OGC® DOCUMENT: 21-019

External identifier of this OGC® document: http://www.opengis.net/doc/PER/t17-D040

Open Geospatial Consortium

OGC TESTBED-17: ATTRACTING DEVELOPERS: LOWERING THE ENTRY BARRIER FOR IMPLEMENTING OGC WEB APIS

ENGINEERING REPORT

PUBLISHED

Submission Date: 2021-11-19
Approval Date: 2021-12-17
Publication Date: 2022-01-21
Editor: Aleksandar Balaban

CONTENTS

LIST OF TABLES

LIST OF FIGURES

I ABSTRACT

This OGC Testbed 17 Engineering Report (ER) documents the work completed in the “Attracting Developers: Lowering the entry hurdle for OGC Web API experiments” task.

OGC Web API Standards are being developed to make it easy to provide geospatial data over the web. These standards provide a certain level of formality to ensure high levels of interoperability. They rigorously define requirements and rules to reduce room for error during interpretation. This rigor sometimes makes the standard documents difficult to read and hence implement. Rather than direct examination of a standard, the majority of developers often prefer to start with implementation guidelines, sample code, and best practice documentation and then refer to the standards document for guidance and clarity.

The Testbed-17 (TB-17) API task served as a foundation for further development and exploration and delivers knowledge necessary for agile development, deployment, and executing OGC Standards-based applications following a “How-To” philosophy with hands-on experiments, examples, and instructions.

II EXECUTIVE SUMMARY

The Testbed-17 API work began with design and conduction of an online Clause 4 and then focused on the results derived from the answers given by the participants: The developers were mostly from the OGC community. The outcome of this survey is that a typical profile of a geospatial developer is generally comparable with the results of similar surveys, such as StackOverflow Developer Survey 2021, which have been done having considered much larger groups of developers with broader interests. The conclusion is that a typical developer from the OGC/geospatial community is an experienced person focused on stable, well-established technologies. Therefore, this ER focuses on newer technologies and approaches to fill the gap between “stable maturity” and “contemporary/emerging trends”.

The value of this ER to interoperability is the exploration of different options for OGC APIs in the context of contemporary development and deployment practices. For interoperability, cloud native services are often based on open source and standards based technology. This helps reduce vendor lock-in and results in increased portability. The ER establishes the repository of How To-examples (GitHub) enabling faster and more efficient geospatial data provision and consumption based on OGC APIs and modern/emerging technology stacks. The API experiments further advance location-based technologies by investigating the compatibility to OGC API Standards with, and the applicability to, technologies such as source code generation and cloud to OGC API Standards. The final result is more efficient development processes, which make building new tools cheaper, more efficient, and less stressful, as well as a broader developer audience attracted to OGC APIs.

The requirements addressed by the work documented in this ER were:

a) Provide easy to understand documentation as a starting point for Web developers to explore OGC APIs standards.

b) Establish and demonstrate a culture of continuous learning and experimentation in the OGC APIs context utilizing DevOps practices.

c) Provide instructions/guidelines to get started with OGC APIs within cloud environments.

d) Expose data sets from cloud storages for consumption using selected technology stacks and compatible OGC APIs.

e) Explore the compatibility of client/server code generation approach in conjunction with OGC APIs.

An overview of recommendations on how to further proceed with the achievements documented in this ER is given in sections Clause 9 and Clause 10.

III KEYWORDS

The following are keywords to be used by search engines and document catalogues.

ogcdoc, OGC document, API

IV PREFACE

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

V SECURITY CONSIDERATIONS

No security considerations have been made for this document.

VI SUBMITTING ORGANIZATIONS

The following organizations submitted this Document to the Open Geospatial Consortium (OGC):

- m-click.aero GmbH

VII SUBMITTERS

All questions regarding this document should be directed to the editor or the contributors:

NAME	ORGANIZATION	ROLE
Aleksandar Balaban	m-click.aero GmbH	Editor
Arne Vogt	52°North GmbH	Contributor
Ignacio Correas	Skymantics Europe SL	Contributor
Sam Lavender	Pixalytics Ltd	Contributor
Karri Ojala	Solenix GmbH	Contributor
Alexander Lais	Solenix GmbH	Contributor

1

SCOPE

1 SCOPE

The scope of this OGC Testbed 17 ER covers the following topics:

- Clause 3 introduces the problem that appears when starting with OGC APIs development. The overview describes the current situation and discusses the requirements set by the testbed Sponsors.
- Clause 4 presents the results of developer survey.
- Clause 5 illustrates the scenarios and experiments of this task.
- Clause 6 describes the task architecture and the services/components.
- Clause 8 explains the concept of code generation for OGC API.
- Clause 9 summarizes the results of this API experiments.
- Clause 10 provides conclusions and gives recommendations for future Work.
- Clause 11 covers the Technology Integration Experiments between the DCS Server and DCS Client.
- Annex A lists the questions used in the developer survey.

2

TERMS, DEFINITIONS AND ABBREVIATED TERMS

2 TERMS, DEFINITIONS AND ABBREVIATED TERMS

This document uses the terms defined in OGC Policy Directive 49, which is based on the ISO/IEC Directives, Part 2, Rules for the structure and drafting of International Standards. In particular, the word “shall” (not “must”) is the verb form used to indicate a requirement to be strictly followed to conform to this document and OGC documents do not use the equivalent phrases in the ISO/IEC Directives, Part 2.

This document also uses terms defined in the OGC Standard for Modular specifications (OGC 08-131r3), also known as the ‘ModSpec’. The definitions of terms such as standard, specification, requirement, and conformance test are provided in the ModSpec.

For the purposes of this document, the following additional terms and definitions apply.

2.1. Terms and definitions

2.1.1. **Application Programming Interface**

An Application Programming Interface (API) is a standard set of documented and supported functions and procedures that expose the capabilities or data of an operating system, application or service to other applications (adapted from ISO/IEC TR 13066-2:2016).

2.1.2. **CI/CD**

Continuous Integration, *Continuous Delivery* is a method to frequently deliver apps to customers by introducing automation into the stages of app development. (source: RedHat)

2.1.3. **DevOps**

a set of practices that combines software development (Dev) and IT operations (Ops).

2.1.4. **Cloud**

a network of computing resources, such as storage, servers, applications and services, that can be used on-demand anywhere and anytime via the Internet NIST.

2.1.5. **Cloud-native**

The concept of building and running applications to take advantage of the distributed computing offered by the cloud delivery model. Cloud native apps are designed and built to exploit the scale, elasticity, resiliency, and flexibility the cloud provides, Oracle.

2.1.6. **Code Generation**

Source code generation is the process of creating programming code from a model such as one of OGC APIs data schemas.

2.1.7. **Containerization**

the packaging of software code with just the operating system (OS) libraries and dependencies required to run the code to create a single lightweight executable—called a container—that runs consistently on any infrastructure, IBM.

2.1.8. **CRUD**

In computer programming, create, read (aka retrieve), update, and delete are the four basic functions of persistent storage.

2.1.9. **OGC APIs**

Family of OGC standards developed to make it easy for anyone to provide geospatial data to the web.

2.1.10. **Technology Stack**

A technology stack or tech stack refers to a set of technologies, software, and tools that are used in the development of applications.

2.2. Abbreviated terms

API	Application Programming Interface
CI/CD	Continuous Integration / Continuous Delivery
CNCF	Cloud Native Computing Foundation
COTS	Commercial Off The Shelf
DevOps	Software development (Dev) and IT operations (Ops)
EDR	Environmental Data Retrieval
IDL	Interface Definition Language
JSON	JavaScript Object Notation
OGC	Open Geospatial Consortium
OWS	OGC Web Services
STAC	Spatio temporal Asset Catalog

3

INTRODUCTION

3 INTRODUCTION

This OGC Testbed 17 ER documents the experiments performed with OGC API Standards for lowering the entry barrier for developers implementing OGC API Standards. The ER describes all developed services and components, the results of Technology Integration Experiments (TIE), provides an executive summary, and describes recommended future work.

4

DEVELOPER SURVEY

4 DEVELOPER SURVEY

A pivotal activity within the TB-17 API experiments task was to design and conduct an online survey. The target survey universe was developers from the geospatial community. The survey objective was to collect information about their profile and behaviors, as well as their needs and their satisfaction with current and emerging OGC API Standards. The survey received responses from 125 participants. Answers from the survey helped the testbed participants understand the current situation, design the API experiments, recommend improvements for OGC APIs documentation, and provide useful tutorials with code examples for the community.

The answers were evaluated graphically and are available in Annex A. The following sections provide a detailed summary of survey's findings:

4.1. Who are our developers?

More than 80% of the respondents reside in Europe and the United States of America; and many of the respondents were experienced developers.

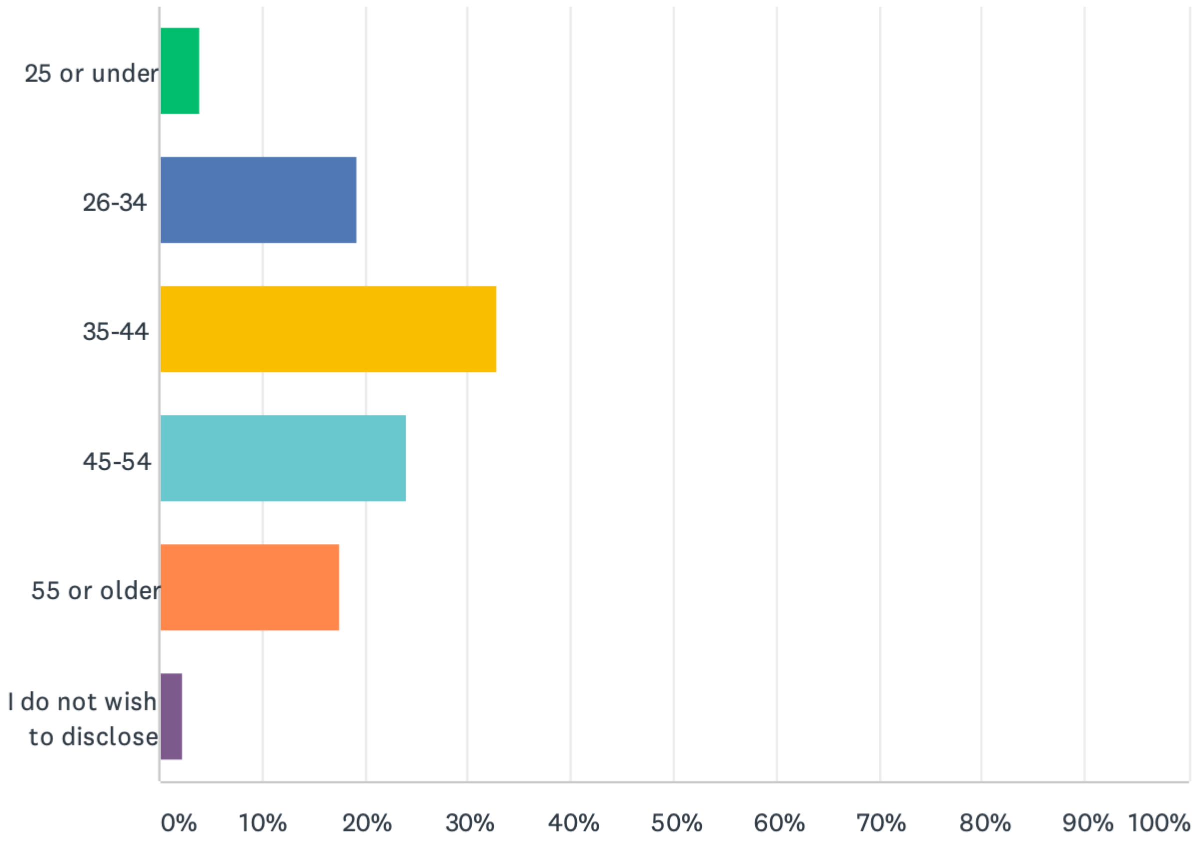


Figure 1

- Almost 75% of developers are older than 35!
- More than 70% of the respondents have Masters degrees or/and doctorates.
- Almost 80% are full-time employed, 20% work for public agencies, 30% for small businesses.
- 53% of the respondents currently reside in Europe and almost 29% in North America
- 60% of them stated that they spend 21 hours or more per week on tasks related to geospatial technologies (Q19).
- 71% of the respondents have been coding for more than 8 years.

4.2. Technology Stack

As expected, Python and JavaScript are the most popular programming languages.

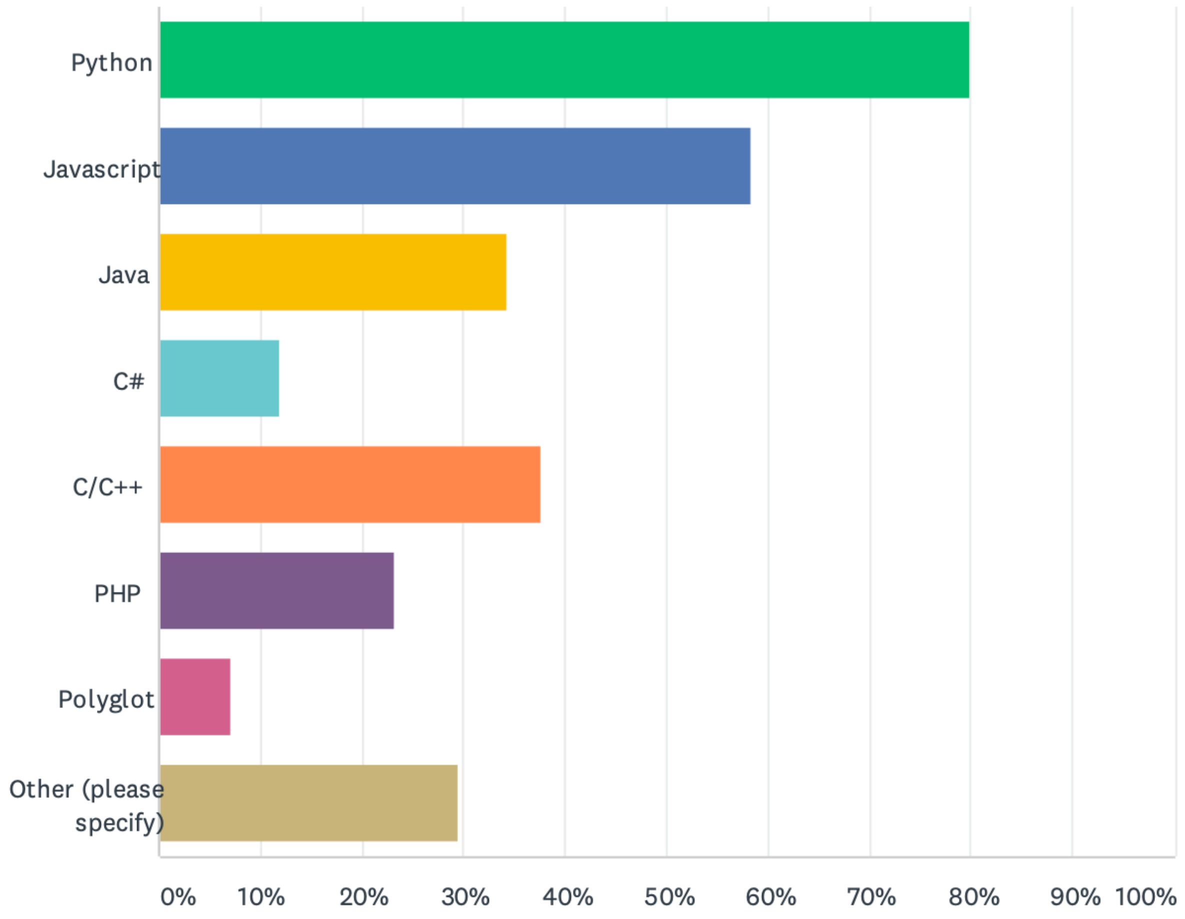


Figure 2

- Regarding programming languages preferences (Q11), developers are most familiar with Python (about 80%) followed by JavaScript (58%). Many of them are also fluent in more traditional C/C++ programming languages (about 38%). Java is still well presented in their technology stacks.
- Developers are very familiar with the mainstream open-source geospatial projects. The most popular is the QGIS framework (for 90.40% of survey participants) followed by PostGIS 82.40%, GeoServer 65.60%, and MapServer 48.80%. This also reflects the fact that features and maps are the most significant and prominent geospatial data formats.

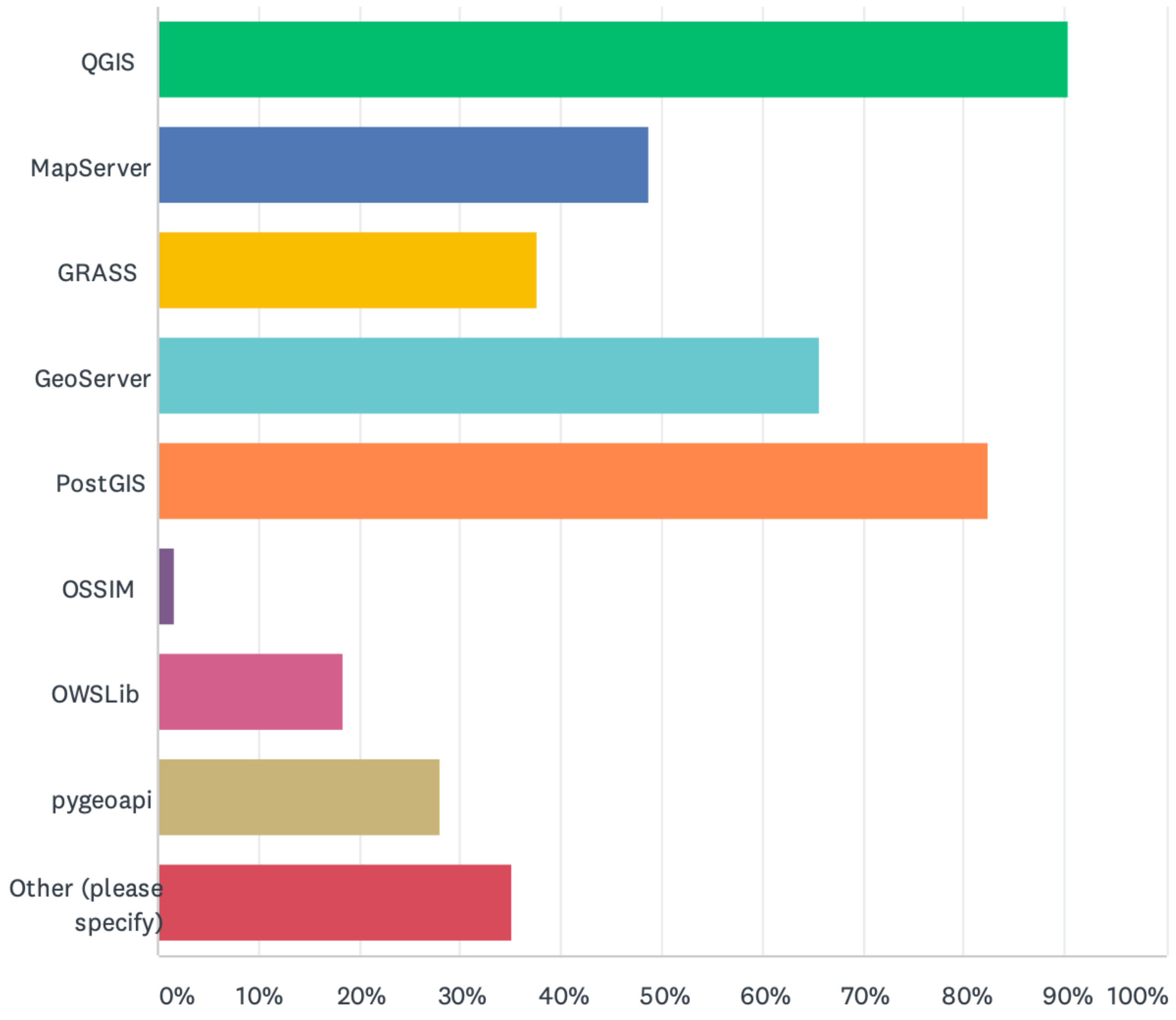


Figure 3

4.3. Learning Habits

- The preferable way to learn a technology seems to be to “follow a documented how-to/tutorial” (30%) and “to read a tutorial” (20%).
- When learning about a new topic from the internet, “full written descriptions” are the most preferable sources.
- Portal Stack Overflow (Q29) seems to be the most important tool for getting support from the community (80%).

4.4. Familiarity with OGC (API) Standards

- About 20% of the respondents have contributed to the development of OGC API Standards on GitHub. Still, these standards are not yet widely used in the community because, although 75% of respondents have either used an OGC Standard or read one of the documentations (Q24), a high percentage (30%) answered that **they don't have any experience involving OGC API Standards** (Q25).

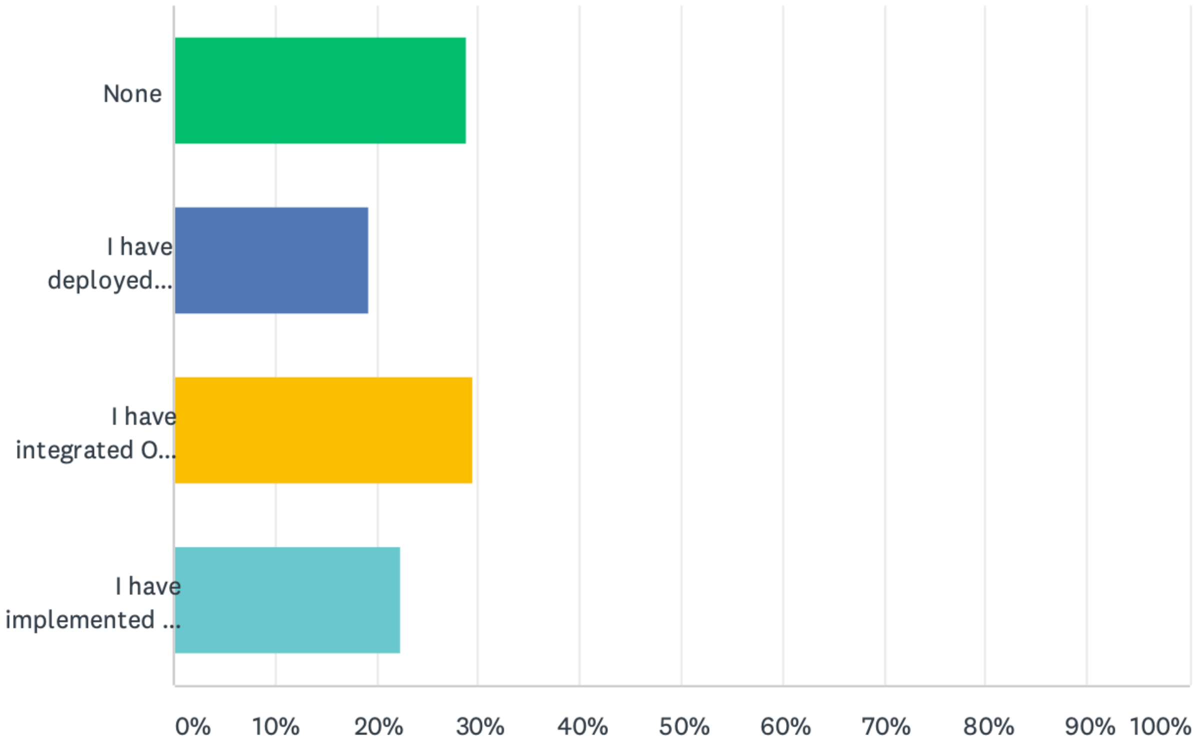


Figure 4

- Features, Maps and Tiles are the most important geospatial standards and therefore the corresponding OGC APIs are of equal importance (Q27).
- Some of the respondents believe OGC API Standards are not easy to learn. On the question of whether OGC API Standards are easy to learn (Q30) almost 30% answered that "Mastering the first API is hard, but then it becomes easier to learn" while 40% thinks that "Some are very easy and **others are extremely difficult** to learn".
- About 57% believes that there is a **lack of best-practice guides** that help differentiate the dos and don'ts", as well as 38% that technical documentation is **hard to read or understand** (Q32)!

4.5. Summary

The survey results suggest that a typical developer from the OGC/geospatial domain is an experienced, well-educated individual focused on stable, well-established technologies. Traditional and new technologies are equally used. For example they follow the recent trends regarding scripting programming languages as they are predominant today.

Considering the results, the API experiments focused on developing examples and how-to documentation as well as newer technologies and approaches to fill the gap between “stable maturity” and “contemporary/emerging trends”.

Annex A lists the questions used to conduct developers survey.

5

API EXPERIMENTS SCENARIOS OVERVIEW

5 API EXPERIMENTS SCENARIOS OVERVIEW

The following figure illustrates the work items and deliverables of this task.

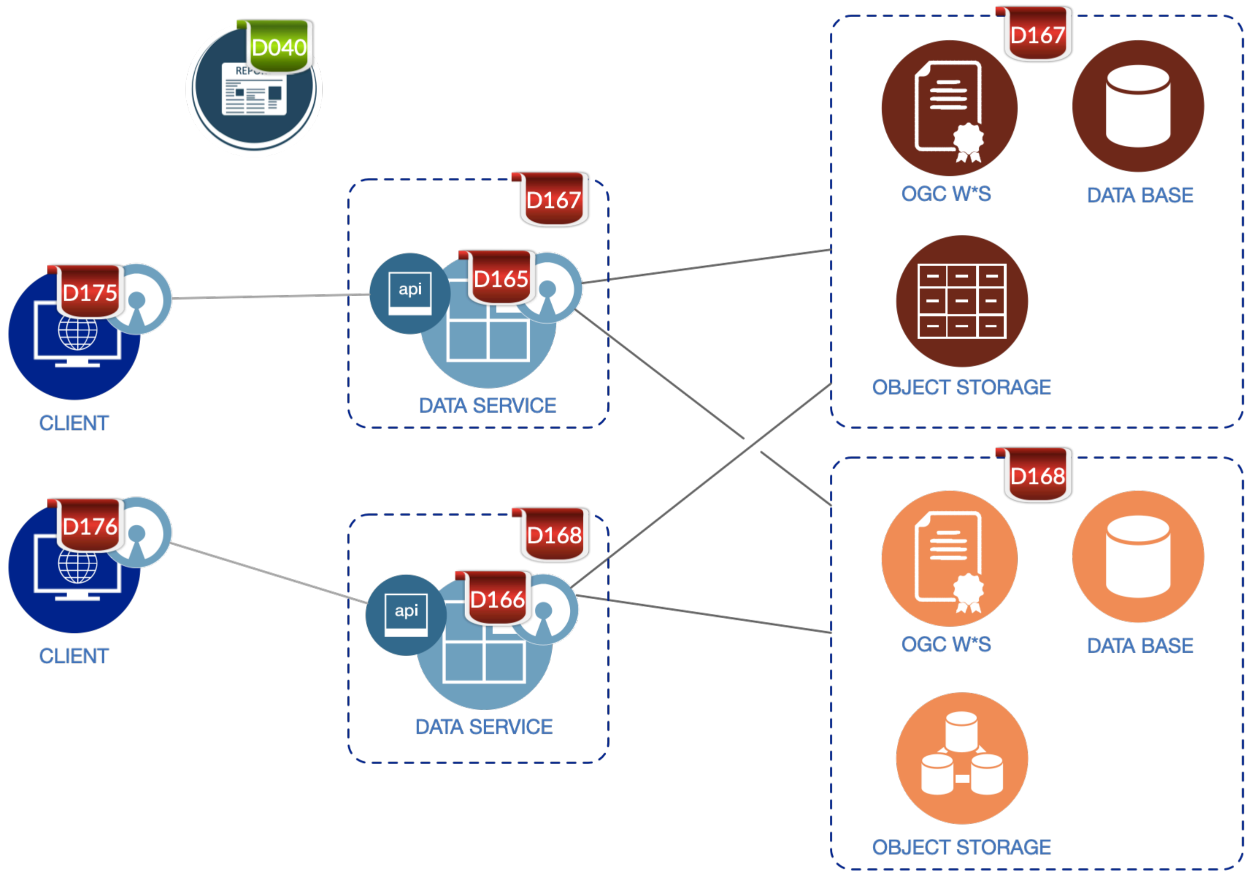


Figure 5 — Deliverables and Packages

5.1. Participants

To test all components and guidance material, the work in this task was shared by participants acting as:

a) Client developers

b) Server developers

c) Data providers

Client developers: Organization O1 implemented and delivered a client-side software library in Python for both OGC API Features and OGC API Environmental Data Retrieval (EDR).

Organization O2 delivered a similar library in JavaScript and TypeScript. Both organizations demonstrated their library in a client application.

Server developers: Organization O3 implemented and delivered a server-side software library in Python for both OGC API — Features and OGC API — EDR. The library supported access and exploration of both OGC API — Feature and OGC API — EDR resources and supported data hosted on cloud object storage, cloud databases, and OGC Web Feature Service (WFS) instances (with constraint functionality). O3 provided a deployable instance together with deployment and execution documentation that allows third parties to test the deliverable(s) as a microservice in different cloud environments. Organization O4 delivered a similar library in JavaScript (alternatively TypeScript).

Data providers: Organization O5 deployed a microservice based on deliverables provided by O3 and O4 in at least two different cloud environments (to the extent possible, as some data backends are cloud specific, others work across cloud environments or are fully cloud independent). In addition, O5 provided data in a database, cloud object storage, and as a Web Feature Service instance to test data access (reuse of existing WFS and/or Web Coverage Service (WCS) instances, even outside of the target cloud environment, is admitted). All of the data sources were available to all organizations in this task and are freely accessible (i.e. object storage and WFS/WCS endpoints are fully accessible, together with cloud native databases as much as possible). Organization O6 performed similarly to O5.

Participants:

- **Data providers:**
  - **Skymantics:** D167 Data Backend and Deployment 1
  - **Pixalytics:** D168 Data Backend and Deployment 2
- **Services (server developers):**
  - **52N:** D165 API Experiments Server (Python)
  - **GMU:** D166 API Experiments Server (JavaScript)
- **Client developers:**
  - **Skymantics:** D175 API Experiments Client (Python)
  - **Solenix:** D176 API Experiments Client (TypeScript)

5.2. API experiment scenarios

The figure below illustrates a generic Testbed-17 OGC APIs experiment component diagram with interactions. A client with a dedicated OGC API software library (1) interacts with a data service that implements one of the OGC APIs (2) at the front-end and includes a cloud native data storage software library (3) at the backend that interacts with cloud native storage, such as cloud databases, cloud object storage, or existing OGC Web Service instances such as WFS.

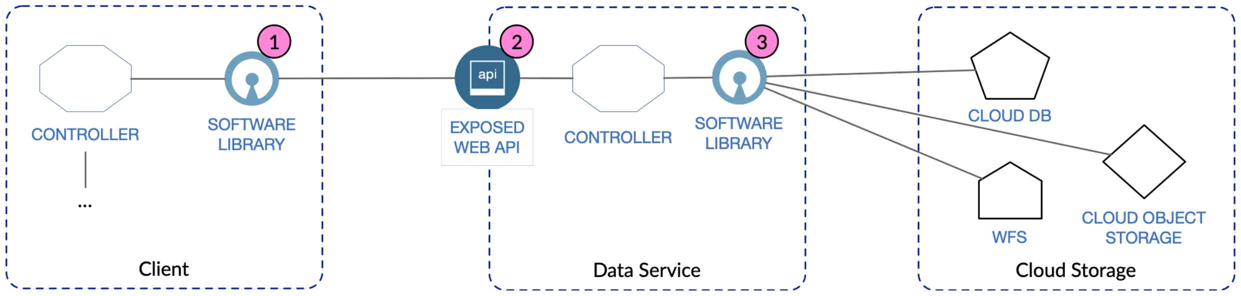


Figure 6 — Generic API experiments scenario

6

ARCHITECTURE AND COMPONENTS

6 ARCHITECTURE AND COMPONENTS

This chapter describes the experiment architecture, focusing on addressing the concepts of cloud-native architecture in conjunction with OGC APIs. Component diagrams are primarily used to describe the structure, while process, deployment and other diagrams were used on demand. Another chapter, on TIE experiments, lists all interactions among components which were used to validate the approach.

6.1. Key Features of a Cloud-Native Application Architecture

“Cloud native” is a design paradigm governed by the Cloud Native Computing Foundation (CNCF). Cloud native applications can be defined as applications running in a containerized environment, capable of moving in and out of the cloud, and that scale horizontally based on varying workloads. They are deployed as microservices, abstracted to underlying infrastructure, and delivered through a DevOps pipeline. Often, cloud-native applications are orchestrated by an orchestration infrastructure. Following are the characteristics as defined by CNCF:

Containerization	Cloud native applications are infrastructure agnostic and use containers. Containers provide the application with a lightweight runtime, libraries, and dependencies that allow the application to run as a stand-alone environment able to move in and out of the cloud —independent of the nuances of a certain cloud provider’s virtual servers or computer instances. With this, containers are able to increase mobility between different environments.
API Driven	The term means to ensure compatibility across platforms and simplify complexity by using APIs. The most common API pattern used in the industry is RESTful API. RESTful APIs are a component of loosely coupled architecture. REST systems interact through standard operations on resources, and do not rely on the implementation of interfaces.
Microservices	The term “microservice” is a design paradigm used in cloud-native applications to break down large components into multiple stand-alone deployable parts. The component is divided and deployed into smaller functional units. This way, maintenance in a module does not affect the other functionalities, since they can independently work on their own.

Horizontal Scaling	A key property of cloud-native applications is the ability to horizontally scale in and out based on the size of their payload.
Service Mesh	A service mesh is a dedicated infrastructure layer for handling service-to-service communication. The service mesh is responsible for the reliable delivery of requests through the potentially complex network topology of services. In practice, the service mesh is implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware.
Delivery Pipeline	The state-of-the art software delivery pipeline is supported by the concepts of Continuous Integration/ Continuous Delivery (CI/CD). CI/CD has the following stages/components: a) Source code is stored in a Repository b) Each time new code is pushed to a Repository, CI is automatically triggered to create a build (image). c) The images are then stored in a Registry or Storage Bucket. d) Through CD, images are pulled from the registry for deployment. e) CD then pushes the image to the target environment (e.g., the Kubernetes cluster).

