OBP

Phase 1 Report

TAGGING AND UI PILOT

\_\_\_\_\_\_\_



[**Overview**](#_c70ai3okxqe) **2**

[**Application Details**](#_j7oq2y17b11c) **3**

[Module 1: Semantic Tagging of OBP-R Documents](#_fsj1kt346qxk) 3

[Overview](#_r6swpwko2o8v) 3

[Technical Approach](#_gk9rry97snl2) 4

[Module 2: Semantically Enhanced Search and Browse](#_jrpahjjvsetv) 6

[Overview](#_vi6ue2ql124n) 6

[Technical Approach](#_on1i02p9z5gh) 7

[Module 3: User Interfaces](#_73mwdbqd59zp) 8

[Overview](#_nw1xnziradck) 8

[Landing Page](#_yvgtzg7jwvtu) 9

[Search Results](#_l8izvft6shzu) 10

[Document Tags and Relationships](#_myaah02aqdk5) 11

[Technical Approach](#_6ky9ekny5prr) 13

[**Continuation of Development**](#_panti0zgces6) **13**

[Improved Semantic Tagging of OBP-R Documents](#_66ggeboml4dl) 14

[Semantically Enhanced Search and Browse Capabilities](#_7xwuv8vtjxhb) 14

[User Interface Improvements](#_8fpgoc3w8ddt) 15

[Architecture and System Improvements](#_i632bzc18as2) 15

[**Appendix 1: System Architecture**](#_tgu20pjzip5s) **16**

# Overview

The OceanBestPractices Repository (OBP-R) is a DSpace-based repository providing simple search capabilities. The goal of this effort is to move toward a more powerful, semantic search tool for users to more effectively find documents within  
the repository.

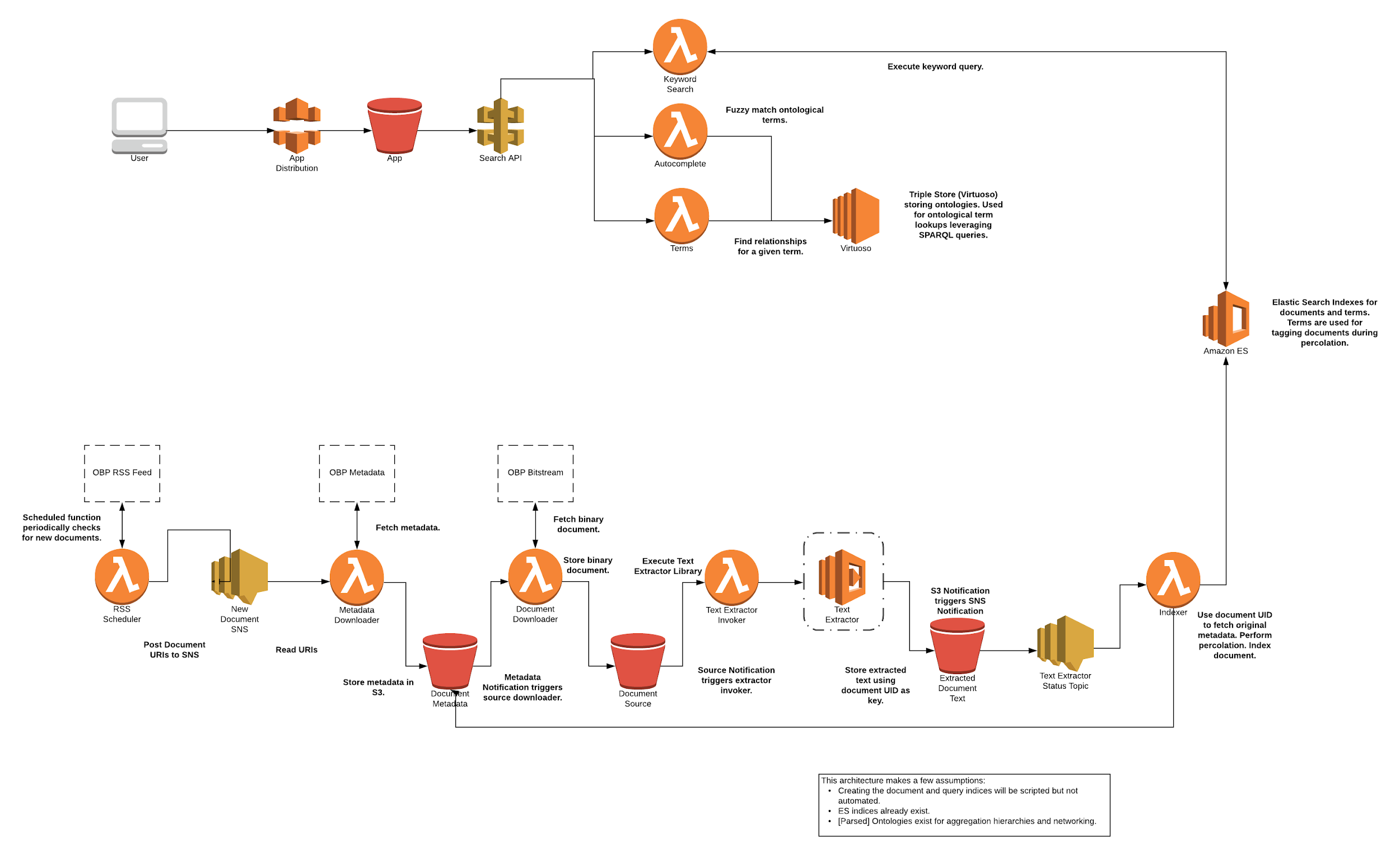
Element 84 was tasked to create an application that ingests new documents found in the OBP-R, maps these documents against specified ontologies, and provides an interface allowing users to search ingested documents in a semantic fashion.

