Georecord

1 Abstract

GeoRecord focuses on developing a scalable, modular web application that allows users to annotate specific geographic locations with comments known as "Geothreads." The project embraces a microservice architecture, leveraging services like user authentication, location management, and user data management to ensure high availability and scalability. React was selected for the frontend due to its modularity and reusability, while FastAPI was chosen for the backend to optimise performance. The system's deployment on AWS, using services like Amazon CloudFront, ECS, and RDS, further enhances its scalability and performance, ensuring a seamless user experience.

The project involved evaluating various architectural options, including layered and monolithic architectures, before settling on microservices for their ability to meet the project's architecturally significant requirements (ASRs). This choice addressed issues related to modularity, scalability, deployability, and availability, though it introduced complexity and potential latency challenges. These challenges were mitigated through careful design, such as using JSON Web Tokens (JWTs) for stateless session management and employing containerisation to streamline deployment.

2 Changes

The first step to creating our project was understanding the system requirements. Whilst we based this off the initial project proposal, there were still some potential changes that could be considered. The following features are the main services required to make the system operational:

- Basic drawing and editing tools,
- ‘Geothreads’ (comment section),
- User authentication and profiles,
- Basic filtering options.

As we delved into each of the following components, we identified a number of changes to make that would improve the system operationally.

Feature: Basic Drawing and Editing Tools
Initial Requirement: Users will be able to **draw** and **edit points** and **polygons** on the map interface.
Adjusted Requirement: Users will be able to **draw polygons** on the map interface.
Rationale: ● We wish to preserve the original context of discussions and comments, frequent changing on polygons can disrupt ongoing discussions as it can create confusion about the place in question. Therefore, we've removed the ability to modify polygons after they're made. ● Having both points and polygons complicates users' understanding of how to interact with the system. Points are also less visible and interactive than polygons, hence our decision to be able to create shapes on the map.

Feature: Geothreads
Initial Requirement: Users can start and participate in geothreads. Will be a pop-up interface similar to **popular discussion threads**, where users can add **comments** and **media files**.
Adjusted Requirement: Users can start and participate in geothreads. Will be a pop-up interface similar to **Reddit** or **Facebook comments**, where users can add **comments.**
Rationale: ● We've chosen to tailor our design to follow likeness to Reddit and Facebook comments. These are already successful discussion boards that exemplify how people know to interact with them. ● To reduce clutter and promote a cleaner, sleek look, we've decided to only include comments in the geothreads. Comments can also be censored when the information is considered invasive, whilst controlling images is not as feasible.

Feature: User Authentication and Profiles

Initial Requirement: Users will have basic profile functionality to manage their account settings and contributions.

Adjusted Requirement: Users will have basic profile functionality to manage their account settings and contributions, as well as a **levelling system** to keep track of how well they've contributed to the software.

Rationale:
- The inclusion of a levelling system gives users something to strive for. It enables them to stay motivated when using the system and allows them to have goals with their active contributions.

Feature: Basic Filtering Options

Initial Requirement: Users will have limited filtering options to search and explore data based on simple criteria **such as location or type of place.** We will include a search bar which will **match search terms to data in discussion forms and places**.

Adjusted Requirement: Users will have limited filtering options to search and explore data **based on location**. We will include a search bar which will **match search terms to location information.**

Rationale:
- We've opted to make the search bar only look for information based on the location. Searching by the data in discussion forms would allow for too many obscure results that could show up, making the users unable to find what they are actually looking for. By restricting this to location information, users are limited to only receiving results around the places they're actually looking into.
- For consistency through our filtering options, we have also decided to base the filter on location only. This global search will also streamline the interaction that users have with the system, requiring less user input for both polygon creation and searching.

Feature: ASRs
Initial Requirement: Availability, Scalability, **Extensibility**
Adjusted Requirement: Availability, Scalability, **Modularity, Deployability**
Rationale: ● As we are creating an MVP, our services should all be easy to adjust. Some of these changes we've made could prove to not be as optimal as we've desired. By ensuring a modular architecture, we are making sure that our system can be improved upon efficiently. ● For the same reason as above, we've opted to ensure our system is deployable. This means our system needs to be efficient when moving from development to production, especially given the range of services we'd like to include in our system. ● We have chosen to remove extensibility as we've included modularity. This is a strong requirement that narrows in on how well our system can be updated. Within the current scope of the system, extensibility doesn't stand as a testable requirement. Whilst it may be necessary to extend the system in the future steps, it remains outside of our current product.

3 Architecture options

While we were considering how to design our system, we had to decide which architecture options would be the most optimal for our system. This led us to three main options - layered, monolithic, and microservice architectures. The ASRs for our system are availability, scalability, modularity, and deployability, which were used as the primary decision maker for which architecture we would use.

3.1 Layered Architecture

A layered architecture divides the system into a number of high-level layers, each of which with a specific role and responsibility. This can be quite good as each layer can focus on its own logic, leading to cleaner code. This approach also promotes modularity, where changes in one later does not affect the others. This makes the system easier to maintain and update as future directions approach.