6.2. Component Description Template

This section provides a template for formal description of components involved in the API experiments. The template is structured based on the following characteristics:

- Functional Description
- Implemented, supported or required OGC APIs
- Used data sources
- Technology Stack
- Components, libraries, configurations

- Development paradigm/process/approach:
  - Model Drives Software Development (MDSD), Domain Specific Language (DSL), Code/API generation?
  - Microservices? Service mesh?
  - Cloud native design/implementation?
- CI/CD:
  - Pipeline
  - Containerization
  - Container orchestration
  - Cloud deployment

7

COMPONENT DESIGN AND IMPLEMENTATION DETAILS

7 COMPONENT DESIGN AND IMPLEMENTATION DETAILS

7.1. D165 API Experiments Server (Python)

The API Experiments Server implementation comprised separate implementations of OGC API — Features (Features) and OGC API — Environmental Data Retrieval (EDR). Both server implementations were developed in the Python programming language and around the Flask web framework. Flask provides the core capabilities to implement web interfaces. The basic implementation of both OGC API Standards is supported by the use of code generators. The model classes and controller classes (server stubs) are automatically generated from the respective OpenAPI specifications of Features and EDR by using the OpenAPI Generator project. OpenAPI Generator offers a specific code generator that produces basic Flask-based executable server applications.

The source code, corresponding documentation and deployment instructions are available on Github.

7.1.1. OGC API — Features Server

The architecture of the Features implementation is structured into modules. The modular approach enables different types of data backends to be easily integrated in a similar way. Within this task, connectivity's for OGC Web Feature Service (WFS) and Elasticsearch backends were implemented.

7.1.1.1. Core Architecture

- **Controller:** Controller classes accept API requests, choose the correct backend to handle a request and return the result to the client. The controller classes (empty stubs) are generated by the OpenAPI code generator along with the model classes.
- **Request transformer:** A request transformer is specific to the backend type. Request transformers convert a request to the OGC API — Features into a corresponding request to the specific data backend. For example a request to the */items* endpoint is converted into a WFS *GetFeature* request.
- **Query transformer:** A query transformer is a backend specific implementation that translates query parameters into the domain specific language of the backend. For example a query transformer translates the datetime filter parameter of the */items* endpoint into a WFS filter expression.

- **Format transformer:** A format transformer is optionally interposed if a data backend has no support for the GeoJSON format.
- **Backend:** A backend stores the configuration for an instance of a backend type (e.g URL of a WFS instance) and determines which type of request transformer has to be used for that specific backend type.

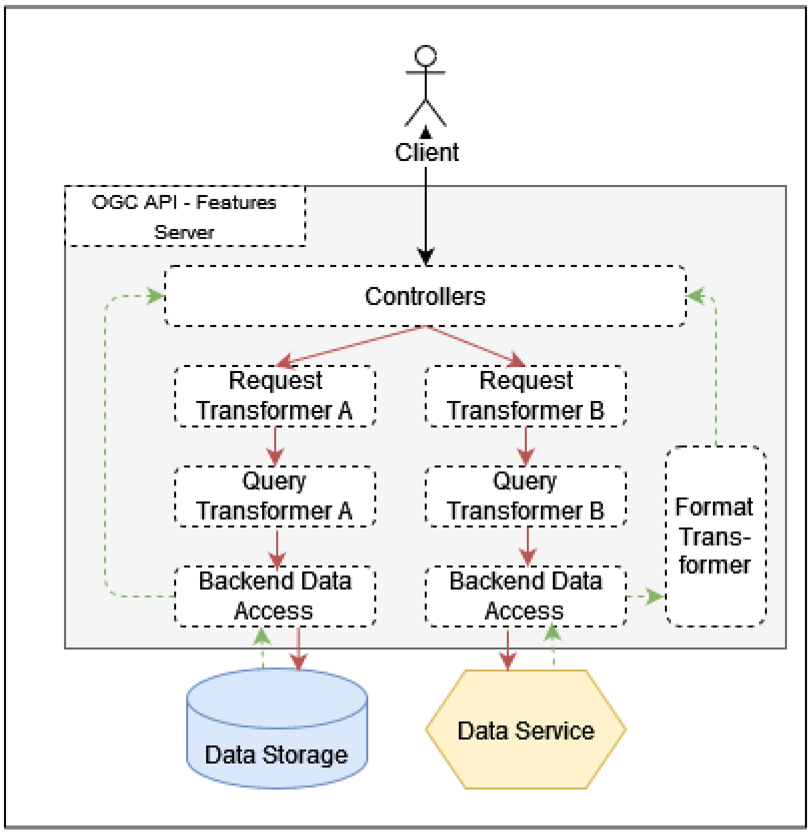


Figure 7 — schematic representation of the architecture of the OGC API — Features implementation with two data backends

7.1.1.2. Data Backends

7.1.1.2.1. OGC Web Feature Service Backend

The WFS backend can be used to create an OGC API — Features wrapper for an arbitrary WFS (2.x) instance. A request to the */items* endpoint is converted to a *GetFeature* WFS request. The metadata for the collection endpoints of Features is automatically obtained and parsed from the WFS capabilities document.

7.1.1.2.2. Elasticsearch Backend

The Elasticsearch backend implementation is mainly meant to be used with the OGC API — Records GeoJSON features that were generated and indexed in the D168 Data Backend and Deployment task. With the Elasticsearch backend the Features implementation can be used as an OGC API — Records implementation at the same time since the API endpoints are identical. The Elasticsearch backend is able to connect to Amazon Web Service (AWS) Elasticsearch Service/OpenSearch Service instances. Therefore the Elasticsearch Python package version *7.13.4* must be used because newer versions of the package do not connect to the AWS flavor of Elasticsearch.

In general the Elasticsearch backend is able to serve generic GeoJSON features indexed in an Elasticsearch index. An Elasticsearch index corresponds to a collection. The assumption is made that the GeoJSON features match the general structure of the OGC API — Records definitions in order to implement the filter capabilities (e.g. datetime filter) of OGC API — Features.

7.1.2. OGI API — EDR (Environmental Data Retrieval) Server

In the same way as the Features implementation, the EDR implementation has a modular structure in order to enable seamless integration of different data backend types. Within this task the capability to perform data queries on NetCDF files was implemented. The EDR server application supports the *position*, *radius* and *area* data query endpoints of the EDR Standard. Though the OGC API — Features and — EDR implementations are separate applications the overall architecture is roughly the same.

When using the code generator there were originally many errors in the generated code for the EDR OpenAPI specification. The main reasons for the errors are the frequent use of the *anyOf* and *oneOf* keywords in the OpenAPI specification and the fact that the EDR specification is more comprehensive and complex compared to the Features specification. In order to mitigate these issues a custom, lean and simplified version of the OpenAPI EDR Standard was created. This custom version supports only two-dimensional, temporal data and only serves netCDF data as response. All data query endpoints that were not planned to be implemented — that means all except *position*, *radius* and *area* — were removed from the implementation to keep it as lean as possible. This resulted in fewer errors in the generated code which could be fixed manually afterwards.

7.1.2.1. Core Architecture

- **Controller:** Controller classes accept API requests, choose the correct backend to handle a request and return the result to the client. The controller classes (empty stubs) are generated by the OpenAPI code generator along with the model classes.
- **Request transformer:** A request transformer is specific to the backend type. A Request transformer converts a request to the OGC API — EDR interface into

a corresponding request to the specific data backend. A request transformer delegates the execution of data query requests to a data query transformer.

- **Data query transformer:** A query transformer is a backend specific implementation that executes data query requests for data served by a data backend.
- **Format transformer:** A format transformer is optionally interposed if a data backend has no support for the NetCDF format. Since the OGC API — EDR implementation only serves netCDF files and has only a NetCDF backend there are no format transformers used.
- **Backend:** A backend stores the configuration for an instance of a backend type (e.g location of a NetCDF file) and determines which type of request transformer has to be used for the specific backend type.

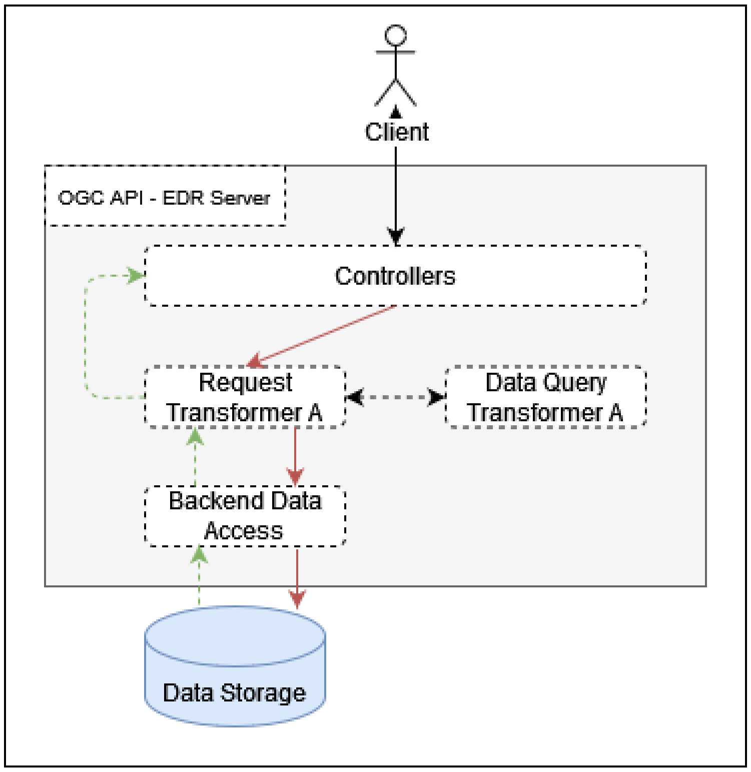


Figure 8 — schematic representation of the architecture of the OGC API — EDR implementation with a single data backend

7.1.2.2. Data Backends

7.1.2.2.1. netCDF backend

The NetCDF backend parses NetCDF files. Each file corresponds to an instance of a collection. The backend is adapted to the use of NetCDF land cover classification data provided by the

D168 Data Backend and Deployment task. The general assumptions about the internal structure of these files is that each file contains a single data array, which has the dimensions *x*, *y* and *time*.

To implement data queries the NetCDF backend implementation mainly uses the Python packages Rasterio, xarray and Shapely. In addition, a package called rioxarray is used. Rasterio is used to clip raster data to a specific geometry and xarray is used to implement filter capabilities like the optional datetime filter. The internal data structure of xarray resembles the structure of a NetCDF file. rioxarray combines the functionalities of Rasterio and xarray which supports convenient use without transformation between (internal) data formats. Shapely is used to parse WKT geometries that are passed to the data query endpoints as query parameters.

7.1.3. Overall Design Decisions

7.1.3.1. Separating Implementations of OGC API — Features and OGC API — EDR

The implementation of OGC API — Features and OGC API — EDR was split into two separate services that do not share any common code base. Integrating Features and EDR into a single API instance was considered but rejected.

Another approach that was also rejected was the creation of a Python package that contained the (common) data model classes and then imported by both applications. The decision for separate applications was based on the difficulties that arose with code generation and that both OGC API Standards contain a few deviations or different concepts despite many similarities in detail.

Regarding code generation, the results for OGC API — Features and especially OGC API — EDR already showed that code generation becomes more error prone with increasing complexity of the underlying OpenAPI specification. The generated code for OGC API — EDR contained more errors compared to the code for the (simpler) Features specification. This led to the necessity to simplify the OpenAPI definition document provided with the OGC API — EDR. An OpenAPI specification that integrates Features and OGC API — EDR would have been even more complex and therefore the generated code would have been flawed and hard to rectify.

From the conceptual point of view, there are some concepts that have similar names but the meaning is slightly different. For example both APIs specify the */collections/{collectionID}* endpoint to retrieve metadata for collection of data. In Features a collection is always a set of features while in OGC API — EDR Standard a collection can also be a coverage. In addition, in OGC API — EDR a collection can have multiple instances such as distinguishing multiple measurement series. These conceptual differences lead to slight differences in the data model. In OGC API — EDR a collection has additional attributes compared to the OGC API — Features Standard. As described, in OGC API — EDR a collection has multiple instances (a list of collections) and furthermore has supplemental attributes to describe measurement parameters.

A common data model for both implementations would also have had to meet the requirements of the OGC API — EDR Standard although some attributes are not required in the OGC API — Features Standard. This means that the same class of data model is used differently in different contexts. This can be seen as an anti-pattern. The OGC API Standards are not yet fully aligned with OGC API — Common, partly because OGC API – Common was still a candidate standard at the time of this testbed. As such, the data models have a few (minor) deviations. Since the first version of the OGC API — EDR Standard was only recently published and adjustments to the specification and data model are to be expected; this could also make a common code base hard to maintain.

7.1.3.2. Development and Deployment

Development, testing and deployment of both server applications was performed using a container-based platform for building applications called Docker. Docker simplifies service dependency management as the developer/user does not need to install dependencies on a local system (except the Docker engine itself). However, deploying and running the applications without Docker is still possible if all dependencies are installed manually. The decision to primarily use Docker stems from the need to have other software libraries installed in addition to simple Python packages. In particular the EDR implementation (respectively the used Python packages like Rasterio) require an installation of GDAL. While the Python package can easily be installed in a virtual environment with simple tools like Venv, the installation of GDAL needs more advanced tools like Conda. The assumption is that Docker is more known and better understood by most developers and users. In addition, with Docker the applications are completely separated from the environment and operation system.

In order to support development and production-ready deployment, the code repository contains Docker files and Docker Compose definitions with development and production profiles. The development profile uses the built-in Flask development server (which is not meant to be used in a production environment) and installs additional debugging dependencies. This allows developers to attach a remote debugger to the running Docker container and debug the application in the development environment of their choice (e.g. Visual Studio Code). The production profile does not contain any debugging dependencies and uses nginx and uWSGI instead of the development server and in order to serve requests fast and securely.

7.2. D166 API Experiments Server (JavaScript)

This section describes the server component as developed, documented and delivered by task D166 (JavaScript) and communicates all experiences and lessons learned back to the server provider to help optimizing documentation and scripts.

7.2.1. Development Environment

This implementation uses JavaScript to develop servers of OGC API-Features and API-Environmental Data Retrieval (EDR). The base system is as follows.

- Debian-based Linux systems (Debian 10 or higher or Ubuntu 18.04 or higher)
- NodeJS >= 10.6

- NPM >= 6.10.0
- GDAL 2 or higher
- PostgreSQL 12+/Proj 6+/PostGIS 3+(optional)

7.2.2. Architecture

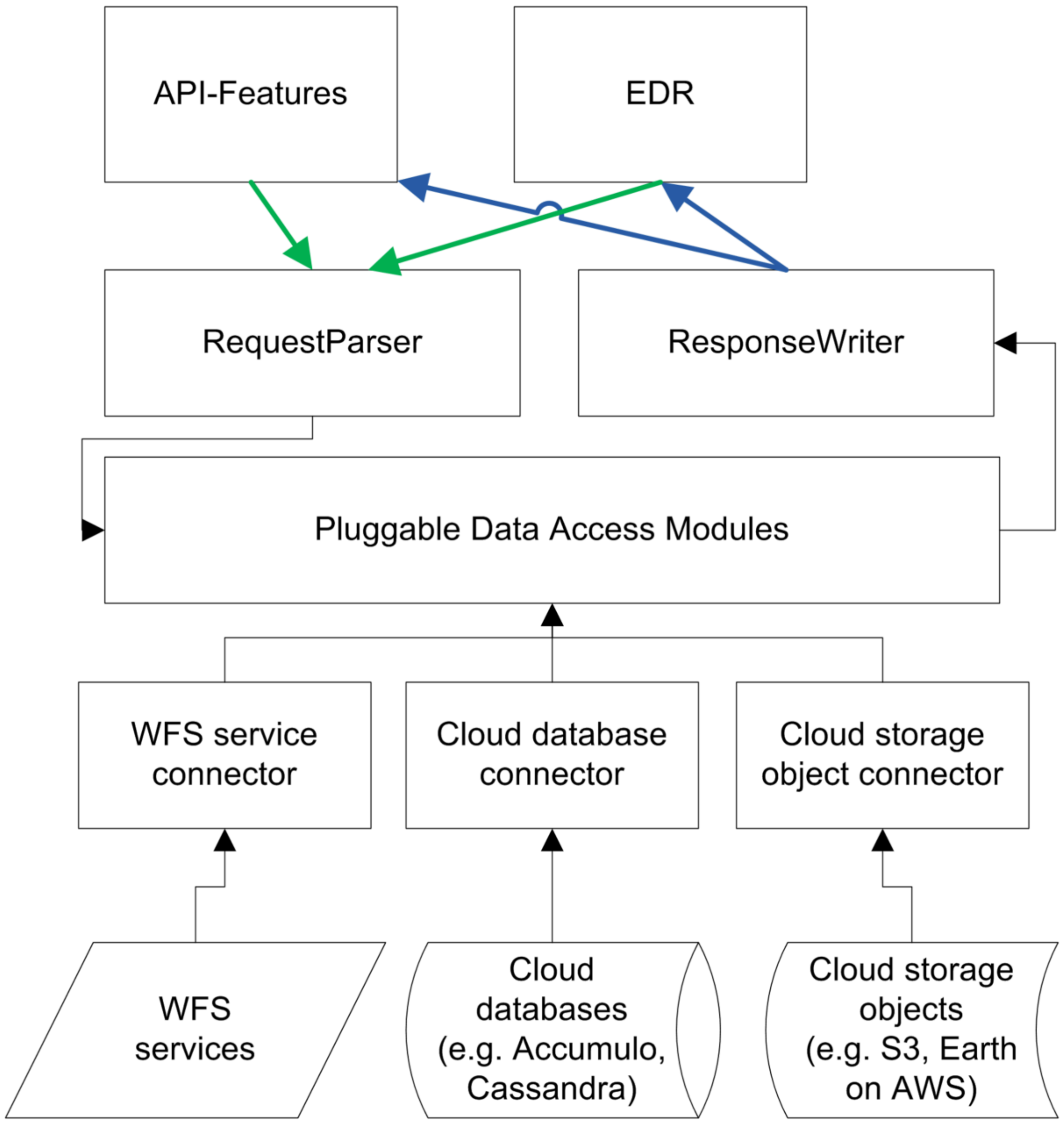

API-Features
EDR
RequestParser
ResponseWriter
Pluggable Data Access Modules
WFS service connector
Cloud database connector
Cloud storage object connector
WFS services
Cloud databases (e.g. Accumulo, Cassandra)
Cloud storage objects (e.g. S3, Earth on AWS)

Figure 9 — Overall architecture

7.2.3. Deployment and Test Services

The demonstration service was deployed at https://aws4ogc17.webmapengine.com/ . This deployment is based on an Amazon Machine Image (AMI).

7.2.3.1. Deployment Through Source Code

This section contains information on how to deploy the JavaScript server implementation through cloning the source code with configuration.

7.2.3.1.1. Requirements of Virtual Machine

The following are tested and recommended system configurations:

- Ubuntu 18.04 or higher or Debian 10 (Buster) or higher
- NodeJS (>= 10.6) and NPM (>= 6.10.0)
- git
- GDAL 2 or higher
- PM2 Process Management
- HTTP server (e.g. Apache or nginx) for proxy

7.2.3.1.2. Cloning of Source

Use git to clone the source into a local work directory.

```
git clone https://github.com/opengeospatial/T17-API-D166.git
```

7.2.3.1.3. OGC API — Features

Change working directory to “T17-API-D166/features”. The following command will pre-install all required libraries.

```
npm install
```

Start the Service

Run the following command to start the server as a service.

```
sudo pm2 start index.js
```

Run the following command to stop the server.

```
sudo pm2 stop index.js
```

Configurations

Service Endpoints

Edit config.js to configure the running port and the external service endpoint. The external service endpoint can be different from the local server endpoint with port if the proxy is set up to redirect the service. If both services (API-Features and EDR) run on the same virtual machine, different ports should be configured.

Configuring Back-end PostGIS Database

Edit the file ‘data/collections.json’ to add a new back-end PostGIS database. The new element should be added under node ‘resources’. The following is one example -

```
"tl_2020_us_county": {
  "type": "collection",
  "title": "Tiger Line US Counties",
  "description": "Counties, US Census, TIGER/Line",
  "keywords": [
    "counties",
    "US",
    "TIGER/Line"
  ],
  "links": [
    {
      "type": "text/html",
      "rel": "canonical",
      "title": "information",
      "href": "https://www.census.gov/geographies/mapping-files/time-series/
geo/tiger-line-file.html",
      "hreflang": "en-US"
    }
  ],
  "extents": {
    "spatial": {
      "bbox": [
        -179.231086,
        -14.601813,
        179.859681,
        71.439786
      ],
      "crs": "http://www.opengis.net/def/crs/OGC/1.3/CRS84"
    }
  },
  "providers": [
    {
      "type": "feature",
      "name": "PostGIS",
      "data": "postgresql://tluser:tl2020@localhost:5432/tl",
      "id_field": "ogc_fid",
      "title_field": "name",
      "table": "tl_2020_us_county",
      "geometry":{
        "geom_field":"wkb_geometry",
        "geom_format":"ewkb"
      }
    }
  ]
},
```

Configuring Back-end WFS Service Endpoints

Edit the file ‘data/collections.json’ to add new back-end WFS Service Endpoints. The new element should be added under node ‘resources’. The following is one example to add one single feature collection -

```
"DC_Building_Footprints": {
  "type": "collection",
  "title": "DC Building Footprints",
  "description": "DC building footprints.",
  "keywords": [
    "DC",
    "US",
    "building",
    "footprint"
  ],
  "links": [
    {
      "type": "text/html",
      "rel": "canonical",
      "title": "information",
      "href": "https://cubewerx.pvretano.com/cubewerx/cubeserv/default/ogcapi/
usbuildingfootprints/collections/DC_Building_Footprints",
      "hreflang": "en-US"
    }
  ],
  "extents": {
    "spatial": {
      "bbox": [
        -77.115085,
        38.810444,
        -76.909707,
        38.99561
      ],
      "crs": "http://www.opengis.net/def/crs/OGC/1.3/CRS84"
    }
  },
  "providers": [
    {
      "type": "feature",
      "name": "WFS202",
      "id_field": "gml_id",
      "typename": "DC_Building_Footprints",
      "data": "https://www.pvretano.com/cubewerx/cubeserv?datastore=
usbuildingfootprints&"
    }
  ]
},
```

All feature collections may be added as the back-end service. The following resource configuration shows an example using WFS capabilities.

```
"daraa": {
  "type": "collections",
  "title": "CubeSERV WFS - Daraa",
  "description": "CubeSERV WFS - Daraa service, support version 2.0.2.",
  "keywords": [
    "WFS",
    "feature",
    "capabilities"
  ],
  "links": [
    {
      "type": "text/html",
      "rel": "canonical",
      "title": "information",
      "href": "https://www.pvretano.com/cubewerx/cubeserv?DATASTORE=
daraa&SERVICE=WFS",
      "hreflang": "en-US"
    }
  ],
  "providers": [
    {
      "type": "collection",
      "name": "WFS202Capabilities",
      "data": "https://test.cubewerx.com/cubewerx/cubeserv/demo?DATASTORE=
Daraa&SERVICE=WFS&REQUEST=GetCapabilities&AcceptVersions=2.0.2&AcceptFormats=
text/xml",
      "removeprefix":"cw"
    }
  ]
},
```

7.2.3.1.4. OGC API — Environmental Data Retrieval

Change working directory to “T17-API-D166/edr”. The following command will pre-install all required libraries.

```
npm install
```

Start the Service

Run the following command to start the server as a service.

```
sudo pm2 start index.js
```

Run the following command to stop the server.

```
sudo pm2 stop index.js
```

Configurations

Service Endpoints

Edit config.js to configure the running port and the external service endpoint. The external service endpoint can be different from the local server endpoint with port if the proxy is set up to redirect the service.

Configuring Back-end PostGIS Database

Edit the file ‘data/collections.json’ to add new back-end PostGIS database. The new element should be added under node ‘resources’. The following is one example -

```
"noaa_global_hourly_surface": {
  "type": "collection",
  "title": "The Integrated Surface Dataset (global, hourly)",
  "description": "The Integrated Surface Dataset (ISD) is composed of worldwide
 surface weather observations from over 35,000 stations, though the best
 spatial coverage is evident in North America, Europe, Australia, and parts of
 Asia. Parameters included are: air quality, atmospheric pressure, atmospheric
 temperature/dew point, atmospheric winds, clouds, precipitation, ocean waves,
 tides and more. ISD refers to the data contained within the digital database
 as well as the format in which the hourly, synoptic (3-hourly), and daily
 weather observations are stored. The format conforms to Federal Information
 Processing Standards (FIPS). ISD provides hourly data that can be used in
 a wide range of climatological applications. For some stations, data may go
 as far back as 1901, though most data show a substantial increase in volume
 in the 1940s and again in the early 1970s. Currently, there are over 14,000
 'active' stations updated daily in the database. For user convenience, a
 subset of just the hourly data is available to users for download. It is
 referred to as Integrated Surface Global Hourly data, see associated download
 links for access to this subset.",
  "keywords": [
    "Integrated Surface Dataset",
    "Global",
    "NOAA"
  ],
  "links": [
      {
        "type": "text/html",
        "rel": "canonical",
        "title": "information",
        "href": "https://www.ncdc.noaa.gov/isd",
        "hreflang": "en-US"
      }
  ],
  "extents": {
    "spatial": {
      "bbox": [
        -180.00,
        -90.00,
        180.00,
        90.00
      ],
      "crs": "http://www.opengis.net/def/crs/OGC/1.3/CRS84"
    },
    "temporal": {
      "interval": [{
          "begin":"1972-01-01T00:00:00Z",
          "end":"1972-12-31T23:59:59Z"
      }],
      "trs": "TIMECRS[\"DateTime\",TDATUM[\"Gregorian Calendar
\"],CS[TemporalDateTime,1],AXIS[\"Time (T)\",future]"
    }
  },
  "data_queries" : {
      "position": {
          "link": {
              "href": "/collections/noaa_global_hourly_surface/position?coords=
{coords}",
              "hreflang": "en",
              "rel": "data",
              "templated": true,
              "variables": {
                  "title": "Position query",
                  "description": "Position query",
                  "query_type": "position",
                  "coords" :{
                      "description": "Well Known Text POINT value i.e. POINT(-
120, 55)"
                  },
                  "output_formats": [
                      "CoverageJSON",
                      "GeoJSON",
                      "IWXXM"
                  ],
                  "default_output_format": "IWXXM",
                  "crs_details": [
                      {
                          "crs": "CRS84",
                          "wkt": "GEOGCS[\"WGS 84\",DATUM[\"WGS_
1984\",SPHEROID[\"WGS 84\",6378137,298.257223563,AUTHORITY[\"EPSG
\",\"7030\"]],AUTHORITY[\"EPSG\",\"6326\"]],PRIMEM[\"Greenwich
\",0,AUTHORITY[\"EPSG\",\"8901\"]],UNIT[\"degree\",0.
01745329251994328,AUTHORITY[\"EPSG\",\"9122\"]],AUTHORITY[\"EPSG\",\"4326\"]]"
                      }
                  ]
              }
          },
          "linkrel":"relative"
      },
      "radius": {
          "link": {
              "href": "/collections/noaa_global_hourly_surface/radius?coords=
{coords}",
              "hreflang": "en",
              "rel": "data",
              "templated": true,
              "variables": {
                  "title": "Radius query",
                  "description": "Radius query",
                  "query_type": "radius",
                  "coords" :{
                      "description": "Well Known Text POINT value i.e. POINT(-
120, 55)"
                  },
                  "output_formats": [
                      "CoverageJSON",
                      "GeoJSON",
                      "IWXXM"
                  ],
                  "default_output_format": "GeoJSON",
                  "within_units": [
                      "km",
                      "miles"
                  ],
                  "crs_details": [
                      {
                          "crs": "CRS84",
                          "wkt": "GEOGCS[\"WGS 84\",DATUM[\"WGS_
1984\",SPHEROID[\"WGS 84\",6378137,298.257223563,AUTHORITY[\"EPSG
\",\"7030\"]],AUTHORITY[\"EPSG\",\"6326\"]],PRIMEM[\"Greenwich
\",0,AUTHORITY[\"EPSG\",\"8901\"]],UNIT[\"degree\",0.
01745329251994328,AUTHORITY[\"EPSG\",\"9122\"]],AUTHORITY[\"EPSG\",\"4326\"]]"
                      }
                  ]
              }
          },
          "linkrel":"relative"
      },
      "area": {
          "link":                  {
              "href": "http://www.example.org/edr/collections/hrly_obs/area?
coords={coords}",
              "hreflang": "en",
              "rel": "data",
              "templated": true,
              "variables": {
                  "title": "Area query",
                  "description": "Area query",
                  "query_type": "area",
                  "coords" :{
                      "description": "Well Known Text POLYGON value i.e.
 POLYGON((-79 40,-79 38,-75 38,-75 41,-79 40))"
                  },
                  "output_formats": [
                      "CoverageJSON",
                      "GeoJSON",
                      "BUFR",
                      "IWXXM"
                  ],
                  "default_output_format": "CoverageJSON",
                  "crs_details": [
                      {
                          "crs": "CRS84",
                          "wkt": "GEOGCS[\"WGS 84\",DATUM[\"WGS_
1984\",SPHEROID[\"WGS 84\",6378137,298.257223563,AUTHORITY[\"EPSG
\",\"7030\"]],AUTHORITY[\"EPSG\",\"6326\"]],PRIMEM[\"Greenwich
\",0,AUTHORITY[\"EPSG\",\"8901\"]],UNIT[\"degree\",0.
01745329251994328,AUTHORITY[\"EPSG\",\"9122\"]],AUTHORITY[\"EPSG\",\"4326\"]]"
                      }
                  ]
              }
          }
      },
      "locations": {
          "link": {
              "href": "/collections/noaa_global_hourly_surface/locations",
              "hreflang": "en",
              "rel": "data",
              "templated": false,
              "variables": {
                  "title": "Location query",
                  "description": "Location query",
                  "query_type": "locations",
                  "output_formats": [
                      "application%2Fgeo%2Bjson",
                      "text%2Fhtml"
                  ],
                  "default_output_format": "application%2Fgeo%2Bjson",
                  "crs_details": [
                      {
                          "crs": "CRS84",
                          "wkt": "GEOGCS[\"WGS 84\",DATUM[\"WGS_
1984\",SPHEROID[\"WGS 84\",6378137,298.257223563,AUTHORITY[\"EPSG
\",\"7030\"]],AUTHORITY[\"EPSG\",\"6326\"]],PRIMEM[\"Greenwich
\",0,AUTHORITY[\"EPSG\",\"8901\"]],UNIT[\"degree\",0.
01745329251994328,AUTHORITY[\"EPSG\",\"9122\"]],AUTHORITY[\"EPSG\",\"4326\"]]"
                      }
                  ]
              }
          },
          "linkrel":"relative"
      }
  },
  "crs": [
      "CRS84"
  ],
  "output_formats": [
      "application/geo+json",
      "text/html"
  ],
  "parameter_names": {
  },
  "providers": [
    {
      "type": "feature",
      "name": "PostGIS",
      "data": "postgresql://noaauser:noaa2020@localhost:5432/noaa",
      "id_field": "ogc_id",
      "locid_field": "station",
      "title_field": "station",
      "table": "noaa_global_surface",
      "geometry":{
        "geom_field":"the_geom",
        "geom_format":"ewkb"
      },
      "time_field":{
        "time_format":"datetime",
        "datetime":"date"
      }
    }
  ]
},
```

Configuring Back-end WFS Service Endpoints

Service instances can be added as back-end. This is especially useful for those with "instances" path.

7.2.4. Running Through Docker

Docker images have to be created for a quick-start of the servers.

7.2.4.1. Running OGC API — Features Through Docker

The following command runs the service directly using Docker:

```
docker run -p 8080:8080 -d eugenegmu/ogc-api-features-javascript
```

To test the server, you may browse to http://localhost:8080 to test the results. This image does not have the PostGIS set up locally. The first three collections do not work properly. Configuration needs to be done if a proper PostGIS database is set up with populated data. If a port is redirected to a different port other than 8080, the configuration needs to be updated. To get into the image, the following command may be used.

```
docker exec -it <container id or name> /bin/bash
```

The container ID or name can be found by running "docker container ls".

7.2.4.2. Running OGC API — EDR Through Docker

The following command runs the service directly using Docker:

```
docker run -p 8080:8080 -d eugenegmu/ogc-api-edr-javascript
```

To test the server, browse to http://localhost:8080/edr/ to test the results. This image does not have the PostGIS set up locally. Again, configuration needs to be done with a proper PostGIS database set up with populated data. If a port is redirected to a different port other than 8080, the configuration needs to be updated. To get into the image, the following command may be used.

```
docker exec -it <container id or name> /bin/bash
```

The container ID or name can be found by running “docker container ls”.

7.2.5. Test Services

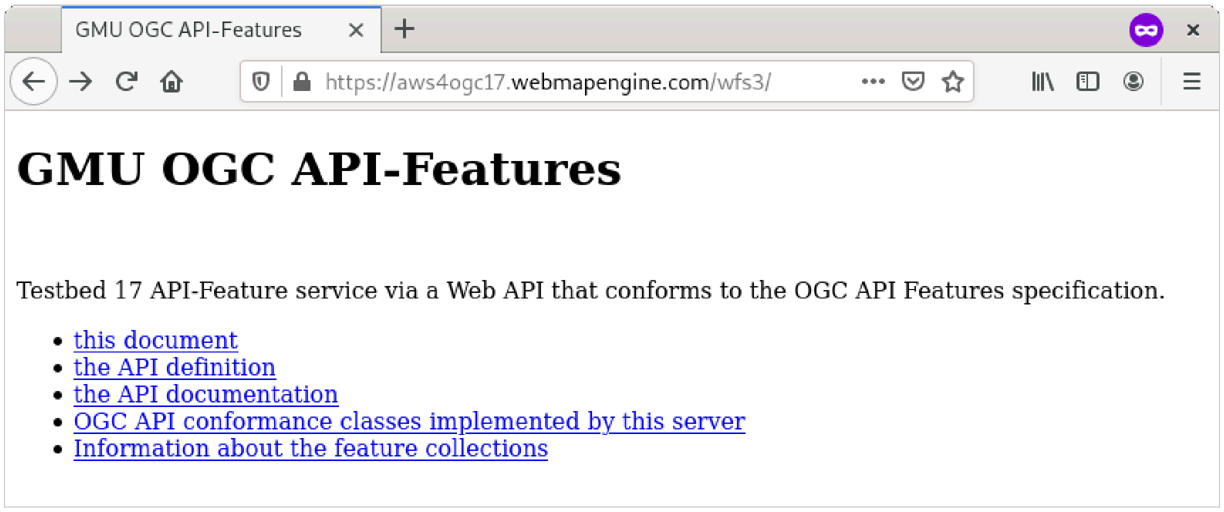
The demonstration deployments can be found at Demonstration Services for OGC Testbed 17.

