# Honey, I shrunk the GIS: Developing scalable and lightweight geospatial software applications with microservices and containerization

Arne Schumacher

German Federal Office for Radiation Protection (BfS), Köpenicker Allee 120, 10318 Berlin - aschumacher@bfs.de

**Keywords:** geospatial software development, microservices, containerization

**Abstract**

This paper presents a containerized and modular approach to developing a geospatial application using Docker and geospatial open source software. The system is designed to ingest, store, manage and visualize radiological measurement data within a portable and scalable framework. Core technologies include PostgreSQL with PostGIS for spatial data storage, GeoServer for OGC-compliant map services, Node.js for package dependencies and OpenLayers for web-based visualization. Each component is isolated in its own Docker container, promoting ease of deployment and maintainability. The architecture demonstrates an effective and transferable solution that can be applied to other geospatial software projects, particularly those seeking to implement a similar setup or initiate the development of containerized applications. This architecture not only simplifies development and deployment but also provides a robust foundation for building more complex geospatial systems that support advanced spatial analytics and statistics or elaborate data pipelines.

## 1. Introduction

### 1.1 Containerization and microservices

Traditional GIS systems are generally monolithic in nature, requiring the installation of bulky desktop applications. Both, proprietary and open source GIS typically follow a 'one size fits all' philosophy despite the fact that in many cases geodata is thematic data with coordinates plotted on a map. Historically, GIS have emerged as full-fledged applications that manage the key components of a GIS - by definition - capturing, storing, analyzing and managing of geodata (IBM, 2025). The monolithic nature of such development presents numerous pitfalls. Monolithic refers to the concept of coupling and a high degree of dependence between the different modules within an application. This tight coupling in software development results in several disadvantages:

- **Reduced flexibility** - need to modify multiple modules when changes occur.
- **Impeded maintainability** - changes in source code may have cascading effects.
- **Limited reusability** - transferring a system to another context can be difficult or impossible.
- **Higher risk of failure** - intertwined systems have a cascading risk of failure.
- **Complex debugging** - identifying the root cause is more difficult.
- **Decreased efficiency** - more resources, time and personnel are required throughout the software development lifecycle.

In contrast to the monolithic approach - in which the software is a single code base, inseparable and components are tightly integrated - modern software development emphasizes modularization and code separation. Decoupling addresses many of the issues mentioned above. Components communicate via well-defined services and interfaces, such as an Application Programming Interface (API), which outlines protocols and communication rules. This encapsulation enables components to evolve independently. For instance, a database management system can completely rewrite its internal data handling while maintaining compatibility through its API - for example, via a Web Feature Service delivering GeoJSON.

**1.1.1 What is the difference between containerization and microservices?** The distinction between the terms containerization and a microservice architecture (MSA) lies in their respective roles within software design and development. Containerization refers to a deployment technique that encapsulates an application module and its dependencies into a container, thereby enhancing portability and scalability. The emphasis is on container images that define and build the running instances. In contrast, a microservice represents an architectural pattern in which each service performs a specific function. These services operate collaboratively within a distributed, service-oriented architecture. Microservices are independently deployable, loosely coupled, and technologically agnostic. (Newman, 2019).

**1.1.2 Advantages of modularization** Both concepts aim to isolate functionality and minimize unintended interactions. "A microservice architecture decomposes a business domain into small, consistently bounded contexts implemented by autonomous, self-contained, loosely coupled, and independently deployable services" (Blinowski et al., 2022). Service orchestration is encapsulated and communication between them is organized through standardized protocols and APIs. As in the physical world and as the word suggests, containers provide the flexibility of lightweight, portable and interchangeable modules that can be used across different environments and between each other. This approach supports a wide range of use cases, such as deploying software, running development or production stacks and much more. Key benefits include:

- **Portability** - containers encapsulate all required libraries, dependencies, software and configurations.
- **Efficiency** - containers use fewer resources than VMs or server infrastructures. They also have a positive effect on

human resources because they lessen dependencies between developers.
- **Scalability** - supports both horizontal and vertical scaling.
- **Isolation** - containers are self-contained and reduce the risk of conflicts.
- **Continuous integration** - testing and production environments can be identical, accelerating development cycles.
- **Clear ownership** - containers have well-defined boundaries, which prevents individuals from interfering with each other's work and thereby improving efficiency.
- **Interoperability** - containers and services can easily be consumed with no special needs in terms of operating system or software.