One of our core requirements is scalability, however, using a layered architecture would not meet this attribute. In a layered architecture, all components are tightly coupled and need to be scaled together. This can be quite inefficient, especially for a system like ours where different features (like geothreads or user authentication) may experience varying levels of demand. With the architecture requiring each layer to communicate with each other, performance could be hindered as latency is introduced. The risk of inefficient layers would lead to a slow-working system that would result in us missing our availability goal. Also, considering the downward dependency principle in the layered architecture, Changes in a low-level module could require changes in the high-level module that depends on it, reducing modularity.

3.2 Monolithic Architecture

In a monolithic architecture, all the system's functionality is handled by a single service. In our case, all four of our services would come together by communicating internally whilst being run on a single platform.

This follows quite a simplistic approach that is simple to develop, deploy, and test, which would be beneficial in early stages. There would also be minimal concerns for performance issues as there would be no heightened latency caused by communication between services.

However, given the simplicity of a monolithic architecture, there are a number of concerns around our required quality attributes. In a monolithic architecture, the entire application must be scaled as a whole. This can be inefficient, especially when features have varying levels of demand. An example of how this would impact us is that we could have an increased demand on the geothreads feature, but would have to scale every feature up to follow. Similarly, if any service has an outage, the entire system would go down, ruining our need for availability.

Updating any part of the system also requires the entire system to be redeployed. This can be very time-consuming and inefficient, especially in the perspective that our system is an MVP. With the idea that we can grow and expand our system to include new features, a monolithic architecture only encourages difficulty to work with over time despite providing a simple design initially.

3.3 Microservice Architecture

A microservice architecture is an architectural pattern wherein a larger service is decomposed into numerous smaller fine-grained “microservices” that are loosely coupled but tightly integrated. Each microservice can be developed, deployed, and scaled independently, as different parts of the software are decoupled. This aligns directly with our demand for modularity. Beyond this, as the services are decoupled, if one microservice fails, the others can continue to function, preventing a complete system outage. This is a key problem around availability with a monolithic architecture that microservices address well.

Each microservice can be scaled according to its needs due to the decoupling of services. This is another issue prevalent and outlined for both layered and monolithic architecture. It is also easier to update microservices than entire systems and layers. Each microservice can be redeployed independently, allowing for faster and safer updates to individual features.

Microservice Architecture directly aligns with each of our four ASRs, however, there are two main concerns with how it considers our systems functionality. Microservices communicate over the network, which can lead to increased latencies. However, with proper design and optimisation, this is a concern that can be mitigated. Microservice architecture is also far more complex than both layered and monolithic. The orchestration of multiple microservices can make the service more difficult to deploy initially and manage over the applications life cycle.

Given the system’s functionality and the identified ASRs, a microservice architecture is definitely the most suitable. It provides the modularity needed for the diverse features, the horizontal scalability to handle growing demand, the availability to ensure a reliable user experience, and the deployability to allow for efficient updates. While there are potential issues around complexity and increased latency, these can all be managed with effective design.

4 System Architecture

GeoRecord is a web application that a user can simply interact with, as seen in Figure 1. It is a system where people can leave different comments about a certain location that they define by drawing a polygon in that area.

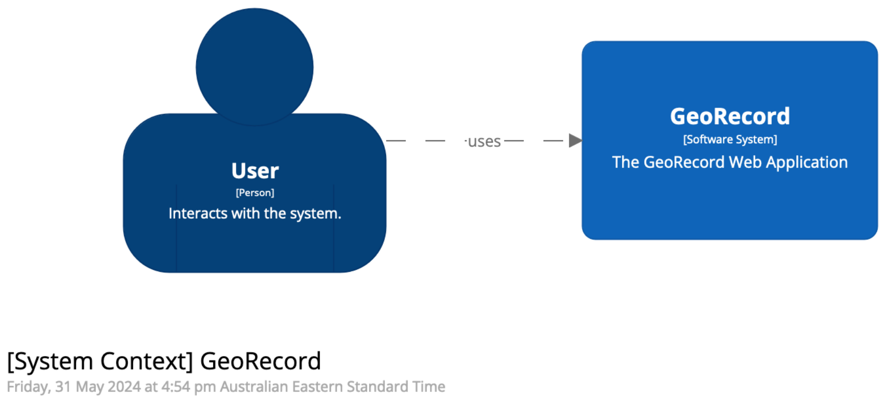

User
[Person]
Interacts with the system.
uses
GeoRecord
[Software System]
The GeoRecord Web Application
[System Context] GeoRecord
Friday, 31 May 2024 at 4:54 pm Australian Eastern Standard Time

Figure 1: System Context Diagram (Blown-Up View)

The design that we ultimately decided as the framework for our system was a microservice architecture. However, that's not to say our work wasn't guided by layered and monolithic architectures as well.

To implement GeoRecord in a microservice architecture, the team first began by modelling the system as a layered architecture. That is, partitioning the elements of the system into the categories of the presentation layer, business layer, persistence layer, and the database layer. This yielded a result of an architecture consisting of three services, a web application interface (presentation layer), a HTTP REST API (business layer), and a database (database layer).