7.2.6. OGC API — Features Test Service

The test service is deployed at API-Features.

Landing Page

The implementation of the OGC API — Features Standard supports two media content types by default: application/json and text/html. Encoding negotiation is supported. For browsers, “text/html” would be the matching format. The following figure shows one example landing page.



GMU OGC API-Features

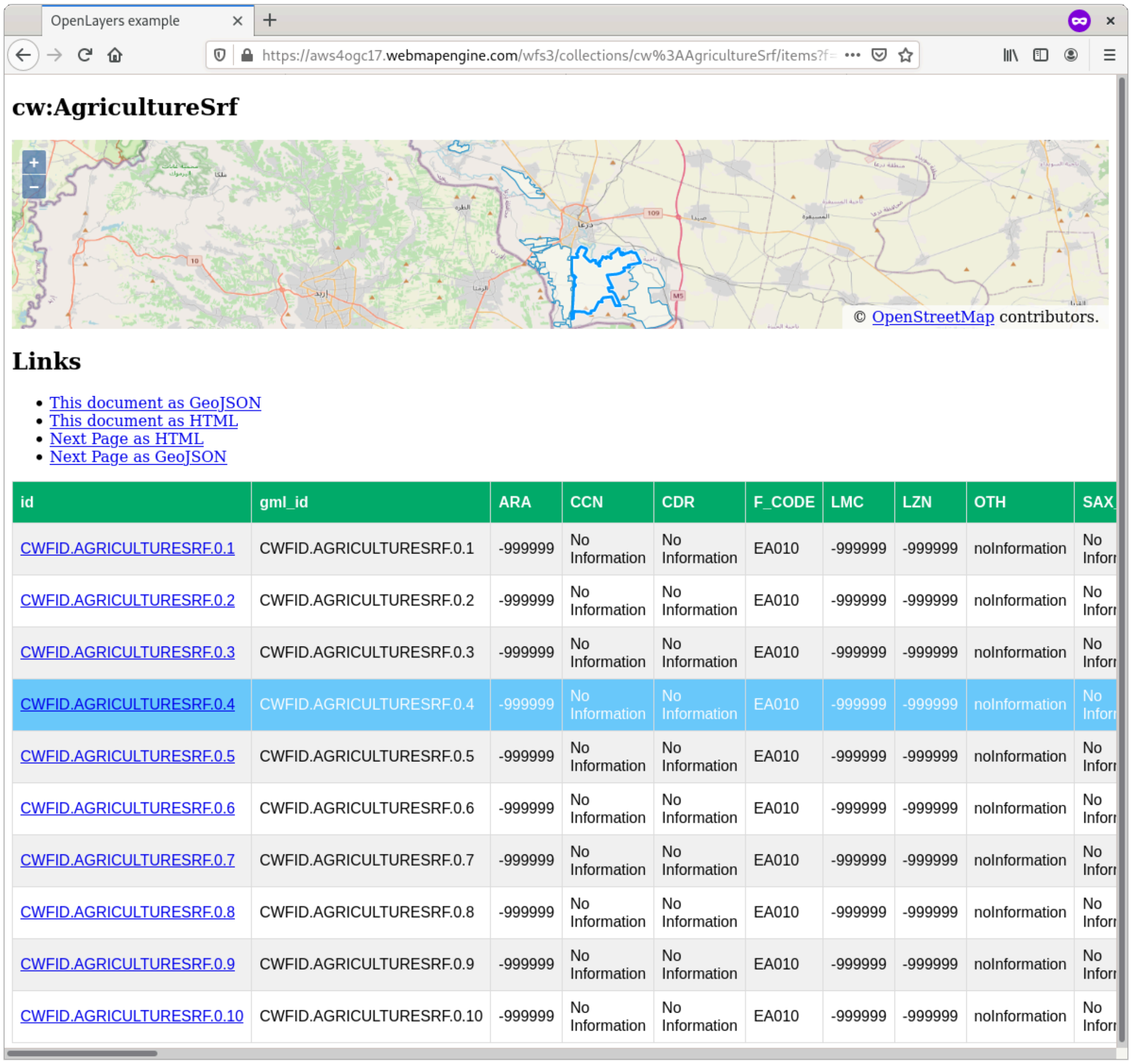
Testbed 17 API-Feature service via a Web API that conforms to the OGC API Features specification.

- this document
- the API definition
- the API documentation
- OGC API conformance classes implemented by this server
- Information about the feature collections

Figure 10 — Landing page of the demonstration OGC API — Features server

Testing API using SwaggerUI: The SwaggerUI is enabled for users to test each API path. This is the response of an API document in text/html.

Features in a collection: The response for collection data supports encodings in both application/geo+json and text/html. The text/html shows a linked map and table for browsing items by pages. The following shows one example of a collection in the browser.


OpenLayers example

https://aws4ogc17.webmapengine.com/wfs3/collections/cw%3AAgricultureSrf/items?f=

cw:AgricultureSrf

© OpenStreetMap contributors.

Links

- This document as GeoJSON
- This document as HTML
- Next Page as HTML
- Next Page as GeoJSON

id	gml_id	ARA	CCN	CDR	F_CODE	LMC	LZN	OTH	SAX
CWFID.AGRICULTURESRF.0.1	CWFID.AGRICULTURESRF.0.1	-999999	No Information	No Information	EA010	-999999	-999999	noInformation	No Infor
CWFID.AGRICULTURESRF.0.2	CWFID.AGRICULTURESRF.0.2	-999999	No Information	No Information	EA010	-999999	-999999	noInformation	No Infor
CWFID.AGRICULTURESRF.0.3	CWFID.AGRICULTURESRF.0.3	-999999	No Information	No Information	EA010	-999999	-999999	noInformation	No Infor
CWFID.AGRICULTURESRF.0.4	CWFID.AGRICULTURESRF.0.4	-999999	No Information	No Information	EA010	-999999	-999999	noInformation	No Infor
CWFID.AGRICULTURESRF.0.5	CWFID.AGRICULTURESRF.0.5	-999999	No Information	No Information	EA010	-999999	-999999	noInformation	No Infor
CWFID.AGRICULTURESRF.0.6	CWFID.AGRICULTURESRF.0.6	-999999	No Information	No Information	EA010	-999999	-999999	noInformation	No Infor
CWFID.AGRICULTURESRF.0.7	CWFID.AGRICULTURESRF.0.7	-999999	No Information	No Information	EA010	-999999	-999999	noInformation	No Infor
CWFID.AGRICULTURESRF.0.8	CWFID.AGRICULTURESRF.0.8	-999999	No Information	No Information	EA010	-999999	-999999	noInformation	No Infor
CWFID.AGRICULTURESRF.0.9	CWFID.AGRICULTURESRF.0.9	-999999	No Information	No Information	EA010	-999999	-999999	noInformation	No Infor
CWFID.AGRICULTURESRF.0.10	CWFID.AGRICULTURESRF.0.10	-999999	No Information	No Information	EA010	-999999	-999999	noInformation	No Infor

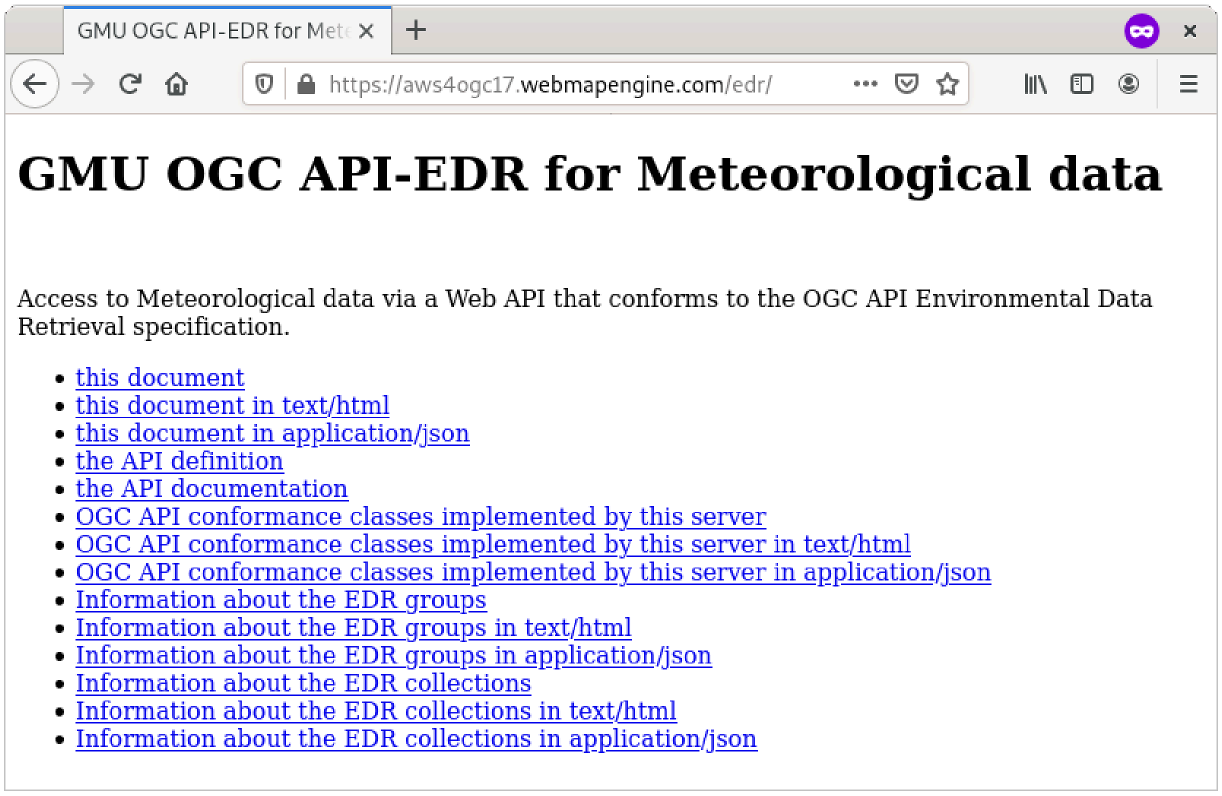
Figure 11 — Collection item page of the demonstration OGC API — Features server

7.2.7. Test Service for OGC API — EDR

The test service is deployed at API-EDR.

Landing Page

The implementation of the API — EDR supports two formats by default: application/json and text/html. Encoding negotiation is supported. For browsers, "text/html" would be the matching format. The following figure shows one example landing page.



GMU OGC API-EDR for Meteorological data

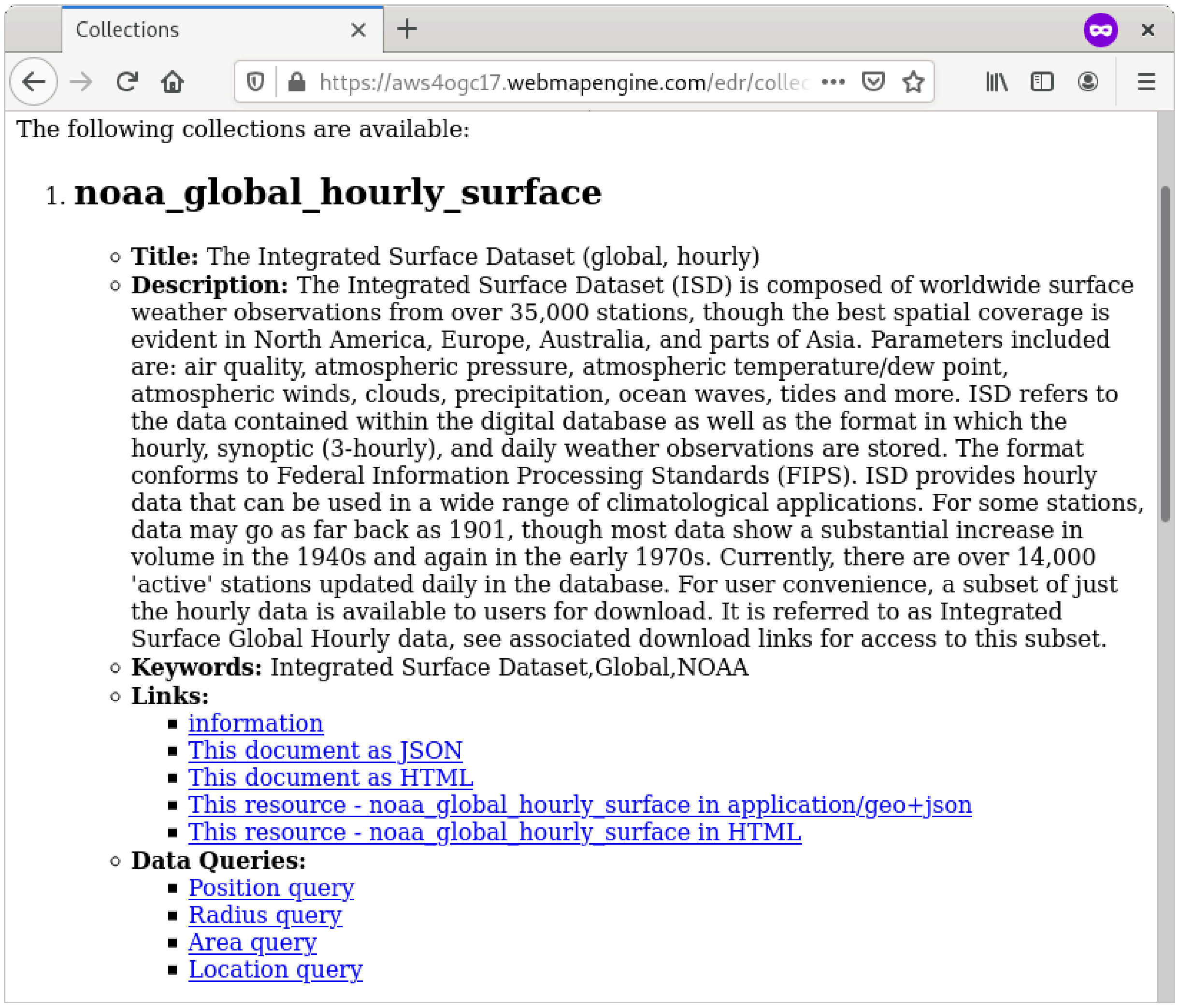
Access to Meteorological data via a Web API that conforms to the OGC API Environmental Data Retrieval specification.

- this document
- this document in text/html
- this document in application/json
- the API definition
- the API documentation
- OGC API conformance classes implemented by this server
- OGC API conformance classes implemented by this server in text/html
- OGC API conformance classes implemented by this server in application/json
- Information about the EDR groups
- Information about the EDR groups in text/html
- Information about the EDR groups in application/json
- Information about the EDR collections
- Information about the EDR collections in text/html
- Information about the EDR collections in application/json

Figure 12 — Landing page of the demonstration OGC API — EDR server

Testing API using SwaggerUI: The SwaggerUI is enabled for users to test each API path. This is the response of an API document in text/html.

Data query in a collection: Several data queries are supported for each collection in the API-EDR service. The following figure shows one example collection which supports queries of position, radius, area, and location.

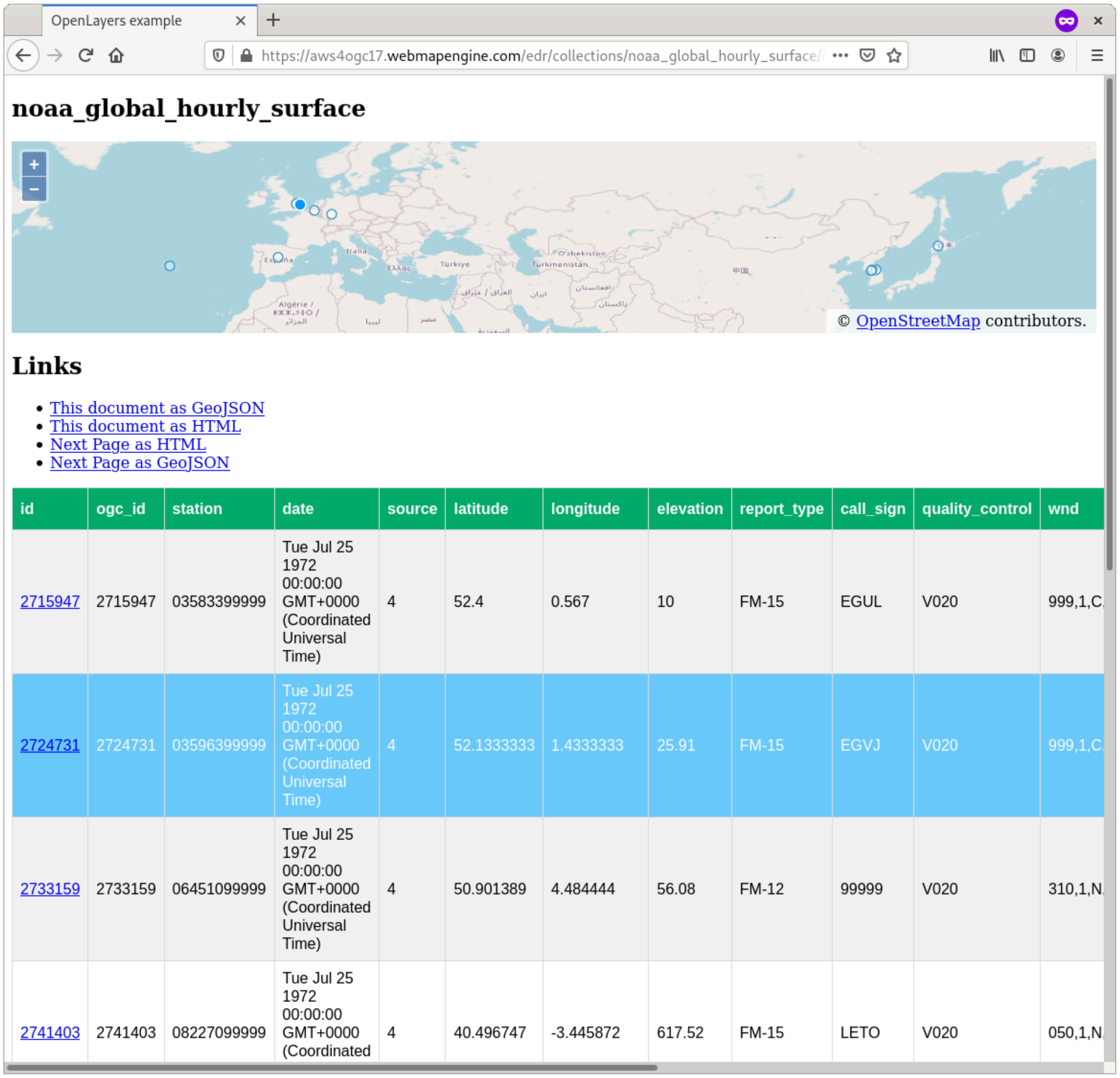


The following collections are available:

1. **noaa_global_hourly_surface**
   - **Title:** The Integrated Surface Dataset (global, hourly)
   - **Description:** The Integrated Surface Dataset (ISD) is composed of worldwide surface weather observations from over 35,000 stations, though the best spatial coverage is evident in North America, Europe, Australia, and parts of Asia. Parameters included are: air quality, atmospheric pressure, atmospheric temperature/dew point, atmospheric winds, clouds, precipitation, ocean waves, tides and more. ISD refers to the data contained within the digital database as well as the format in which the hourly, synoptic (3-hourly), and daily weather observations are stored. The format conforms to Federal Information Processing Standards (FIPS). ISD provides hourly data that can be used in a wide range of climatological applications. For some stations, data may go as far back as 1901, though most data show a substantial increase in volume in the 1940s and again in the early 1970s. Currently, there are over 14,000 'active' stations updated daily in the database. For user convenience, a subset of just the hourly data is available to users for download. It is referred to as Integrated Surface Global Hourly data, see associated download links for access to this subset.
   - **Keywords:** Integrated Surface Dataset,Global,NOAA
   - **Links:**
     - information
     - This document as JSON
     - This document as HTML
     - This resource - noaa_global_hourly_surface in application/geo+json
     - This resource - noaa_global_hourly_surface in HTML
   - **Data Queries:**
     - Position query
     - Radius query
     - Area query
     - Location query

Figure 13 — The Collections page of the demonstration OGC API — EDR server

The response for collection data supports encodings in both application/geo+json and text/html. The text/html shows a linked map and table for browsing features by pages. The following shows one example of a collection in the browser. The request is https://aws4ogc17.webmapengine.com/edr/collections/noaa_global_hourly_surface/items?f=text%2Fhtml&datetime=1972-07-25T00:00:00.000Z/1972-07-25T23:59:59.000Z



OpenLayers example

https://aws4ogc17.webmapengine.com/edr/collections/noaa_global_hourly_surface/

noaa_global_hourly_surface

© OpenStreetMap contributors.

Links

- This document as GeoJSON
- This document as HTML
- Next Page as HTML
- Next Page as GeoJSON

id	ogc_id	station	date	source	latitude	longitude	elevation	report_type	call_sign	quality_control	wnd
2715947	2715947	03583399999	Tue Jul 25 1972 00:00:00 GMT+0000 (Coordinated Universal Time)	4	52.4	0.567	10	FM-15	EGUL	V020	999,1,C
2724731	2724731	03596399999	Tue Jul 25 1972 00:00:00 GMT+0000 (Coordinated Universal Time)	4	52.1333333	1.4333333	25.91	FM-15	EGVJ	V020	999,1,C
2733159	2733159	06451099999	Tue Jul 25 1972 00:00:00 GMT+0000 (Coordinated Universal Time)	4	50.901389	4.484444	56.08	FM-12	99999	V020	310,1,N
2741403	2741403	08227099999	Tue Jul 25 1972 00:00:00 GMT+0000 (Coordinated	4	40.496747	-3.445872	617.52	FM-15	LETO	V020	050,1,N

Figure 14 — Collection item page of the demonstration OGC API — EDR server

7.2.8. Extensions to the Library

The implementation of the library for both API-Features and API-Environmental Data Retrieval supports extension through the following mechanisms: One is the extension to support different back-end data sources and another is the extension to support different response formats.

7.2.8.1. Add a new backend data source

The Model extension manages the backend data models. The Data provider extension needs to support query interfaces as defined. An example provider can be seen in PostGIS.js.

7.2.8.2. Support new format

The Writer extension manages the supported formats. Each writer needs to support both collection items and individual features. TextHtml.js is an example writer.

7.2.9. Lessons learned

The following cover discussions about the implementation of API -Features and API — EDR services using JavaScript.

- **Usability of OpenAPI-Generator**: The stubs generated by the OpenAPI-Generator provide a solid base for implementing the service. Minor modifications are necessary to work with specific data types.
- **Limitations of supporting libraries in pure JavaScript**: There are fewer libraries available in pure JavaScript. Many require a system call to libraries from modules or programs implemented in other languages, such as C/C++ and Python. This increases the complexity in deployment and running environment.
- **Working with different backend data sources**: Different backend data have different levels of support on realizing the interface functions. File-based dataset may require specific programs to support the spatiotemporal queries. Databases with spatial modules have full spatiotemporal query support with proper indexing. Legacy services, such as old versions of WFS, have various capabilities on supporting queries depending on their implementations.
- **Extensibility of API libraries**: Response format supports media type negotiation. Backend services are pluggables with different modules. Extensions are supported in two aspects — backend data source provider and output data.
- **Performances**: Certain profiled query-functions of API-EDR may not be easy to achieve ideal response time when a large dataset is in process. In the AWS cloud, one approach may be to set up a high-performance data service to serve data. For example, AWS RDS database service can be used to serve data. The PostGIS module can be configured to connect with AWS RDS PostgreSQL/PostGIS. The capability of handling big data can be achieved through a similar cloud-based database as the service.

7.3. D167 Data Backend and Deployment

The D167 Data Backend and Deployment consists of three different components:

a) A tiles server generator, generating the tiles from OpenStreetMap data and publishing them through an Apache Web Server

b) An OpenStreetMap dataset generator, storing the data in PostGIS datasets (in AWS, CloudFerro or CloudSigma) or as cloud objects in AWS S3 buckets

c) A system to publish array-oriented datasets in a tiled manner, through a STAC catalog and storing the resulting NetCDF files in AWS S3 buckets

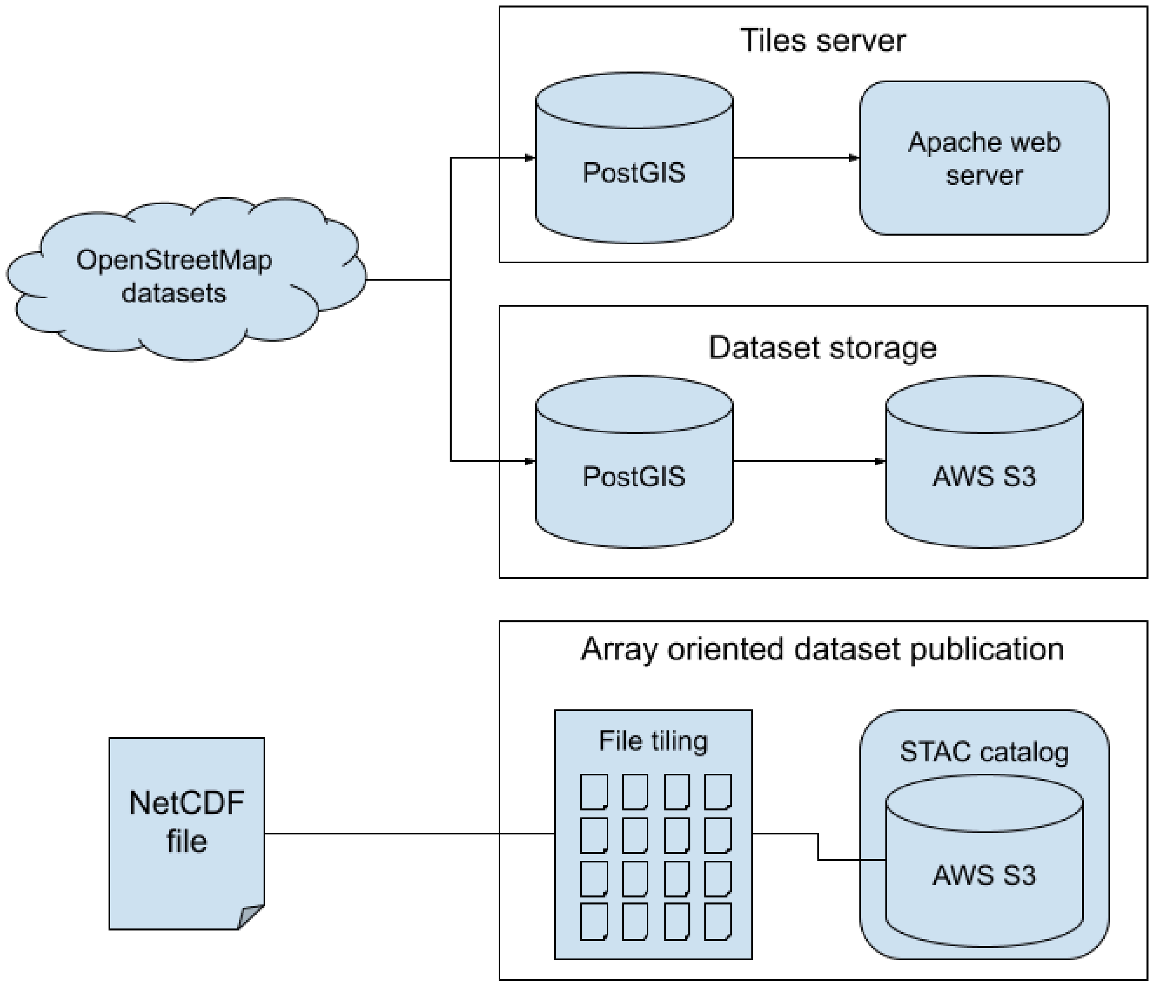


Figure 15 — Component diagram

The focus for all the components was in automation and ease of execution, following a DevOps approach and lowering the learning curve for new developers to deploy advanced environments. For example, the script for the tiles server generator covers the following actions:

a) Installs/compiles all the software dependencies.

b) Installs a PostgreSQL/PostGIS database and sets it up.

c) Downloads and configures a stylesheet.

d) Downloads OSM data for the specific country and inserts it into the database.

e) Downloads additional shapefiles and fonts.

f) Sets up Apache HTTP Server with rendered.

g) Launches Apache Server.

The process for deploying a tiles server using a Docker container was also tested and documented. In addition, a simple HTML/JavaScript client to test the tiles server was developed and added to the repository.

The idea of tiling NetCDF files was to experiment with solutions for data providers to offer array oriented datasets to service providers that can be consumed on demand or to select the data of interest based on the bounding boxes extension. As the concept of tiling is well understood in the geospatial world, instead of publishing a large NetCDF covering the whole world, the content can be tiled by levels, chopping the content into smaller chunks that are more easily transferable through the network. These smaller NetCDF files can be published through a STAC catalog that documents the extent of each file, allowing service providers to automate the search for the data of interest.

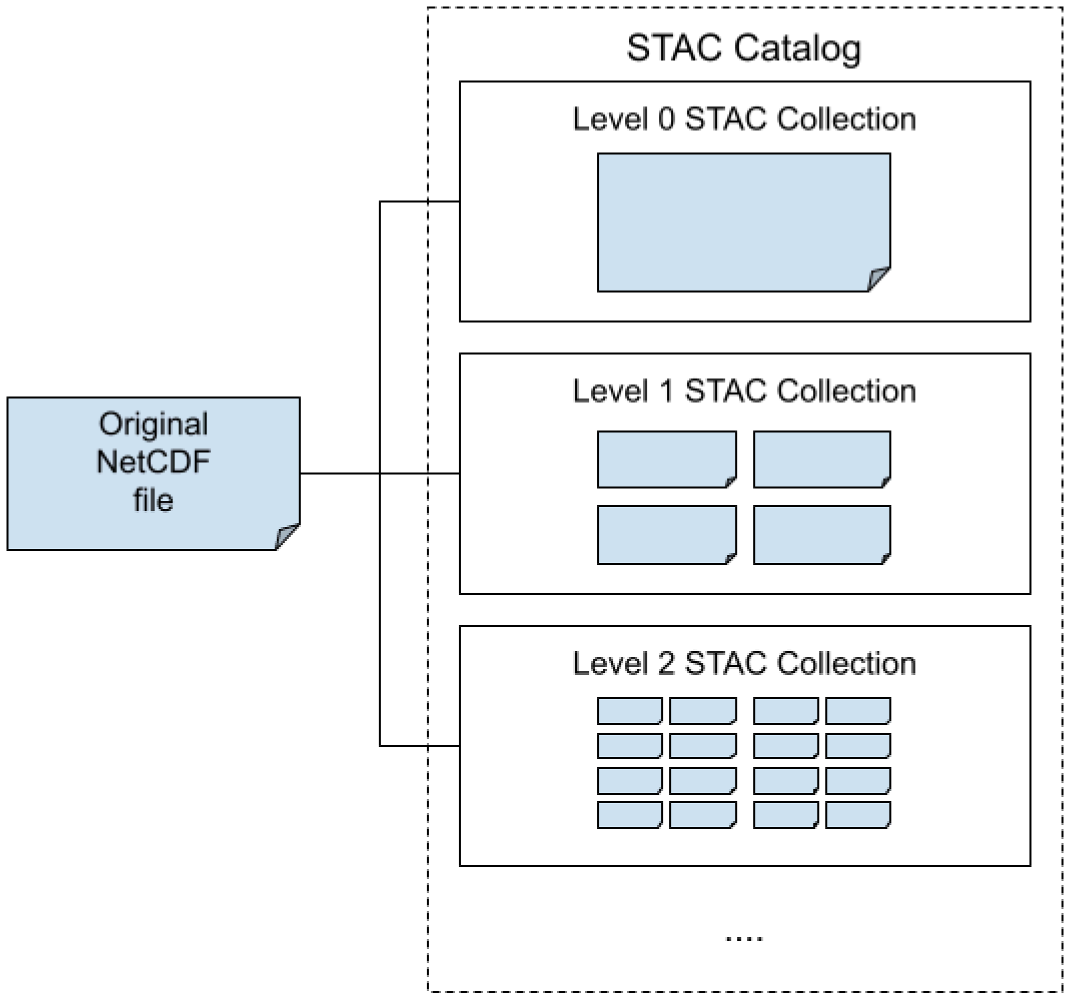


Figure 16 — NetCDF tiling concept

The source code and detailed documentation on how to deploy these components are available at https://github.com/opengeospatial/T17-API-D167. The datasets are deployed in the following endpoints:

Database Access

- AWS:
    - IP: 18.189.112.137
    - Port: 5432
    - User: apipostgres
    - Pwd: @P$_2021!
    - Database: dbapi4
    - Tables: geojson_places, geojson_waterways, geojson_polygons
- CloudSigma:
    - IP: 162.213.36.126
    - Port: 5432
    - User: dbasigma
    - Pwd: Db@_S$gma2021!
    - Database: gis
    - Tables: geojson_places, geojson_waterways, geojson_polygons
- CloudFerro:
    - IP: 185.178.85.176
    - Port: 5432
    - User: apipostgres
    - Pwd: Db@_F$rro2021!
    - Database: dbferro01
    - Tables: geojson_places, geojson_waterways, geojson_polygons
    - Example: psql -W -U apipostgres -p 5432 -h 185.178.85.176 dbferro01

AWS S3 buckets

- Polygons (2883 elements):
  - https://ogc-polygons.s3.us-east-2.amazonaws.com/
  - Examples: https://ogc-polygons.s3.us-east-2.amazonaws.com/332195921.json, https://ogc-polygons.s3.us-east-2.amazonaws.com/704263360.json
- Points (1776 elements):
  - https://ogc-points.s3.us-east-2.amazonaws.com/
  - Examples: https://ogc-points.s3.us-east-2.amazonaws.com/1143922944.json, https://ogc-points.s3.us-east-2.amazonaws.com/1242125200.json

Tiles server

- Area: Northeast Spain
- End point: http://3.143.92.80/tileserver/{z}/{x}/{y}.png
- Example: http://3.143.92.80/tileserver/0/0/0.png

7.3.1. Challenges and lessons learned

The scripting and automation approach used in completing this deliverable was to deploy datasets and services in the cloud. This approach provides numerous benefits such as:

a) The deployed datasets and their environments are similar, easing the maintenance effort.

b) Human errors are minimized.

c) The time required for deployment is greatly reduced.

d) Deep technical knowledge is not needed, lowering the learning curve and allowing for new developers to quickly become productive.