For these reasons, they are well suited for cloud infrastructures. "Cloud computing is inherently rooted on virtualization technologies" (Combe et al., 2016). Containers integrate tightly with the host system, minimizing the software overhead typical of virtual machines or physical servers. Virtualization and to a larger degree containerization enable more precise hardware allocation. Containers offer a self-sufficient ecosystem including components such as operating system, software and dependencies, often deployable with minimal effort.

Containerization is also considered to be very fast. While images may take time to pull and assemble all layers - starting, stopping, pausing or removing Docker containers usually occurs in seconds. Both processes, building images and running the containers require little to no human interference and can be highly automated.

The acceleration of development cycles and the concept of the 12 Factor App outline best practices for cloud-native application development, focusing on availability, portability and maintainability. For a program "in the 1960s and 1970s, it was enough [...] to run" (Lerner, 2014). Today, applications must be scalable, maintainable, portable and highly available. Docker Hub and similar registries facilitate sharing these applications. Containerization and microservices are part of an "architectural style as a way of working with systems in which scale is a factor" (Nadareishvili et al., 2016). The spread of container technologies is a response to the growth pressure big tech companies faced when the volume of activity outgrew the capacity of traditional technology choices. The growing popularity of containerization date back to 2013 when Docker was founded and to 2011 when the term microservice was coined and the 12 Factor App was introduced (Blinowski et al., 2022).

**1.1.3 The 12 Factor App Methodology** The Twelve-Factor App pursues similar goals but from a slightly different perspective. It emerged from the challenge developers faced when attempting to deploy locally tested applications into production environments. The primary objective was to minimize divergence between development and production. To address this, a set of twelve principles was established to promote portability, maintainability, scalability, and resource efficiency. This methodology has had a significant influence on the design of cloud-native, containerized, and microservice-based infrastructures. Originating in the context of software-as-a-service (SaaS) applications (Wikipedia, 2025), it continues to have widespread implications for modern software architecture. The core principles include the following:

- **I Codebase** - keep only one codebase in a tracked version control system.
- **II Dependencies** - all dependencies should be declared and isolated.
- **III Config** - store config in environment variables.
- **IV Backing Services** - treat backing services as attached resources.
- **V Build, release, run** - the delivery pipeline should strictly consist of build, release, run.
- **VI Processes** - execute the app as one or more stateless processes.
- **VII Port binding** - expose services by specified ports.
- **VIII Concurrency** - scale out via the process model.
- **IX Disposability** - fast startup and graceful shutdown.
- **X Dev/Prod parity** - all environments should be as similar as possible.
- **XI Logs** - treat logs as event streams.
- **XII Admin Processes** - run admin/management tasks as one-off processes.

Particularly, "I Codebase", "II Dependencies", "III Config", "VII Port binding" and "VIII Concurrency" had a significant influence on this project, as will be discussed in the methodological chapter.

**1.1.4 Limitations of containerization and microservice architectures** In spite of these advantages, containerization is not always the optimal choice. Determining the most suitable software development approach requires consideration of factors such as the application's size, complexity, and purpose. Containerization may not be the best fit under the following conditions:

- **Size** - For small applications with a limited number of users, modularization may complicate the development unnecessarily.
- **Simplicity** - Containerization adds complexity and tools and techniques must be learned by developers. It is also harder to understand the software holistically.
- **Performance and latency** - In-process calls are generally faster than over-the-network communication.
- **Security** - Vulnerabilities in the container runtime or misconfigurations can lead to security issues; a monolithic application can be secured more easily.
- **Debugging** - More complex to debug and find errors. Monolithic software have simpler workflows, are easier to monitor, log and troubleshoot.
- **Resource Overhead** - Although lightweight, each container adds CPU and memory overhead.