The work was broken up into three modules for development:

**Module 1**: Semantic tagging of OBP-R documents

**Module 2**: Semantically enhanced search and browse capacities

**Module 3**: User Interfaces



**Figure 1:** Overall system architecture (see Appendix for larger version).

# Application Details

## **Module 1:** Semantic Tagging of OBP-R Documents

### Overview

A modular infrastructure was developed in order to support the key features of  
Module 1:

* Automatic discovery of new documents
* Metadata and document ingest and text extraction
* Semantic tagging, indexing, and availability

The system built to support Module 1 was developed around key Amazon Web Services (AWS) technologies including server-less Lambda functions, Simple Notification Service, a managed Elasticsearch cluster, and S3 for storage. In addition to AWS, the system relies on the DSpace API for document discovery and metadata retrieval.

The system queries the DSpace RSS feed on a regular interval in order to discover recently published documents. Leveraging the DSpace API the system fetches the metadata and original PDF source document, and stores these artifacts on S3. Artifacts stored on S3 automatically trigger the text extraction routines, which fetch and extract text from the original PDF document. The raw document text is once again stored  
on S3.

Text extraction automatically triggers Lambda functions that fetch the document metadata and raw text. Document text is then percolated against an existing Elasticsearch “tags” index, which includes pre-defined queries matching tags extracted from the ingested ontologies supported by the system. The percolator, or reverse search, generates a list of matching tags that are then combined with the original document metadata. Document metadata and raw text are then added to the Elasticsearch documents index and are available for discovery.

### 

**Figure 2:**  Module 1 architecture.

### Technical Approach

Module 1 is built to leverage easily accessible and transferable technologies provided by AWS. Specifically, Module 1 is an almost entirely server-less system built on top of AWS’ Lambda service. A significant benefit to a server-less infrastructure is that cost is reduced by only being charged when the system is in use. Since Lambda functions are executed on-demand, costs are reduced by not having to pay for running servers while the system is not in use.

Lambda functions are written in JavaScript leveraging a Node.js environment. These technologies allow for an easily transferable code base if the system is moved out of an AWS environment.

Document metadata and sources are stored in various buckets in S3, which provides a cost effective and highly available storage solution.