However, developing these scripts required considerably more effort than a single deployment and they also require more technical knowledge. Scripts need to be structured and be reusable, as well as handle errors or generic configurations.

Besides, configuring cloud services and storage needs to be carefully set up and tested, because an error or untidiness in a deployment might result in security risks or unnecessary high costs. These risks are multiplied when developing a script that will be reused, especially if users are different and have a lower technical knowledge than the original developer.

The databases were deployed in three different clouds: AWS, CloudSigma and CloudFerro. There were not substantial differences among them from the functional point of view. AWS seemed more usable but eventually all three offered a similar set of features.

The extraction and handling of OpenStreetMap data was more troublesome than initially expected, with complex SQL queries required to extract specific datasets, including coordinate reference system transformation. These queries are not an important challenge for a seasoned geospatial developer, but they might be a major blocker for newcomers.

The NetCDF tiling concept resulted in a relatively easy-to-understand concept and its implementation was simple enough. However, testing with 0 tiling levels, it required 10 times more storage space as the same data needed to be packed for all the levels, and each file had considerable overhead. Moreover, the structure of the STAC catalog became more complex. Instead of the original NetCDF file being an item in a STAC catalog, the tiling structure required a sub-catalog containing a collection for each tiling level which then would publish each tiled file as an item.

During the TIEs, the server API implementations were deployed in AWS following the instructions documented in the repositories. The technical level required for these deployments was adequate for an experienced IT professional, but it was too high for newcomers. As part of future work, documentation in the repositories should be improved to ease access for less experienced professionals.

The experience with Docker when deploying D165 and D166 server APIs was positive: The process was simple and easily encapsulated. However, in real-life environments the risks of over-architecting should be considered among different alternatives before making a decision on how to deploy a server API.

7.4. D168 Data Backend and Deployment

The aim of D168 was to be the provider of data and deploy a server component as developed, documented and delivered: Communicating all experience and lessons learned back to the server provider to help with optimizing documentation and scripts.

The testing involved storing the data in specific cloud environments, primarily AWS with some parallel testing of CreoDIAS, with a specific focus on cloud object storage. The API Experiments Server tested was D165.

7.4.1. Development & Experiments

7.4.1.1. Experiment 1: Data catalogs stored within AWS S3 bucket & OpenSearch

Pixalytics has an Earth Observation (EO) dataset that is currently stored in a directory/file structure on an AWS Expandable File Storage (EFS).

Copernicus Sentinel-1 and Sentinel-2 data were used to generate a land cover classification at 10-meter resolution to investigate sand dams as part of a project that focuses on sustainable sand extraction in Kenya. In addition, the datasets include other products derived from Sentinel-2 (vegetation related) and products derived from other satellite missions at coarser spatial resolutions.The dataset can be viewed online at https://voila.notebook.eo4sas.com/ as CaseStudy1 (one of three Jupyter Notebooks running as interactive webpages).

Within the framework of this activity, the dataset was transitioned to an accessible S3 bucket with the catalogue defined through the two options of STAC (https://stacspec.org/) and OGC API — Records. The image files were originally in GeoTIFF format but transitioned to Cloud Optimized GeoTIFF (COG) and NetCDF for the experiments.

This experiment involved five steps, with the code stored as shown in Figure 1:

a) Convert GeoTIFF files to COG and NetCDF format using `convert_gtiff.py` in *utils* folder

b) Create OGC Record or STACcatalogs using `create_catalog.py` in *build_catalog* folder of D168

c) Manual push of catalogs to AWS S3 buckets

d) Push catalogs to Elasticsearch using `upload_esearch.py` in *deploy_catalog* folder of D168

e) Deployment of D165 server providing API access to the catalogs with the specification in JSON files

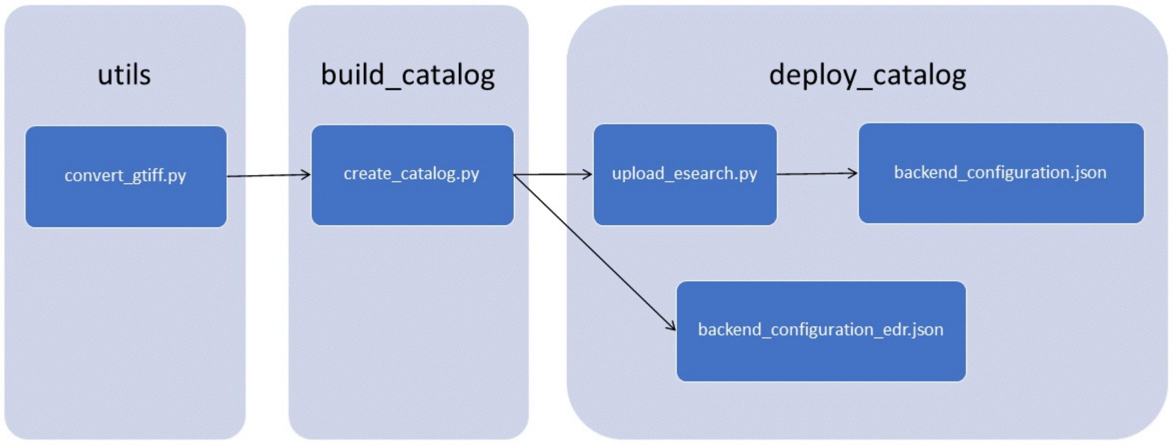


Figure 17 — Representation of the D168 GitHub repository with the code folders (light blue) and code components (dark blue)

7.4.1.1.1. Catalogs within AWS S3 bucket

The catalogs were created using Python code shared via the T17-API-D168 GitHub repository, under the *build_catalog* subfolder. The code uses yaml files (*test-configuration.yaml* and *eo4sas-record.yml*) to store configuration information, and rely on existing open-source code for STAC (pystac) and OGC API — Records (modified version of pygeometa to create the main catalog file) catalog creation. The GitHub ReadMe describes how to install the used anaconda environment and run the catalog creatin code.

Once created, the static catalogs are manually uploaded to the specifically setup S3 bucket that has public read access.

Examples of what is currently available via the S3 bucket include:

- STAC catalog v0-8 created using pystac for GeoTiFFs:
  - main JSON: https://pixalytics-ogc-api.s3.eu-west-2.amazonaws.com/eo4sas-catalog-stac-v0-8/collection.json
  - image JSONs, e.g. https://pixalytics-ogc-api.s3.eu-west-2.amazonaws.com/eo4sas-catalog-stac-v0-8/20200831T101156_rgb_classification/20200831T101156_rgb_classification.json
- STAC catalog v0-8 created using pystac for NetCDFs:
  - main JSON: https://pixalytics-ogc-api.s3.eu-west-2.amazonaws.com/eo4sas-catalog-stac-nc-v0-8/collection.json
- OGC API — Records catalog v0-8 created using pygeometa for GeoTiFFs:
  - main JSON: https://pixalytics-ogc-api.s3.eu-west-2.amazonaws.com/eo4sas-catalog-records-v0-8/catalog.json
  - image JSONs e.g.: https://pixalytics-ogc-api.s3.eu-west-2.amazonaws.com/eo4sas-catalog-records-v0-8/eo4sas-record1.json
- OGC API — Records catalog v0-8 created using pygeometa for NetCDFs:
  - main JSON: https://pixalytics-ogc-api.s3.eu-west-2.amazonaws.com/eo4sas-catalog-records-nc-v0-8/catalog.json

7.4.1.1.2. Catalogs deployed via Amazon OpenSearch (previously called Elasticsearch) Service with D165 Server

To support querying the catalogs, they were also deployed to Elasticsearch using Python code shared via the T17-API-D168 GitHub repository, under the *deploy_catalog* subfolder. The code uses a YAML file (*es_upload_conf.yaml*) for configuration of the deployment and then upload to Elasticsearch.

At the start of the Testbed 17 activity, OpenSearch was called Elasticsearch but since then the two have diverged. To maintain compatibility with what was developed the AWS deployment to Elasticsearch 7.10 was used. Also, version 7.13.4 of the Python Elasticsearch library needs to be used as more recent versions produce critical error messages.

- Elasticsearch endpoint on AWS [needs an IAM user account with authentication for access]: https://search-ogc-t17-d168-yhvlgzft2zhuvdssiaejkyq5lq.eu-west-2.es.amazonaws.com
    - STAC Catalog indices (for version 0-8): STAC-index (GeoTIFFs) STAC-index-nc (NetCDFs)
    - OGC Records indices (for version 0-8): records-index (GeoTIFFs) records-index-nc (NetCDFs)

7.4.1.1.3. Deployment using D165 server

The deployment as an API using the D165 server is available at:

- OGC API — Features server with three catalogs (CubeWerx's alongside Elasticsearch versions of Records and STAC GeoTiFF catalogs): http://ogcapi.pixalytics.com:8080/
- OGI API — EDR implementation with a single multi time-step NetCDF: http://ogcapiedr.pixalytics.com:8080/

7.4.1.2. Experiment 2: Catalogs created and deployed using the CloudFerro CreoDIAS platform

As a second test, the D168 code was also run on the CreoDIAS platform. The data was still stored in the AWS S3 bucket, but pulled to CreoDIAS and the catalogs generated and deployed using the D165 server. These deployments have not been kept active so cannot be accessed.

7.4.2. Lessons learned

A summary of lessons learned is:

- Existing GitHub repositories for STAC (pystac) and OGC API Records (pygeometa) were helpful to both understand the catalog structure and implement the Python code to generate the catalogs.
- To create the Records catalog, API weekly discussions alongside the documentation in ogcapi-records repository were very helpful.
- For AWS, a combination of the online documentation and virtual training/events was used to get the necessary knowledge to setup the required IAM permissions and Amazon OpenSearch configuration.
- For CreoDIAS, an introductory webinar was attended and then the Virtual Machine setup following the provided guidance and what was already understood from using AWS.

- At the start of the activity Pixalytics had not deployed catalogs. From what was learnt during the API Experiments, the participants feel confident about future deployment of catalogs using the approaches tested.

7.5. D175 API Experiments Client — Python

The D175 API Experiments Client — Python implements three different OGC APIs, each available in its own GitHub repository:

- OGC API — Features, available at https://github.com/opengeospatial/T17-API-D175-Features
- OGC API — EDR, available at https://github.com/opengeospatial/T17-API-D175-EDR
- OGC API — Routes, available at https://github.com/opengeospatial/T17-API-D175-Routes

All of the implemented APIs share a common structure and base code, as well as an automated and encapsulated deployment process that pulls the code from the respective repository. These implementations were conceived as data-centric clients, focusing the efforts in making them easy and quick to deploy and providing immediate access to the data.

Experiments were carried out in the auto-generation of client libraries for OGC API — Features and — EDR, based on the definition of a specific server API. The software used to auto generate the client libraries was OpenAPI Generator, in particular the python client (experimental). One of these experiments, auto generated against CubeWerx's Daraa API, has been registered in its own repository and is available for deployment and tests at https://github.com/opengeospatial/T17-API-D175-Features_autogenerated.

Each client implementation consists of two different components:

a) A backend, built in Python 3 on top of Flask, a popular micro web framework. The backend implements the OGC API client calls to interact with the API server and fetch the data.

b) A frontend, built in HTML/JavaScript, receiving the interactions from the user through a web browser and translating those to the client backend, as well as visually displaying the data fetched from the server.

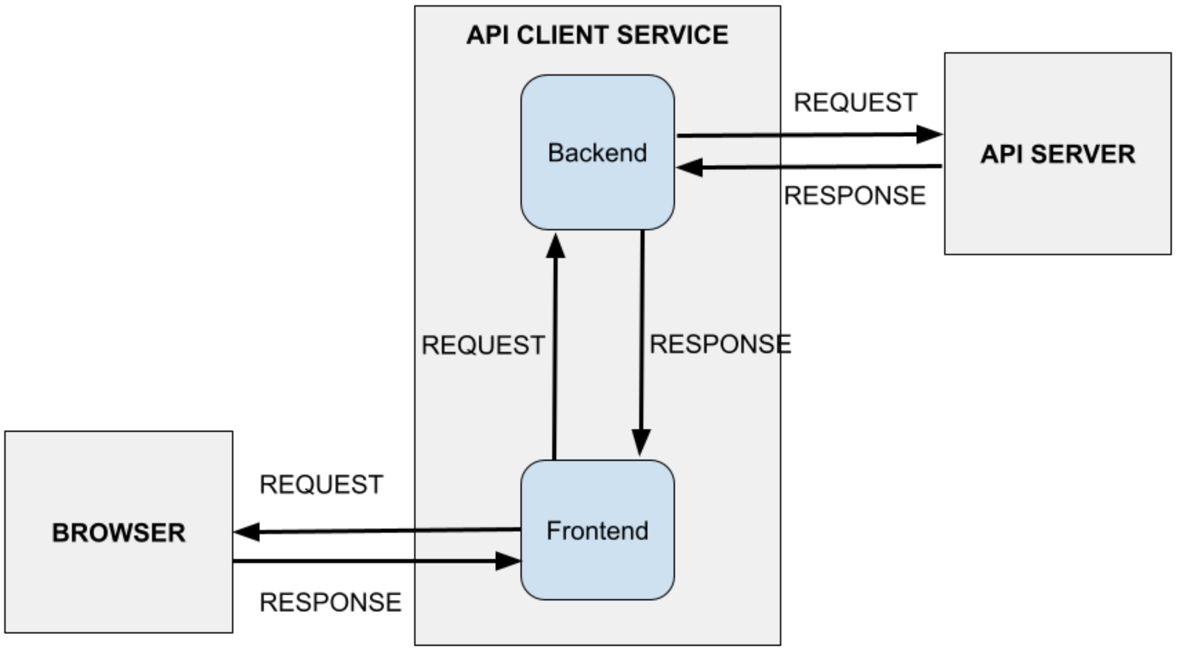


Figure 18 — Client architecture

This two-layered architecture enables separating the browser-client interface from the client-server interface, providing a greater flexibility in the design of the user interface. Clients can be deployed locally, but also remotely, at a different location than the API server or the user's browser.

Clients use two common design patterns with this framework: The application dispatching pattern and the application as a package. This allows for a clean directory structure and readable code, making the different clients easy to maintain and deploy. Each client is launched in its own Python virtual environment, installing its own version of the required libraries without messing with the existing installation in the operative system.

Once the client is launched, it automatically reaches the server landing page and fetches all the important information, which is offered to the user via buttons, texts and drop-down menus. The goal is for the user to explore the API capabilities in a graphical and convenient way.

The User Interface for all clients have similar designs, with a main map to display results, a list of buttons to easily explore the API and a series of buttons and fields to make the most common queries. This user interface is stored in HTML, CSS and JavaScript files with a very simplistic design, but there is no limit as to the level of sophistication they can attain.

7.5.1. Challenges and lessons learned

7.5.1.1. Client auto-generation

Although in theory code auto-generation seems like a very interesting idea, the reality is that it is an error-prone process that requires a well-defined API and might not be worth the effort. This is particularly true considering that implementing a REST API client is not particularly difficult. The main challenges encountered in the client auto-generation can be summarized in the following points:

a) The OpenAPI Generator is not really stable. The latest versions are too buggy and can crash in many ways. For our case, OpenAPI Generator v4.0.0 was considered stable enough and could be useful but it was yet not perfect. There were still some minor bugs, which could be mitigated by editing the API definition.

b) Server API definition must comply with OpenAPI v3. The generator is particularly picky with numeric data types.

c) OpenAPI Generator has extensive parameterization and it requires a lot of documentation reading before being able to use it.

d) Generated libraries can be used only with the target API.

e) Changes in the target API can easily break the compatibility with auto generated clients.

f) Reality check: OpenAPI Generator is a good idea in theory but in reality, there are plenty of APIs with incomplete definitions, typos in OpenAPI docs, inconsistent definitions or published data are inconsistent with OpenAPI definitions.

On the other hand, client auto-generation did provide important benefits:

a) It is an API-oriented approach, the focus shifts away from code.

b) It moves towards standardization of the API structure and methods across programming languages.

c) It is easy to expand to other OGC APIs.

d) It is easy to maintain/update existing APIs.

e) Documentation and unit tests can be auto generated along with the source code.

7.5.1.2. Data-centric clients

A second important set of lessons learned came from the experiments with data-centric clients. Such clients are more focused on quick access to the data than on the feature completion.

The main challenges in the development of these clients were to set up an installation in a properly isolated environment, as well as the implementation of the invisible automation that allows the immediate visualization of data. The use of standard OGC API building blocks is instrumental in solving the latter.

However, there was an aspect that showed some limitations in the current definition of OGC API standards. The approach was for the client to automatically connect to a server API, explore it and populate the user interface with all the options and capabilities offered through the API. Having a standard and easy way to request from the server API landing page which methods and encodings are accepted before making the first request would be very useful. This could be done using the HTTP method OPTIONS and could facilitate the automation of API exploration.

When carrying out the integration experiments, some approaches to implementing the server APIs added extra complexity in the client side. For example, although `type` is not a property strictly required for link objects, it saves considerable implementation time when developing a client that automates the API exploration. A significant amount of effort had to be spent in adding the code necessary to handle link objects that did not include a `type` property that could lead to requests for non-GeoJSON formats. An interesting improvement would be to require the property `type` in all link objects, or at least to recommend its use in the standard documentation.

Alternatively, the client could have added the format in the request with the parameter `f=json` to ensure the format of the response. However, there is no specification for the value of this parameter in the standard and, in fact, other participants used different values in their server implementations, such as `f=application/json` and `f=application/geo+json`. The proper way would be to access the API definition, if it exists, and find the possible values for parameter `f`, but the automation for this is cumbersome.

Overall, data-centric clients have proven an interesting approach, providing the following benefits:

a) All deployed environments are similar, even for different types of APIs and clients, which facilitates maintenance.

b) Human error factors are minimized during the environment configuration and deployment.

c) The time required to complete a deployment is minimal, which makes it useful for test environments or pilots.

d) Users do not need advanced technical knowledge to set up the environment or deploy a client, lowering the entry barriers, and facilitating the entry-level for new developers to OGC APIs.

e) Clients are light-weight and performant and allow for an automated exploration of APIs, which can become a handy tool for experienced OGC server developers and deployers to test their deployments or visually explore the data they publish.

7.6. D176 API Experiments Client — TypeScript

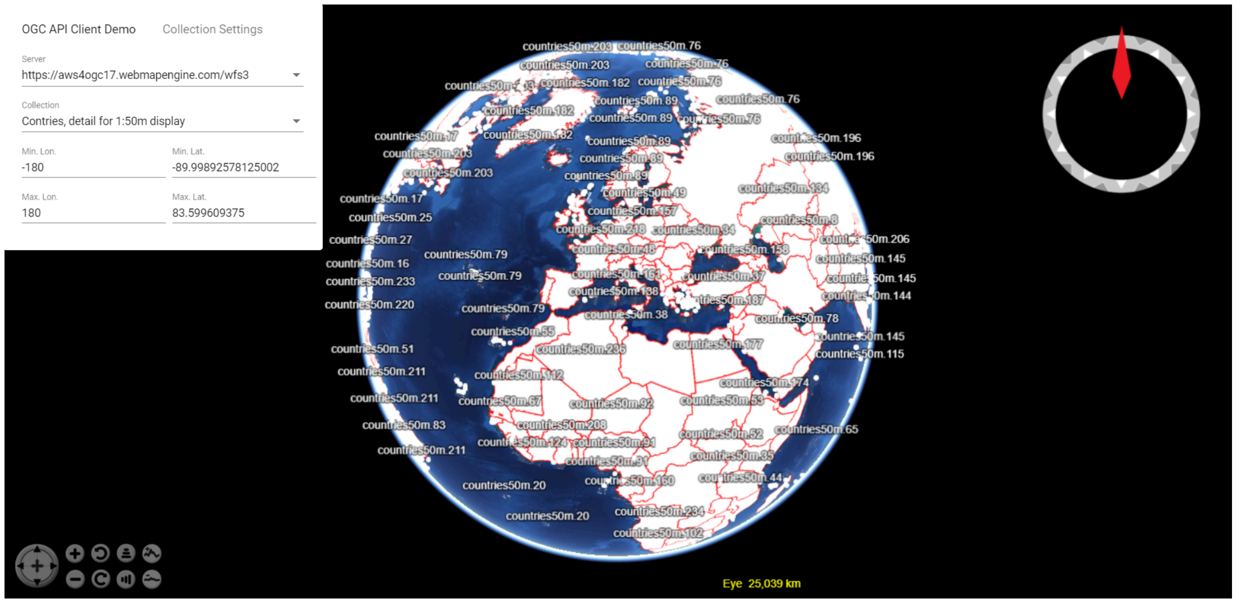
The TypeScript API Experiment client consists of three main components:

a) The API clients that are generated based on the corresponding OpenAPI specifications. (https://github.com/opengeospatial/T17-API-D176)

b) A connector for Web WorldWind that ties in the API responses into visual layers on WorldWind. (https://github.com/karriojala/WebWorldWind-OGCTB17)

c) A demonstrator that shows the API client's capabilities in a real-world use case integrating with WorldWind, which bridges to plain JavaScript usage. (https://github.com/opengeospatial/T17-API-D176-dev)

The demonstrator is available to try here: https://ogctb17-apis-breithorn.solenix.ch/#/earth


OGC API Client Demo Collection Settings

Server: https://aws4ogc17.webmapengine.com/wfs3

Collection: Contries, detail for 1:50m display

Min. Lon.: -180
Min. Lat.: -89.99892578125002
Max. Lon.: 180
Max. Lat.: 83.599609375

Figure 19 — Demonstrator application using NASA WebWorldWind

The most interesting aspects about this experiment are:

- **Execution in a Browser**
  Browsers exhibit particular challenges for the execution and communication with remote services as they are executed in the restricted JavaScript sandbox in a

browser. Considerations such as HTTP vs. HTTPS access, Cross-Origin Resource Sharing (CORS) and limitations in processing of data are to be considered.

- **Exploration of the API Client** in a real world scenario, using Web WorldWind.
- **TypeScript** with strong typing that significantly improves the available tooling for developers.

TypeScript is suitable for NodeJS environments and browsers, which spans a wide range of use cases.

The demonstration and discussed use cases focus on front-end development and visualizations in browsers, and less on complex processing or workflows of the results. Consequently, the developed use cases demonstrate the integration and common data request patterns for such web-based applications.

In addition to the OGC APIs Client source code, the documentation accompanying the source code provides examples for common use cases and explains the utility of specific end points and collections.

An important aspect of getting started with the OGC APIs is to get the initial bearings and identify where to find specific data, to learn the names and concepts used, and to learn how to explore available data sets. The documentation touches on these points as well, guiding a new user in getting started with the OGC APIs, data providers and data sets alike.

7.6.1. API Clients and Code Generation

This component focused on exploration and experimentation with OGC API — Features and OGC API — EDR.

Following the findings for generating code based on OpenAPI specifications for OGC APIs and considering the goal of creating a generic client that follows the standards as closely as possible, having an independent client API for each of the supported OGC API Standards was the final decision.

The back-end used was the `typescript-angular` generator, which a strongly typed client library that uses Angular's `HttpClient` and various reactive patterns in the backend. This backend was chosen to demonstrate the generic use of code generators and as excellent fit for the angular based demonstrator.

The Web WorldWind API is plain JavaScript with basic (`require.js` based) dependency management and no awareness of TypeScript or Angular. The extension to WorldWind was thus made in a way that allows consuming resources retrieved via the APIs, independent of the specific client. Consequently, a developer could also generate a pure JavaScript or non-Angular TypeScript client that would be able to provide those same responses, which could be rendered appropriately by WorldWind.

Finally, the OpenAPI specification provided by the Daraa-data server that is hosted by CubeWerx was used to exercise a compound API client, which can be found as `api-daraa` in the API clients repository.

7.6.2. OGC Code Sprint and Discrete Global Grid Systems

During October 2021 OGC API Code Sprint the client was extended with a demonstration for retrieving and visualizing DGGS (Discrete Global Grid Systems) zones. For this purpose, auto-generated TypeScript client was used in the same way as with OGC API — EDR and OGC API — Features.

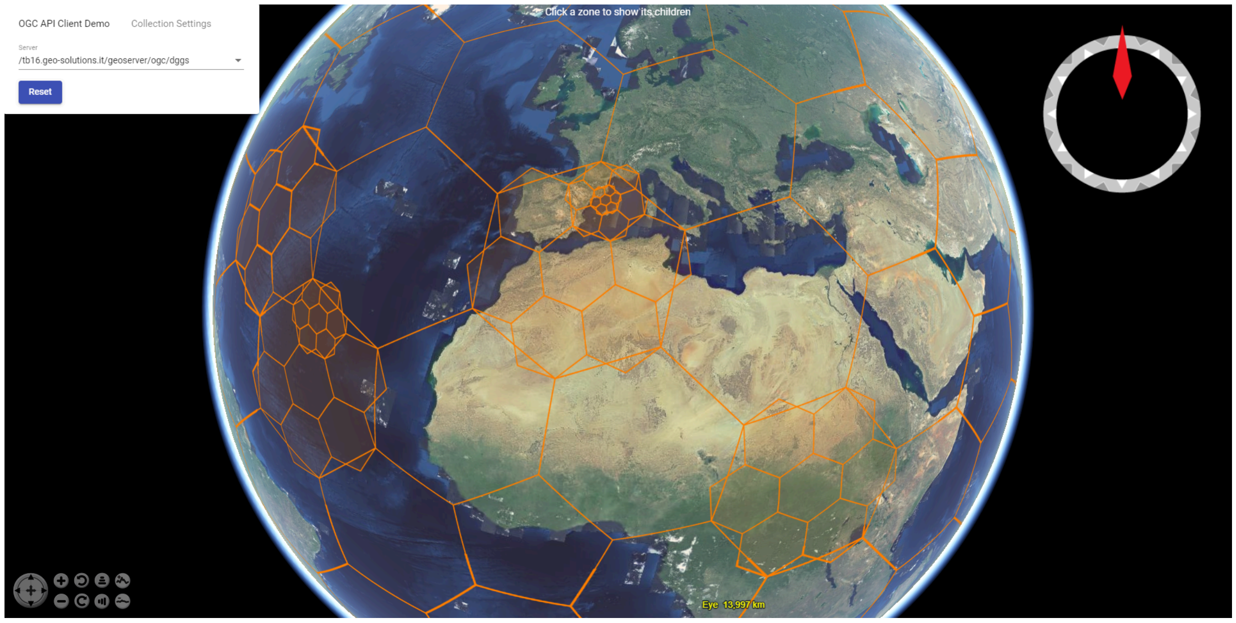

OGC API Client Demo
Collection Settings
Server
/tb16.geo-solutions.it/geoserver/ogc/dggs
Reset
Click a zone to show its children
Eye 13,997 km

Figure 20 — Demonstrator application showing DGGS zones

The top-level H3 zones were retrieved from the server and visualized on the globe. Clicking on the zones loads the children of the zones.

This demonstrates that code-generation can be a quick way to extend the capabilities of an existing client to support new APIs.

7.6.3. Technology Integration Experiments

The client is integrated with the API servers from George Mason University (GMU) and 52°North (52N). For the former, both OGC API — EDR and OGC API — Features were integrated. For the latter, only retrieval of data in NetCDF format is not yet supported on the client. Users can select the server using a drop-down menu where the configured endpoints are displayed. The client proxies the 52N server to add HTTPS compatibility.

The following functionality was implemented:

- Select server.

- Select collection.
- (EDR) Select location.
- (EDR) Select parameters (temperature, wind etc.).
- (EDR) Select time (slider between temporal extend start and end).
- (EDR) Data is displayed as a point on the map with a number label.
- (Features) Define bounding box.
- (Features) Data is displayed as polygons, lines or points.

Issues detected were related to inconsistencies with bounding box definition format varying between collections, and the temporal extent of the EDR collections not fully matching the available data.

The generated APIs work largely out of the box with the servers and helped to reduce the implementation effort and focus on the data visualization demonstration.

7.6.4. Future improvements

Further improvements can be made to improve quality of presentation:

- Support for paginating the data (instead of just showing first 1000 entries)
- Better visualization of EDR data (colors, plots, etc.)
- Support for NetCDF data visualization

The testbed participants also considered whether to publish the source code in a dedicated WorldWind fork and make it available for users and developers. This action would require clarification of which parts of the source code would be the most beneficial for the community. The generated code for APIs could also be published in the npm package Registry for JavaScript.

8

OPENAPI CODE GENERATION

8 OPENAPI CODE GENERATION

The OpenAPI specification language, previously called *Swagger*, can be used to express the details of a RESTful API.

Resource paths, examples, data payload schemas, mandatory and optional parameters, as well as alternative representations can be expressed using OpenAPI.

The OGC APIs to be used in this activity are all backed by corresponding OpenAPI specifications. Because the APIs are very versatile and the OpenAPI specifications are comprehensive, the participants in this testbed activity explored the use of code generators for creating server and client-side code with the goal of accelerating development, easing new developers into the OGC APIs and providing example client and server implementations that can be used to explain concepts, yet cover the full range of capabilities.

The following sections discuss the use of OpenAPI in the OGC and the findings, issues, workarounds and recommendations that were the result of the work described in this ER.

8.1. Overview of OpenAPI

There are two goals for using OpenAPI:

a) Create comprehensive documentation of an intended communication interface.

b) Automatic generation of server and client-side code stubs as well as data transfer objects or structs.

The first point aids in the design, discussion and formalization of APIs during their inception and can be refined or adapted based on later implementation details.

The second point is particularly useful for creating compatible and comprehensive clients in a new programming language without requiring one to develop the boilerplate code for communication, message exchange, data representation, encoding and other well-understood challenges.

The OGC has embraced OpenAPI for documenting the new generation of OGC APIs and has used OpenAPI to design, iterate and ratify the new versions of the respective successor APIs to WMS, WFS, WPS and others.

OpenAPI is also an excellent means of providing the currently applicable capabilities of a particular server, where the server can provide an OpenAPI document on one of its endpoints. This is mandated by the OGC API — Common candidate standard.

In the frame of this testbed activity, the focus was on the use of OGC API — Features and OGC API — EDR and in demonstrating in an understandable fashion how to use those APIs in a manner that is accessible to developers that are not necessarily familiar with geospatial processing in general or the new OGC APIs in particular.

The following sections discuss various aspects that were identified in the course of this activity.

8.2. Examples of Code Generation

OpenAPI based code generators are popular throughout the industry, wherever REST APIs are used and where OpenAPI has been used to formalize the interface.

Examples for successful use of generated clients are:

- The Kubernetes API clients that are available in a variety of languages. Similar to OGC API based servers, Kubernetes provides access to the *extensible* REST API that is currently available via OpenAPI specification endpoint.
- Cloud providers (Amazon, Google, Microsoft) provide REST APIs for communicating with their services. Software that interfaces with said services will often use generated communication stubs to interface with those services.

Large API surfaces — the part of a program that your users can openly interact with — having a large number of endpoints, data types, and/or implementations in different programming languages can benefit from code generation of the 'boilerplate' code to interface with a particular API. This leaves developers with more time to develop the added value by consuming or providing the services behind those APIs.

As mentioned in the introduction, the OGC API compliant servers are similarly set up with detailed APIs, which make it more difficult to create a comprehensive client implementation. Similarly, server implementations need to start somewhere as well and can benefit from a generated foundation.

8.3. Caveats with Generated Code and Code Generators

The OpenAPI specification is very accommodating for API designers and developers in getting their ideas across and provides many degrees of freedom. Some of the aspects that would be beneficial for a more formal specification are optional and can be left out.

There are a limited number of functional OpenAPI code generators, which in turn have a large number of different back-ends that support the programming language, framework and programming style of choice.

The generated code is generally self-contained with proper namespaces, which is the desired behavior for any client library, or starting point for a server implementation.

There are a few caveats with using generated code:

- A new self-contained set of code with namespaces is generated for each OpenAPI specification.
  In the case of the OGC APIs, the OGC API — Features and OGC API — EDR standards are two separate such specifications.
  Any declarations in the API, even if they were included from the same underlying fragment, become part of the specific namespace. A `Coverage` response data structure for `Features` and for `EDR` will become two different structs or classes. There was no automated way to consolidate such shared declarations at the time of writing this ER.

- Manually changed code may be destroyed on re-generation.
  Inheritance or composition can work around these issues, where the needed manual changes are added to the generated functionality.

- Not all of the features of OpenAPI are easily transferable to all programming languages.
  A specific case identified during this activity were responses with `anyOf` or `oneOf` declarations, which would require union types, a higher-level abstraction or appropriate ‘holders’ and ‘markers’ that indicate which of the possible data types was returned.

Finally, there were also caveats with using the code generators themselves:

- The OpenAPI specification, which started as Swagger and was originally developed by the commercial entity SmartBear, is now an open standard. Multiple implementations of code generators are available with various back-ends of varying completeness and maturity.

- Because programming languages have different functionality, paradigms, supporting runtime frameworks and varying degrees of developer participation, the generated code can vary significantly in terms of resilience, error detection, recovery and verbosity of error messages during code generation.

Fortunately, there were no issues identified with the software license of the resulting generated code, which should encourage experimentation and use of OpenAPI code generators.

8.4. Generating Code based on OGC OpenAPI Specifications

Code generators rely on consistent and clear instructions. Any ambiguity that would be fine for a human developer may trip up the code generator. This includes typos in declarations, missing links to other resources or incompatible data structure declarations.

Considering that the various OGC API Standards can be combined into a single server that provides the functionality of different APIs, some of the commonly available resources are shared or augmented with additional fields by the various APIs.

The approved OGC APIs, or those that are in development at the moment, are primarily aimed at human implementers. When using code generators, a few typos, inconsistent specifications of common endpoints and limitations of the code generators (as mentioned before) were identified.