The decision regarding which architectural paradigm to adopt remains a complex one and is rarely clear-cut. However, as software projects grow in size and complexity, the advantages of containerization and microservices generally outweigh their disadvantages. The separation of modules enables greater flexibility in updating, relocating, and replacing code and data, thereby simplifying the development of adaptable or "liquid" software. (Kalske et al., 2017). Additionally, the good integration and acceptance in cloud-based environments have convinced big tech companies such as Amazon, eBay, IBM, Zalando, Spotify, Uber, Airbnb, LinkedIn, Twitter, Groupon and Coca-Cola to adopt this strategy (Su et al., 2024). The flexibility and agility offered by containerization and microservices increase service innovation and enable businesses to respond more effectively to changing environments and trends (Hasan et al., 2023).

### 1.2 Containerization and OGC-compliant services

Containerization and OGC-compliant services complement each other well. With the advent of the internet and the growing volume of geospatial data from sensors and devices, distributing geodata beyond traditional GIS platforms has become increasingly important. To encourage a wider dissemination of geodata, the OGC has developed interoperable standards. As server-client architectures gained prominence, standards like the Web Map Service (WMS) and the Web Feature Service (WFS) became increasingly popular. The exchange of geodata on the internet is grounded in standards that propagate the distribution of data according to the FAIR principle; findable, accessible, interoperable and reusable. Acceptance of these principles depends on well-defined and harmonized standards that operate independently of operating systems and proprietary software.

Particularly the more modern standards - e.g. OGC API - Features or OGC API - Maps - provide RESTful alternatives that neatly integrate with microservice and cloud-native design patterns. The standards facilitate the scalability of geospatial software that can be deployed in web-based and/or mobile applications. Just like traditional OGC services their aim is to encourage and facilitate the use of geospatial vector and raster data.

The following chapter outlines the methodological approach of the containerization project presented for this case study - more specifically, the visualization of radiological data on a client-side web map. The case study is fully based on open source software including Docker, PostgreSQL, GeoServer, OpenLayers and Node.js.

## 2. Methodology

The objective of this project is to develop a containerized geospatial application that follows a modular approach while meeting the core GIS requirements of data ingestion, storage, management, and visualization. The thematic content is radiological sensor information gathered by German and European authorities. The application should be open-source, portable and deployable with minimal effort. Each component is encapsulated in a separate Docker container and communicates with others via standardized interfaces.

### 2.1 Project essentials

Docker and Podman are two well-known containerization tools. This project uses Docker to containerize the application. Docker enables a Platform-as-a-Service (PaaS) model, aligning with various XaaS paradigms (where X may represent Infrastructure, Software, Data, AI, etc.). It can encapsulate all components of an IT ecosystem, including the operating system, software, data, configurations, libraries, frameworks and dependencies.

The repository is published at:
`https://github.com/arneschum/bfsrad.git`
and provides a configuration that pulls the following components as an image and later as a running service (container). The project implements a 4-tier container orchestration to visualize the geodata:

1. PostgreSQL
2. pgAdmin
3. GeoServer
4. Node.js

The key components include the following images and containers:

**Container 1: PostgreSQL with PostGIS** PostgreSQL and PostGIS are the backend that manage the geodata. It contains a spatially enabled PostgreSQL database (PostGIS). If need be the size of the images can be significantly reduced by serving flat files through e.g. a GeoPackage instead of an entire database management system and its graphical user interface (GUI) pgAdmin. However, due to the many advantages of a DBMS, i.e. multi-user editing and user and role management, transaction control, data consistency and data integrity, backup and recovery etc., a DBMS outperforms any file-based approach.

The database holds the content of the application in form of a denormalized table “measdata”. This table is the source for GeoServer’s Web Map Service.