Within the layered architecture, it is apparent that the HTTP REST API business layer is a monolith that could be decomposed. A functional decomposition was undertaken of this layer, which resulted in it being partitioned into three key services; authentication, location management, and user data management. The resulting microservice architecture is described in Figure 2.

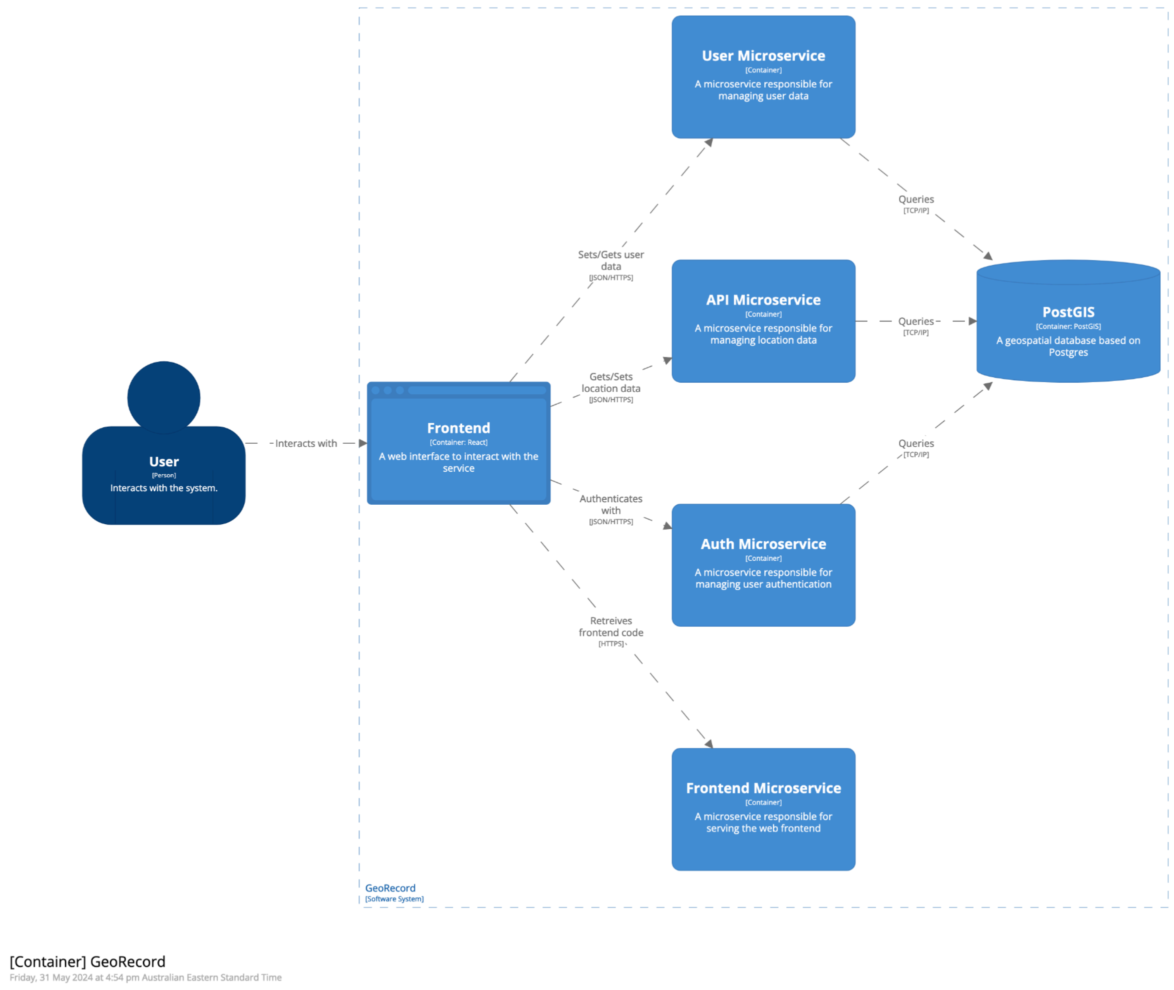


Figure 2: Container Diagram (Blown-Up View)

4.1 Presentation Architecture

The Frontend is the primary interface for the user, developed using React and accessed via a web browser. It is composed of several components, including the Auth Hook, which manages authentication state and communication, the Location Hook, which manages location state and communication, and the User Hook, which manages user data communication.

The selection of React as the basis of the project's user interface was informed by the project's architecturally significant requirements. React's component-based architecture offers modularity and reusability, enabling the development team to efficiently build and maintain a complex, dynamic UI with a rich set of features. This approach synergises with the project's ASRs of modularity and scalability.

Using Amazon CloudFront was highly beneficial for scalability and software performance. The CDN distributes the load of delivering content by routing user requests to the nearest edge location, which can cache responses and provide lower-latency routes to the origin servers. This not only reduces latency but also increases the speed of content delivery, enhancing the user experience. A CDN can also handle sudden traffic spikes and distribute traffic among multiple servers, preventing any single server from becoming a bottleneck. This greatly improves scalability as the system can serve a large number of users simultaneously without degrading performance. The CDN then forwards the web traffic to the Application Load Balancer.