There are three possible approaches for handling this circumstance:

a) Create distinct API clients for the different OGC API Standards and have the application decide, based on configuration, which client to use for a particular endpoint, the metadata provided for a particular collection or using other heuristics. This is the most standards-compliant approach that covers the functionality provided by the OGC API Standards. However, at the same time functionality is limited as it will not cover any server-specific extensions.

b) Generate clients for different APIs and manually merge them into a single client. This is the most work for a developer and is closest to developing a client from scratch. At the same time, it might be the most interesting solution for a server implementation, where the majority of work is in filling out the business logic behind the various resource handlers.

c) Implement a server endpoint that provides an OpenAPI definition document that is specific to this server. This definition document can also be used to bootstrap server development. A client based on such a server-specific OpenAPI definition document can utilize any specific extensions provided by the server, but is not guaranteed to work with any other implementation. The solution could be considered for highly-specialized servers that push the boundaries of what an existing OGC API provides.

The right approach depends on the scenario, desired level of interoperability and control that developers exert over the resulting code.

Approach 1 is the closest to the standards but also the most susceptible to minor deviations and errors in the OpenAPI definition documents used.

Approach 2 is the standards-adhering way to develop a multi-API server based on generated code.

Approach 3 leaves the most flexibility and gets the most out of the generated code in a coherent and unified code base, but may be limited in terms of interoperability if the developer is not careful.

In the frame of this activity, the generators for Python, JavaScript and TypeScript were exercised in more detail and information on those is provided in detail in the respective component descriptions in the Components Overview.

8.5. Recommendations to the OGC for using OpenAPI

The main recommendations as outcome from using OpenAPI code generators as part of this activity are the following:

- Use code generators, in addition to SwaggerUI, ReDoc and other OpenAPI tools, in order to verify the consistency, completeness and correctness of OGC APIs in accordance with OpenAPI specifications.
- Provide feedback to the generator developers when bugs or issues are found in the general generator framework or a particular language back-end.
- Include commonly used declarations for common endpoints, data transfer types and responses in order to ensure full consistency.
- Find an agreed upon approach for the composition of different APIs in a single server and/or client, which allow implementation in a single server and single client side by side.

These recommendations, if considered, may lead to such OGC API Standards (or profiles of those standards), which would be more suitable for composition and code generation.

9

RESULTS AND FINDINGS

9 RESULTS AND FINDINGS

This task conducted six distinctive API experiments to validate the applicability (and provide documentation and How-Tos) of OGC APIs in the combination with some of the most popular technology stacks and cloud solutions. While detailed elaboration can be found in the section dedicated to a specific component, this one provides the recap.

OGC APIs define state of the art RESTful web interfaces and JSON encoded data types in order to support the development of standardized geospatial web applications. The following figure represents the variety of APIs. Note that, as of the 1st of December 2021, only two of them (framed in green) had been published as approved OGC Standards while all others were draft specifications. By the end of December 2021, OGC API — Processes had also been published as an approved OGC Standard (raising the number of published approved OGC API Standards to three).

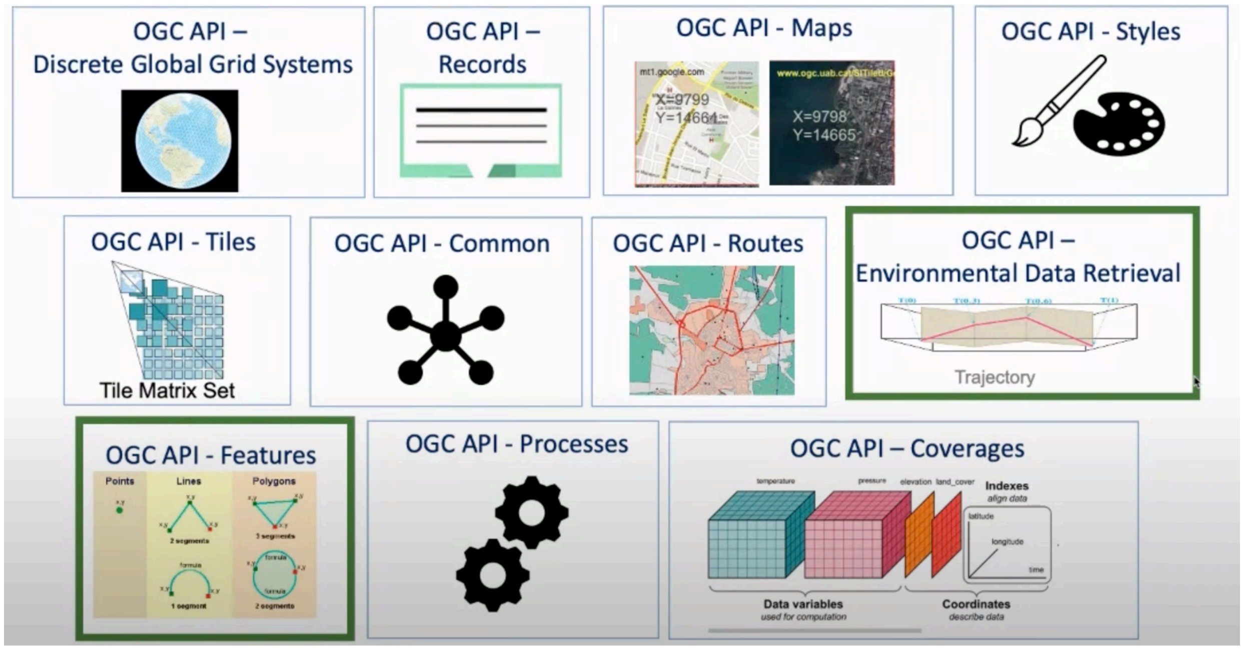


Figure 21 — The OGC API Family

API experiments performed several exercises with OGC API — Features and — EDR, as well as draft OGC API — Records, and OGC API — Routes.

All services created for experiments can be considered as being microservices because they are independent, technologically heterogeneous, containerized, and communicate over well-defined, lightweight APIs. They can be deployed in the cloud, as well.

Source code and detailed installation guidelines for all of six API Experiments are located in the GitLab repository under following URLs:

a) D165 Server (Python)

b) D166 Server (JS/TypeScript)

c) D167 Data Backend (AWS, CloudFerro and CloudSigma)

d) D168 Data Backend (AWS, CloudFerro and OGC API — Records)

e) D175 Client — Python

f) D176 Client — TypeScript

API Experiments Clause 7.1 successfully implemented two microservices based on OGC API — Features (Features) and OGC API — Environmental Data Retrieval (EDR). Both services are developed using Python programming language and geospatial libraries. The source code was partially generated out of specifications using Open API Generator.

When using the code generator there were originally many errors in the generated code for the EDR OpenAPI specification. The main reasons for the errors are in the OpenAPI Standard and the fact that the OGC API EDR Standard is more comprehensive and complex compared to the OGC API — Features specification. In order to mitigate these issues a simple profile of the OpenAPI based OGC API — EDR Standard was created. It supports only two-dimensional, temporal data.

The Elasticsearch search engine was used as backend data storage for features and records (OGC API — Features and Records implementations).

An Earth Observation (EO) dataset, Copernicus Sentinel-1 and Sentinel-2 data, was used to generate a land cover classification to monitor/investigate certain natural resources (sand dams) in Kenya. Within the framework of Clause 7.4 activity, the dataset was transitioned to an accessible Amazon Simple Storage Service (Amazon S3) bucket with the catalog defined through the two options of OGC API — Records and STAC. These were then dynamically deployed using Elasticsearch, through AWS OpenSearch, so they could be accessed by Clause 7.1.

Experiment Clause 7.2 demonstrated the applicability of JavaScript programming language for geospatial microservice implementations. The Technology Stack validated here was based on a server-side JavaScript environment Node JS, accompanied with widely used and mature geospatial databases PostGIS, and PostgreSQL.

The experiments carried out in Clause 7.3 (Data Backend and Deployment) were based on an idea of tiling NetCDF files (array oriented scientific data sets) to select them based on geographic extension of bounding boxes via OGC API — Tiles. Doing so, smaller NetCDF files can be published through a STAC catalog that documents the extent of each file, allowing a service provider to automate the search for the data of interest.

The following methods/technologies were demonstrated:

a) Generate and deploy a server for OGC API — Tiles

b) Use of 3 distinctive cloud environments (AWS, CloudSigma, CloudFerro) with PostGIS deployments.

c) AWS S3 deployment of a set of GeoJSON objects

d) NetCDF files publishing following a tiles approach.

Operating System-level virtualization service Docker was used (as container environment) to render the tiles before launching the service.

The D176 API Experiments Client — TypeScript demonstrated the use of TypeScript to request the data and a virtual globe API called NASA WorldWind to create 3D visualization of geographical information. A client code was partially generated based on the corresponding OpenAPI endpoint specifications. A connector for NASA Web WorldWind was created that loads API call response data sets into visual layers on WorldWind. A demonstrator showed the API client's capabilities in a real-world use case integrating with WorldWind, which bridges to plain JavaScript usage.

The D175 API Experiments Client — Python API Experiments Client was implemented in Python. Several geospatial Python libraries were tested, as well. The software used to automatically generate the client libraries was OpenAPI Generator. Code generation tests were successful for microservice implementations compatible with OGC API — Features and OGC API — EDR. Generated client libraries were capable of making requests to those servers endpoints and receive the subsequent responses.

10

FUTURE WORK

10 FUTURE WORK

The OGC API approach is based on newer technologies that did not exist during development of initial OGC Web Services (OWS). As OGC APIs mature towards official, approved standards (during this experiment only two specifications, OGC API — Features and OGC API — EDR were officially released by OGC, while others were available as drafts and partially used), the demand for tutorials, and code examples which would cover all API specifications will certainly grow.

A recommendation for future work would be to provide code examples and tutorials for each of OGC API Standards. The examples would specify data, cloud, server and client implementations, and CI/CD pipeline including deployment and integration tests. The code examples should preferably be made available on GitHub (or another online Git repository service) and the CI/CD pipeline also be specified using services from that repository.

In the context of cloud-native applications and OGC APIs, the future work recommendation is to conduct more advanced experiments with geospatial microservices applied on OGC APIs. Cloud-native applications use microservices. They share the following characteristics:

a) Implement a specific task within a larger domain context.

b) Each is developed autonomously and can be deployed independently.

c) Each is self-contained encapsulating its own data storage technology, dependencies, and programming platform.

d) Each runs in its own container and communicates using standard communication protocols such as HTTP or AMQP.

Some of these concepts were demonstrated during this task. Individual microservices utilizing OGC APIs were developed and deployed in cloud environments. More advanced experiments are recommended for future experiments. For example, an experiment including all OGC APIs compatible services (Feature, Maps, Tiles, EDR, etc.) covering a whole application domain created for demonstration purposes, and featuring services, which offer distinctive data sets and work all together in a service mash to fulfil a geospatial “business objective”. Such “experiments” would therefore test the fitness of, and provide referent examples for all OGC API Standards presented in a more complex geospatial application domain context.

It will not always be possible to respond to queries synchronously. Currently only OGC API – Processes specifies support for asynchronous communication, whereas other OGC API Standards do not specify how to handle any asynchrony. Different services may propose different best practices. Future work might explore different implementation options in the context of OGC APIs and provide code examples.

OGC APIs are RESTful web APIs. In certain situations, an alternative API approach for geospatial data might be useful as a service model for OGC APIs to address specific needs. That alternative API technology could be GraphQL. GraphQL is a query language and server-side runtime for application programming interfaces (APIs) that is especially well suitable to query on demand from complex graphs of linked, dependent data.

11

TECHNOLOGY INTEGRATION EXPERIMENTS (TIE)

11 TECHNOLOGY INTEGRATION EXPERIMENTS (TIE)

The TIEs for the API Experiments task are documented around the interactions between the server, client, and data storage components:

- **Data:**
  - D167 Data Backend and Deployment 1
  - D168 Data Backend and Deployment 2
- **Services:**
  - D165 Server (Python)
  - D166 Server (JavaScript)
- **Clients:**
  - D175 API Experiments Client (Python)
  - D176 API Experiments Client (TypeScript)

The TIEs are divided into the part dedicated to the data deployment (preparation for the provision) and the client/service interactions. OGC API — Features, EDR, and Tiles were used to access services and the interaction with client components via these APIs were tested and documented.

11.1. Data backend and deployment

Table 1 — Server-Backend TIE Summary Table

SERVER\\BACKEND	D167 SKYMANTICS	D168 PIXALYTICS
D165 52N	2/4	4/4
D166 GMU	2/4	2/4

Table 2 — TIE Functional Test

#	FUNCTION	DESCRIPTION	CLIENT ACTION	SERVER RESPONSE	SUCCESS CRITERION
1	DockerDeploy Features Server	Pull OGC API — Features container and deploy server	Pull the Docker container and then deploy the server using the default backend configuration	Landing page is deployed	Landing page is visible and can be interacted with
2	DockerDeploy Features Server with backend data	Edit the default backend configuration and then deploy the server	Deploy the server using the modified backend configuration	Landing page is deployed	Landing page is visible and the information for the backend datasets is available
3	DockerDeploy EDR Server	Pull container and deploy the Environmental Data Retrieval (EDR) server	Pull the Docker container and then deploy the server using the default backend configuration	Landing page is deployed	Landing page is visible and can be interacted with
4	DockerDeploy EDR Server with backend data	Edit the default backend configuration and then deploy the server	Deploy the server using the modified backend configuration	Landing page is deployed	Landing page is visible and the information for the backend datasets is available

11.2. Service invocations and data consumptions

Table 3 — Server-Client TIE Summary Table

SERVER\\CLIENT	D175 SKYMANTICS	D176 SOLENIX
D165 52N	6/6	5/6
D166 GMU	6/6	6/6

Table 4 — TIE Functional Test

#	FUNCTION	DESCRIPTION	CLIENT ACTION	SERVER RESPONSE	SUCCESS CRITERION
1	LandingPage Features	Request, receive, parse landing page	Form landing page request, parse response, display and/or act on it	Receive landing page request, response with accepted / requested format (f=json, html)	Client displays API “data” link and/or navigates to it
2	Collection Features	Request, receive, parse collection page	Form collection page request, parse response, display and/or act on it	Receive collection page request, response with accepted / requested format (f=json, html)	Client displays list of collections available or processes it
3	Items Features	Request, receive, parse items page	Form items page request, parse response, display and/or act on it	Receive items page request, response with accepted / requested format (f=json, html)	Client displays feature items received
4	LandingPage EDR	Request, receive, parse landing page	Form landing page request, parse response, display and/or act on it	Receive landing page request, response with accepted / requested format (f=json, html)	Client displays API “data” link and/or navigates to it
5	Collection EDR	Request, receive, parse collection page	Form collection page request, parse response, display and/or act on it	Receive collection page request, response with accepted / requested format (f=json, html)	Client displays list of collections available or processes it
6	Items EDR	Request, receive, parse items page	Form items page request, parse response, display and/or act on it	Receive items page request, response with accepted / requested format (f=json, html)	Client displays feature items received

A

ANNEX A (INFORMATIVE) OGC INNOVATION PROGRAM DEVELOPER SURVEY

A

ANNEX A (INFORMATIVE) OGC INNOVATION PROGRAM DEVELOPER SURVEY

The objective of this survey was to deliver the ideas to improve the code resources, tools, and documentation available for developers of geospatial enabled applications, by learning directly from them what works best.

a) **What is your age group?**

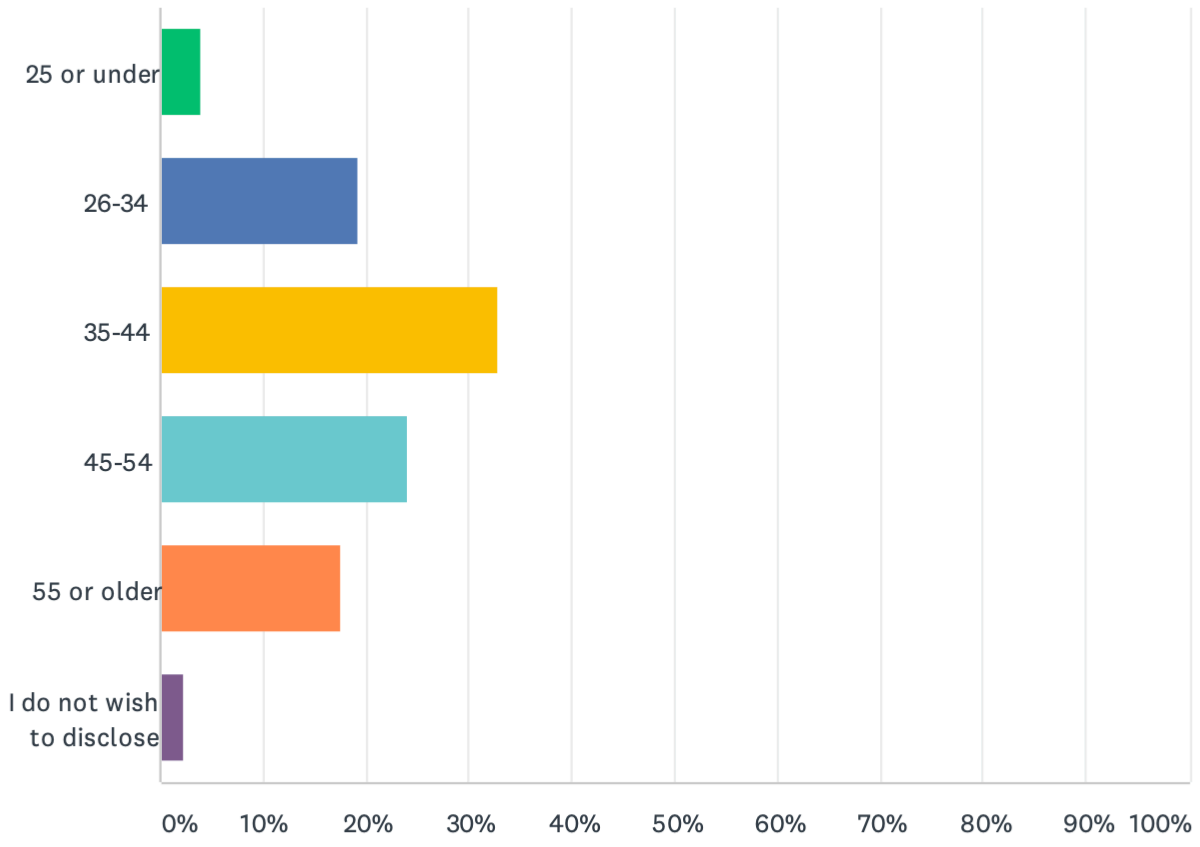


Figure A.1

ANSWER CHOICES	RESPONSES	
25 or under	4.00%	5
26-34	19.20%	24
35-44	32.80%	41
45-54	24.00%	30
55 or older	17.60%	22
I do not wish to disclose	2.40%	3
TOTAL		125

Figure A.2

b) **Which region do you currently reside in?**

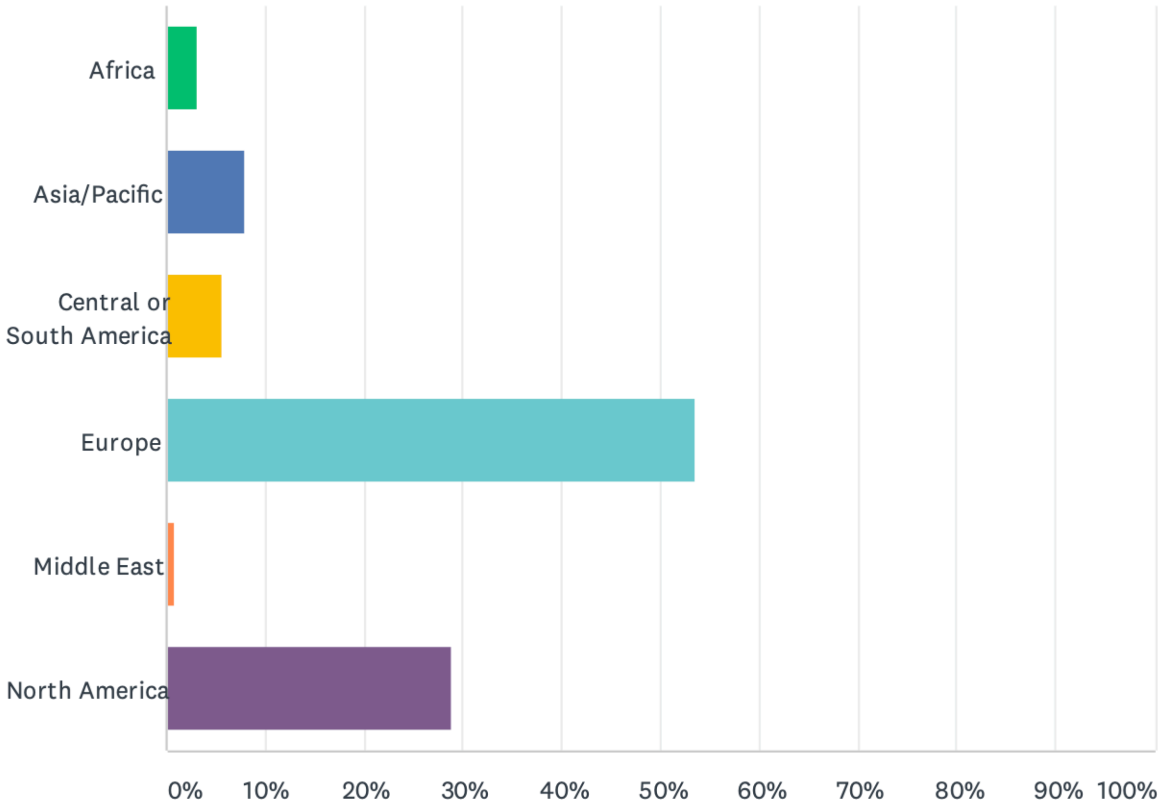


Figure A.3

ANSWER CHOICES	RESPONSES	
Africa	3.20%	4
Asia/Pacific	8.00%	10
Central or South America	5.60%	7
Europe	53.60%	67
Middle East	0.80%	1
North America	28.80%	36
TOTAL		125

Figure A.4

c) **What is the highest degree or level of school you have completed?**

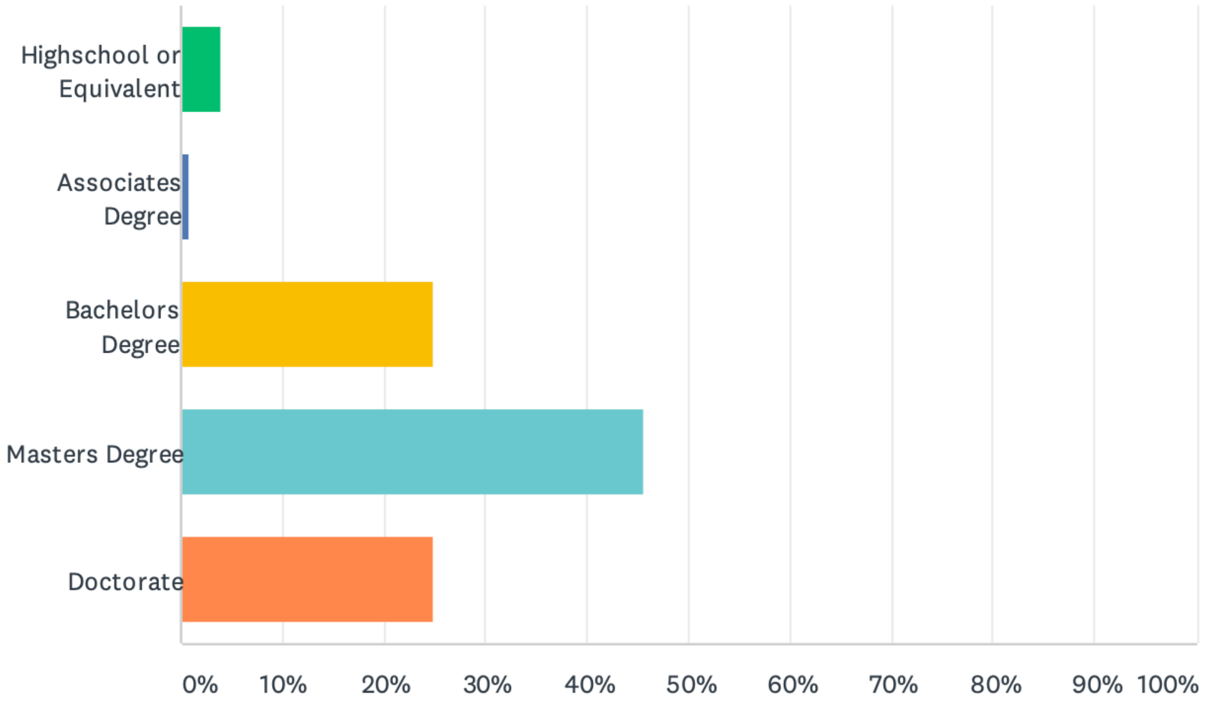


Figure A.5

ANSWER CHOICES	RESPONSES	
Highschool or Equivalent	4.00%	5
Associates Degree	0.80%	1
Bachelors Degree	24.80%	31
Masters Degree	45.60%	57
Doctorate	24.80%	31
TOTAL		125

Figure A.6

d) **What of the following options best describes you?**

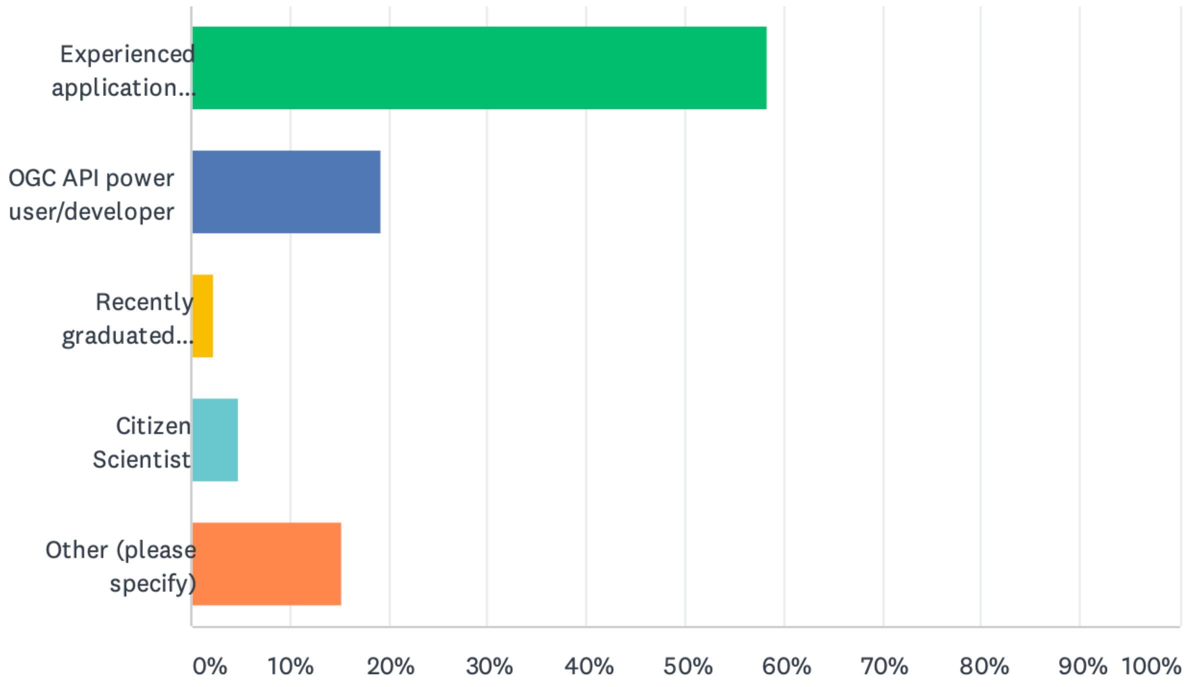


Figure A.7

ANSWER CHOICES	RESPONSES	
Experienced application developer	58.40%	73
OGC API power user/developer	19.20%	24
Recently graduated engineer	2.40%	3
Citizen Scientist	4.80%	6
Other (please specify)	15.20%	19
TOTAL		125

Figure A.8

e) **What is your employment status?**

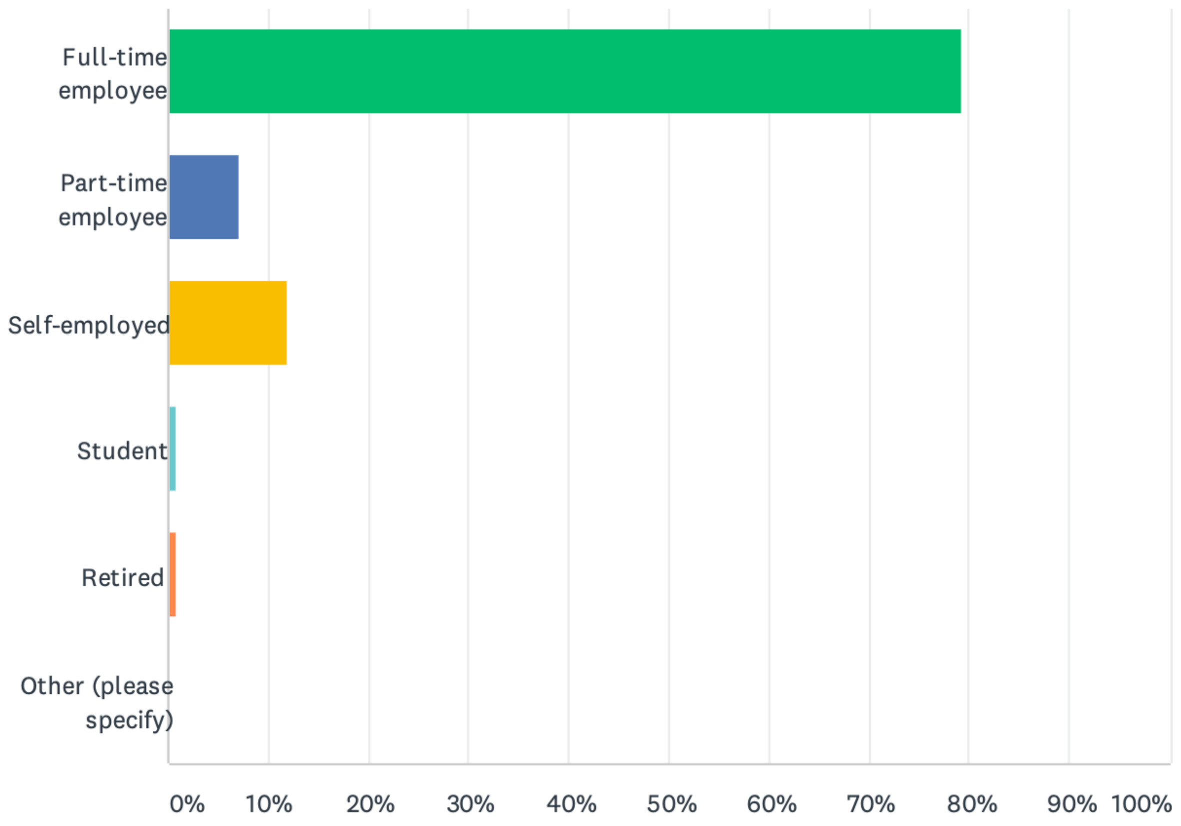


Figure A.9

ANSWER CHOICES	RESPONSES	
Full-time employee	79.20%	99
Part-time employee	7.20%	9
Self-employed	12.00%	15
Student	0.80%	1
Retired	0.80%	1
Other (please specify)	0.00%	0
TOTAL		125

Figure A.10

f) **If employed, what best describes your organization?**

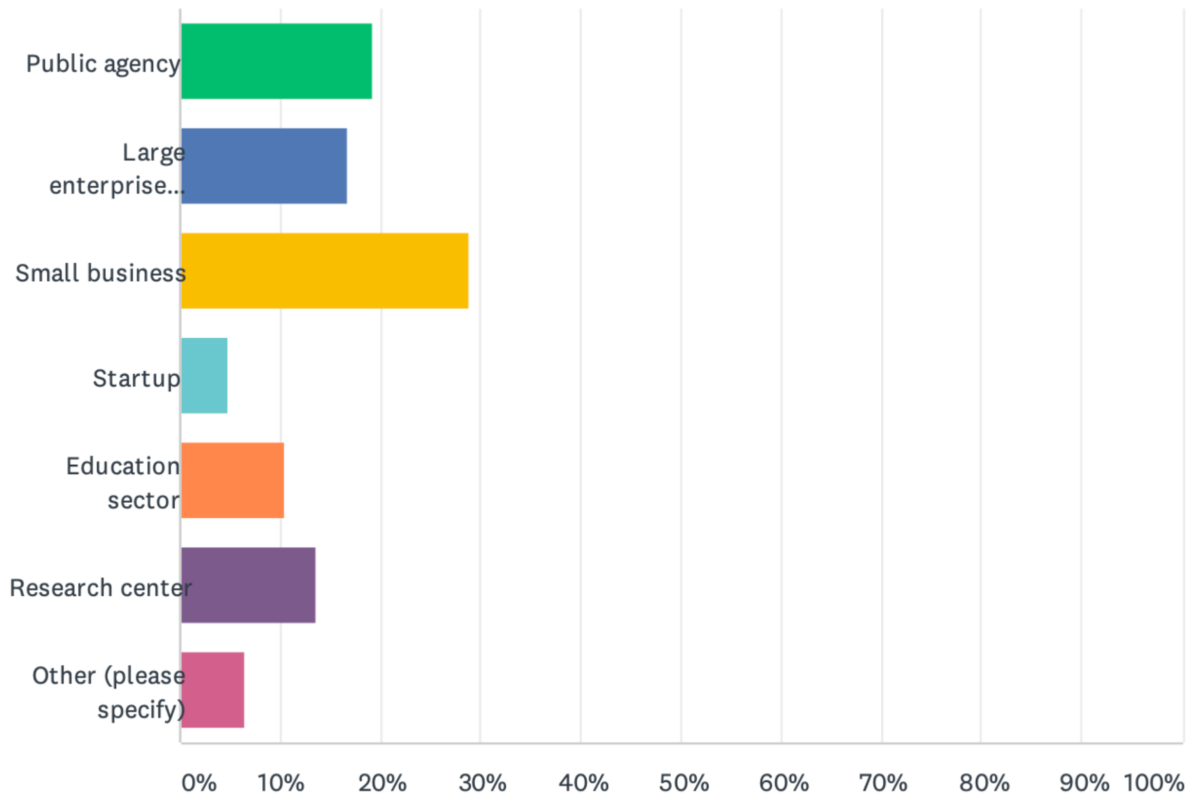


Figure A.11

ANSWER CHOICES	RESPONSES	
Public agency	19.20%	24
Large enterprise organization	16.80%	21
Small business	28.80%	36
Startup	4.80%	6
Education sector	10.40%	13
Research center	13.60%	17
Other (please specify)	6.40%	8
TOTAL		125

