Modernizing geospatial services: an investigation into modern OGC API implementation and comparative analysis with traditional standards in a web application

Sudipta Chowdhury[1,2], Dietrich Schröder[2], Hamidreza Ostadabbas[1], Mohammad Hosseingholizadeh[1], Frank Friesecke[1]

[1] die STEG Stadtentwicklung GmbH, Germany - (hamidreza.ostadabbas, mohammad.hosseingholizadeh, frank.friesecke)@steg.de
[2] Stuttgart University of Applied Sciences, Germany - sudipta.urp10@gmail.com, dietrich.schroeder@hft-stuttgart.de

Keywords: Modernizing Geospatial Services, OGC API, GeoServer, pygeoapi, Web Application Architecture, ALKIS

Abstract:

The study explores the transition from traditional geospatial service standards to modern Open Geospatial Consortium (OGC) API standards in web applications, focusing on urban development management. The main goal is to compare the performance and practical implications of integrating modern and traditional geospatial technologies. Two prototype system architectures were formulated based on the underlying principle of three-tier architectures. Database operations were facilitated by PostgreSQL (PostGIS), while server-side functionalities employed GeoServer and pygeoapi for data publication and OpenLayers served as the frontend for data visualization. The primary data source for this study is ALKIS (Authoritative Real Estate Cadastre Information System of Germany). The investigation encompasses two principal facets: a theoretical evaluation of two distinct server implementations utilizing conventional standards (GeoServer) and contemporary standards (pygeoapi), alongside a practical testing phase. Theoretical comparisons underscore GeoServer's robustness, well-established user base, and comprehensive feature set, along with its highly efficient folder structure and detailed, user-friendly documentation. In contrast, pygeoapi is characterized by its emphasis on simplicity and utilization of modern technologies such as OpenAPI for implementing a RESTful API. During hands-on testing, it was observed that pygeoapi consistently exhibited longer rendering times than GeoServer. Moreover, as the feature count increased, both platforms showed a linear escalation in rendering times. To address prolonged rendering times in pygeoapi, incorporating vector tiles led to a significant reduction in rendering times. Regarding the affect of different data format, PostgreSQL (PostGIS) consistently outperforms other data formats used in pygeoapi, while Shapefile and PostgreSQL (PostGIS) perform well in GeoServer. This research aims to effectively integrate geospatial technologies, bridging the gap between established standards and emerging APIs in web applications.

1. Introduction

What if we could imagine an urban landscape where traffic flows seamlessly, energy consumption is optimized for efficiency and resources are allocated perfectly? With the ongoing expansion of cities, their challenges are also increasing. Geographic Information System (GIS) is considered as a potent technology which makes it possible for better understanding the territories (cities and rural) and manage them in an integrated and efficient way (Gonçalves and Virtudes, 2020). However, as a result of the advancement of information technologies and the rise in web usage in general, WebGIS apps play a significant role in urban management information system for the spatial planning entities (Gonçalves and Virtudes, 2020). Geospatial data visualization is a fundamental feature of GIS-based web applications, enabling the presentation of information about various spatial features within urban areas or any geographical region. A primary objective of the web application is to enable urban planners or architects to become more independent in terms of spatial data usage without extensive GIS experience. However, the effectiveness of WebGIS is highly dependent on the underlying Geo-services, which requires a thorough examination of traditional and modern approaches to meet the increasing demands of urban planning applications. In spite of these advantages, WebGIS still faces a number of challenges, particularly interoperability issues pertaining to heterogeneous data sources (Rowland et al., 2020). To address interoperability gaps, the Open Geospatial Consortium (OGC) has introduced standards.

Since its inception at 1994, OGC has become an integral part of the world's information infrastructure by providing standards i.e., Open Web Standards (OWS) that enable developers to create quickly and easily exchangeable information systems (OGC-Standards, 2023). Until recently, OWS was comprised of three popular standards: Web Map Service (WMS), Web Feature Service (WFS), Web Coverage Service (WCS) (Assefa, 2018). Recently, OGC has started working on developing a new generation of standards called OGC API, which is the new reinvention of how geospatial/location information is shared, accessed, integrated, and analyzed (OGC-Context, 2023). The idea behind this OGC API is to make things simpler for anyone to integrate and use geographical data to the web, as well as to combine this data with any other kind of information by following the approach of OpenAPI (OGC-Context, 2023). This new standards are considered as a resource centric API which is the combination of traditional standards (WxS) and modern web development practices (OGC-Context, 2023). Therefore, the central problem addressed by this study is the need to explore and evaluate the transition from traditional geospatial standards to modern OGC API standards.

In the world of WebGIS, there are various open source software that implements the traditional web standards, such as MapServer, GeoServer, QGIS Server. In addition, pygeoapi implements the modern OGC standard protocol and simplifies the process of sharing, discovering and accessing geospatial data over the Internet. As this research focuses on conducting a comparative analysis between traditional and modern geospatial standards, so the traditional implementation of this standards was done by GeoServer and the modern standards was implemented by pygeoapi, in order to identify the strengths, weak-

nesses and overall performance of these platform. The aim of this research is to explore the potential for enhancing web applications through a comparative analysis of the integration of modern and traditional geospatial technologies based on their performance and practical implications. To achieve this aim, several objectives have been formulated. Firstly, a prototype system architecture for a web application for urban data management was developed. Secondly, a PostgreSQL database was designed as an extension of an existing database. Thirdly, the prototype was implemented using the same database and system specifications, with the pygeoapi and GeoServer selected as geospatial services. Fourthly, existing case studies and real-world implementations of pygeoapi and GeoServer in web applications was analyzed. Finally, a comparative performance analysis between pygeoapi and GeoServer was conducted under varying levels of concurrent requests.