4.2 Business Layer Architecture

The business layer of the GeoRecord system operates as a collection of microservices, as seen in Figure 2, each serving a specific purpose. This is a characteristic of a microservice architecture, where an application is structured as a collection of loosely coupled services. In the case of GeoRecord, these include the Auth Microservice for managing user authentication, the API Microservice for managing location data, the User Microservice for managing user data, and the Frontend Microservice for serving the web frontend. Each microservice communicates with the frontend and queries the PostGIS database as needed. This design allows each microservice to operate independently, yet work together to deliver the overall functionality of the system.

Separating the business layer into four main microservices ensures that our system meets the specified ASRs. Each microservice outlined previously operates independently, where this modularity only promotes scalability, deployability, and availability. As the load on a particular service increases, more instances of that service can be deployed without affecting the others. This allows the system to effectively handle increasing loads and user demands. Each of the services automatically scale up and down as required to meet user load as efficiently as possible. Section 6.4 elaborates on the success our application had in doing this.

We also used FastAPI as our web framework for building our REST API. It is a modern, fast, easy-to-use framework that offers several advantages when compared to others such as Flask. FastAPI is one of the fastest Python frameworks currently available, significantly faster than Flask. As we are using a microservice architecture, it’s important to consider and optimise our design to counteract potential latency issues. FastAPI also possesses support for asynchronous request handling. This can lead to significant performance improvements for IO-bound applications, such as GeoRecord. Our system often needs to wait for data to be retrieved from or stored in the database during its interactions with the user, especially when handling requests that involve significant data processing. A good example of this would be when the system is querying the geospatial data in the PostGIS database. The performance of GeoRecord is directly influenced by the speed of its IO operations. This is why the inclusion of technologies that can improve IO operations, such as FastAPI and AWS services, is beneficial to the operation of our system.

4.3 Database Layer Architecture

The database layer of the GeoRecord system is represented by a RDS database with the PostGIS extension. PostGIS is an open-source software program that adds support for geographic objects to the PostgreSQL object-relational database. PostgreSQL has strong capabilities around reliability, feature robustness, and performance. This high performance and reliability contributes to the overall availability of the system. The database's ability to handle complex queries and large amounts of data also enhances the system's scalability. The Auth, API, and User Microservices all query the PostGIS database.

Currently, the PostGIS database is hosted on an AWS RDS provisioned instance. AWS RDS makes it easy to set up, operate, and scale a relational database in the cloud, which enhances the deployability and scalability of the database. However, after the MVP, it would be optimal to move to a serverless RDS. This transition will further improve the scalability of the database as it allows the database to automatically adjust capacity to match the application's needs, resulting in improved performance and increased efficiency.

4.4 Deployment

The GeoRecord system is deployed and hosted on AWS. Figure 3 details how it is divided into several key components.

The frontend of the GeoRecord system is run on the customer's computer and is accessed via a web browser. This setup ensures that the system is widely accessible to users on various devices and platforms. This frontend communicates with the AWS Global infrastructure, specifically with CloudFront. The CDN efficiently routes and caches traffic, ensuring high availability and performance of the system. It also enhances the system's scalability as it can handle a large number of requests without degrading performance.

AWS US-EAST-1 is the primary hosting environment for the backend services and the database of the GeoRecord system. The web traffic from the CDN is routed to an AWS Elastic Load Balancer which routes requests to the microservices. It ensures high availability by efficiently distributing incoming traffic across multiple targets. The microservices are hosted on AWS Elastic Container Service (ECS). ECS is a highly scalable container orchestration service that supports Docker containers. It allows us to easily run and scale our containers on AWS without having to maintain the underlying infrastructure. An application autoscaling policy is in place for each of the ECS instances, allowing them to scale to meet increased load on the system and scale down for efficiency.

The PostGIS database is hosted on RDS. This is a tool that makes it easy to set up, operate, and scale a relational database in the cloud. RDS enhances the deployability and scalability of the database.

GeoRecord offers an intuitive web platform for location-based interactions, supported by a streamlined microservice architecture, robust business layer, and efficient database management.