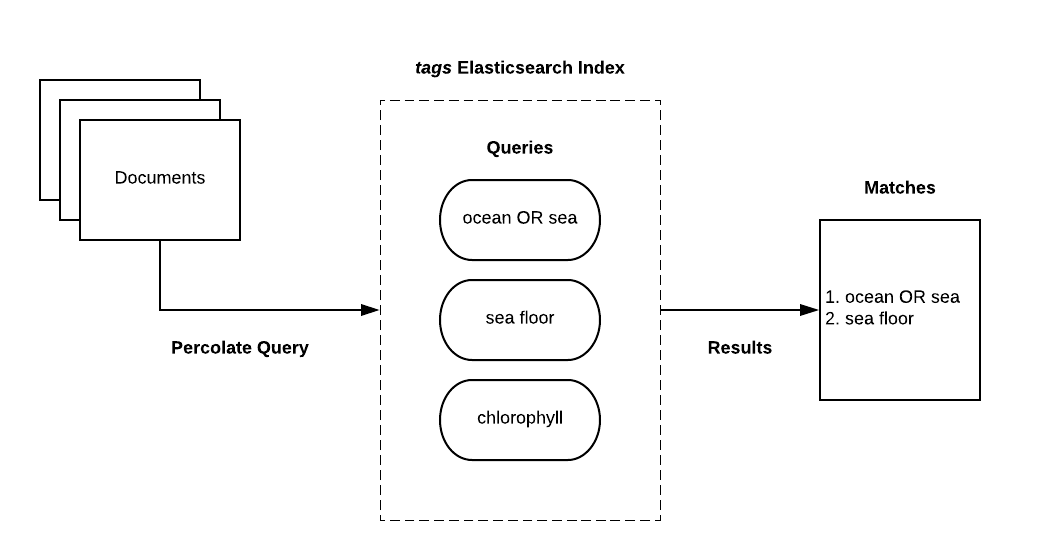
Specific Module 1 tasks, such as ingest and text extraction, are triggered by a combination of S3 notifications and Simple Notification Service (SNS) topics. When document metadata is made available in S3, a notification triggers the next Lambda function. The system utilizes the document’s **uuid** metadata field in order to uniquely identify documents and metadata across S3 buckets.

In order to provide modularity, S3 notifications may publish to existing SNS topics which can be subscribed to by other AWS services. In the case of Module 1, Lambda functions subscribe to SNS topics in order to be triggered when new data is made available. This provides a modular framework where specific components can be swapped in for different technologies if so desired.

Document indexing and search is provided by Elasticsearch, a distributed and powerful search engine. Module 1 leverages AWS’ managed Elasticsearch service which provides an easy-to-configure Elasticsearch cluster with minimal spin up time. Document metadata and raw text are added to an Elasticsearch index, tokenized, and made available for search.

In order to support semantic tagging of documents the Elasticsearch Percolator is used. This feature, which you can read more about at <https://www.elastic.co/blog/percolator>, provides a reverse search interface for matching queries. Tags are fetched from multiple ontologies which are ingested into a self-hosted instance of Virtuoso - a triple-store database that provides a SPARQL query API. Elasticsearch queries are then created from the tags and added to the *tags* index - a different index from where the document metadata is stored. When a document is percolated we ask Elasticsearch which queries would match the given document. From the returned queries from the *tags* we parse out the individual tags and add them to the document metadata.

The Percolator allows us to leverage the efficiency of Elasticsearch to provide us with matching semantic tags given a document and its metadata. The Percolator index can be updated at any time to improve or change the queries that match a document. For example: currently in Module 1 the Percolator index is designed to match exact tags for a given document. The indexed queries should be updated to match tags with OR logic in order to support exact synonyms.



**Figure 3:**  Tagging documents with the Elasticsearch Percolator.

The *tags* and *documents* indices are created in Elasticsearch using a set of Ruby scripts provided in the project repository. The *tags* index can be populated using additional sets of Ruby scripts found in the project.

Module 2 components are defined across multiple Cloud Formation templates which allow for quick and automated deployments via AWS.