**Container 2: pgAdmin** pgAdmin is the GUI for PostgreSQL data management. pgAdmin is not necessary but a complementary tool to access data in a convenient way. The DBMS can also be fully accessed by the `psql` command line tool without the need for a GUI.

**Container 3: GeoServer** The third component is the web map server GeoServer. GeoServer serves spatial data using OGC standards. It connects to the PostgreSQL/PostGIS database and publishes the table containing the sensor data. GeoServer is responsible for delivering the data as map layers that can be consumed by external clients.

The Web Map Service is the OGC standard used in this project. It is one the most well-established OGC standards and serves server-side, georeferenced map images (e.g. png, jpg or giff) over the internet. It does not, per se, serve raw data or geospatial datasets - for that, the Web Feature Service (WFS) is used for vector data and the Web Coverage Service (WCS) for raster data. The only additional information that can be queried beyond the thematic map image itself is through the `GetFeatureInfo` request, which returns information about the selected feature at the clicked location.

Every WMS layer is associated with a style that defines how the vector objects - points, lines and polygons - are rendered on the map. In this project a XML-based Styled Layer Descriptor (SLD) file named "measdata.sld" is used to style objects based on two attributes; shape and size. SLD is the only fully native styling option in GeoServer, other options like CSS, YSLD (YAML-based) and MBStyle (JSON-based) can be installed as extensions. To use these alternatives, the corresponding *.jar files must be placed in the appropriate folder, for this project this is: `./geoserver-init/styles` (Figure 1).

**Container 4: Client application** The client webpage includes a basic triplet of HTML, CSS & JS code. Additionally, OpenLayers to visualize the data and node.js for package dependencies is included. Node.js is a convenient way to fulfill the needs of the client application such as OpenLayers. Vite serves as the local development and testing server.

The images used here include configuration details about software components as well as the application’s source code. As such, containerization merges the roles of the software developer - responsible for writing the application and defining its dependencies - and the system administrator - responsible for system configuration and resource deployment (Nickoloff and Kuenzli, 2019). While this merging of roles may seem at odds with the previously mentioned advantages and separation of logic, it is necessary in this context to build containers efficiently and to produce consistent, reproducible images for cross-environment use.

### 2.2 Containerization for fault tolerance, resilience and high availability

Reproducible images are very important for failover or switchover mechanisms on georedundant (geographically redundant) servers. Each server can be a physical machine, a virtual instance (VM) or a container. For the first two options, automation tools like Ansible or Puppet can streamline this setup. Such approaches, however, are generally more inefficient and resource-intensive than containerization. "Unlike traditional virtualization or paravirtualization technologies, [containers] do not require an emulation layer or a hypervisor layer to run and instead use the operating system's normal system call interface. This reduces the overhead required to run containers and can allow a greater density of containers to run on a host" (Turnbull, 2014). Similarly, Jaramillo et al. argue that Docker containers enable "the sharing on operating system and supporting libraries, which is more lightweight, prompt and scalable than Hypervisor based virtualization. These features make it ideally suited for applications deployed in microservice architecture" (Jaramillo et al., 2016).

### 2.3 Pulling images and starting containers

Docker pulls images - like the four listed above - from its registry. They are the backbone of every container. It gathers all necessary instructions to consistently build the same image across different environments. For single containers this is usually specified in the Dockerfile with the keyword `FROM`. For multiple and orchestrated containers and to serve a full stack software application this is typically achieved in the docker-compose.yml with the keyword `image`. These files provide step-by-step instructions for assembling the necessary image(s) including configuration details.

These plain text files require minimal storage and can be version-controlled in a Git repository (see Section 1.1.3) containing all the information needed to reproduce exact replicas using a single docker command `docker compose up --build` (see below) or two commands at most `docker compose build` and `docker compose run`. The build command assembles images; if no elaborate customization is required and the image is available on the docker hub, it can also be retrieved using the `docker compose pull` command. Essentially, build or pull retrieves the image and run creates the running instance (i.e. execution) of the image. A built image is a read-only object and thus immutable. Containers, in contrast, are writable running processes based on these images.