2. Material and Methods

2.1 Data Analysis and Processing

The ALKIS data was used as the source of the test data in this research. ALKIS data includes both spatial data and non-spatial data, interconnected with varying cardinalities. Besides that, it encompasses details about land parcels, buildings, administrative area units, and points containing diverse information such as house numbers, parcel numbers, road names, reservations, land register data, current land use, traffic details, vegetation, water bodies, and more. Apart from the ALKIS data, the official House Coordinates of Germany (HK-DE) data was also used here which defines the precise spatial position of buildings with addresses across Germany. ALKIS data, typically distributed as NAS files in XML format, were processed by utilizing the 'norGIS-ALKIS-Import' software developed by norBIT to convert them into a PostgreSQL database. The resulting dataset was stored in a database named the 'original database', preserving the original ALKIS data structure. Subsequently, a project-specific database was created using Django, and Python scripting was employed to transfer the ALKIS data from the original database to the new database while adhering to the specifications outlined by the Steg planning department. The figure 1 shows the overall workflow of data processing.

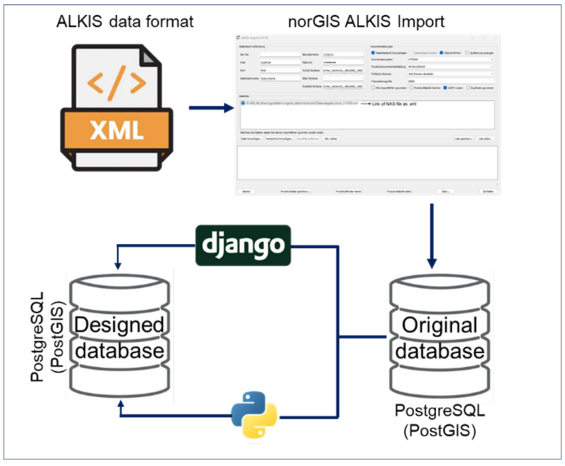

Figure 1. Workflow of the data processing

2.2 Dataset

The web application utilized ALKIS data from both Karlsruhe and Frost cities for testing, enabling a comparative analysis between GeoServer and pygeoapi. Both Karlsruhe and Frost ALKIS datasets contains all the pertinent information typical of ALKIS data mentioned in 2.1. Karlsruhe, is situated in southwestern Germany on the eastern bank of the Rhine River neighboring Rhineland-Palatinate to the northwest and France to the west (Karlsruhe-Erleben-EN., 2023). The total area of land parcel is 31.06 km^2 (Karlsruhe-Erleben-EN., 2023). According to ALKIS data, the total number of land parcel and buildings of Karlsruhe are 11,602 and 17,852 respectively. Forst is positioned in the northern part of the Karlsruhe district encompassing an expanse of 11.47 km^2 (Landeskundliche-Informationssystem, 2024). ALKIS data further reveals that Forst consists of 4,891 land parcels and 5,737 buildings.

2.3 Methodology

The methodological flowchart shown in figure 2 illustrates a systematic approach to implement this research which includes, a WebGIS prototype system architecture, database design, data preparation, service setup, architecture implementations and comparative analysis. The methodology is presented in detail below.

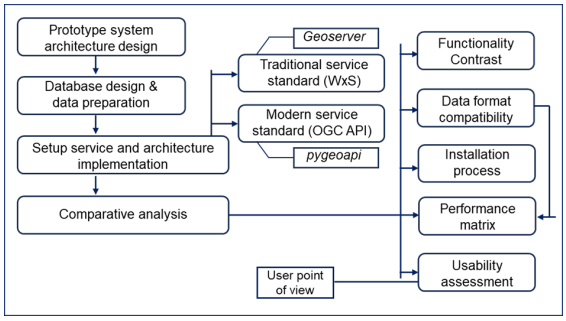

Figure 2. Methodology

2.3.1 Prototype System Architecture Design: WebGIS prototype system architecture serve as a foundational blueprint, outlining its structure and design principles, which guide its development. As far as the architecture of a WebGIS is concerned, the architecture based on three levels is the most commonly used (Pascaul et al., 2012). In this study, two prototype system architectures were formulated based on the underlying principle of three-tier architectures. In the presentation tier, on the client side, JavaScript libraries (OpenLayers, jQuery) as a mapping library as well as Bootstrap, HTML5, and CSS were used to design the web app (client side). The application tier, situated on the server side, comprised map services. In the first prototype system architecture, GeoServer (figure 3), adhering to traditional Open Geo-spatial Consortium (OGC) standards such as Web Map Service (WMS), Web Feature Service (WFS) was employed. Conversely, the second prototype system architecture incorporated pygeoapi (figure 4), a Python implementation aligned with modern OGC API standards. At the database tier, situated at the foundation of the architecture, spatial data was stored in a PostgreSQL database with the PostGIS extension.