Figure A.12

g) **Which of the following positions is closest to yours?**

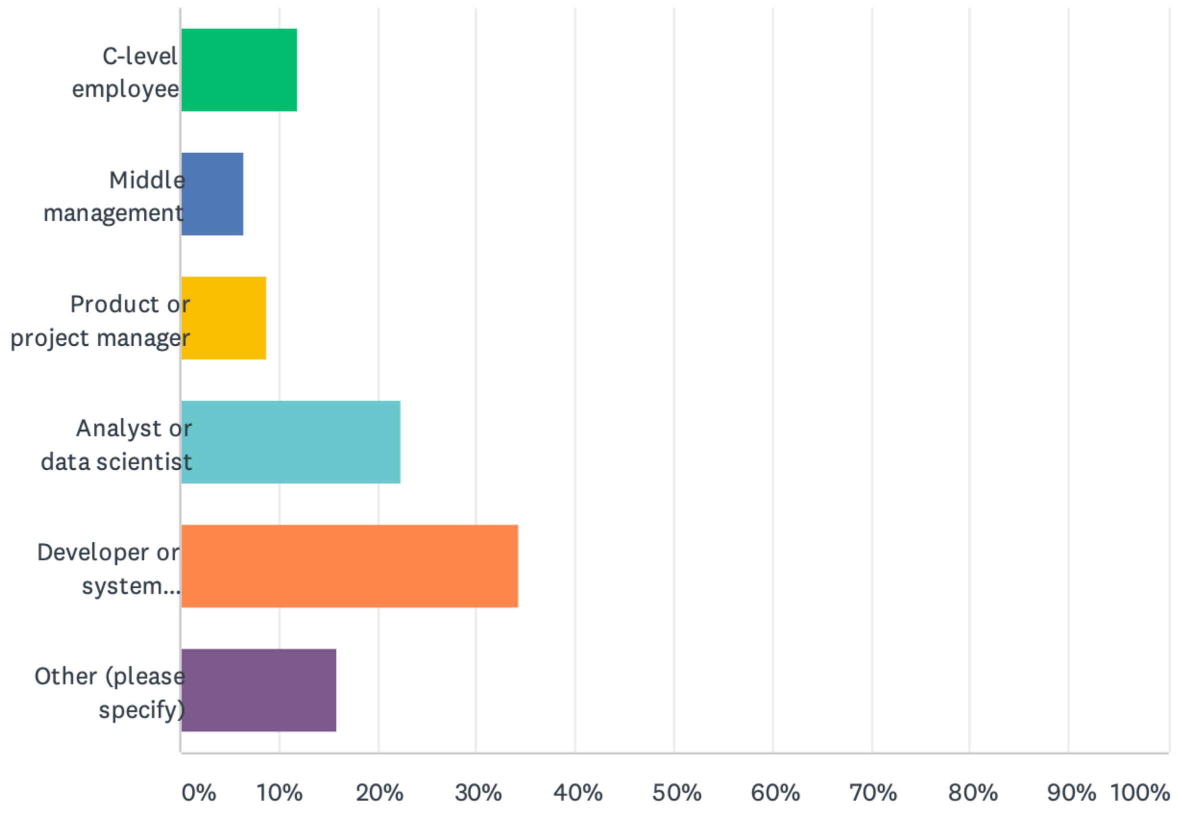


Figure A.13

ANSWER CHOICES	RESPONSES	
C-level employee	12.00%	15
Middle management	6.40%	8
Product or project manager	8.80%	11
Analyst or data scientist	22.40%	28
Developer or system administrator	34.40%	43
Other (please specify)	16.00%	20
TOTAL		125

Figure A.14

h) **How long have you been coding?**

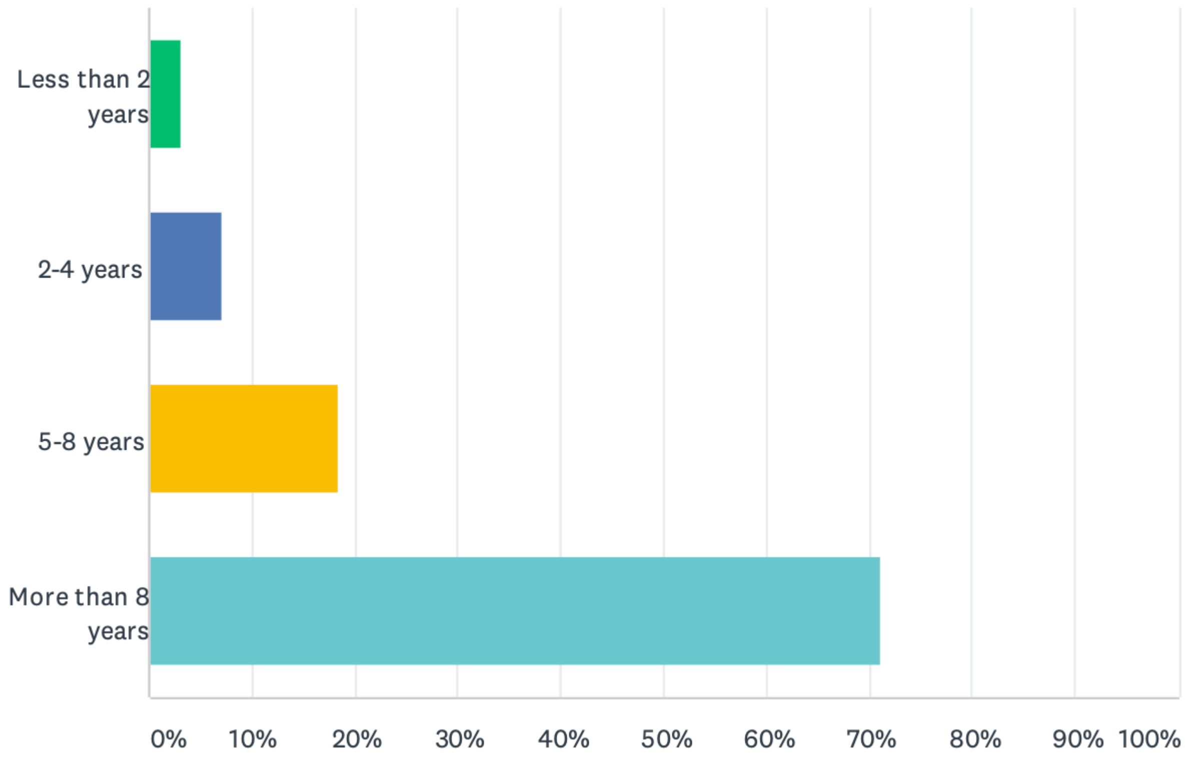


Figure A.15

ANSWER CHOICES	RESPONSES	
Less than 2 years	3.20%	4
2-4 years	7.20%	9
5-8 years	18.40%	23
More than 8 years	71.20%	89
TOTAL		125

Figure A.16

i) **Please select the one sentence that best describes your software development role (If none apply, please specify other):**

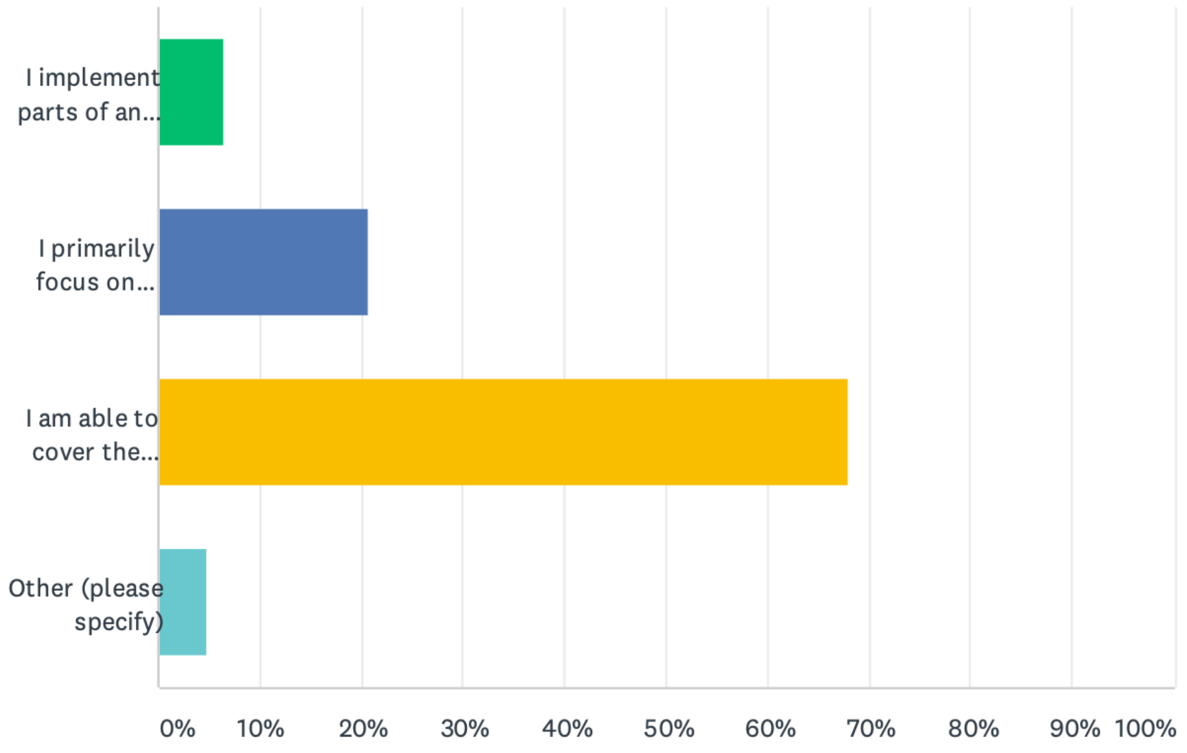


Figure A.17

ANSWER CHOICES	RESPONSES	
I implement parts of an application as instructed by another, senior team member.	6.40%	8
I primarily focus on architectural and stack decisions and delegate this information to coders.	20.80%	26
I am able to cover the entire software lifecycle from design all the way to development.	68.00%	85
Other (please specify)	4.80%	6
TOTAL		125

Figure A.18

j) **What Operating Systems are you most familiar with?**

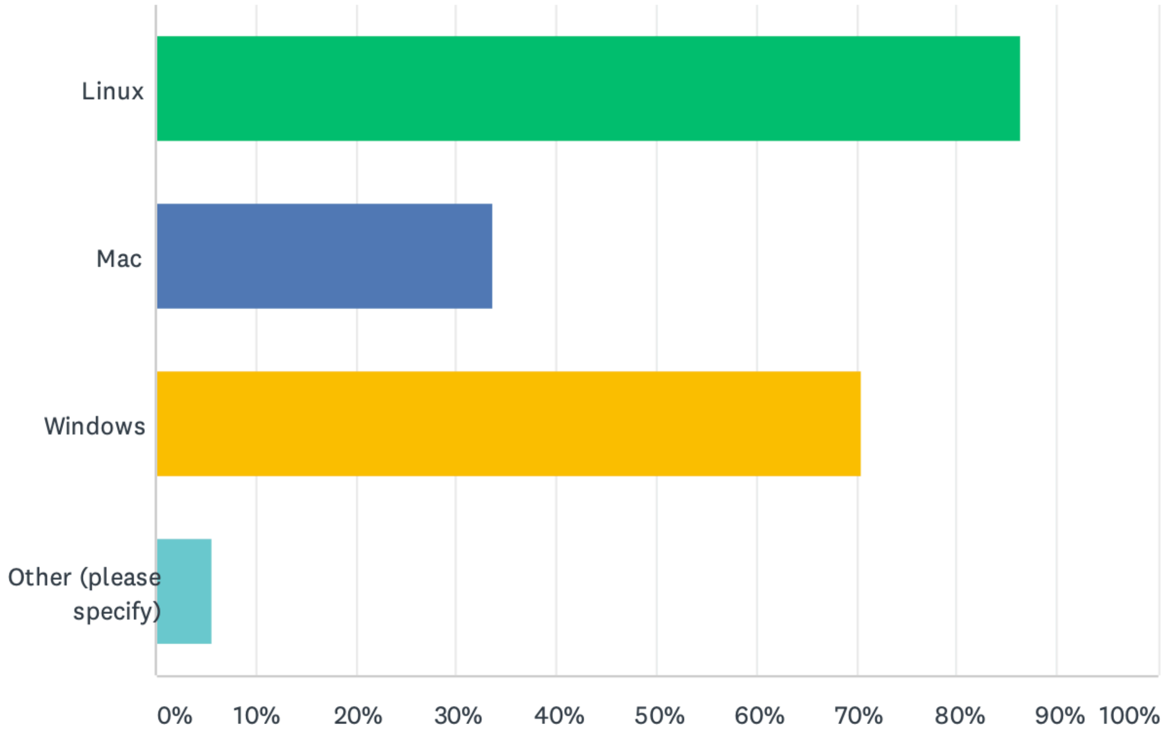


Figure A.19

ANSWER CHOICES	RESPONSES	
Linux	86.40%	108
Mac	33.60%	42
Windows	70.40%	88
Other (please specify)	5.60%	7
Total Respondents: 125		

Figure A.20

k) **What programming languages are you most familiar with (mark all that apply)?**

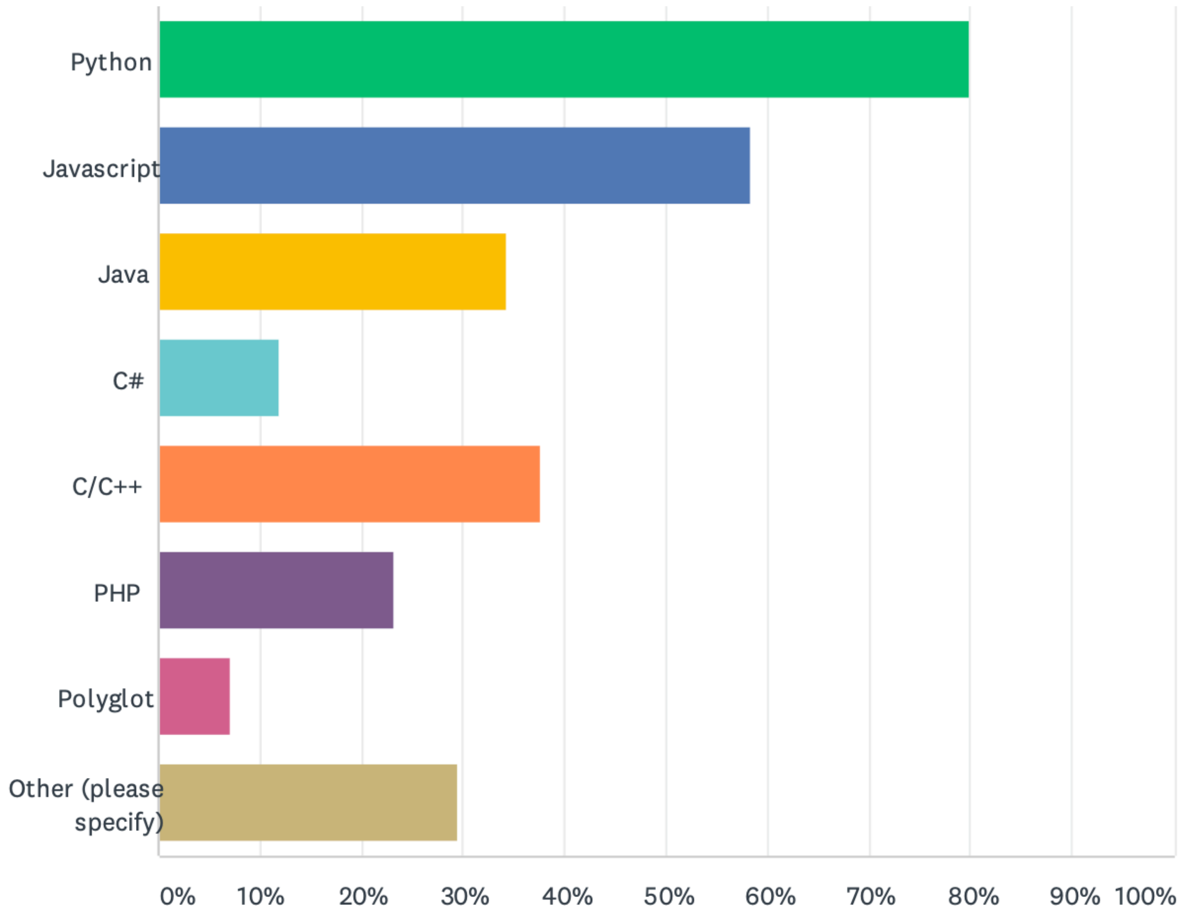


Figure A.21

ANSWER CHOICES	RESPONSES	
Python	80.00%	100
Javascript	58.40%	73
Java	34.40%	43
C#	12.00%	15
C/C++	37.60%	47
PHP	23.20%	29
Polyglot	7.20%	9
Other (please specify)	29.60%	37
Total Respondents: 125		

Figure A.22

l) **You prefer a presenter who:**

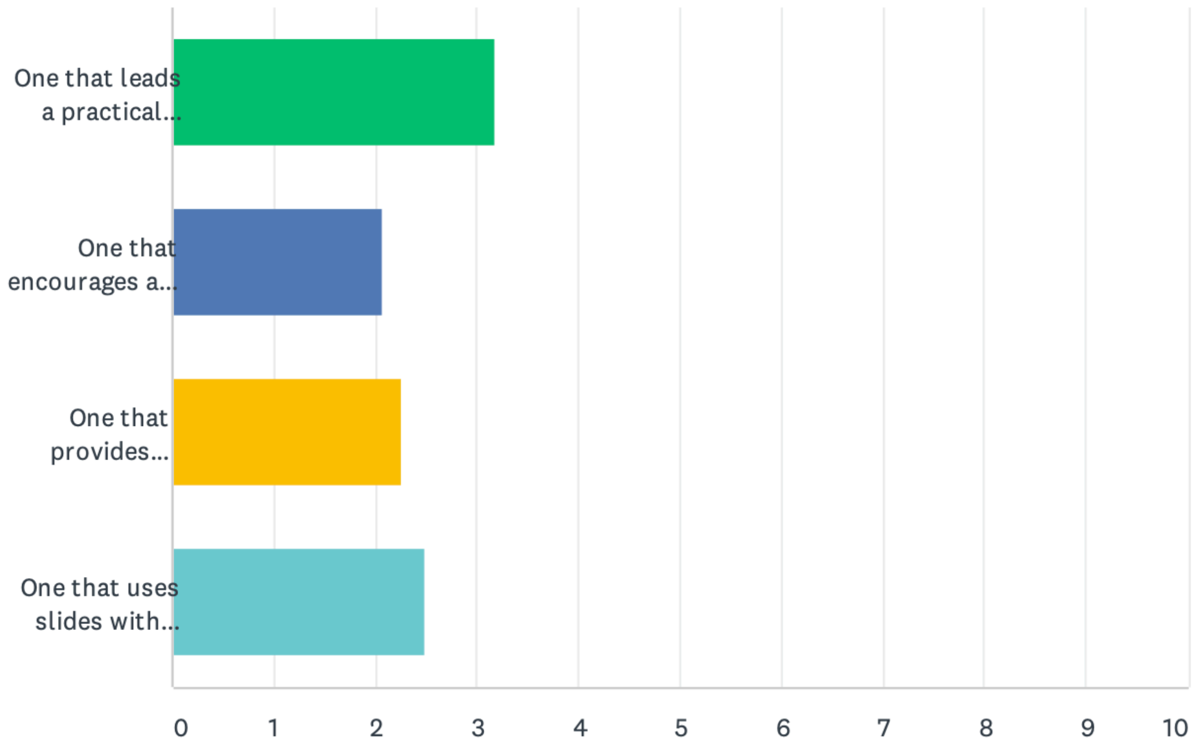


Figure A.23

	1	2	3	4	TOTAL	SCORE
One that leads a practical session or workshop	53.60% 67	19.20% 24	17.60% 22	9.60% 12	125	3.17
One that encourages a good group discussion	8.80% 11	26.40% 33	28.80% 36	36.00% 45	125	2.08
One that provides written material for further reading	13.60% 17	28.00% 35	28.80% 36	29.60% 37	125	2.26
One that uses slides with diagrams, charts, maps, and graphs	24.00% 30	26.40% 33	24.80% 31	24.80% 31	125	2.50

Figure A.24

m) **You want to learn to use a new technology. You would most likely:**

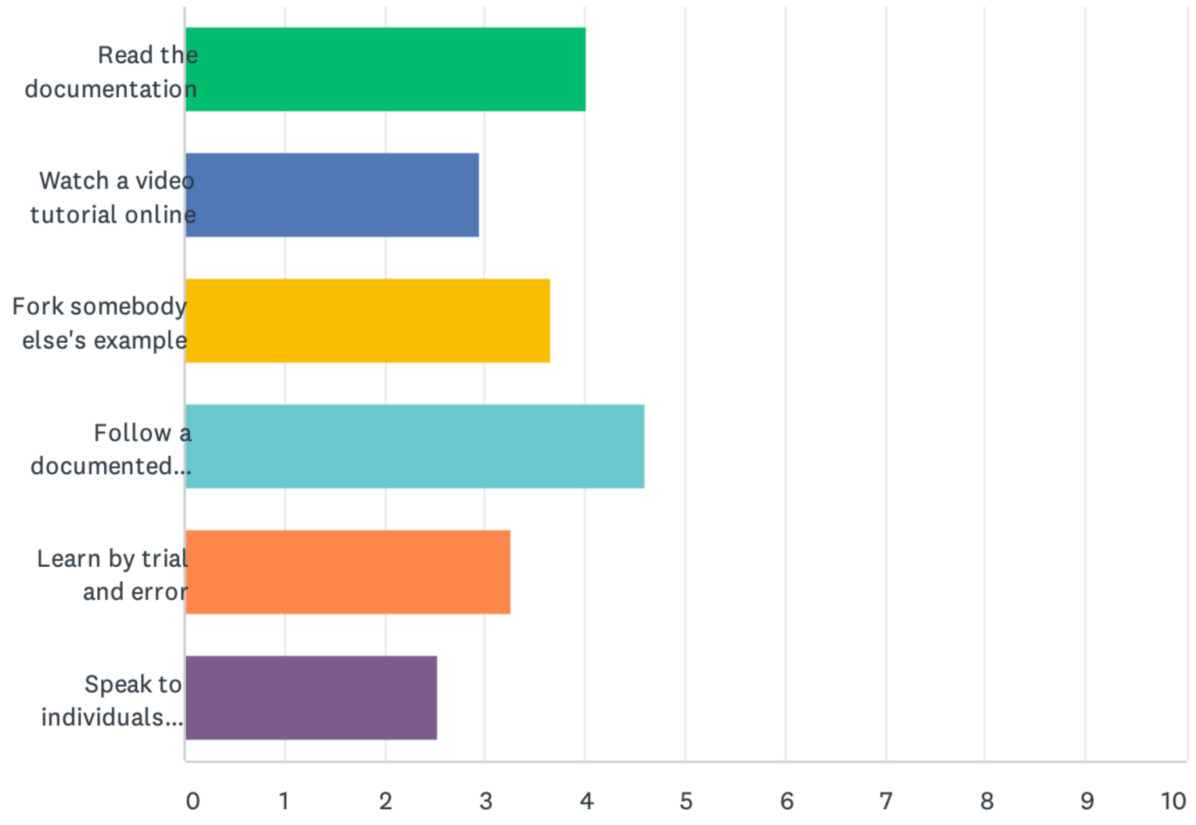


Figure A.25

	1	2	3	4	5	6	TOTAL	SCORE
Read the documentation	20.00% 25	19.20% 24	28.00% 35	14.40% 18	12.00% 15	6.40% 8	125	4.02
Watch a video tutorial online	11.20% 14	13.60% 17	8.80% 11	18.40% 23	20.80% 26	27.20% 34	125	2.94
Fork somebody else's example	16.80% 21	14.40% 18	24.00% 30	15.20% 19	21.60% 27	8.00% 10	125	3.66
Follow a documented how-to/tutorial	32.80% 41	30.40% 38	15.20% 19	12.00% 15	5.60% 7	4.00% 5	125	4.61
Learn by trial and error	16.00% 20	12.00% 15	14.40% 18	17.60% 22	19.20% 24	20.80% 26	125	3.26
Speak to individuals that know about the technology	3.25% 4	10.57% 13	9.76% 12	22.76% 28	21.14% 26	32.52% 40	123	2.54

Figure A.26

n) **You want to learn how to play a new card game. You would most likely:**

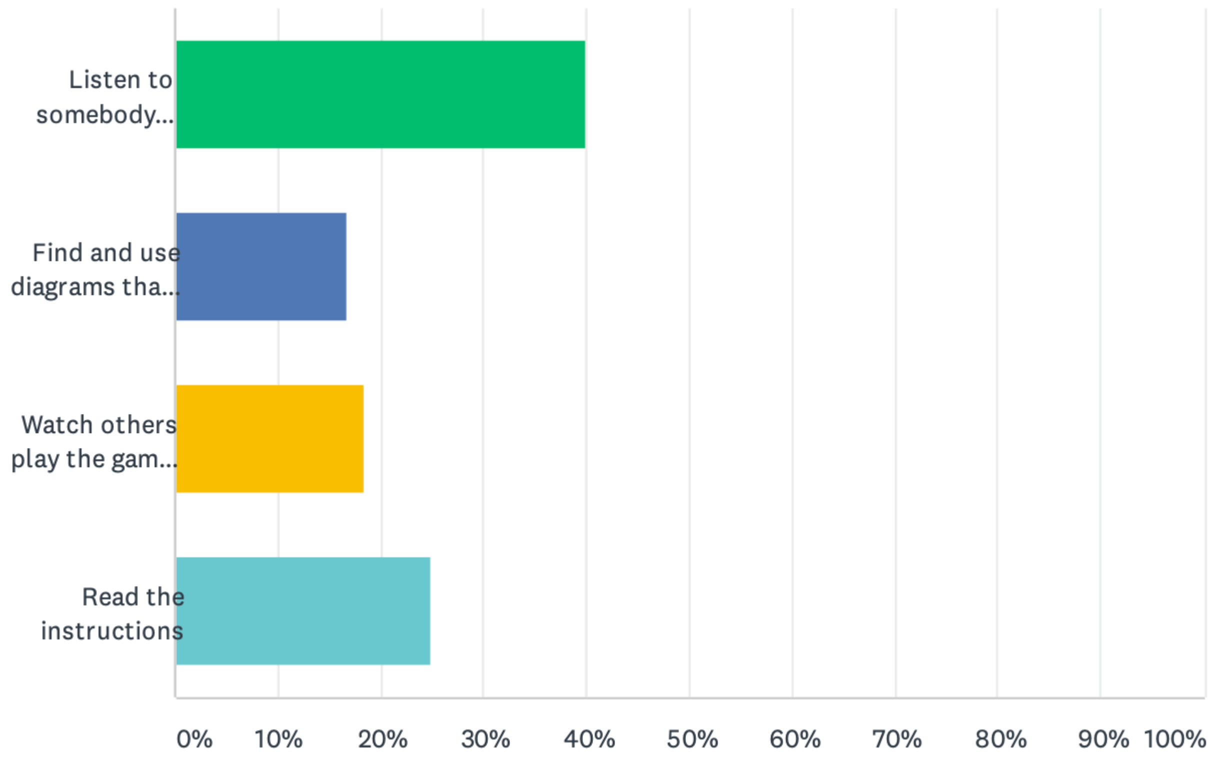


Figure A.27

ANSWER CHOICES	RESPONSES	
Listen to somebody explain the game and ask questions	40.00%	50
Find and use diagrams that explain core parts of the game such as stages, types of moves, and strategies	16.80%	21
Watch others play the game and then join in	18.40%	23
Read the instructions	24.80%	31
TOTAL		125

Figure A.28

o) **You just bought a new camera and want to learn how to use it. You would most likely:**

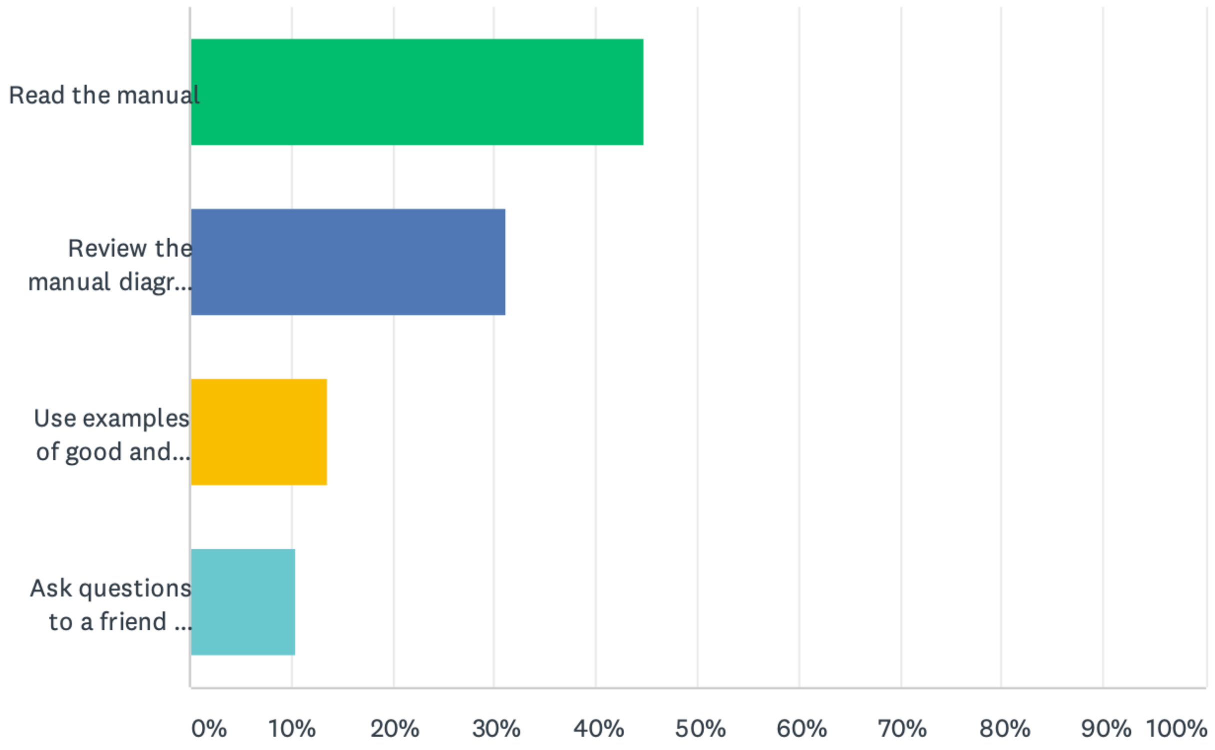
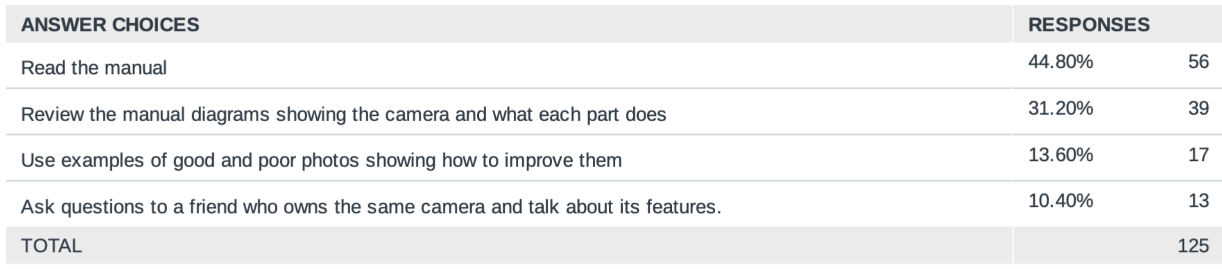


Figure A.29



ANSWER CHOICES	RESPONSES	
Read the manual	44.80%	56
Review the manual diagrams showing the camera and what each part does	31.20%	39
Use examples of good and poor photos showing how to improve them	13.60%	17
Ask questions to a friend who owns the same camera and talk about its features.	10.40%	13
TOTAL		125