### 2.4 Persisting data in a container

A key concern of containers is the persistence of data. "Data written to the container layer doesn't persist when the container is destroyed. This means it can be difficult to get the data out of the container if another process needs it [...] You can't easily extract the data from the writeable layer to the host, or to another container" (Docker Homepage, 2025). There are several reasons why data exchange between the host and a container is necessary, especially for reading logfiles or writing (e.g. configuration changes, dynamic content or to update datasets). Docker supports shared storage that allows both the host and container to access data in real time.

Two key concepts enable this:

- **Volumes** are persistent storage resources that remain intact even after containers are removed. They are managed by the Docker daemon and are ideal for persistent storage. Since they are more isolated, they are considered safer than bind mounts.
- **Bind mounts** link a directory from the host machine to the container. These are not managed by Docker and are well-suited for real-time data access from the host.

Volumes play a crucial role in the configuration of open-source software and the exchange of data. The remainder of this chapter introduces the volumes used in this project to support the creation of reproducible containers.

### 2.5 Project volume details

Figure 1 illustrates the directory structure of the project and highlights the core components of the repository. Data ingestion into the DBMS is handled through the `./db-init` folder. GeoServer objects such as the workspace, data store, layer, and style - will be created in the `./geoserver-init` folder. The client visualization and web interface are contained within the `./app` folder.

```
|-- app
|   |-- index.html
|   |-- main.js
|   |-- css.css
|   |-- package.json
|-- db-init
|   |-- init.sql
|   |-- measdata.csv
|-- geoserver-init
|   |-- create_geoserver_objects.sh
|   |-- setup.sh
|   |-- styles
|   |-- |-- measdata.sld
|-- .env.example
|-- postgres_data
|-- docker-compose.yml
```

Figure 1. Directory structure of project (condensed for illustration)

The project uses Docker volumes for PostgreSQL, GeoServer and node.js. For simplified configuration and for data loading, PostgreSQL and GeoServer are connected to multiple volumes each. More specifically the setup includes the following volumes:

`./postgres_data:/var/lib/postgresql/data/`

The postgresql data directory contains important configuration details related to security (pg_hba.conf) and hardware allocation and administration (postgresql.conf). This folder needs to be empty when the image is first built!

`./db-init:/docker-entrypoint-initdb.`

The docker-entrypoint-initdb.d directory (inside the container) contains an init.sql file that holds all the SQL commands necessary to initialize the database. It includes both Data Definition Language (DDL) and Data Manipulation Language (DML) commands. This script creates the database, builds the data model, and populates the tables. A denormalized GeoServer-compatible view called measdata is included, containing a geometry column named geom as a spatial data type. The data is loaded from a comma-separated-value (CSV) file named measdata.csv. As a volume, this folder is accessible both inside and outside the container, enabling quick updates of the data upon restart.

The following steps are executed when the container is started:

1. **(Re)create the Database**
   `DROP DATABASE IF EXISTS bfsrad;`
   `CREATE DATABASE bfsrad;`
2. **Connect to the Database**
   `\connect bfsrad`
3. **(Re)create the table**
   `DROP TABLE IF EXISTS measdata;`
   `CREATE TABLE measdata (...);`
4. **Insert Data**
   `COPY measdata FROM 'measdata.csv' CSV HEADER`

`./geoserver-init:/init`

Similar to PostgreSQL's init.sql, the geoserver-init folder includes a shell script (create_geoserver_objects.sh) that automates the creation of:

1. The workspace: groups the objects in a namespace
2. The data store: the connection to the database
3. The style - the folder 'styles' can be used to load any number of styles which need to be added to GeoServer.