2.3.2 Database Design and Data Preparation: The database structure for the 'die STEG Stadtentwicklung GmbH' urban development project is designed to include six primary layers, each containing both spatial and non-spatial data elements. These layers include parcels, buildings, landmarks, red lines, future development areas and visual points. The spatial data from ALKIS is combined with information from field inspections and in-house processing to populate the parcel and

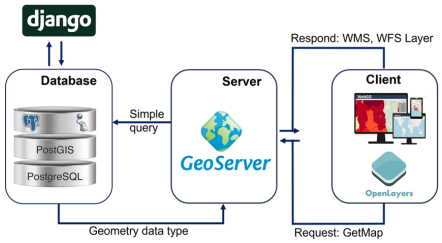

Figure 3. Prototype system architecture using GeoServer

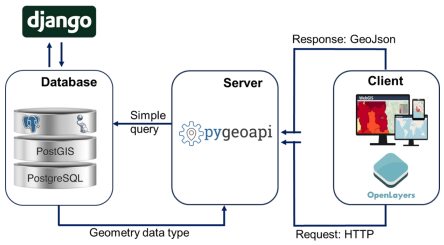

Figure 4. Prototype system architecture using pygeoapi

building layers. The data on landmarks, red lines and future development areas are collected independently during the field inspections. This database design adheres to a fundamental principle: the segregation of geometry and non-geometry information into separate tables, with views created to amalgamate both datasets based on unique identifiers. Such an approach facilitates adaptability and customization, catering to diverse project requirements. Moreover, this design fosters consistency and scalability, enabling the creation of new views while preserving the original spatial structure. Notably, a one-to-one relationship between spatial and non-spatial tables is typically maintained; however, for scenarios involving varying land prices across different years, a one-to-many relationship is established.

2.3.3 Setup Services and Architecture Implementations: In this phase of the methodology, a number of geospatial components were installed to create a robust infrastructure for conducting a comparative analysis. These components included PostgreSQL with the PostGIS extension, GeoServer, pygeoapi and OpenLayers. The pygeoapi installation process offered flexibility with options such as cloning directly from GitHub or using the official Docker image. OpenLayers, on the other hand, was installed by downloading the library from the official website and initializing it within an HTML container through JavaScript code. Regarding the interconnection between elements of the system architecture, the system architecture starts at the front-end interface when the user interacts with the map and triggers a request to the GeoServer for specific data. The GeoServer then establishes communication with the database, extracts the requested information and then transfers it back to OpenLayers. There is therefore no direct connection between OpenLayers and PostgreSQL, but the GeoServer acts as an intermediary. As an intermediary, GeoServer understands the spatial data in the database, interprets incoming requests into SQL or spatial queries, retrieves the data and formats it for transfer to the front end. The seamless connection between GeoServer and PostgreSQL is critical to the efficient management and delivery of spatial data in a WebGIS application. In terms of pygeoapi, it sends requests to the middleware to fetch geographic data and updates the UI accordingly. These requests can be made using standard web protocols such as HTTP or HTTPS (Cerciello and Simones, 2022). It works with the modern HTTP paradigm of the OGC API and uses HTTP verbs such as GET (Used for retrieving data or information), POST (Used for creating or updat-ing data), PUT (Used for updating data), DELETE (Used for deleting data). Communication uses HTTP status codes (e.g. 200, 201, 400) to indicate results and relies heavily on content negotiation to access relevant media types (Cerciello and Simones, 2022). In addition, pygeoapi supports JSON (JavaScript Object Notation), a widely used format among web developers that complies with the principles of RESTful (Representational State Transfer) web services (Cerciello and Simones, 2022).

2.3.4 Comparative Analysis: A comprehensive comparative analysis was conducted to evaluate various parameters between GeoServer and pygeoapi. These parameters encompassed installation processes, folder structure organization, data format compatibility, layer preview functionality, rendering performance, and usability. The analysis extended to assessing rendering times for different feature counts, data format impacts on efficiency, and symbology display. The rendering time of the layer was systematically collected as part of the experimental process. This involved utilizing the network section of Google Chrome Developer Tools, a widely recognized and reliable tool for performance analysis in web development. Specifically, the tool enabled the precise measurement of the time taken for the layer to render within the web application.

2.4 Web-app Implementation: Overview of Implemented Functionalities

In both pygeoapi and GeoServer based web-app, fundamental functionalities are provided within the frontend interface, ensuring a cohesive and robust geospatial exploration experience. To ensure methodological consistency in comparison, both web applications exhibit equivalent functionality while interfacing with distinct server endpoints, originating from pygeoapi and GeoServer, respectively. These functionalities include a visual legend for layer interpretation, dynamic layer switching for focused data exploration, and seamless zooming capabilities for navigation. Additionally, tools for length and area measurements facilitate accurate spatial analysis, while attribute popups offer detailed information on individual layers. Attribute-based queries further empower users by enabling precise extraction of insights based on specified attribute criteria. Overall, the frontend interface of both pygeoapi and GeoServer depicted in the Figure 5.

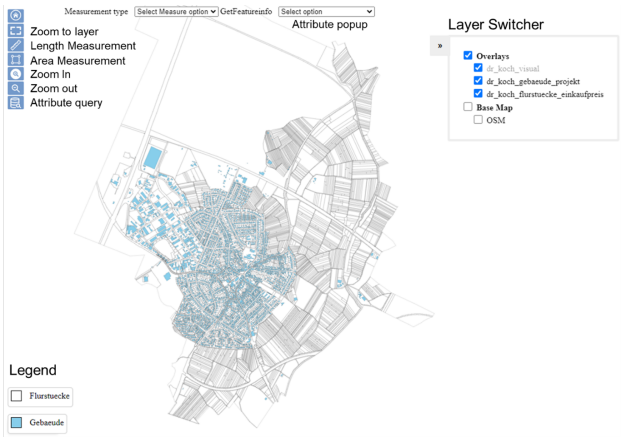

Figure 5. Overall functionalities in the webapp

3. Result: Comparative Analysis

3.1 GeoServer and pygeoapi at a glance in terms of Functionalities

GeoServer, originating in 2001, is a robust, Java-based open-source server tailored for spatial data publishing while conforming to OGC standards. GeoServer's Graphical User Interface (GUI) streamlines tasks like data uploading and symbology adjustments without manual coding. Additionally, GeoServer offers a REST interface for interaction with Python scripts, accommodating both manual and automated workflows. In contrast, pygeoapi, introduced in 2018, represents a Python server implementation aligned with OGC Standards. It establishes a RESTful API endpoint using OpenAPI, GeoJSON, and HTML. Unlike GeoServer, pygeoapi lacks a direct user interface for data publishing tasks but incorporates Swagger, facilitating API documentation and testing. This dynamic interface enables users to explore, comprehend, and test available API endpoints effectively.