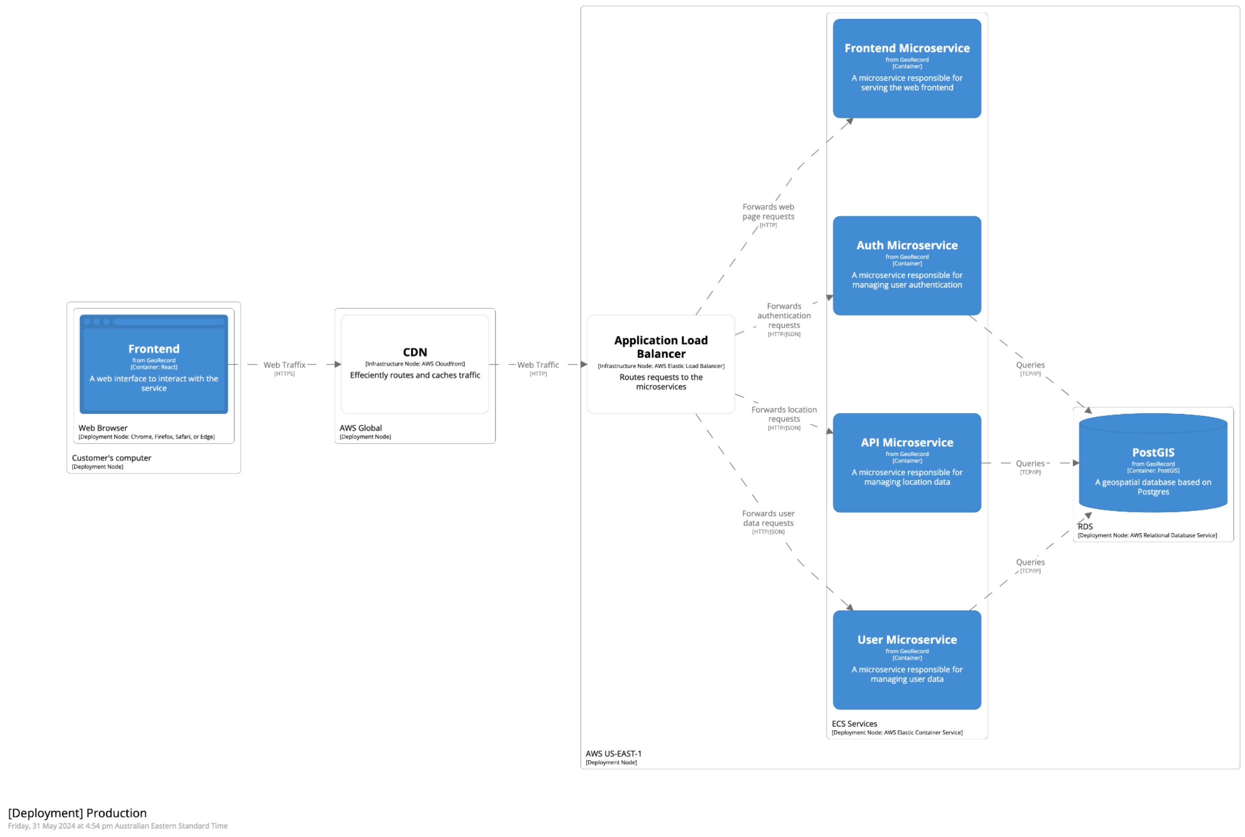

Frontend
from GeoRecord
[Container: React]
A web interface to interact with the service
Web Browser
[Deployment Node: Chrome, Firefox, Safari, or Edge]
Customer's computer
[Deployment Node]
Web Traffic
[HTTPS]
CDN
[Infrastructure Node: AWS Cloudfront]
Effeciently routes and caches traffic
AWS Global
[Deployment Node]
Web Traffic
[HTTP]
Application Load Balancer
[Infrastructure Node: AWS Elastic Load Balancer]
Routes requests to the microservices
Forwards web page requests
[HTTP]
Forwards authentication requests
[HTTP/JSON]
Forwards location requests
[HTTP/JSON]
Forwards user data requests
[HTTP/JSON]
Frontend Microservice
from GeoRecord
[Container]
A microservice responsible for serving the web frontend
Auth Microservice
from GeoRecord
[Container]
A microservice responsible for managing user authentication
API Microservice
from GeoRecord
[Container]
A microservice responsible for managing location data
User Microservice
from GeoRecord
[Container]
A microservice responsible for managing user data
ECS Services
[Deployment Node: AWS Elastic Container Service]
Queries
[TCP/IP]
Queries
[TCP/IP]
Queries
[TCP/IP]
PostGIS
from GeoRecord
[Container: PostGIS]
A geospatial database based on Postgres
RDS
[Deployment Node: AWS Relational Database Service]
AWS US-EAST-1
[Deployment Node]
[Deployment] Production
Friday, 31 May 2024 at 4:54 pm Australian Eastern Standard Time

Figure 3: System Deployment Diagram (Blown-Up View)

5 Trade-offs

Throughout the development of GeoRecord, numerous trade-offs had to be evaluated and mitigated to ensure that the architecturally significant requirements of the system were fulfilled. These trade-offs had to be made as a result of the microservice architecture option being selected over a monolithic or layered architecture.

5.1 Distributed State

One of the fundamental challenges choosing a microservice-based architecture over a monolithic architecture is the management of distributed state. One example of this distributed state was the user authentication tokens. Traditionally, in a monolithic application, session states can be stored locally, allowing for straightforward and efficient user session management. However, the distributed nature of microservices necessitates a different approach to share session states across services. This shift brought to the forefront the trade-off between the convenience of local state management in monoliths and the scalability and resilience offered by microservices.

To address this challenge, several options were considered. Initially, storing session data in the database layer, facilitating shared access across services, was considered. However, this solution threatened to introduce the database as a bottleneck, as every request would require an additional database query. This contradicts the projects' ASR of scalability so another solution was required. After evaluating the trade-offs, this trade-off was mitigated by implementing JSON Web Tokens (JWT) for session management. JWTs provided a stateless mechanism to maintain user sessions, significantly alleviating the burden on the database and enhancing the system's scalability.