Automation of GeoServer objects can be achieved through:

- The GeoServer REST API, or
- Direct configuration of the GeoServer data directory

In this project, the REST API is accessed via a shell script that is executed when the container starts. All database tables must exist before GeoServer may setup its own objects. It must be able to access the PostgreSQL tables during startup; otherwise the logs will show errors and GeoServer will fail to start. The docker-compose.yml ensures PostgreSQL is running before starting GeoServer through the keyword:

```
geoserver:
  ..
  depends_on:
    - postgres
```

Docker can complete this setup within seconds, as the configuration resides in the container (not the image), enabling fast container initialization. Testing has shown that the containers start significantly faster than GeoServer itself, which undergoes various setup processes. For this reason, the script setup.sh (Figure 2) serves the purpose to wait until GeoServer is ready before it sets up the objects.

```
#!/bin/bash
catalina.sh run &
PID=$!

echo "Waiting for GeoServer to start..."
until curl -s -u $GEOSERVER_USER:$GEOSERVER_PASS \
http://localhost:8080/geoserver/rest/about/version.xml\
> /dev/null; do
  sleep 5
done

echo "GeoServer is ready. Configuring..."
bash /init/create_geoserver_objects.sh
wait $PID
```

Figure 2. Shellscript waiting for GeoServer to be ready

Once GeoServer is ready, the required objects can be initialized. Figure 3 illustrates the creation of the workspace and the data store. The figure has been shortened for illustration purposes.

```
# 1. Create Workspace
curl -u $GEOSERVER_USER:$GEOSERVER_PASS -XPOST -H \
"Content-type: text/xml" \
-d "<workspace><name>${WORKSPACE}</name></workspace>"\
"${GEOSERVER_URL}/rest/workspaces"

# 2. Create PostGIS Store
curl -u $GEOSERVER_USER:$GEOSERVER_PASS -XPOST -H \
"Content-type: text/xml" -d "<dataStore>
      <name>${STORE}</name>
      <connectionParameters>
        <entry key=\"host\">${PG_HOST}</entry>
        <entry key=\"port\">${PG_PORT}</entry>
        <entry key=\"database\">${PG_DB}</entry>
        <entry key=\"user\">${PG_USER}</entry>
        <entry key=\"passwd\">${PG_PASSWORD}</entry>
        <entry key=\"dbtype\">postgis</entry>
      </connectionParameters>
    </dataStore>" \
"${GEOSERVER_URL}/.../.../${WORKSPACE}/datastores"

# 3. Publish Layer (table must exist in PostGIS)
curl -u ...
# 4. Publish style (found in folder ./styles)
curl -u ...
# 5. Associate the style with the layer
curl -u ...
```

Figure 3. Shellscript to create GeoServer objects (shortened for illustration)

`./app and node.js`

Node modules and client information are mounted in the app folder - both on the host and in the container (./app). This folder (commonly also referred to as "src" in Node.js projects) hosts the client application and contains its source code, including the main webpage. Typically, index.html serves as the application's entry point. It is a simple webpage that loads and displays the Web Map Service via OpenLayers using `ol.TileWMS`.

During the build process, numerous libraries (including OpenLayers) are downloaded, particularly in the node_modules folder, which includes all required dependencies. Similarly, the PostgreSQL image downloads various tools and configuration files during its initial build. However, these files should not be included in the Git repository, as doing so would unnecessarily increase its size and complicate dependency management. Consequently, these files and folders are listed in the `.gitignore` file. As the name implies, this file ensures that the specified files and directories are excluded from the Git staging area and the main repository.

The core repository (Figure 1) only contains lightweight text configuration files that define:

- **Image building and service configuration:** specifies which images to pull (services), which ports to expose, and the volumes each service connects to.
- **Login credentials:** sensitive credentials are passed into the docker-compose.yml via environment variables. These should never be stored directly in the repository - even if the repository is private. Instead, an .env.example file is included as a template. Users should copy and rename this file to .env and customize it for their environment. This .env file is the only file that should differ between individual clones of the repository.
- **Client application:** contains all the web page source code, libraries and node module definitions for the client interface.

