right direction. Docker also provides configuration for automatic restart on failure, which means there's no developer effort required to restart failed microservices. The last of the benefits to be mentioned with regards to Docker in this paper is the fact that it provides a very convenient and efficient update and rollback mechanism. Multiple versions of a given docker image can exist on a given host system, and if a service must be rolled back it's as simple as changing the desired container version. By using an industry standard storage mechanism for container images such as AWS Elastic Container Registry, updates can be distributed nearly instantly, and old versions can be stored in case they become necessary. With the foundational tech stack defined, the creation of the actual microservices became the primary task. This involved the creation of four separate microservices. The first microservice, which is referred to as the "UI host service", was responsible for serving the static files resulting from compiling the user interface. The second microservice, referred to as the "sensor data service", is responsible for serving sensor data to the client application on request. The third microservice was the "GeoData service" and is responsible for serving geographic data to the client, such as various map tiles at various zoom levels. The fourth and final microservice is referred to as the "router service" and was responsible for presenting the REST API of the other three microservices in a unified manner to prevent the client application from needing to resolve individual services before making requests. As the implementation was to fit into a microservice architecture, each individual service (and thus responsibility) could be written in the language best suited to the task. The router service was implemented using NGINX, which is a high-performance multipurpose HTTP proxy and or file server. By writing simple routing rules, we were able to unify all microservices with sub-millisecond overhead into a single point of access. The GeoData service is implemented in TypeScript using the NodeJS runtime, and map tile data is stored remotely in an AWS S3 Bucket. An AWS S3 Bucket is a high-performance, cost-effective cloud storage system. This allows the service image itself to remain small as well as allowing easy reconfiguration of map tile information without requiring an image rebuild. The UI host service is implemented using another NGINX server, this time configured to serve the build artifacts for the React web UI. The backend service is implemented using TypeScript on the NodeJS runtime. NodeJS is chosen for its ability to handle thousands of concurrent IO bound requests with minimal CPU and memory requirements. In addition, TypeScript is a very development friendly language and allows for rapid iteration times. As we were re-creating preexisting functionality provided by an older web app, our base feature set was predefined. This base feature set was sourced from prior work done by Zach Preston at the University of Idaho We needed several core features, including but not limited to:

- Render a map comprised of multiple levels of drone images
- Plot sensor data in multiple types of graphs
- Compute values not presented by sensors
- Provide a login mechanism
- Interface with and edit sensor metadata
- Create and delete sensor metadata
- Create and delete sensor collection information (logical groups of sensors containing all instances of a given type of sensor)

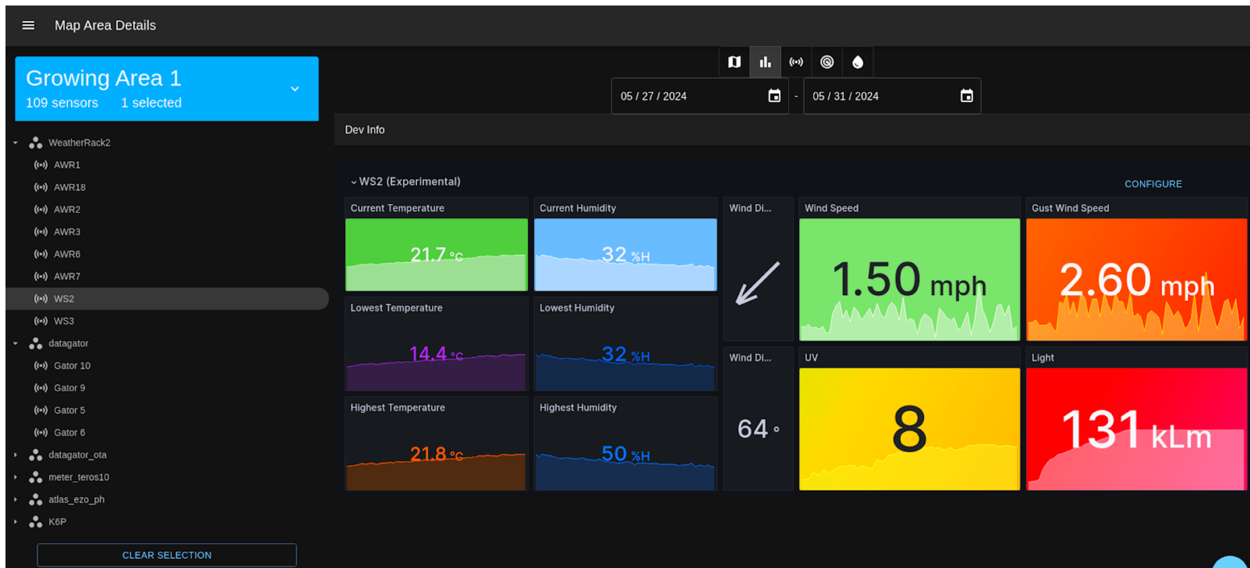
It is important to note that this microservice, on its own, does not solve the interoperability issue presented by other systems. It requires a MongoDB database with schema information and data storage. Other systems use different databases entirely and may not have schema information beyond the structure of database table information. One microservice, on its own, will not be compatible with all data storage implementations. Rather, this microservice encapsulates the responsibility of interfacing with data stored by the SCARECRO system (Everett et al.). Data access services for other systems will be implemented in their own services in accordance with the principles of microservice architecture design.