## **Module 2:** Semantically Enhanced Search and Browse

### Overview

Search and browse are exposed to clients (see Module 3) via a highly available and publicly accessible API built on top of AWS’ API Gateway service. The API provides multiple endpoints to support search, semantic relationship graphs, general system statistics, and tag autocompletion. Clients can request search results by sending keywords entered by the user or by filtering current search results by semantic tags. The API provides all results in JSON format.

In addition to the API, Module 2 also includes an instance of Virtuoso - a database engine which stores and makes available our system’s semantic ontologies. The Virtuoso instance runs on AWS’ EC2 service and exposes a SPARQL query endpoint.

### 

**Figure 4:** Module 2 architecture.

### Technical Approach

The search and browse APIs are built on AWS’ API Gateway service which provides a highly available, distributed, and scalable API. API Gateway allows for multiple environments exposing the same API. Currently, only a *prod* API is available. Multiple endpoints are provided which support the client built in Module 3 but could also support 3rd party clients if so desired:

**/documents**

* **GET**
  + Find documents matching a given query parameter keyword. Multiple parameters are supported in the query including *term*, *sort,* and *limit*.

**/autocomplete**

* **GET**
  + Provides a list of tags that match the given term in the query parameter. Useful for providing a list of potential matching tags as the user types in the search box.

**/statistics**

* **GET**
  + Returns system statistics including number of documents, tags, and ontologies.

**/terms**

* **/graph**
  + Returns an object describing the semantic relationships between the given term as a query parameter and related semantic tags (e.g. is\_a, part\_of, etc.).

All API responses return data in JSON format and are easily parsed by a  
3rd party application.

While API Gateway exposes endpoints useful for building a user facing application, the actual implementation of the API is handled by AWS’ Lambda service. This provides for scalable on-demand execution and cost efficiency. Each API method has a dedicated Lambda function written in JavaScript in a Node.js environment. These standard technologies allow for easy migration to an on-premise infrastructure in the future.

In some cases the API methods execute queries against the Elasticsearch cluster and the Virtuoso instance running on AWS’ managed Elasticsearch service and EC2 respectively. The Lambda functions execute a query and parse the results before returning a JSON response. In some cases, the response from Elasticsearch is passed through to the client without any intermediary parsing.