To build the images and start the services in a single command, the command `docker-compose up` with the `--build` can be used:

```
docker compose up -d --build --force-recreate
```

The command pulls all the required images:

```
docker image ls
```

```
REPOSITORY         TAG      IMAGE         ... SIZE
node               latest   cd86d0acabd6 ... 1.11GB
postgis/postgis    latest   9acc3941fc21 ... 609MB
dpage/pgadmin4     latest   73ce6afcb76d ... 487MB
osgeo/geoserver    latest   b070c291a0d7 ... 685MB
```

Figure 4. List of Docker images

and starts the containers as services:

```
docker ps
```

```
ID      IMAGE           ... PORTS              NAMES
fd87... node:latest     ... *:3001->5173/tcp   nodejs
cc20... osgeo/geoserver... *:8904->8080/tcp    geoserver
83e5... dpage/pgadmin4  ... *:5050->80/tcp     pgadmin
84b8... postgis/postgis... *:5432->5432/tcp    postgres
```

Figure 5. List of Docker containers: * represents `0.0.0.0`

All client applications are now accessible at the following URLs:

```
PostgreSQL: psql -h localhost postgres postgres
GeoServer: http://localhost:8904/geoserver/web/
Webpage: http://localhost:3001/
```

Each component runs in its own image and container, promoting a service-oriented architecture. It is generally considered best practice to separate applications into independent processes, as demonstrated here, to enable a modular and interconnected service infrastructure. This modularization helps to untangle complex monolithic architectures and facilitates clean, efficient communication through APIs, well-defined addresses, and exposed ports.

This streamlined deployment model makes containerization particularly well-suited for horizontal scaling. In combination with orchestration tools like Docker Swarm, Kubernetes, OpenShift and load balancers such as HAProxy or Traefik, the infrastructure can be optimized for high availability and large-scale deployment. Containers can be replicated in a n:m relationship between components like the DBMS and GeoServer, allowing scalability based on traffic demands.

Furthermore, the project's structure and its composition with Docker is currently tailored for a development environment. There are several options, some of which were mentioned earlier, to significantly downscale the system for production use:

1. The PostgreSQL GUI pgAdmin can be completely omitted.
2. The DBMS can be reduced to a more compact database such as SQLLite or GeoPackage.
3. Node.js and Vite, used as development tools, consume significant resources and are not required in the final production setup.

## 3. Results and discussion

### 3.1 Real-World Application: Radiological Emergency Preparedness

The application showcases how radiological measurement data can be deployed for various tasks from routine monitoring to emergency response. The Federal German Office for Radiation Protection (Bundesamt für Strahlenschutz) is responsible for detecting, assessing, and reacting to nuclear and radiological events. It collaborates with European member states and international agencies (e.g., Euratom, IAEA). In an emergency, it must rapidly collect, analyze, and distribute information while proposing protective measures to reduce the impact of nuclear fallout.

Scalability is a key concern during emergencies, as public interest can surge dramatically, leading to heavy traffic on web platforms. Both internal and external access to information must remain stable and fast. For nearly a decade, the agency has emphasized component-based software development and containerization, thereby reducing the overhead of system administration and increasing fault tolerance.

Containerization has improved continuous integration and deployment (CI/CD), scalability, maintainability, portability, resilience (e.g. recovery of services) and performance of geospatial applications. Data updates can be deployed in seconds, and the application can be scaled across multiple servers to meet increased demand.

### 3.2 Discussion

The current implementation only plots data in OpenLayers. However, the client now has access to a powerful DBMS, a web map server, and client-side tools such as node modules, JavaScript, and OpenLayers. This provides extensive opportunities to develop the client further. OpenLayers is a comprehensive library that supports transactional editing, visualization, and analysis of geospatial data. When used in conjunction with additional frameworks or libraries, it can serve as a solid foundation for developing a rich client application.

Figure 6 shows the radiological data plotted on the map in the border triangle of Switzerland, France and Germany. The project visualizes a single layer containing approximately 5,100 measurement points (as of 9 May 2025) from the Integrated Measurement and Information System for the Surveillance of Environmental Radioactivity (IMIS), hosted by the BfS. In Figure 6 these are visualized as points. Additionally, data reported by European partner organizations are displayed as squares. The number of layers, data sources and features can be extended at any time. This project is intended to serve as an introductory example for individuals interested in developing containerized software applications.