3.2 Installation Process

In terms of installation, both GeoServer and pygeoapi offer accessible processes, although with different focuses and approaches. GeoServer being more user friendly particularly for the windows user due to its automated installation procedure equipped with a user-friendly interface, guides the user through the installation steps. Besides that, GeoServer offers several installation options, including Linux binaries, Windows binaries, Windows installers, web archives and Docker containers (Geoserver-Installation, 2024). On the other hand, pygeoapi offers a wider range of installation options, including developer (manual installation), Python package, Docker, Kubernetes, Conda, UbuntuGIS and FreeBSD (pygeoapi Install, 2024). Furthermore, both pygeoapi and GeoServer offer Docker container options for installation which provides a lightweight, portable, and scalable solution for packaging applications and their dependencies (Dimensiona, 2023). In addition, after installing the GeoServer, all packages or libraries are already embedded in the GeoServer, so no additional installation of packages is required. As for pygeoapi, various Python packages need to be installed, which are required for publishing different types of data, such as GDAL for uploading shapefiles; sqlalchemy, geoalchemy2 and psycopg2-binary are required for PostgreSQL (pygeoapi Features, 2024).

3.3 Organization of Folder Structure

GeoServer has an well-structured folder structure that facilitates organized data management. Users have the option of structuring the data according to projects. For example, if there are projects in Karlsruhe and Forst, separate folders can be created with the project names. Within these workspaces, users can define stores options to specify the data format to upload, such as postGIS, geopackage or shapefile. In the Layers section, all layers or views required for a project can be published in different formats. In addition, layer groups and styles can be published using previously created style files. Figure 6 illustrates the organization of the folder structure.

In contrast, pygeoapi lacks a folder organization system for specific projects, and all layers are directly published together on the collection page, but user can verify whether the layers are properly published or not. In order to publish, users must use

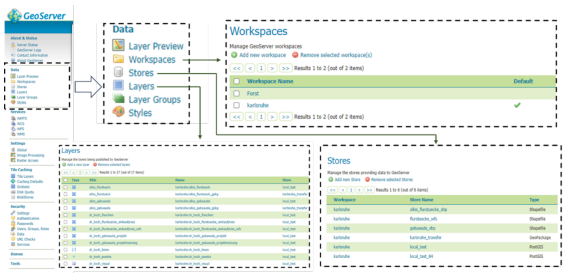

Figure 6. Organization of folder structure in GeoServer

the YAML file, which follows the OpenAPI concept; no direct user interaction could be found by the author from the front end part. Beyond publishing layers, essential information like database connections, metadata, and other related details must be specified in distinct sections within the YAML file.

3.4 Data Format Compatibility

In comparing GeoServer and pygeoapi's data format compatibility, while both platforms support various formats for storing and handling geospatial data, pygeoapi offers a broader range of formats compared to GeoServer. This extensive format support in pygeoapi provides users and developers with increased flexibility and options for storing and sharing data. Put simply, while both GeoServer and pygeoapi offer basic compatibility with common formats, pygeoapi goes one step further by supporting additional formats, potentially serving a wider range of data sources and applications. Table 1 describes the wide range of data formats supported by both pygeoapi and GeoServer.

pygeoapi	GeoServer
CSV (point), Elasticsearch, ERDDAP Tabledap Service, ESRI Shape file, GeoJSON, MongoDB, PostGIS, SQLiteGPKG, SensorThings API	Shapefile, GeoTIFF, PostGIS, GPKG, Oracle, Microsoft SQL Server

Table 1. Data Format Compatibility: pygeoapi vs. GeoServer (pygeoapi Features, 2024, GeoServer-Dataformat, 2024)

3.5 Layer Preview

While both GeoServer and pygeoapi provide a layer preview to check the positioning of the published layers, GeoServer is characterized by the fast loading (259ms) of WMS files, resulting in a faster data display. In contrast, pygeoapi uses GeoJSON files for the layer preview, which can lead to slower rendering times (7.7s), especially for larger feature sets. In addition, pygeoapi uses Leaflet as the frontend for the layer preview, with rendering times being similar regardless of whether Leaflet which is used by the pygeoapi for layer preview or the OpenLayers client, as used in this study.

To enhance the efficiency of layer preview in pygeoapi, one approach is to limit the number of features visualized, a parameter configurable through the YAML configuration file. In the context of layer preview visualization, it's noteworthy that GeoServer lacks a background base map behind the layers, potentially leading to confusion regarding the accurate positioning of the data. On the contrary, pygeoapi addresses this limitation by incorporating OpenStreetMap as base map, ensuring a clear visual context and aiding in the verification of data alignment within the correct coordinate system.