5.2 Local development

The complexity of setting up a local development environment is markedly higher in a microservices architecture than in a monolithic setup. The requirement to run and coordinate multiple services simultaneously presents a notable hurdle, affecting developer productivity and increasing the setup time for new team members.

This trade-off was mitigated through the use of containerisation. A "Dockerfile" was produced for each service, orchestrated by a single docker-compose file. However, some microservices depended on cloud-native technologies. In this case, a suitable replacement was found for local development, such as Nginx for AWS Application Load Balancers and Postgres for AWS Relational Database Service. This setup allowed developers to use a single `docker-compose` command to initiate all required services. Consequently, the use of containers and container orchestration tools not only addressed the challenge of local development complexity but also ensured reproducibility of builds across development and production environments.

5.3 Orchestration & deployment

A key advantage of monolithic and layered architectures is the ease with which they can be deployed and operated. Typically, these applications will run within a single virtual machine, minimising the surface area that operators must maintain. However due to the large number of services that must interoperate, microservices require a significantly larger amount of infrastructure to run. Increasing the difficulty of deployment and operational overhead.

To mitigate these complexities introduced to the deployment and management processes, infrastructure-as-code (IaC) was leveraged. This allowed the deployment of services, networking infrastructure, and other components to be entirely automated, enabling the entire service to be deployed with a single “terraform apply” command. This allowed us to fulfil the modularity ASR of the system without sacrificing the deployability ASR.

6 Non-Functional Requirements

The architecture of the GeoRecord system is well-suited to delivering and meeting our identified quality attributes.

6.1 Availability

GeoRecord is designed to have high availability, primarily due to the AWS infrastructure that it is deployed on. The high availability is implemented using an AWS Elastic Load Balancer that will balance incoming traffic for its microservices, diminishing the downtime by preventing one service from causing failure. AWS ECS with an autoscaling policy further enhances availability by automatically scaling up or down based on the load, ensuring that services remain available even during traffic spikes. This couples well with meeting the scalability requirement and is demonstrated in Figure 4. The use of AWS CloudFront as a CDN also contributes to availability. CloudFront caches content at edge locations, reducing latency while handling large volumes of requests efficiently, which ensures that users experience minimal disruption and downtime.

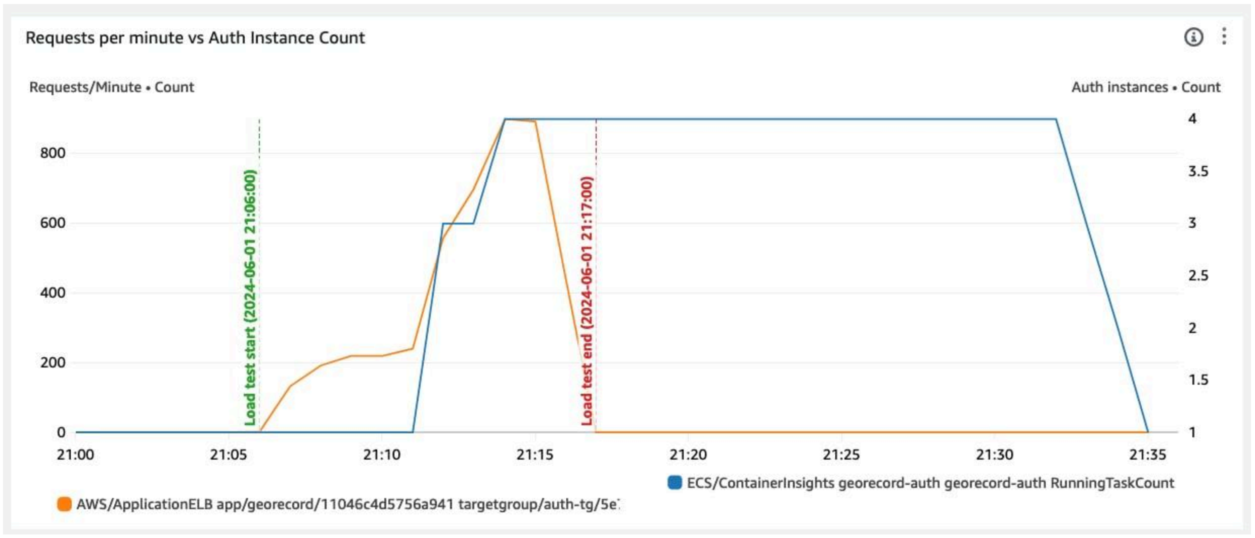


Figure 4: Auth service load test - Requests per minute (orange) vs Authentication microservice instance Count (blue)

6.2 Deployability

One of the strong suits regarding the architecture of GeoRecord is deployability. This is attributed to the microservices approach and their deployment through containerisation. Each microservice is deployed independently using Docker containers managed by AWS ECS. This setup simplifies the deployment process, allowing for continuous integration and continuous deployment practices. Changes to individual microservices can be deployed without affecting the entire system, reducing the risk associated with deployments and making it easier to roll back updates if necessary.