The application presented here is a deliberately simplified example, designed to reduce a geospatial application to its essential components. From this foundation, it can be extended into more complex web-based systems. Its primary goal is to demonstrate the potential for rapid deployment across multiple locations to ensure availability, resilience and fault tolerance. Currently, the BfS operates three georedundant servers located in Berlin, Munich, and Freiburg. These servers run a multi-component software system used to monitor environmental radioactivity.

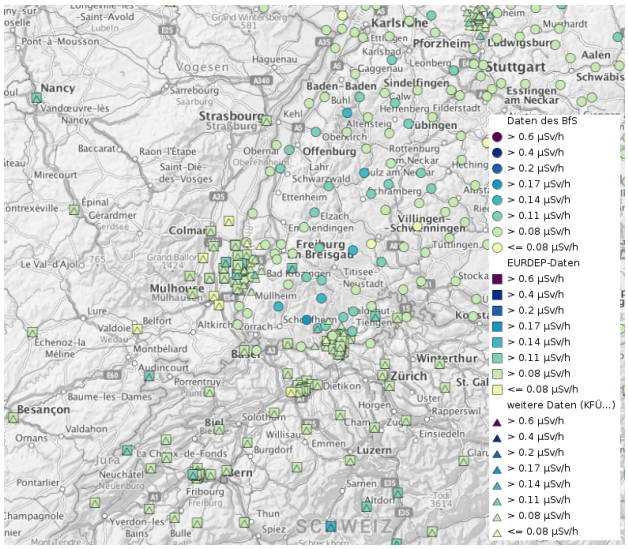


Figure 6. Visualization of radiological data

## 4. Conclusions

This case study demonstrated the benefits of adopting a containerized, service-oriented architecture for geospatial applications. By isolating each functional component - data ingestion, storage, service and visualization into independent Docker containers, the system achieves modularity, scalability, and ease of maintenance. The use of open-source tools like PostgreSQL/PostGIS, GeoServer, and OpenLayers ensures adaptability to a wide range of GIS use cases. This architecture not only simplifies development and deployment but also provides a robust foundation for building more complex geospatial systems that support advanced spatial analytics and statistics or elaborate data pipelines.

## References

Blinowski, G., Ojdowska, A., Przybyłek, A., 2022. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE access*, 10, 20357–20374.

Combe, T., Martin, A., Di Pietro, R., 2016. To docker or not to docker: A security perspective. *IEEE Cloud Computing*, 3(5), 54–62.

Docker Homepage, 2025. Docker engine – storage. `https://docs.docker.com/engine/storage/`.

Hasan, M. H., Osman, M. H., Novia, I. A., Muhammad, M. S., 2023. From monolith to microservice: measuring architecture maintainability. *International Journal of Advanced Computer Science and Applications*, 14(5).

IBM, 2025. Docker engine – storage. `https://www.ibm.com/think/topics/geographic-information-system`.

Jaramillo, D., Nguyen, D. V., Smart, R., 2016. Leveraging microservices architecture by using docker technology. *SoutheastCon 2016*, IEEE, 1–5.

Kalske, M., Mäkitalo, N., Mikkonen, T., 2017. Challenges when moving from monolith to microservice architecture. *International Conference on Web Engineering*, Springer, 32–47.

Lerner, R. M., 2014. At the forge: 12-factor apps. *Linux Journal*, 2014(245), 5.

Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M., 2016. *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.".

Newman, S., 2019. *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media.

Nickoloff, J., Kuenzli, S., 2019. *Docker in action*. Simon and Schuster.

Su, R., Li, X., Taibi, D., 2024. From Microservice to Monolith: A Multivocal Literature Review. *Electronics*, 13(8), 1452.

Turnbull, J., 2014. *The Docker Book: Containerization is the new virtualization*. James Turnbull.

Wikipedia, 2025. Twelve-factor app methodology. [Online; accessed 14-May-2025].