Figure A.30

p) **When learning on a new topic from the Internet you prefer:**

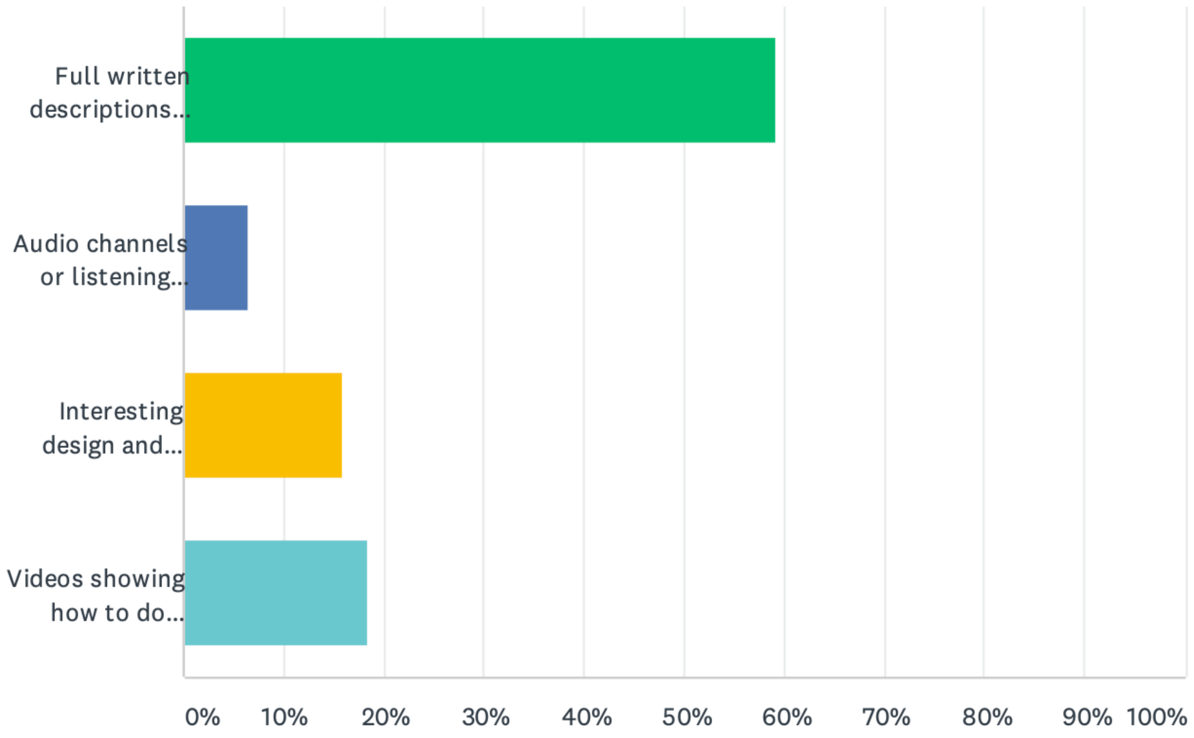


Figure A.31

ANSWER CHOICES	RESPONSES	
Full written descriptions, lists, and explanations	59.20%	74
Audio channels or listening to podcasts or interviews	6.40%	8
Interesting design and visual features	16.00%	20
Videos showing how to do something.	18.40%	23
TOTAL		125

Figure A.32

q) **You just purchased a wooden table that came in parts but you cannot find the documentation. What would you look for on the Internet?**

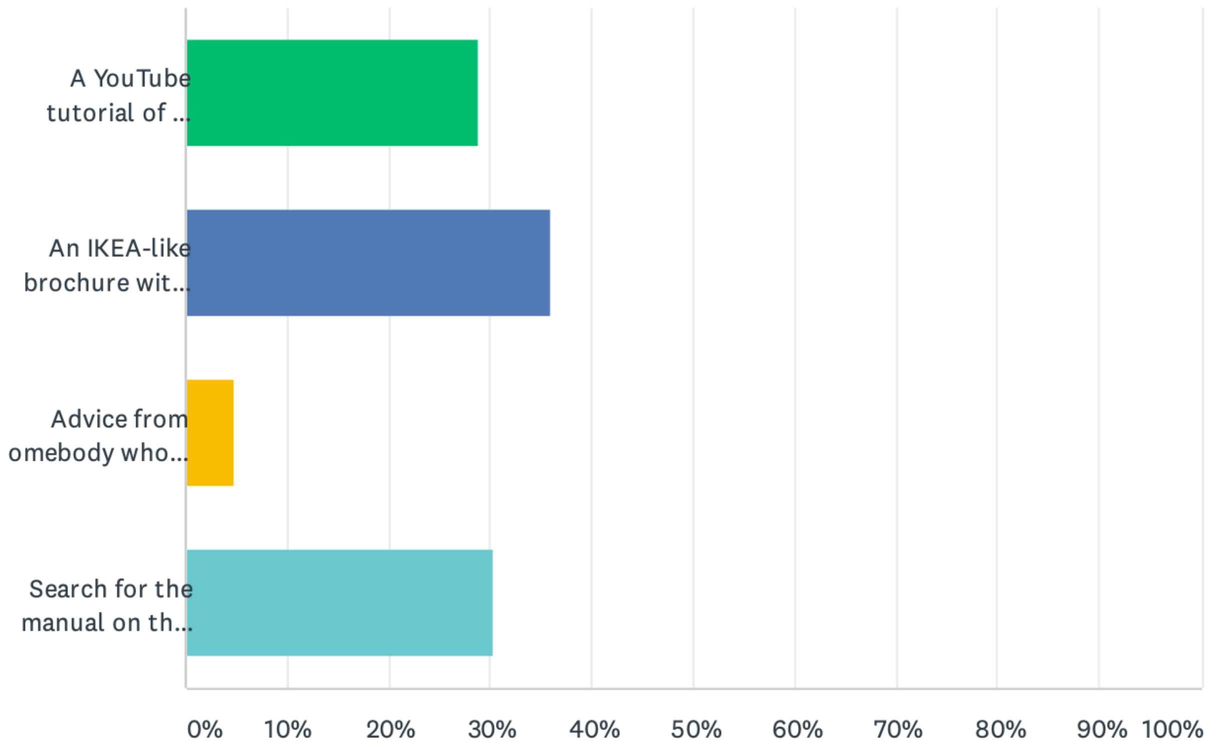


Figure A.33

ANSWER CHOICES	RESPONSES	
A YouTube tutorial of an individual assembling a similar table	28.80%	36
An IKEA-like brochure with diagrams showing each stage of assembly	36.00%	45
Advice from somebody who has put a similar table together before	4.80%	6
Search for the manual on the Internet and read it through	30.40%	38
TOTAL		125

Figure A.34

r) **Do you have experience with geospatial technology?**

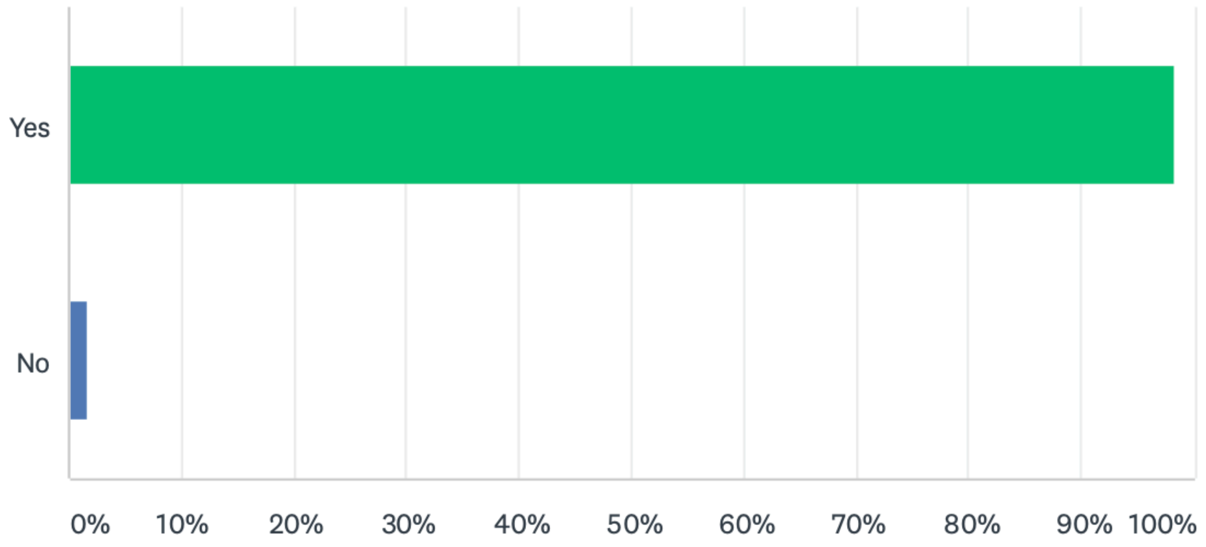
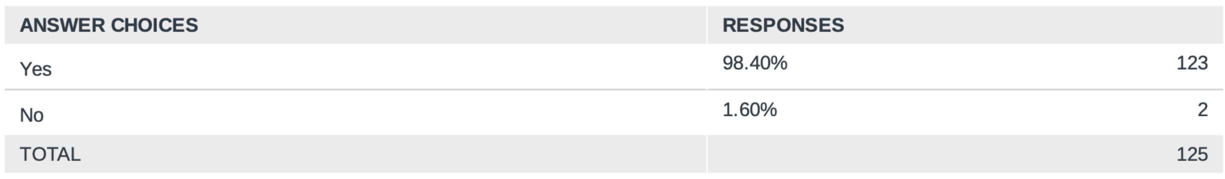


Figure A.35



ANSWER CHOICES	RESPONSES	
Yes	98.40%	123
No	1.60%	2
TOTAL		125

Figure A.36

s) **How many hours per week do you spend in tasks related in one way or another to geospatial technology?**

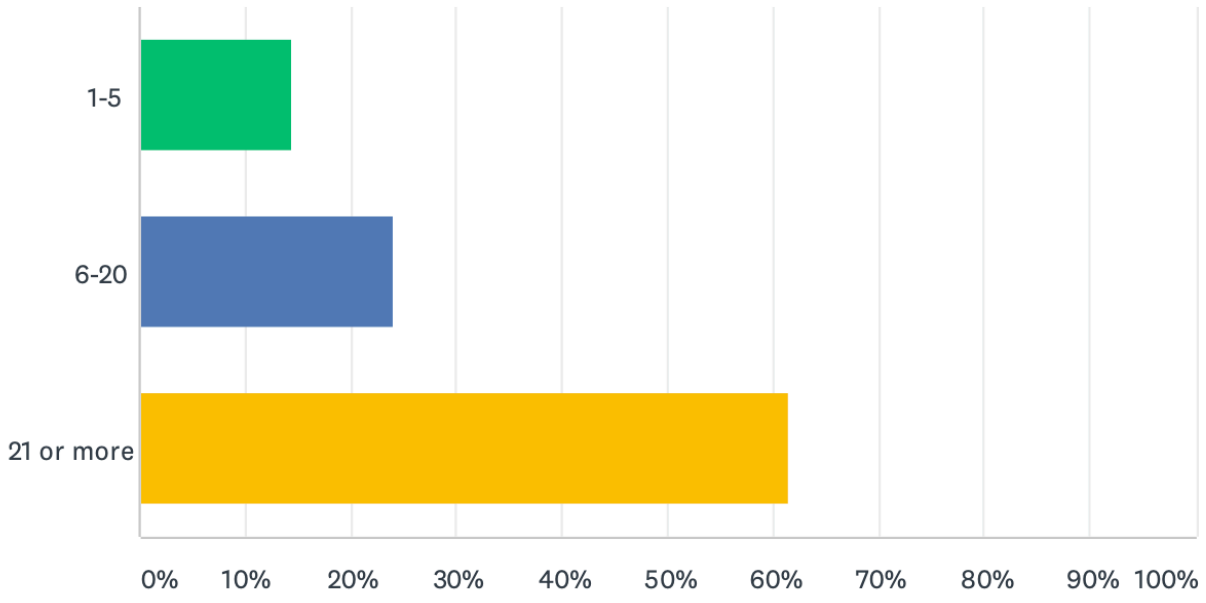


Figure A.37

ANSWER CHOICES	RESPONSES	
1-5	14.40%	18
6-20	24.00%	30
21 or more	61.60%	77
TOTAL		125

Figure A.38

t) **Which of these open-source geospatial projects are you familiar with (mark all that apply)?**

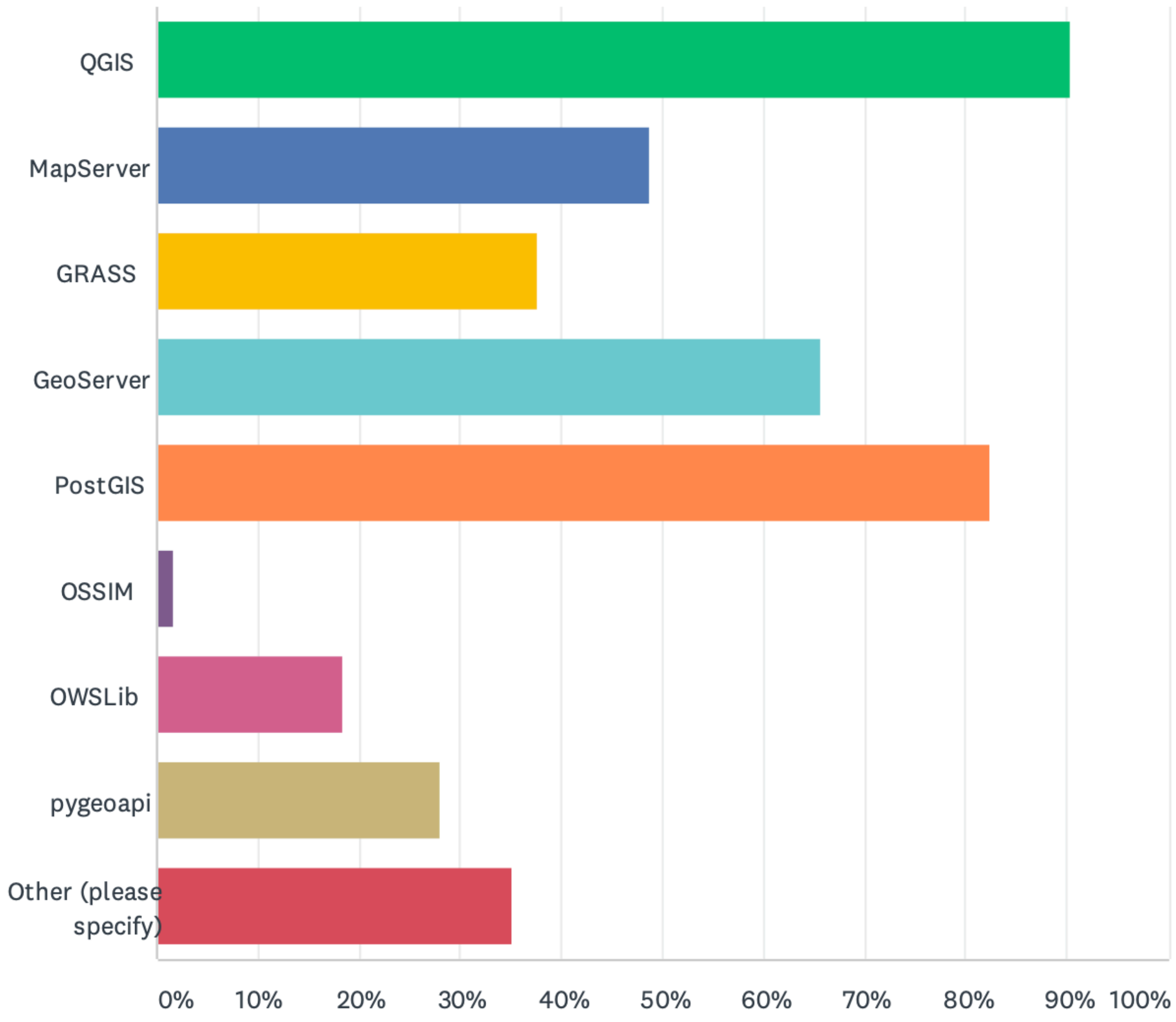


Figure A.39

ANSWER CHOICES	RESPONSES	
QGIS	90.40%	113
MapServer	48.80%	61
GRASS	37.60%	47
GeoServer	65.60%	82
PostGIS	82.40%	103
OSSIM	1.60%	2
OWSLib	18.40%	23
pygeoapi	28.00%	35
Other (please specify)	35.20%	44
Total Respondents: 125		

Figure A.40

u) Are you involved in any FOSS projects?

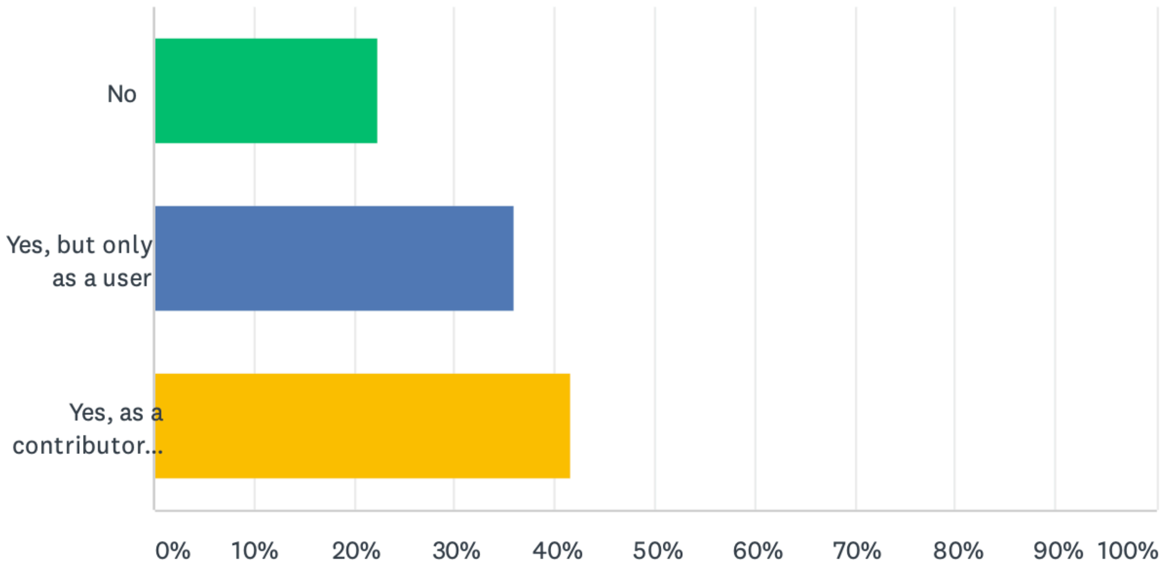


Figure A.41

ANSWER CHOICES	RESPONSES	
No	22.40%	28
Yes, but only as a user	36.00%	45
Yes, as a contributor (code, creating issues, documentation, etc.)	41.60%	52
TOTAL		125

Figure A.42

v) **If you replied yes to the previous question, is this something you do at work?**

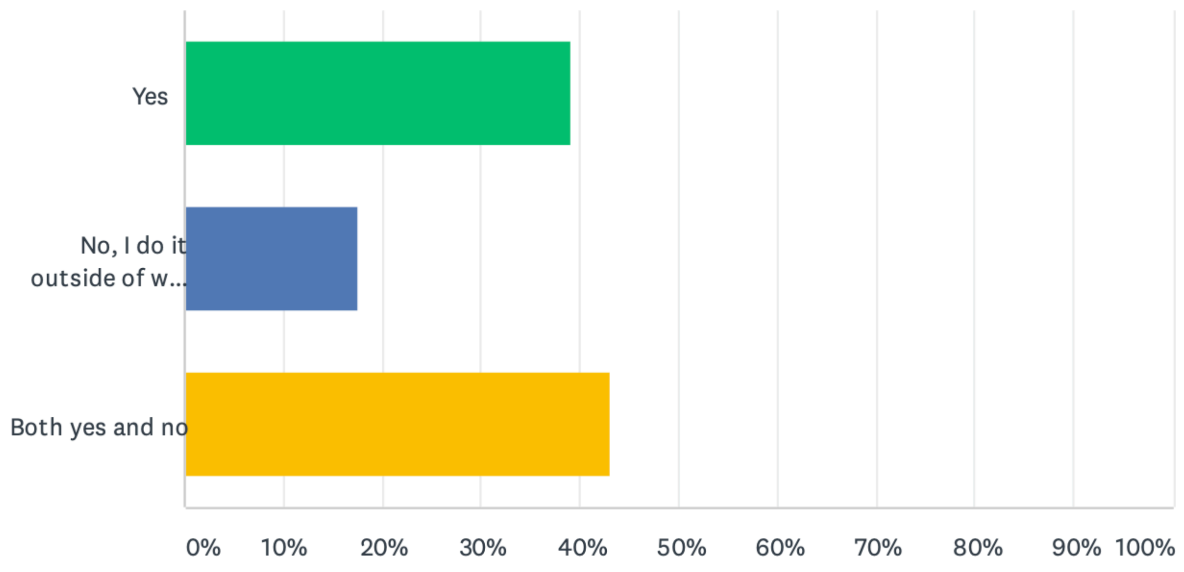


Figure A.43

ANSWER CHOICES	RESPONSES	
Yes	39.20%	49
No, I do it outside of work hours	17.60%	22
Both yes and no	43.20%	54
TOTAL		125

Figure A.44

w) **What is the use of Geospatial technology in your work? What problems do they solve in your organization or for your customers?**

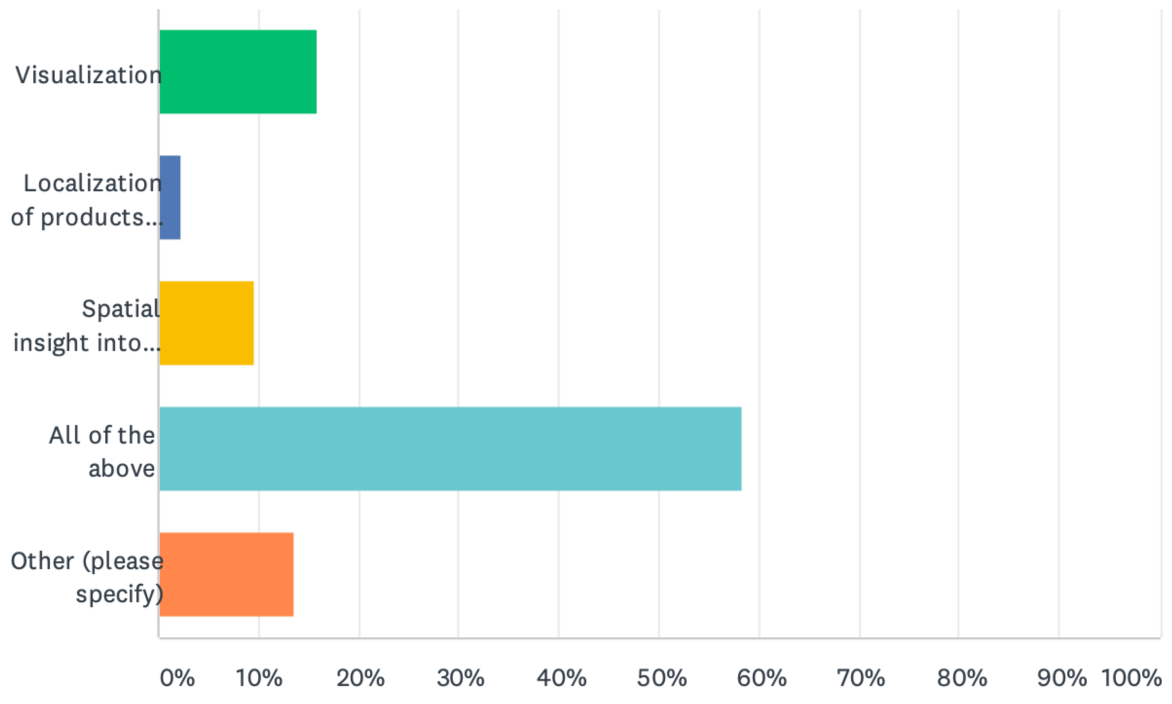


Figure A.45

ANSWER CHOICES	RESPONSES	
Visualization	16.00%	20
Localization of products and/or services	2.40%	3
Spatial insight into organizational or customer data	9.60%	12
All of the above	58.40%	73
Other (please specify)	13.60%	17
TOTAL		125

Figure A.46

x) **What experience do you have with OGC and OGC standards (mark all that apply)?**

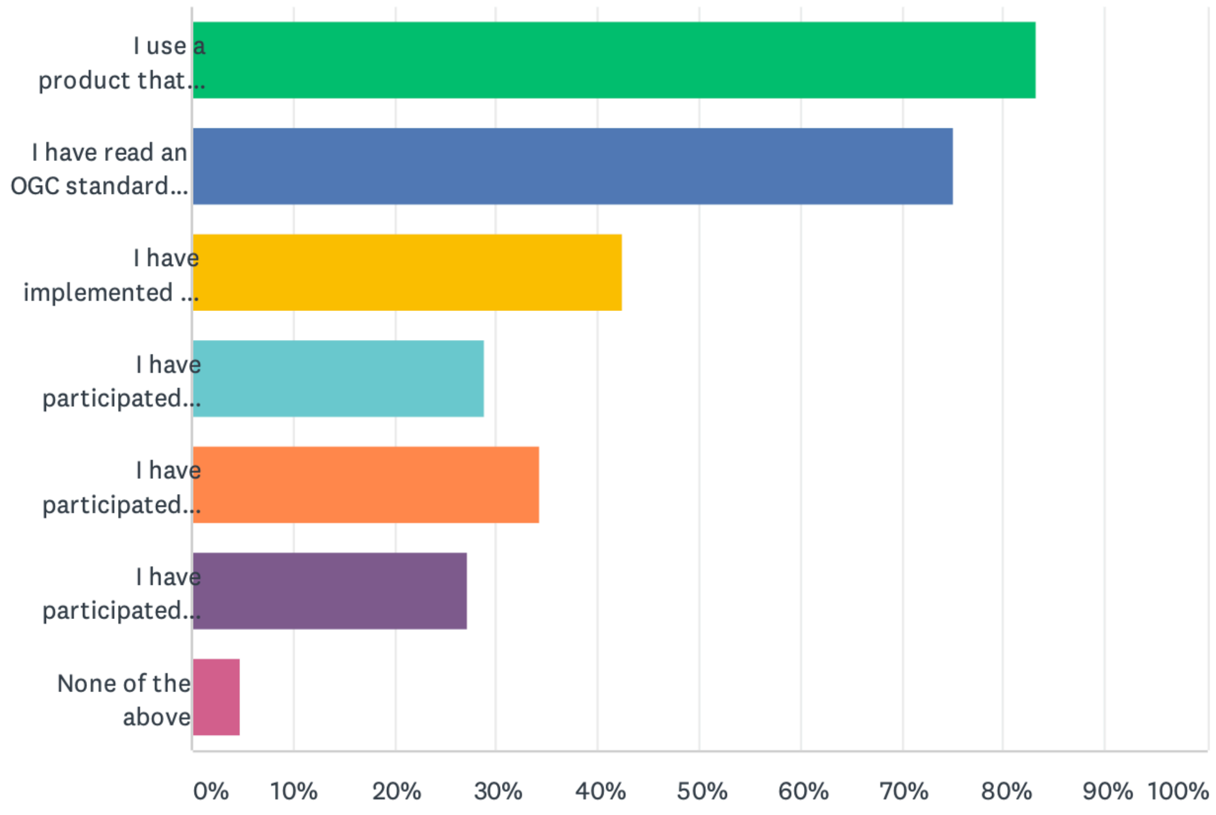


Figure A.47

ANSWER CHOICES	RESPONSES	
I use a product that implements an OGC standard	83.20%	104
I have read an OGC standard document	75.20%	94
I have implemented an OGC standard	42.40%	53
I have participated in an OGC Initiative (Pilot, Testbed, etc.)	28.80%	36
I have participated in n OGC Member Meeting	34.40%	43
I have participated in an OGC Domain or Standards Working Group	27.20%	34
None of the above	4.80%	6
Total Respondents: 125		

Figure A.48

y) **What experience do you have with OGC APIs?**

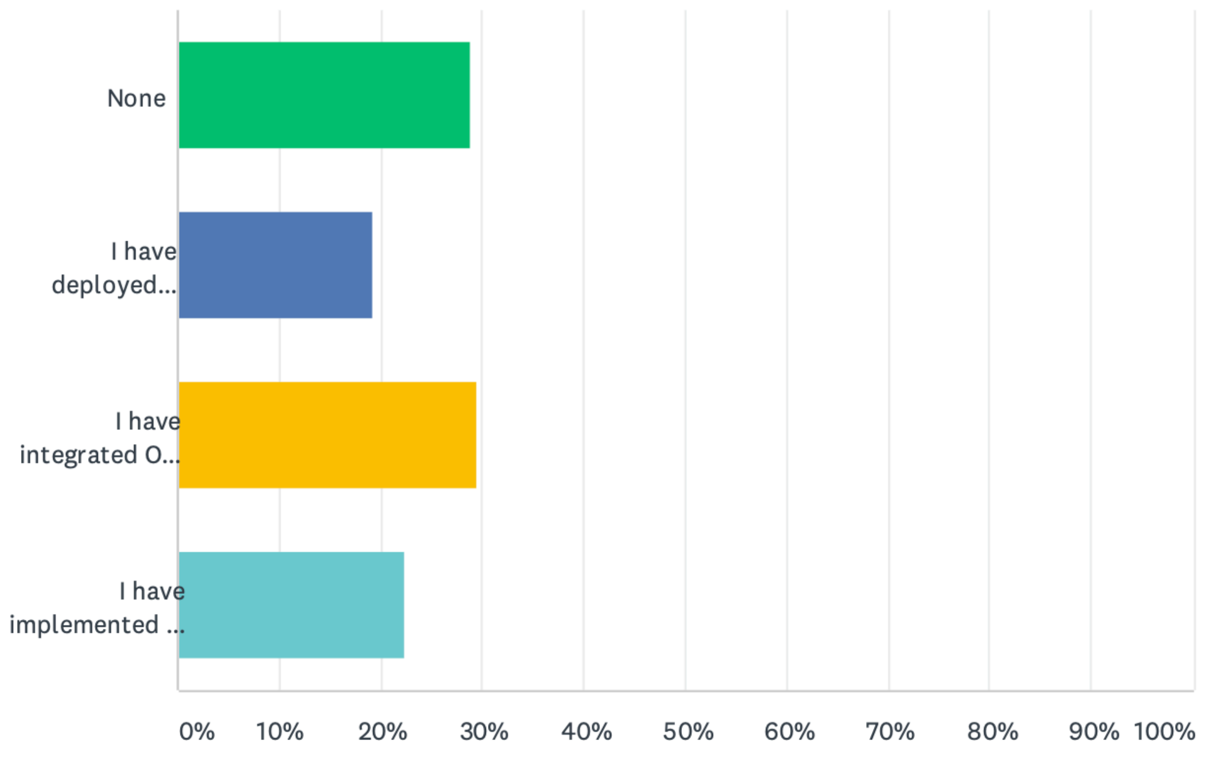
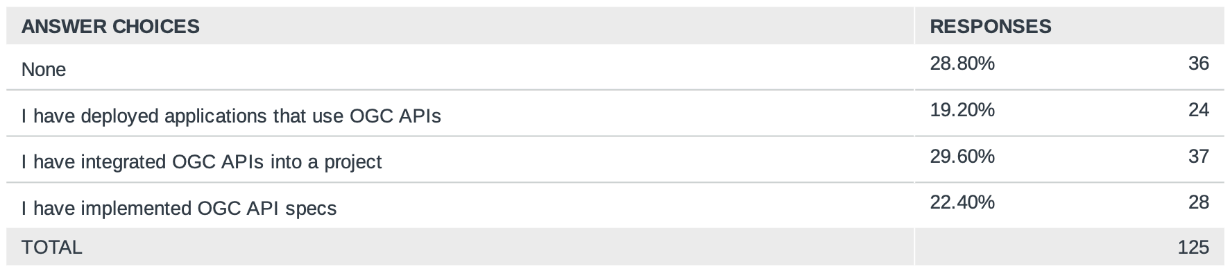


Figure A.49



ANSWER CHOICES	RESPONSES	
None	28.80%	36
I have deployed applications that use OGC APIs	19.20%	24
I have integrated OGC APIs into a project	29.60%	37
I have implemented OGC API specs	22.40%	28
TOTAL		125

Figure A.50

z) **Which OGC APIs are important for your work (mark all that apply)?**

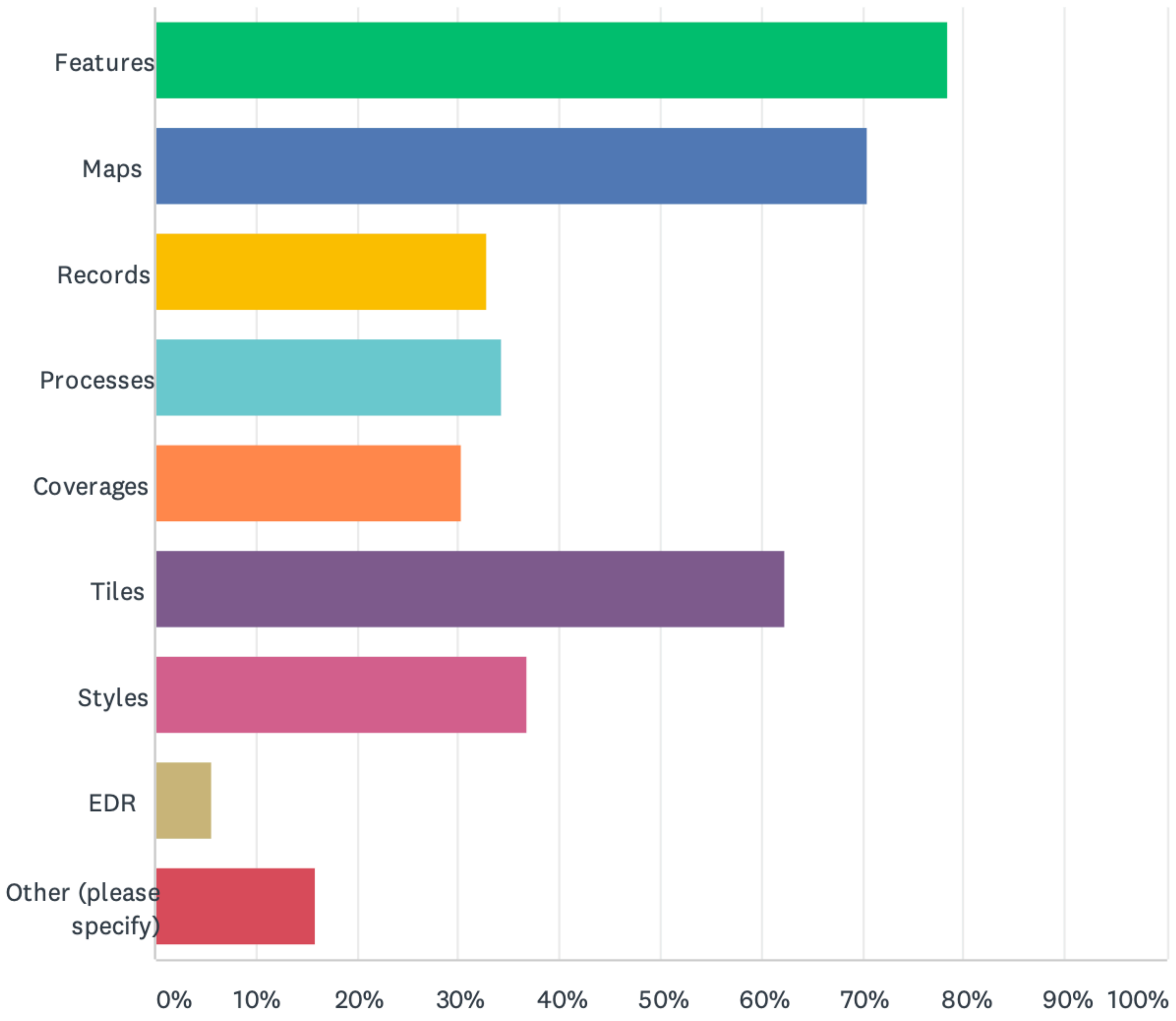
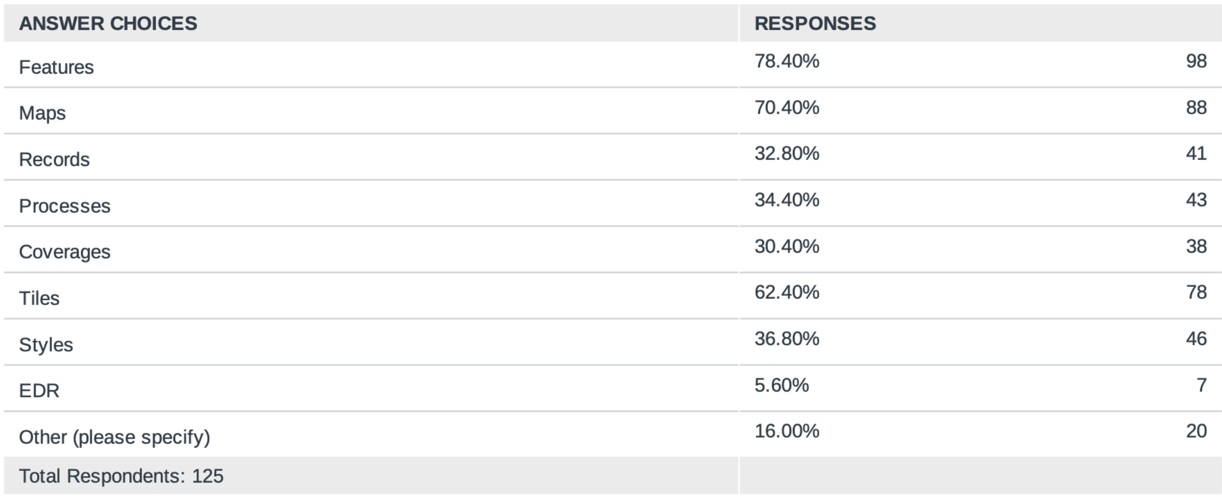