The future plan to transition to a serverless RDS for the database layer will further enhance deployability by allowing automatic scaling of the database resources in response to application demands without manual intervention.

6.3 Modularity

GeoRecord's architecture excels in modularity due to its microservices design. Each microservice (Auth, API, User) operates independently, which aligns with the project's ASRs of modularity and scalability. This modularity allows for easier maintenance and updates, as changes in one service do not directly impact others. The use of React for the frontend also supports modularity through its component-based architecture, enabling the development team to build and maintain a complex UI with reusable components.

6.4 Scalability

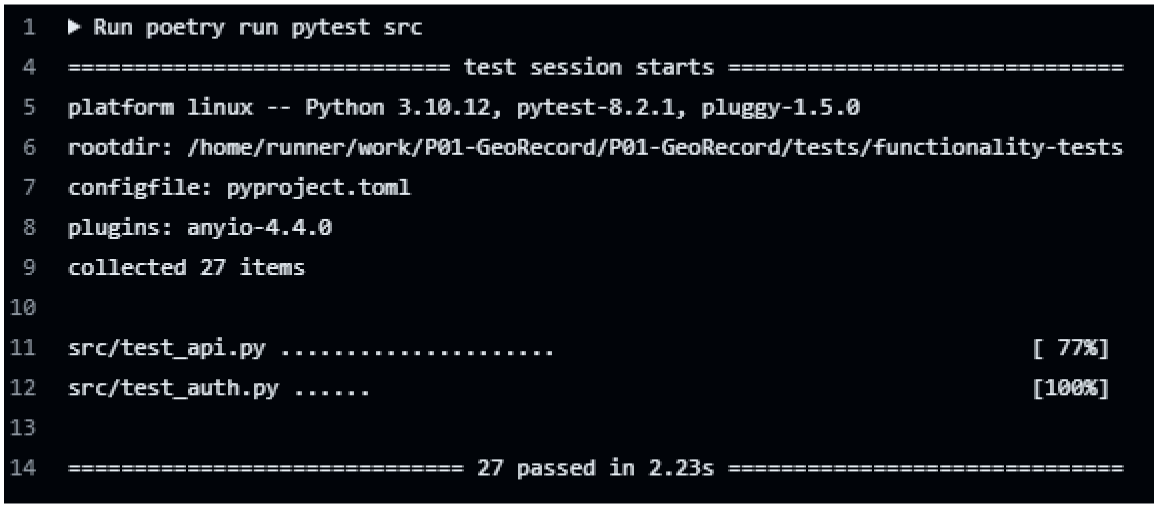
Scalability is both tested and argued effectively within the systems architecture. Using AWS ECS for managing microservices supports horizontal scaling by adding or removing container instances based on demand. The autoscaling policy ensures that the number of active instances increases with the load, as shown in the load test results in Figure 4 where the system scales up with increased requests per minute and scales down when the load decreases.

The PostGIS extension on the RDS database, coupled with the potential shift to a serverless RDS, ensures that the database can handle large volumes of data and complex queries efficiently. AWS CloudFront also aids scalability by handling traffic spikes and distributing content load across multiple servers, ensuring consistent performance even under high demand.

The use of microservices, AWS infrastructure, and containerisation provides a robust framework that meets these ASRs effectively. The test results corroborate the system's ability to scale dynamically in response to load, while the architectural design supports high availability, ease of deployment, and modular development.

7 Functional Requirements

The GeoRecord system delivers full functionality regarding the scope outlined in the original project proposal and the changes described in Section 2**.** An outline of each test case conducted to ensure that our system meets its functional requirements can be found in our GitHub repository here. Figure 5 displays the successful execution of the test cases using poetry.



```
▶ Run poetry run pytest src
============================ test session starts ==============================
platform linux -- Python 3.10.12, pytest-8.2.1, pluggy-1.5.0
rootdir: /home/runner/work/P01-GeoRecord/P01-GeoRecord/tests/functionality-tests
configfile: pyproject.toml
plugins: anyio-4.4.0
collected 27 items

src/test_api.py .....................                                   [ 77%]
src/test_auth.py ......                                                 [100%]

============================= 27 passed in 2.23s ==============================
```

Figure 5: Tests Run with Poetry (test results)

7.1 User Authentication and Profiles

Test cases for successful login, user registration, and handling of invalid login attempts ensure that only authorised users can access the system. Specifically, tests for successful login (TC1) and new user registration (TC2) confirm that users can authenticate and create profiles without issues. Additional tests for registering existing users (TC3), incorrect email login (TC4), and wrong password attempts (TC5) validate that the system correctly handles errors and prevents unauthorised access. A health check test (TC6) confirms that the authentication service is operational.

7.2 Basic Drawing Tools

The test for registering new locations (TC7) confirms accurate and efficient data entry, while the test for invalid location registration (TC8) ensures that the system maintains data integrity by rejecting erroneous inputs. The retrieval of registered locations (TC9) verifies that users can access and manage their geospatial data effectively.

7.3 Geothreads

The ability to create threads for specific locations (TC11) and retrieve these threads (TC12) confirms that users can initiate and access discussions seamlessly. Additional tests for creating comments on threads (TC13) and retrieving these comments (TC14) validate that the system supports active and ongoing user engagement.