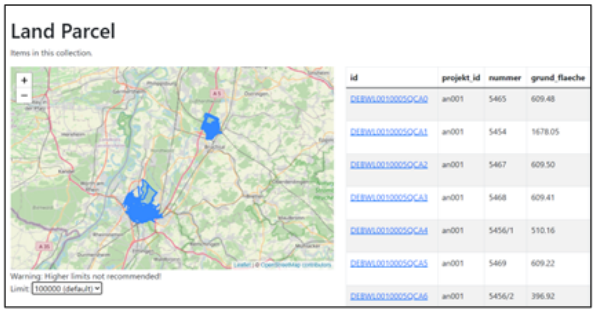

Figure 7. Layer preview of pygeoapi in the server side

Figure 8. Layer preview of GeoServer in the server side

3.6 Performance Analysis

This section describes a comprehensive analysis of performance in terms of rendering time for different levels and features in GeoServer and pygeoapi. The investigation focuses on recognizing the significant dynamic shifts in rendering time in response to different zoom levels as well as the increasing amount of features in the Web Feature Service (WFS), Web Map Service (WMS) and GeoJSON formats. In addition, the effects of different data types, e.g. from PostGIS, shapefiles and GeoPackage (.gpkg), on the performance of layers are examined.

Figure 9 analyses the rendering time of the GeoServer Web Feature Service (WFS) as a function of different zoom levels and different feature sets. The different colours within the bars correspond to the different zoom levels from 14 to 18. It is noticeable that the changes in the zoom levels have only a negligible influence on the rendering time. Specifically, the rendering time fluctuates slightly between 0.43s and 0.49s with a feature count of 4,891. However, when the number of features increases from 4,891 to 23,319, a significant linear escalation in rendering time becomes apparent. The average rendering time increases significantly from 0.46s to 2.15s. This observed pattern indicates a scalability problem associated with an in-creased number of features.

Similarly, Figure 10 shows the dynamics of the rendering time of the GeoServer Web Map Service (WMS) over different zoom levels and feature sets. In contrast to the observed behavior of the GeoServer (WFS), the GeoServer (WMS) shows no detectable economies of scale. This means that the rendering time remains independent of variations in the zoom levels or the number of features. Within the data set, it can be seen that the rendering time has a marginal range that fluctuates between 228 ms and 232 ms in response to changes in the zoom levels for a feature count of 4,891. Similarly, increasing the feature count from 4,891 to 23,319 results in rendering time variations between 216 ms and 227 ms. This proves that the number of features has no distinguishable effect on the rendering time.

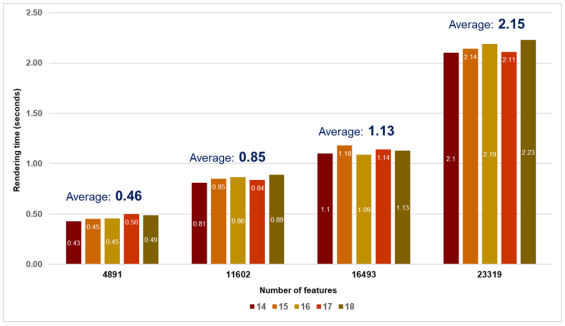

Figure 9. Rendering time of GeoServer (WFS) in relation to varying zoom level and different no. of features

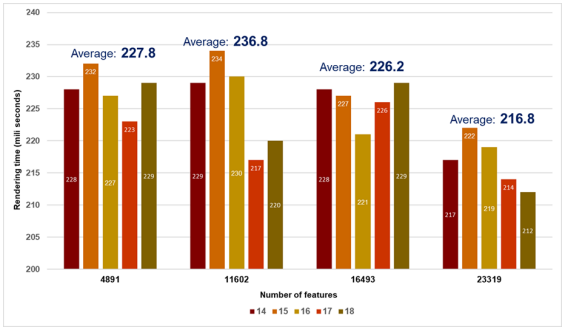

Figure 10. Rendering time of GeoServer WMS in relation to varying zoom level and different no. of Features

Figure 11 describes the rendering time characteristics of pygeoapi GeoJSON across different zoom levels and different feature sets. Similar to the results in GeoServer (WFS), there is a consistent rendering time across different zoom levels from 1.50s to 1.74s for a feature count of 4,891. However, as the number of features increases, a linear clear escalation in rendering time becomes apparent. The sharpest increase in rendering time is observed at the transition from 1.6s at 4,891 to 7.52s for the highest number of features of 23,319. This indicates a higher sensitivity to larger feature datasets, which may indicate scalability issues.

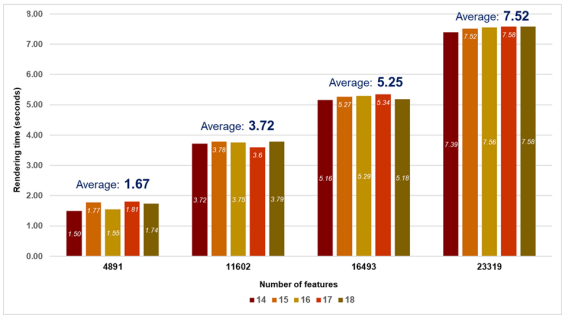

Figure 11. Rendering time of pygeoapi GeoJSON in relation to varying zoom level and different no. of Features

Figure 12 shows a comparative analysis of the rendering times (in seconds) for the GeoServer Web Map Service (WMS), the GeoServer Web Feature Service (WFS) and pygeoapi at different feature counts (4,891, 11,602, 16,493 and 23,319) based on the polygon feature. Here the rendering time is represented for the one specific zoom level which is 16. There are some notable observations when evaluating these results. Overall as a result, GeoServer (WMS-Tiles) renders in a fairly consistent manner, ranging from 0.219s to 0.23s as feature counts increase

from 4,891 to 23,319. This indicates robust performance under different feature sets. Contrastingly, GeoServer (WFS-GML output) displays a more noticeable increase in rendering time. Starting at 0.454s for 4,891 features, the rendering time escalates to 2.19s for 23,319 features. pygeoapi (GeoJSON), in comparison, demonstrates rendering times that are notably higher than both GeoServer (WMS and WFS). Commencing at 1.55s for 4,891 features, the rendering time substantially increases to 7.56s for 23,319 features. This substantial increment may suggest certain performance implications or efficiency considerations associated with the pygeoapi platform.