Figure A.51



ANSWER CHOICES	RESPONSES	
Features	78.40%	98
Maps	70.40%	88
Records	32.80%	41
Processes	34.40%	43
Coverages	30.40%	38
Tiles	62.40%	78
Styles	36.80%	46
EDR	5.60%	7
Other (please specify)	16.00%	20
Total Respondents: 125		

Figure A.52

aa) **What technology do you use to implement or deploy OGC APIs?**

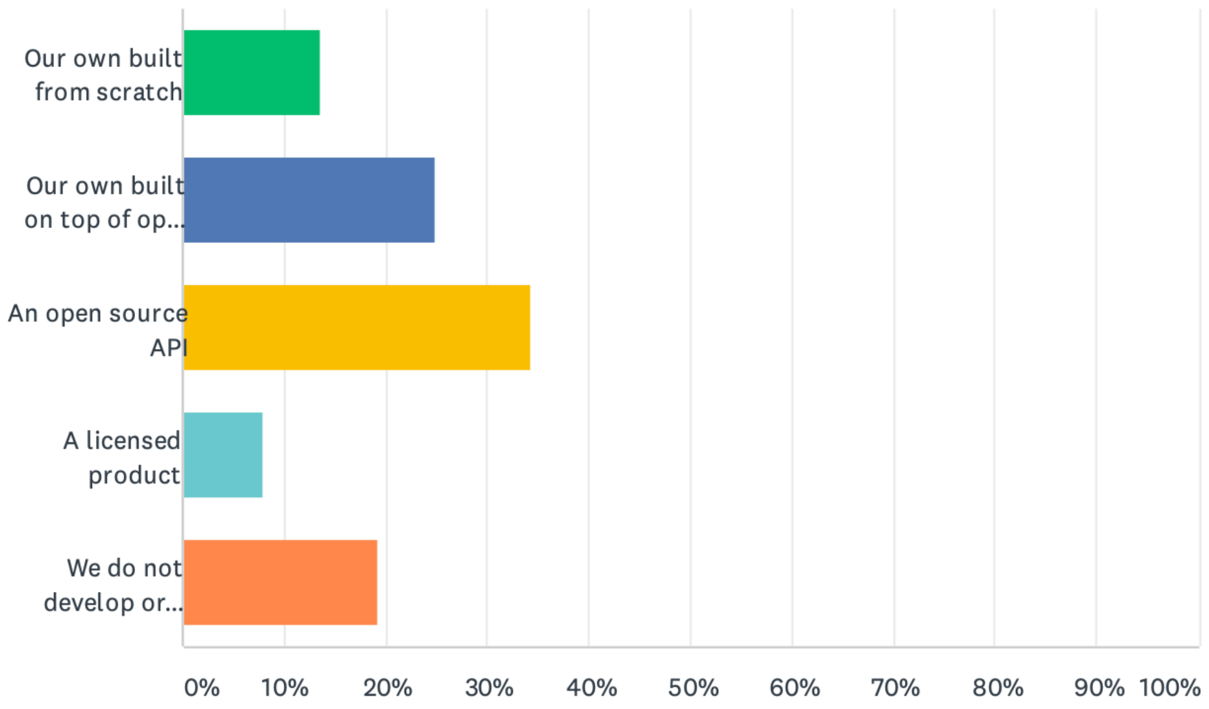


Figure A.53

ANSWER CHOICES	RESPONSES	
Our own built from scratch	13.60%	17
Our own built on top of open source libraries	24.80%	31
An open source API	34.40%	43
A licensed product	8.00%	10
We do not develop or deploy OGC APIs	19.20%	24
TOTAL		125

Figure A.54

ab) **Which tools do you use regularly in your work (please select all that apply)?**

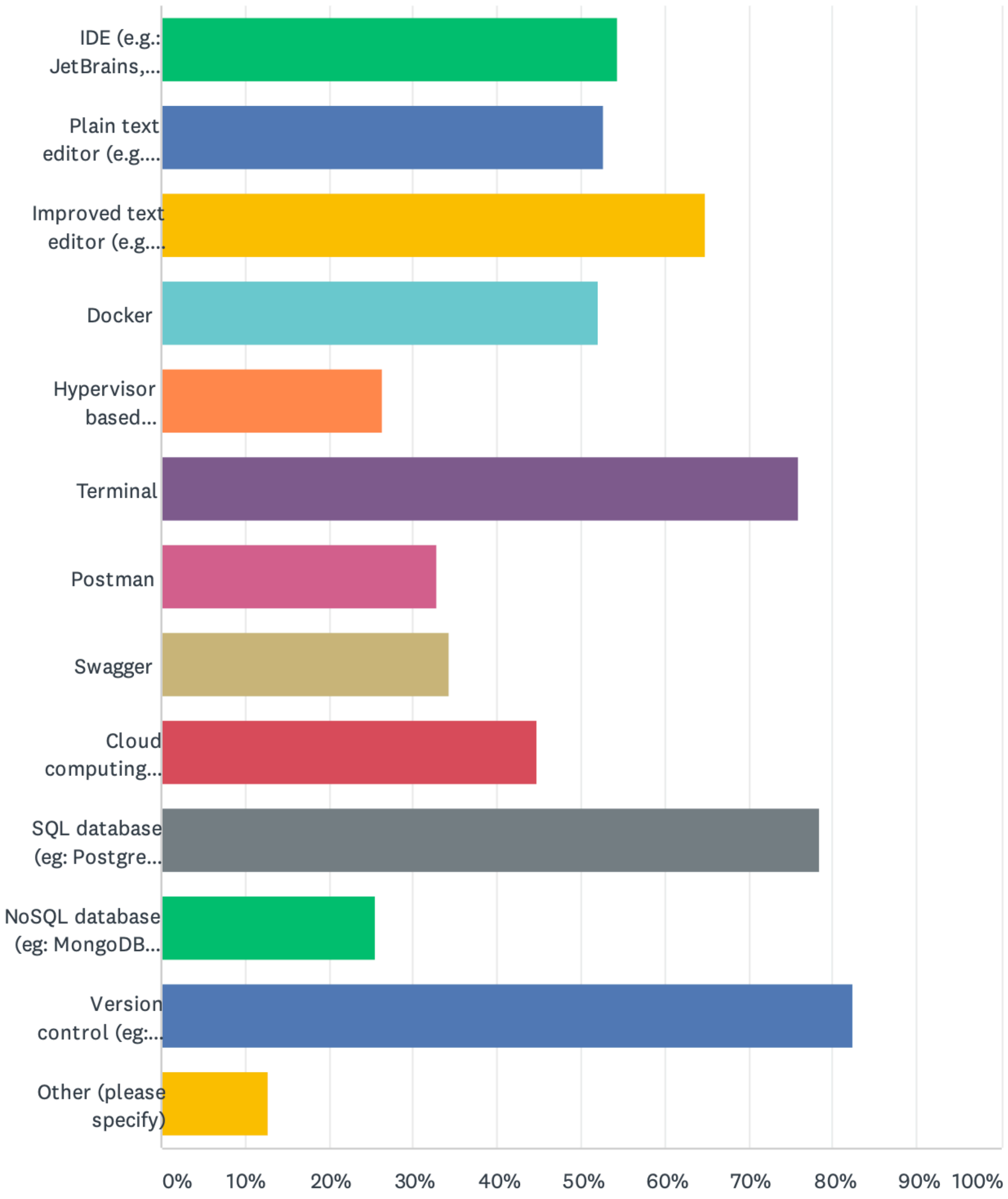
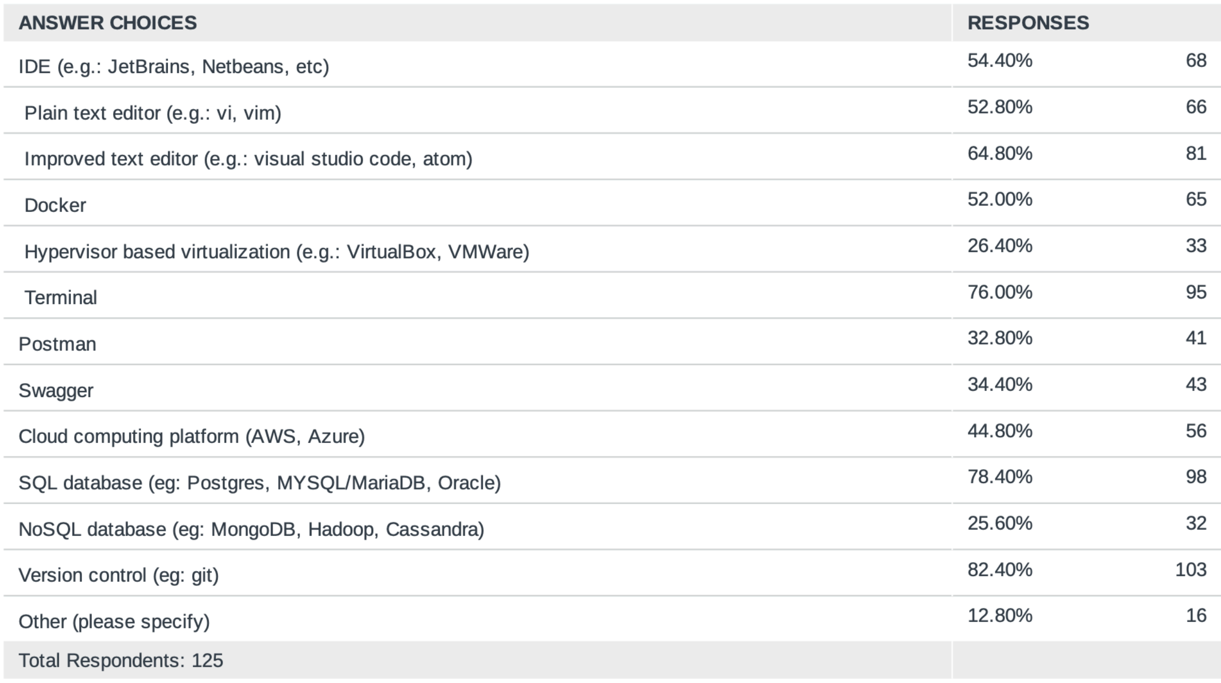


Figure A.55



ANSWER CHOICES	RESPONSES	
IDE (e.g.: JetBrains, Netbeans, etc)	54.40%	68
Plain text editor (e.g.: vi, vim)	52.80%	66
Improved text editor (e.g.: visual studio code, atom)	64.80%	81
Docker	52.00%	65
Hypervisor based virtualization (e.g.: VirtualBox, VMWare)	26.40%	33
Terminal	76.00%	95
Postman	32.80%	41
Swagger	34.40%	43
Cloud computing platform (AWS, Azure)	44.80%	56
SQL database (eg: Postgres, MYSQL/MariaDB, Oracle)	78.40%	98
NoSQL database (eg: MongoDB, Hadoop, Cassandra)	25.60%	32
Version control (eg: git)	82.40%	103
Other (please specify)	12.80%	16
Total Respondents: 125		

Figure A.56

ac) **Which tools or channels do you use to get support from the developer community (please select all that apply)?**

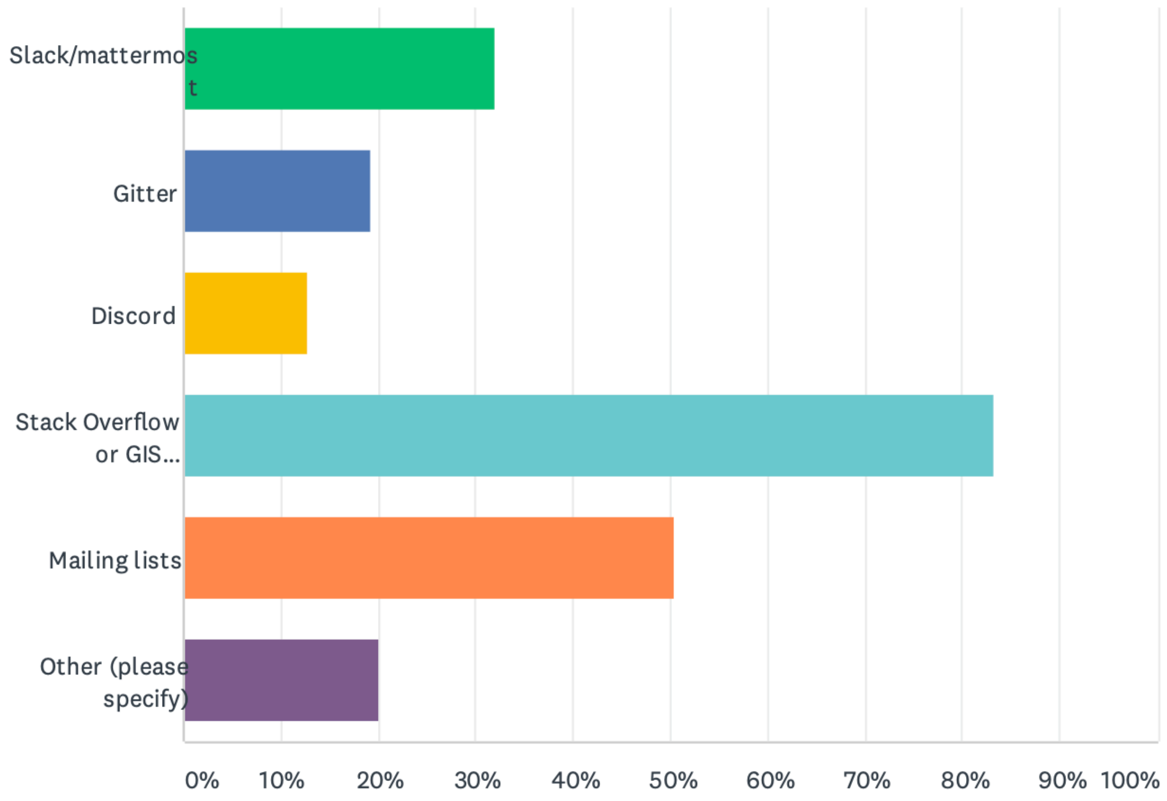


Figure A.57

ANSWER CHOICES	RESPONSES	
Slack/mattermost	32.00%	40
Gitter	19.20%	24
Discord	12.80%	16
Stack Overflow or GIS StackExchange	83.20%	104
Mailing lists	50.40%	63
Other (please specify)	20.00%	25
Total Respondents: 125		

Figure A.58

ad) **Do you consider OGC APIs easy to learn?**

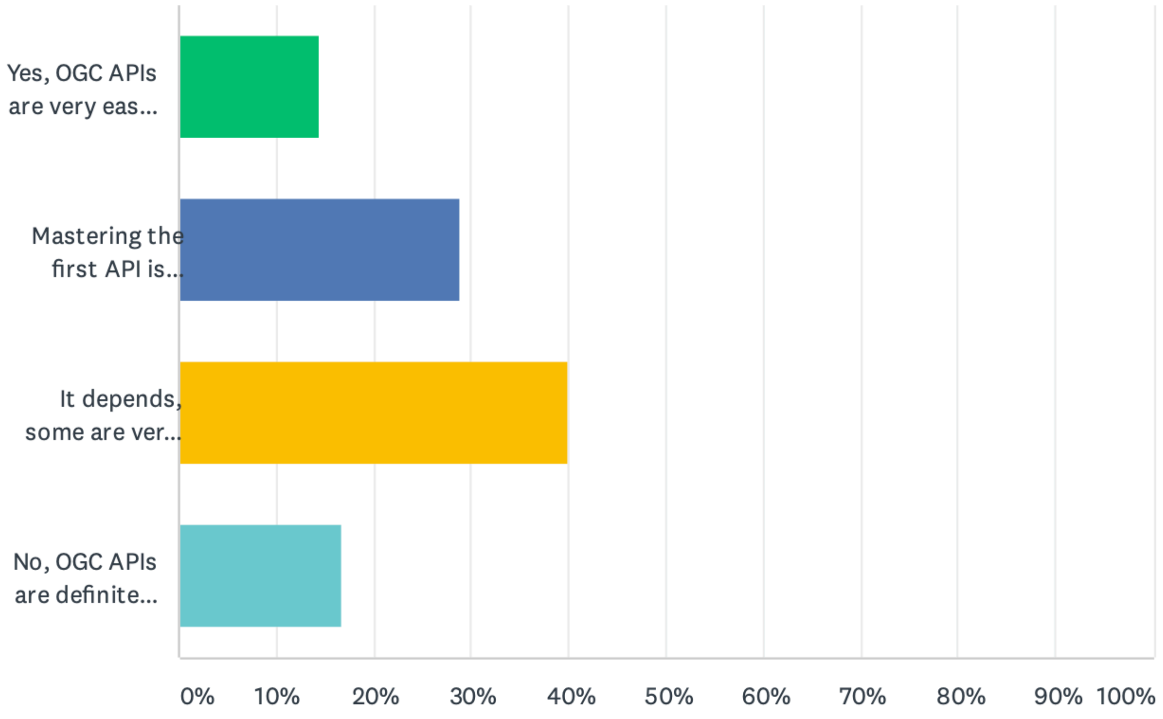


Figure A.59

ANSWER CHOICES	RESPONSES	
Yes, OGC APIs are very easy to learn	14.40%	18
Mastering the first API is hard, but then it becomes easier to learn	28.80%	36
It depends, some are very easy and others are extremely difficult to learn	40.00%	50
No, OGC APIs are definitely hard to learn	16.80%	21
TOTAL		125

Figure A.60

ae) **Have you read an OGC standard specification document?**

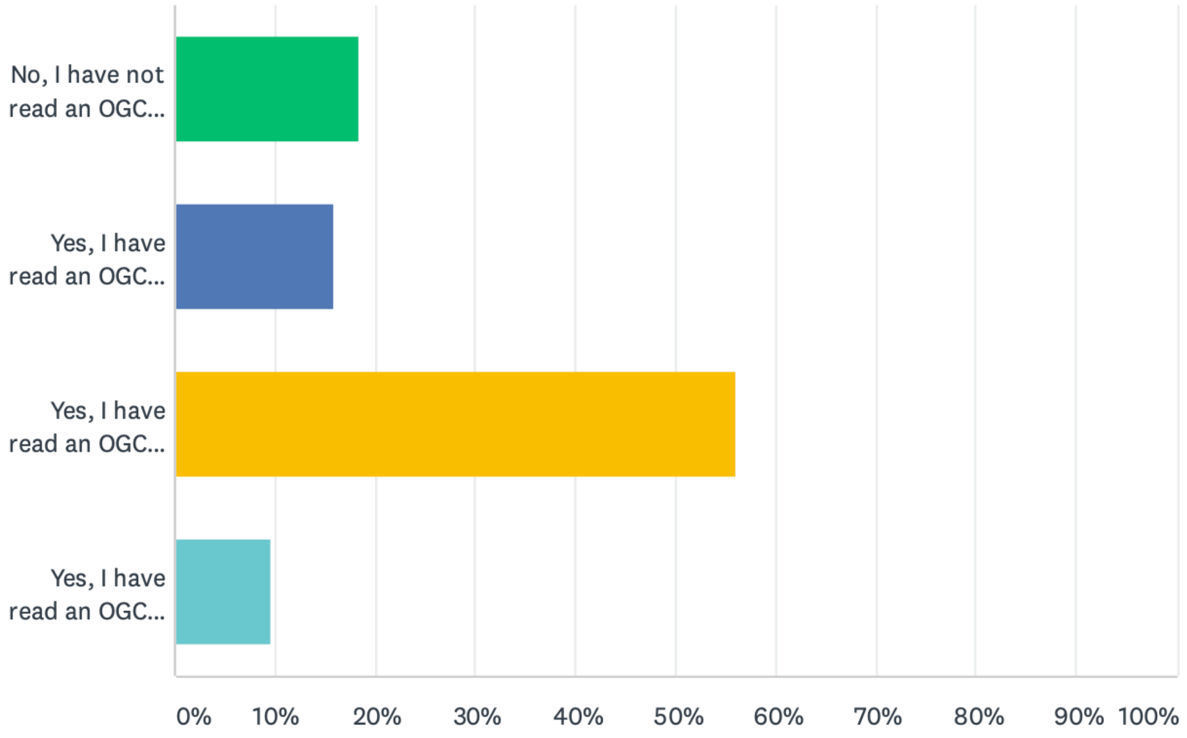


Figure A.61

ANSWER CHOICES	RESPONSES	
No, I have not read an OGC spec document	18.40%	23
Yes, I have read an OGC spec document, but it was not helpful	16.00%	20
Yes, I have read an OGC spec document, and it helped solve some specific issues	56.00%	70
Yes, I have read an OGC spec document, and I did not need any other help	9.60%	12
TOTAL		125

Figure A.62

af) **What are the main challenges you face when developing, deploying and maintaining Geospatial technology (mark all that apply)?**

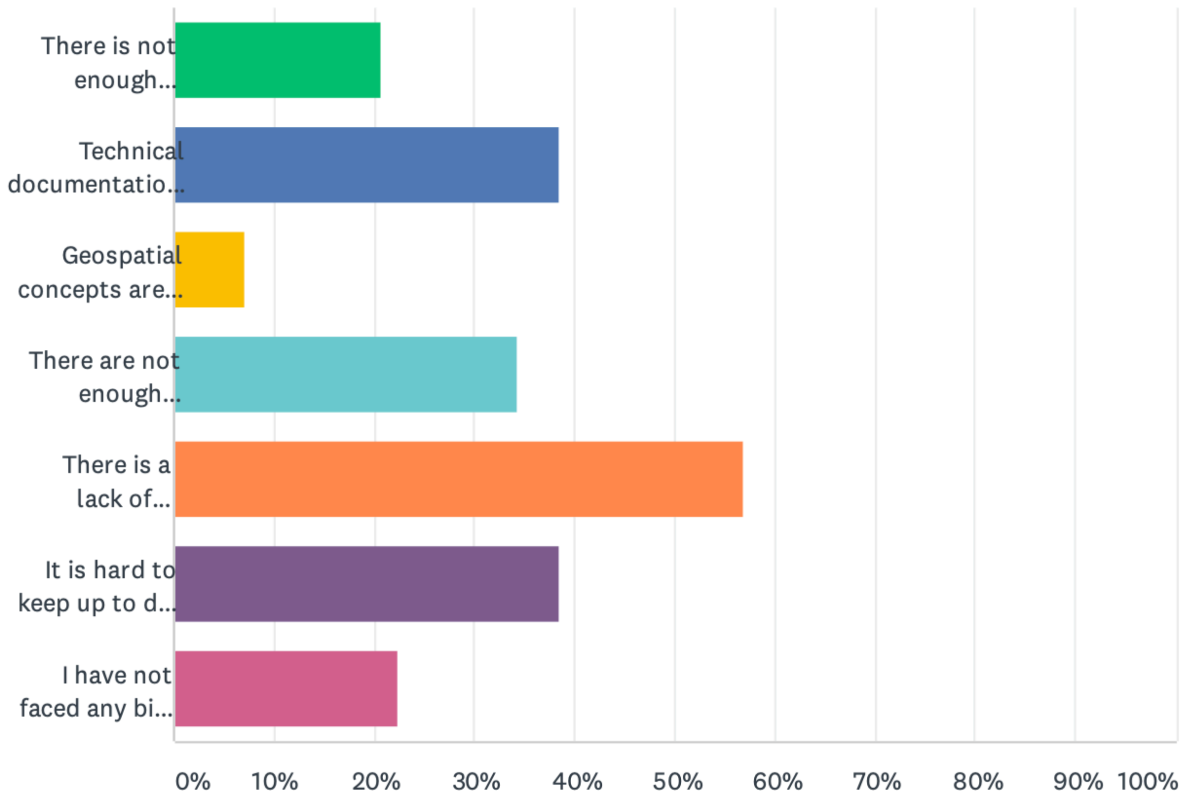


Figure A.63

ANSWER CHOICES	RESPONSES	
There is not enough technical documentation available online	20.80%	26
Technical documentation that is available is hard to read or understand	38.40%	48
Geospatial concepts are difficult to grasp	7.20%	9
There are not enough tutorials available online	34.40%	43
There is a lack of best-practice guides that help differentiate the dos and donts	56.80%	71
It is hard to keep up to date with technology development	38.40%	48
I have not faced any big challenges	22.40%	28
Total Respondents: 125		

Figure A.64

ag) **What sources of documentation and support do you use to answer questions and resolve issues related to Geospatial technologies (mark all that apply)?**

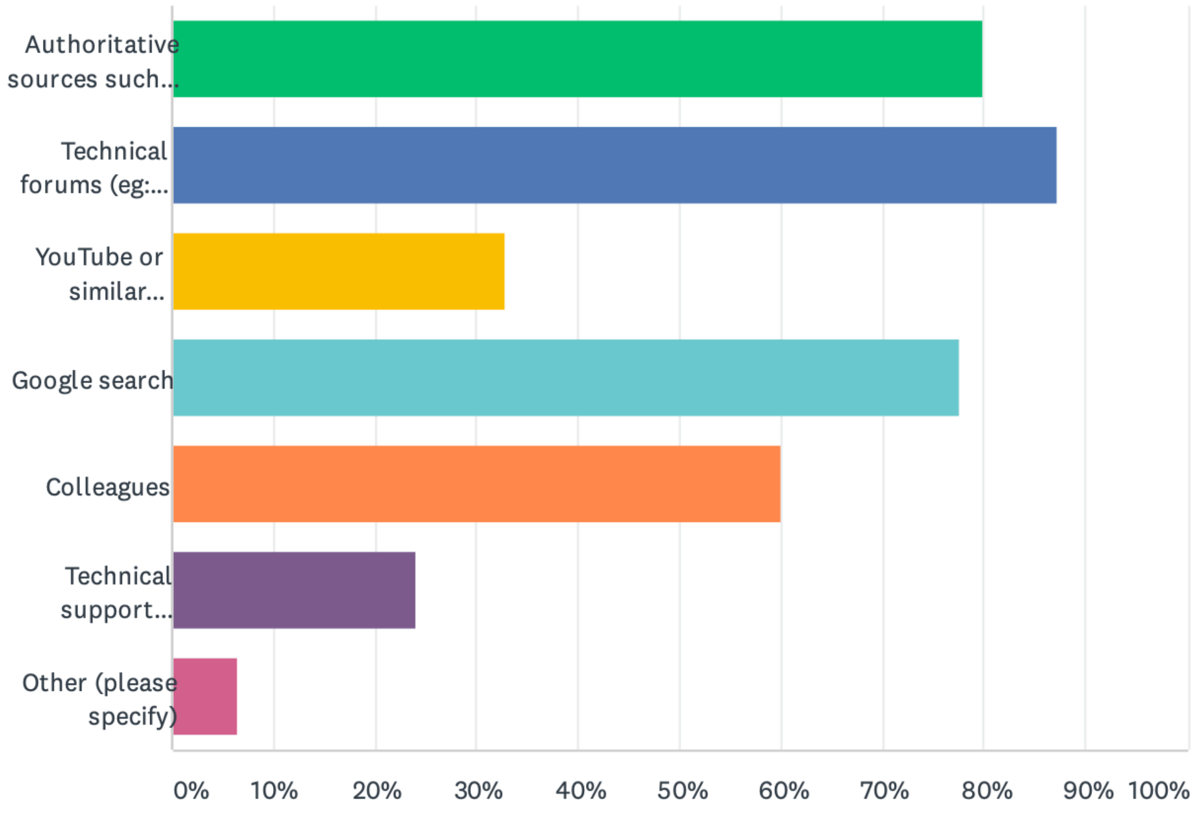


Figure A.65

ANSWER CHOICES	RESPONSES	
Authoritative sources such as official documentation of a specific technology or standard specifications	80.00%	100
Technical forums (eg: stackoverflow, stackexchange)	87.20%	109
YouTube or similar tutorials	32.80%	41
Google search	77.60%	97
Colleagues	60.00%	75
Technical support services (internal or external)	24.00%	30
Other (please specify)	6.40%	8
Total Respondents: 125		

Figure A.66

ah) **Have you ever contributed to a draft specification using Github?**

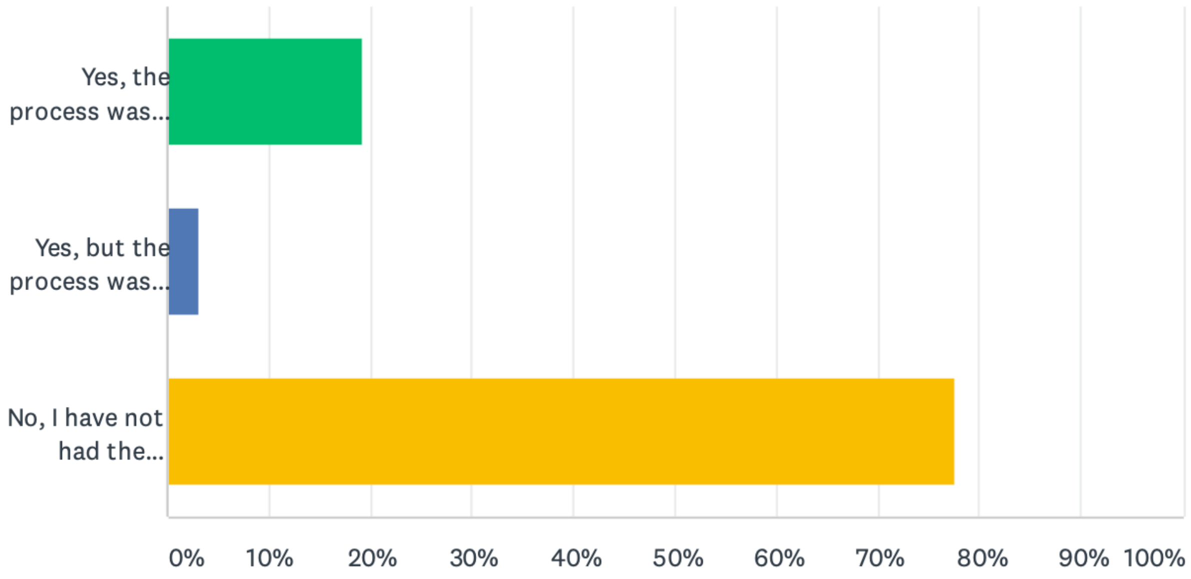


Figure A.67

ANSWER CHOICES	RESPONSES	
Yes, the process was straightforward	19.20%	24
Yes, but the process was difficult or confusing	3.20%	4
No, I have not had the opportunity	77.60%	97
TOTAL		125

Figure A.68

B

ANNEX B (INFORMATIVE) REVISION HISTORY

B

ANNEX B (INFORMATIVE) REVISION HISTORY

DATE	RELEASE	AUTHOR	PRIMARY CLAUSES MODIFIED	DESCRIPTION
May 15, 2021	.1	A. Balaban	all	initial version
Nov 01, 2021	.2	A. Balaban	all	initial draft version
Nov 15, 2021	.3	A. Balaban	all	second draft version
Nov 15, 2021	.4	A. Balaban	all	second draft version
Nov 18, 2021	.5	A. Balaban	all	final draft version

BIBLIOGRAPHY

BIBLIOGRAPHY

1. H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, T. Schaub: IETF RFC 7946, *The GeoJSON Format*. Internet Engineering Task Force, Fremont, CA (2016). https://raw.githubusercontent.com/relaton/relaton-data-ietf/master/data/reference.RFC.7946.xml

2. M. Jones, J. Bradley, N. Sakimura: IETF RFC 7519, *JSON Web Token (JWT)*. Internet Engineering Task Force, Fremont, CA (2015). https://raw.githubusercontent.com/relaton/relaton-data-ietf/master/data/reference.RFC.7519.xml

3. OGC GML in JPEG 2000 (GMLJP2) Encoding Standard, http://docs.opengeospatial.org/is/08-085r8/08-085r8.html

4. OGC API — Features, http://docs.opengeospatial.org/is/17-069r3/17-069r3.html

5. OGC API — Environmental Data Retrieval Standard, https://docs.ogc.org/is/19-086r4/19-086r4.html