7.4 Basic Filtering Options

The GeoRecord system's basic filtering functionality has been validated through a test that queries locations within a specified bounding box (TC10). This test confirms that users can filter and retrieve geospatial data based on defined criteria, ensuring that the system can handle basic search and sorting requirements.

The conducted test cases comprehensively address the primary functional requirements of GeoRecord. Secure user authentication and profile management are validated through rigorous testing. Drawing and editing tools are supported by related location management functionalities. Geothreads successfully facilitates user interactions and discussions. Basic filtering options are proven to work effectively, allowing for efficient data querying.

8 Reflection

8.1 Team Processes

At the start of our project, we lacked a solid plan for our development approach, which led to increased workloads later in the assignment. We could have considered using different agile frameworks to support our project management. This would have enabled everyone to understand clear goals for the project as well as staying on top of our direction. Utilising GitHub's issue tracker more effectively could have facilitated better tracking of bugs and task statuses, also assisting the project in the same way that agile would have.

The development process was stressful due to inadequate team management and an uneven distribution of work. There was little consistency in code quality and a poor understanding of the software architecture, leading to time-consuming bug-fixing and high communication overhead. Improving team management practices, such as adopting a layered architecture to reduce complexities, could enhance productivity and cohesion in future projects. Keeping more detailed meeting minutes and maintaining permanent records of discussions would have made it easier to catch up absent members and address recurring issues promptly through the project development.

The absence of detailed documentation for deploying the app locally caused significant challenges for some team members, emphasising the importance of clear guidelines from the start. This makes us question how people that know nothing about our project would understand the deployment of the application. This experience underlined the need for comprehensive documentation, which ensures that all potential users, including our team, can set up and understand the development environment efficiently.

Nonetheless, communication still played a vital role in our team's success. Regular discussions ensured alignment on the project's requirements and functionality. This continuous dialogue was crucial in overcoming the challenges posed by varying levels of familiarity with new technologies among team members, helping to distribute workloads more evenly and efficiently.

8.2 Project Operation

Implementing microservices allowed us to achieve horizontal scalability, enabling team members to work on different components simultaneously without causing merge conflicts. Early decisions about the database specifics and architecture ensured a smoother backend implementation and saved time in refactoring. However, better API documentation, including expected response formats and call requirements, is crucial. This would aid in the integration of new features and microservices, providing clarity and extensibility in development.

Our biggest early challenge was the need for detailed API specifications, which led to confusion and inefficiencies. Having a pre-populated, centralised development database or scripts to populate local databases with dummy data would have simplified testing and ensured consistency across development environments. We also faced issues with rendering polygons on the map, highlighting the need for more sophisticated rendering logic to improve performance.

The architecture could benefit from the introduction of further performance improvements to improve its scalability and responsiveness. One of the most computationally expensive operations for the database and API was querying which locations were within the bounding box of the user's view. One idea that was discussed early on in the project to optimise this was the use of a fast in-memory key-value store to reduce the location query times and improve responsiveness. Both Redis and AWS Elasticache support geospatial queries, meaning that it could be possible to store a copy of all locations in one of these databases to alleviate the load on the RDS instance.

Another potential enhancement for the project would be globally distributing the microservices and implementing a caching infrastructure. This approach could significantly reduce location query latencies by minimising the distances that API requests need to travel. To achieve this, we could use the location cache previously described and leverage AWS Elasticache's cross-region replication. Configuring CloudFront to route to the AWS region with the best latency for the user would further optimise performance. Implementing these measures would not only improve responsiveness but also ensure a smoother user experience by delivering faster query results. However, it is important to note that such optimisations are currently unachievable due to the data centre restrictions placed on the UQ AWS accounts.

8.3 Future Directions

In future projects, following test-driven development would ensure that all requirements are met before code merges into the main branch, reducing the need for extensive code reviews and minimising integration issues. Using agile frameworks to manage tasks would also improve team communication and task tracking. Balancing the adoption of familiar and new technologies could potentially lead to faster development and better resource allocation, allowing more time to refine the end product. This would be a consideration to weigh up before project development and requires a deeper understanding of everyones current skillset, ability to learn, and ability to adapt to emerging software.

Working on user experiences early on is crucial as it guides the design of our APIs. This approach makes sure that the API and front-end development complement each other, making collaboration easier and more effective. Creating early mockups or prototypes can help align all team members understanding of how different components interact, leading to a more cohesive and efficient development process. Understanding user experience from the start of the project ensures that our team takes a human approach to designing the software with the goal of creating for the end users at the forefront of our architecture.

From a technical perspective, further performance improvements are needed to enhance the architecture's scalability and responsiveness. One major issue was the computational expense of querying locations within the user's view. Optimising this with a fast in-memory key-value store like Redis or AWS Elasticache could alleviate the load on the RDS instance. Exploring global distribution of microservices and caching infrastructure could also minimise latencies and improve user experience by reducing the distance API requests travel. These advanced techniques, although beyond our current scope, represent exciting possibilities for future projects.