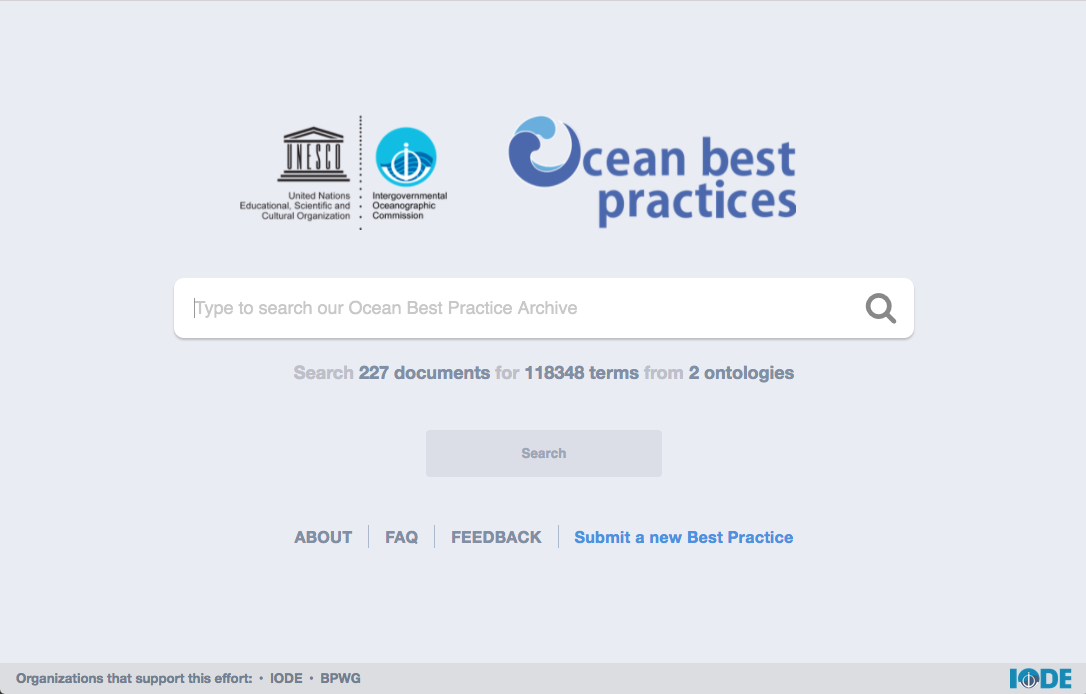
## **Module 3:** User Interfaces

### Overview

A modern, simple user interface was developed to enable users to find relevant documents of interest. The user interface is composed of two views: a landing page view (<https://www.oceanbestpractices.org/>) and a search exploration view (<https://www.oceanbestpractices.org/search>).

#### Landing Page

The landing page allows users to begin their search by entering a query in the prominently displayed search box. As the user begins typing in the search box, ontology matches are suggested. The page also provides basic metrics about the repository -- number of documents and number of terms from how many ontologies. The homepage also includes relevant logos and footer links.



**Figure 5**: Homepage of the application.



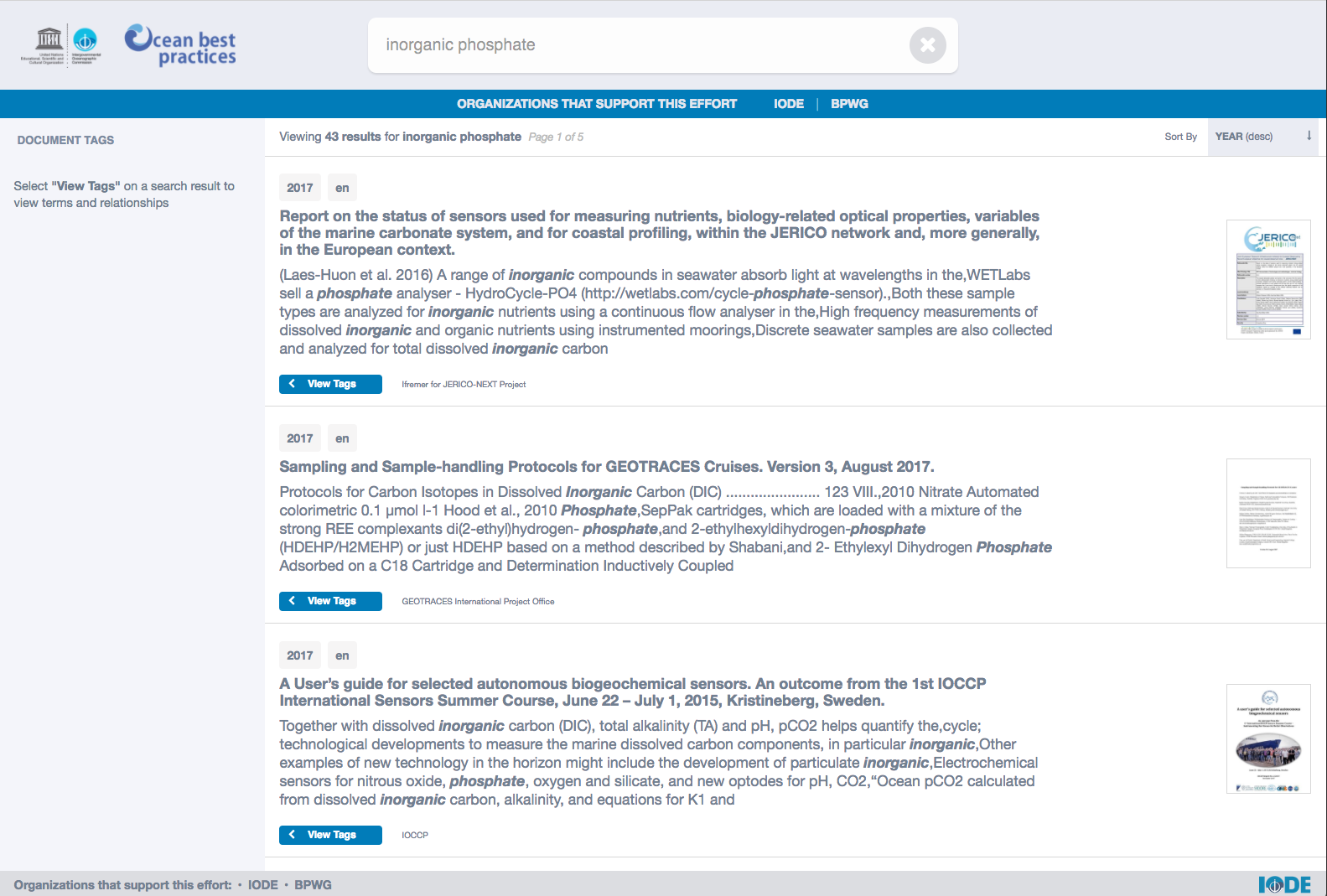
**Figure 6:** Predictive search based on ontological terms.