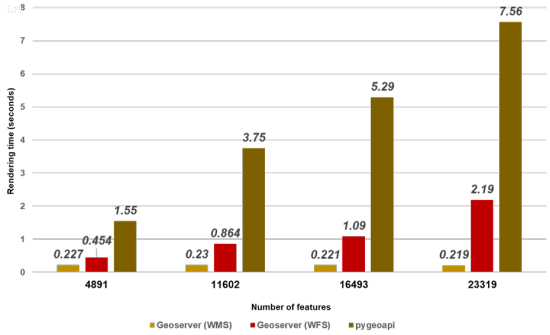

Figure 12. Comparison rendering time of GeoServer WFS, WMS and pygeoapi

Figure 13 shows a comparative analysis of the rendering times (in seconds) for the GeoServer using different data formats - PostGIS, Shapefile (.shp) and GeoPackage (.gpkg). The analysis focuses in particular on 16 zoom levels and a data set with 16,493 features. The results in Figure 13 show different rendering performances for the individual data formats in GeoServer which indicates that PostGIS has a higher efficiency compared to the other data formats such as Shapefile and GeoPackage, having a slightly longer rendering time.

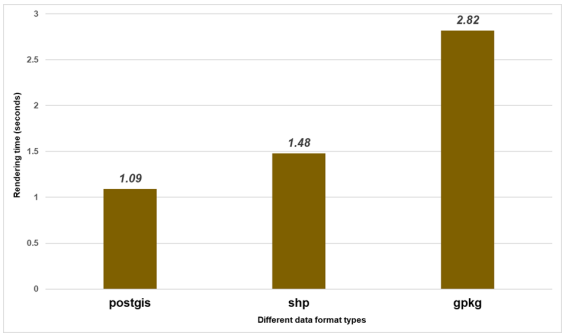

Figure 13. Performance differences because of the different data format in GeoServer

As part of pygeoapi, a comprehensive analysis was carried out covering an extended range of data formats, in particular PostGIS, Shapefile (.shp), GeoJSON, WFS (Web Feature Service) and GeoPackage (.gpkg). The investigation maintained consistency of key parameters, e.g. 16 zoom levels and a dataset of 16,493 features. It is noticeable that pygeoapi offers the flexibility to incorporate traditional data services such as WFS and WMS, a feature that is particularly beneficial for web applications built on modern architectures. Figure 14 shows the perceived performance differences between the different data formats in pygeoapi. Due to PostGIS's spatial extensions for PostgreSQL, an efficient relational database management system, it is very effective for displaying and delivering geospatial data due to its optimized spatial indexing and query capabilities (Tjukanov and Topi, 2018). GeoJSON also shows remarkable performance (6.25s). However, the rendering quality decreases for data formats such as WFS and GeoPackage (more than 9s).

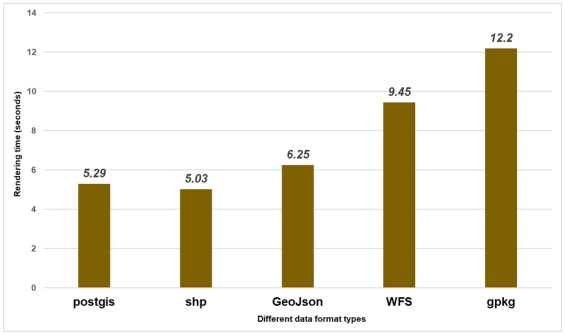

Figure 14. Performance differences because of the different data format in pygeoapi

3.7 Symbology and Legend

GeoServer presents a comprehensive suite of symbology options tailored to optimize the visualization of geospatial data. Central to this toolkit is the Styled Layer Descriptor (SLD). While SLD files can be seamlessly integrated into GeoServer via its REST interface, occasional manual adjustments might be required for proper uploading. Moreover, GeoServer extends its capabilities with CSS styling, YSLD, and support for Mapbox Styles through the MBStyle extension. pygeoapi itself does not have built-in symbology options or styling capabilities. Styling is typically handled by client applications that consume data served by pygeoapi. In this research, symbology was declared manually on the OpenLayers, front-end part while publishing the layer to the front from the server side.

The comparison between GeoServer and pygeoapi reveals differences in their handling of symbology and legend graphics. GeoServer utilizes the GetLegendGraphic operation to generate legend images, often resulting in lower quality output due to its reliance on raster graphics. In contrast, pygeoapi lacks server-side support for legends, necessitating a client-side approach. Despite this, pygeoapi offers clearer and more detailed symbology through vector graphics, surpassing GeoServer in terms of detail and scaling quality.

3.8 Usability Assessment

The usability assessment based on a survey of 12 participants provided insights into the overall user experience, loading speed, map display, recommendations, and preferences between GeoServer and pygeoapi in the context of a WebGIS application. In comparison, for the pygeoapi-based web application, 25% of users said they were satisfied, while the majority of 75% said their experience was good (Figure 15b). Of particular note, participants were unanimous (100%) in favor of GeoServer in terms of the speed of loading and map display, underlining its superiority in these aspects (Figure 15a). In terms of recommendations, 50% of users recommended both platforms, showing a balanced preference (Figure 15d). However, 33% of respondents, especially those who identified themselves as developers, exclusively recommended pygeoapi, underlining its appeal within the developer community (Figure 15d). Conversely, 17% of users recommended GeoServer (Figure 15e). Interestingly, 42% of users saw no significant difference between GeoServer and pygeoapi when it came to rating

the smoother panning, zooming in, and zooming out (Figure 15e). However, 41% preferred pygeoapi for these functionalities, while 17% preferred GeoServer (Figure 15e).