Desktop view of the Map Area Details Page



Mobile views of the upper and lower sections of the Map Area Details Page.



Desktop view of the Map Area Details Page quick sensor information view with one selected sensor

Plot of 264,162 points rendered live in the browser as part of the sensor data graphing page.

Results

The initial results for such a system are extremely promising. It is currently deployed on a vineyard in Virginia, which allows us to trial it in real usage conditions. Performance is excellent; the system currently has all service instances deployed on a single AWS VM (2 CPU cores, 2GB of RAM) and is still capable of handling hundreds of requests per second. Docker continues to be exceedingly useful; several rollbacks have been required throughout the duration of the system's operation, and they occurred without incident. Upgrades are also extremely easy; the entire deployment can be updated in less than 10 commands. The initial implementation of the GeoData service involved storing the tile data in the container image and serving the tiles with NGINX, which bloated the size of the image to the point of running the AWS VM out of storage. This was

corrected by the implementation described in the previous section, and stands as testimony to the principle that so long as the public facing API remains the same, the entire implementation of a given microservice can be altered or outright replaced without affecting the wider system. The researchers consider this to be successful in demonstrating the flexibility of the system. This system has been used to determine where varietals will be planted, determine whether frost damage may occur or have occurred, assist with sensor installation and reinstallation, assist with installing new sensors, and visualize correlations between different data sources and fields. According to users of the software, the features regarding editing, adding, and removing sensor instances from the system are helpful in growing operations. In addition, the ability to see sensor status at a glance on the map is regarded as extremely useful.

Issues

Due to an unfortunate oversight, the authentication system is currently part of the sensor data service. This lapse in architectural follow-through has not yet been corrected and continues to cause issues with verifying client identity across microservices. At present, the GeoData service cannot verify the identity of a user before serving map tiles. This allows clients to view map tile information regardless of their authentication status. This will be corrected later this year by removing the authentication implementation from the sensor data service and placing it into its own microservice, which will then expose a private API for other microservices to verify the identity of a given client.

Future Directions

Moving forwards, a refactor of certain system components will be necessary. This will correct architectural issues regarding authentication as well as provide a better standard of API documentation and introspect ability. The web UI codebase will be reorganized to encourage more component re-use as well as improve performance, most notably by decreasing the size of individual components to facilitate a better load time. When the refactor is complete, the project will be made open source in an effort to provide an open system for which people can easily integrate their existing data collection systems.

Conclusion

Through this research and development, the researchers have demonstrated the advantages of an agricultural data visualization system designed to be modular, easy to develop for, user friendly, highly performance, and accessible anywhere. The use of a microservice architecture proved to be invaluable when making the system modular and safe.

References

Everett, M., et al. “51. The SCARECRO System: Open-Source Design for Precision Agriculture Adoption Gaps.” *Precision Agriculture ’23*, 2 July 2023, doi:10.3920/978-90-8686-947-3_51.

Kubicek, Petr, et al. “Prototyping the Visualization of Geographic and Sensor Data for Agriculture.” *Computers and Electronics in Agriculture*, vol. 97, Sept. 2013, pp. 83–91, doi:10.1016/j.compag.2013.07.007.

Lindblom, Jessica, et al. “Promoting Sustainable Intensification in Precision Agriculture: Review of Decision Support Systems Development and Strategies.” *Precision Agriculture*, vol. 18, no. 3, 21 Dec. 2016, pp. 309–331, doi:10.1007/s11119-016-9491-4.

Tan, Li, et al. “An Extensible and Integrated Software Architecture for Data Analysis and Visualization in Precision Agriculture.” *2012 IEEE 13th International Conference on Information Reuse & Integration (IRI)*, Aug. 2012, doi:10.1109/iri.2012.6303020.