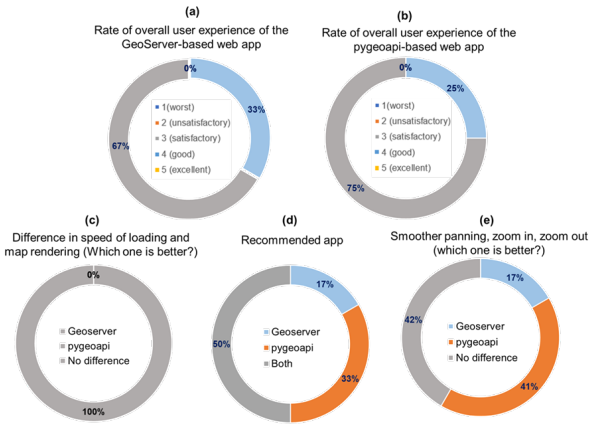

Figure 15. User Perspectives about GeoServer and pygeoapi

3.9 Vector Tiles as a Solution of Faster Rendering Speed of pygeoapi

It was seen in the previous analysis that, the pygeoapi has the higher rendering time of up-loading data in compare to the GeoServer. A solution can be to address to this problem is vector tiles. A significant advantage of vector tiles over fully rendered image tiles lies in their dynamic styling capabilities, which enable them to be adjusted on-the-fly without requiring the download of additional tiles (Vector-Tiles, 2024). At this moment, the providers of the vector tiles of pygeoapi are MVT-tippecanoe, MVT-elastic, MVT-proxy, WMTSFacade as a remote data source (pygeoapi Vector-Tiles, 2024). In this particular scenario, the tiles are pre-rendered for a designated layer within the local directory through the application of Tippecanoe. Subsequently, the layer is published by leveraging the configuration file of pygeoapi. Figure 16 represents the prerendered vector tiles data of Karlsruhe published from pygeoapi and OpenStreetMap (© OpenStreetMap contributors) as basemap. The findings demonstrate a notable reduction in rendering time, decreasing from 7.56 seconds to 459 milliseconds when transitioning from GeoJSON to vector tiles.

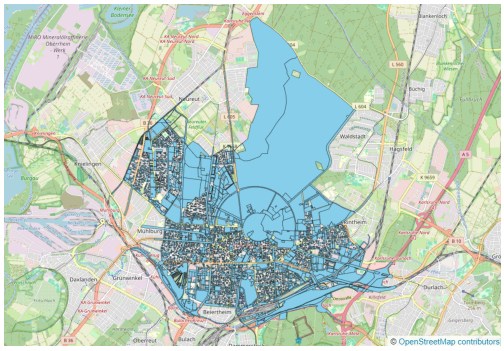

Figure 16. Using vector tiles in pygeoapi

4. Discussion

In the course of research, several significant findings have been achieved on the effectiveness, efficiency and usability of modern OGC API standards compared to traditional geospatial data methods. These findings range across a broad spectrum of areas, including the fine distinctions of web application architectures, the handling of different geospatial data formats, performance benchmarks under different workload scenarios, and usability considerations. The general findings of this research are presented below.

The chosen architecture in this study comprises three levels: Presentation (front-end), Application (server-side) and Database. OpenLayers as well as Bootstrap, HTML5 and CSS were chosen for the front-end due to their versatility and robust mapping capabilities. GeoServer, which adheres to traditional OGC standards, was chosen for the server side, while pygeoapi, which is based on modern OGC API standards, ensures greater efficiency. The spatial data is stored in a PostgreSQL database with the PostGIS library, which enables efficient management of the geographical objects. With this three tier architecture, web application architectures can achieve optimal performance, flexibility and usability, leading to improved geospatial services in city government and beyond.

GeoServer and pygeoapi offer different benefits: GeoServer offers a user-friendly GUI for seamless data management and a quick installation process, while pygeoapi offers versatility with multiple installation options. In addition, GeoServer has an organized folder structure for data management, while pygeoapi lacks project based organization but compensates with a comprehensive YAML file approach. In addition, GeoServer is characterized by high layer preview display speed, while pygeoapi provides a basemap for better spatial context. GeoServer boasts a well-established community with extensive documentation, structured development processes and comprehensive user manuals aimed at users of all skill levels. In contrast, pygeoapi, which is relatively new, has a smaller community but offers a Python-based RESTful API with Swagger integration that provides a dynamic interface for API documentation and testing. Despite GeoServer's user-friendly documentation that caters to a wide variety of users, pygeoapi documentation requires a solid technical understanding for effective use.

GeoServer and pygeoapi are characterized by the processing of geodata formats for web applications. While GeoServer supports common formats such as PostGIS, Shapefile and GeoTIFF, pygeoapi offers all common formats that GeoServer also provides, extended by additional formats such as CSV, Elasticsearch and MongoDB. Besides that, pygeoapi also supports traditional services like WMS, WFS. This extended support by pygeoapi expands the possibilities of data storage and data exchange and improves the applicability in various applications. To summarize, both platforms offer basic compatibility, but pygeoapi broader format support covers a wider range of user requirements. In the practical tests of rendering times due to the different data formats in GeoServer and pygeoapi, it was found that in both cases PostGIS had the highest efficiency, while GeoPackage lagged slightly behind. In pygeoapi, PostGIS and Shapefile were the most efficient, while GeoJSON also performed well. However, formats such as WFS and GeoPackage had longer rendering times. Overall, PostGIS proved to be the most effective format for displaying and providing geodata on both platforms.

The performance comparison between the pygeoapi and GeoServer with different numbers of concurrent requests reveals several important findings. While the Web Map Service (WMS) in GeoServer shows consistent rendering times regardless of

the number of features or zoom level, the Web Feature Service (WFS) which uses GML as data format shows a linear increase in rendering times for larger feature datasets, indicating scalability issues, but no effect of rendering time because of the zoom level. On the other hand, the rendering times of pygeoapi (GeoJSON) are consistently higher compared to GeoServer, indicating optimization opportunities. However, both platforms show minimal impact on rendering times due to additional functionalities at a certain zoom level, which underlines their stability with simultaneous requests. Overall, GeoServer shows a more consistent rendering performance, especially in scenarios that require established Geo-services.

The usability assessment revealed that while GeoServer excelled in loading speed and map display, pygeoapi achieved higher overall user satisfaction. However, both platforms received balanced recommendations from users, indicating different preferences and strengths of the two platforms. In summary, the study compares modern OGC API standards represented by pygeoapi with traditional geospatial methods such as GeoServer for web applications. While GeoServer offered stable rendering performance, pygeoapi was more attractive to developers. The results emphasize the importance of considering project requirements and user preferences when choosing between these platforms for web GIS applications.

5. Conclusions

This work has explored the area of modernizing geospatial technologies through the adoption of Open Geospatial Consortium (OGC) API standards in web applications, particularly in the context of urban development management. Through theoretical assessments and practical testing, the study has provided valuable insights into the transition from traditional geospatial service standards to modern OGC API standards. The comparison between GeoServer and pygeoapi revealed differences in terms of performance, data compatibility and deployment processes, highlighting the strengths and limitations of the two platforms. While GeoServer offers a comprehensive feature set, pygeoapi focuses on simplicity and modern technologies. In addition, the study of data formats highlighted the importance of considering the differences of each format when optimizing geospatial data services. The work also proposes actionable insights to improve the performance of web applications, such as the inclusion of vector tiles to mitigate rendering time issues. Overall, this research contributes to the advancement of geospatial technologies by bridging the gap between established standards and emerging APIs, paving the way for more efficient and effective web applications in the field of urban development management.

References

Assefa, D., 2018. Developing Data Extraction and Dynamic Data Visualization (Styling) Modules for Web GIS Risk Assessment System (WGRAS). *Master Thesis in Geographical Information Science*.

Cerciello, Antonio, J., Simones, 2022. emotional cities: Mapping the cities through the senses of those who make them: Architecture definition and code for the generic sdi. Technical report, DTU.

Dimensiona, 2023. What is docker and what are its advantages?–dimensiona. https://www.dimensiona.com/en/what-is-docker-and-what-are-its-advantages (2024-01-02).

GeoServer-Dataformat, 2024. Supported data formats—geoserver 2.26.x user manual. https://docs.geoserver.org/main/en/user/extensions/importer/formats.html (2024-03-04).

Geoserver-Installation, 2024. Installation—geoserver 2.26.x user manual. https://docs.geoserver.org/main/en/user/installation/index.html (2024-03-02).

Gonçalves, O., Virtudes, A., 2020. Smart Urban Planning at Local Scale: e-Master Plan. *KnE Engineering*, 214–227. DOI: 10.18502/keg.v5i5.6943.

Karlsruhe-Erleben-EN., 2023. Facts and figures. https://www.karlsruhe-erleben.de/en/city-portrait/facts-figures (2023-09-30).

Landeskundliche-Informationssystem, 2024. Frost-detail page-leo-bw. https://www.leo-bw.de/web/guest/detail-gis/-/Detail/details/ORT/labw_ortslexikon/5360/Forst(2024-02-09).

OGC-Context, 2023. Ogc api context. https://ogcapi.ogc.org/ (2023-09-26).

OGC-Standards, 2023. Open geospatial consortium standards—osgeo-live 7.9 documentation. https://live.osgeo.org/archive/7.9/en/standards/standards.html (2023-09-23).

Pascaul, M., Alves, E., De Almeida, T., De França, G. S., Roig, H., Holanda, M., 2012. An architecture for geographic information systems on the web-webgis. *Proceedings of the GEOProcessing 2012 Fourth International Conference on Advanced Geographic Information Systems, Applications, and Services, Valencia, Spain*, 209–214.

pygeoapi Features, 2024. Publishing vector data to ogc api-features. https://docs.pygeoapi.io/en/stable/data-publishing/ogcapi-features.html (2024-03-03).

pygeoapi Install, 2024. pygeoapi: Install. https://docs.geoserver.org/main/en/user/installation/index.html (2024-03-03).

pygeoapi Vector-Tiles, 2024. Publishing tiles to ogc api-tiles—pygeoapi 0.16.dev0 documentation. https://docs.pygeoapi.io/en/latest/data-publishing/ogcapi-tiles.html (2024-03-18).

Rowland, A., Folmer, E., Beek, W., 2020. Towards self-service gis—combining the best of the semantic web and web gis. *ISPRS international journal of geo-information*, 9(12), 753. https://doi.org/10.3390/ijgi9120753.

Tjukanov, Topi, 2018. Why should you care about postgis?—a gentle introduction to spatial databases. https://medium.com/@tjukanov/why-should-you-care-about-postgis-a-gentle-introduction-to-spatial-databases-9eccd26bc42b (2024-03-10).

Vector-Tiles, 2024. Vector tiles introduction-tilesets-mapbox. https://docs.mapbox.com/data/tilesets/guides/vector-tiles-introduction/ (2024-03-18).