#### Search Results

After submitting a query, the user is taken to a search results page where they are presented with initial results and can further refine and adjust their query. Each search result provides document metadata (published year, language, title, thumbnail, and publisher) as well as a snippet showing the most relevant location within the document for the search term.

Search results can be sorted relevancy, year, title, author, and publisher. Multi-page navigation is provided for queries with greater than 10 results.

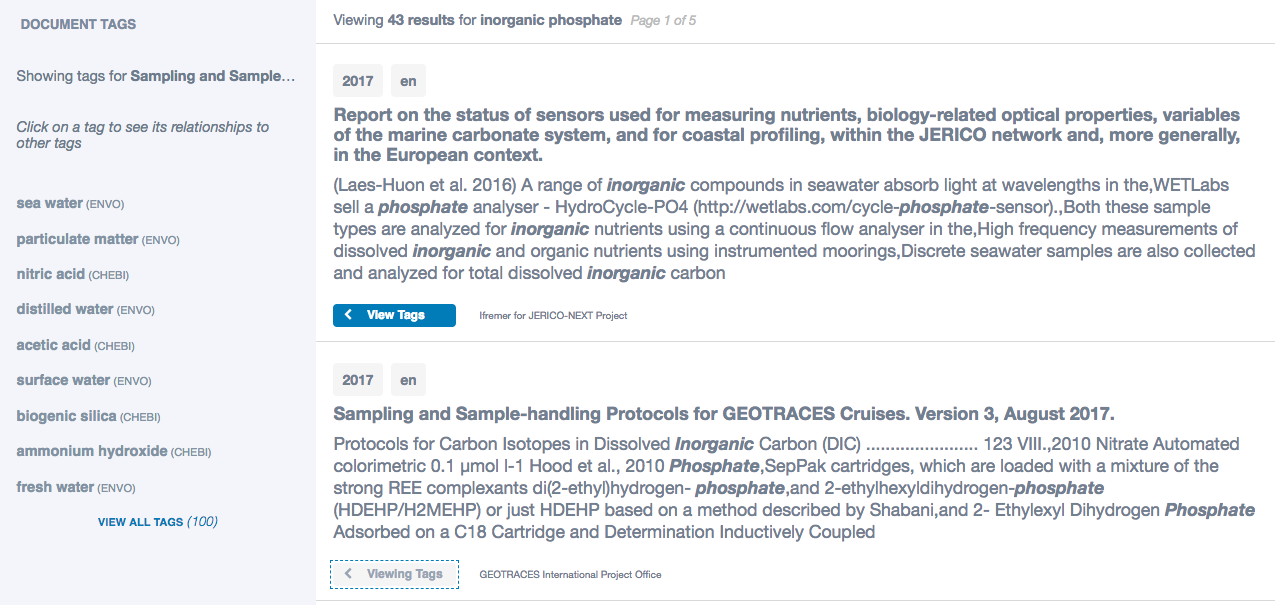
Clicking on a document title within the results opens a new tab that loads the OBP-R page for that document.



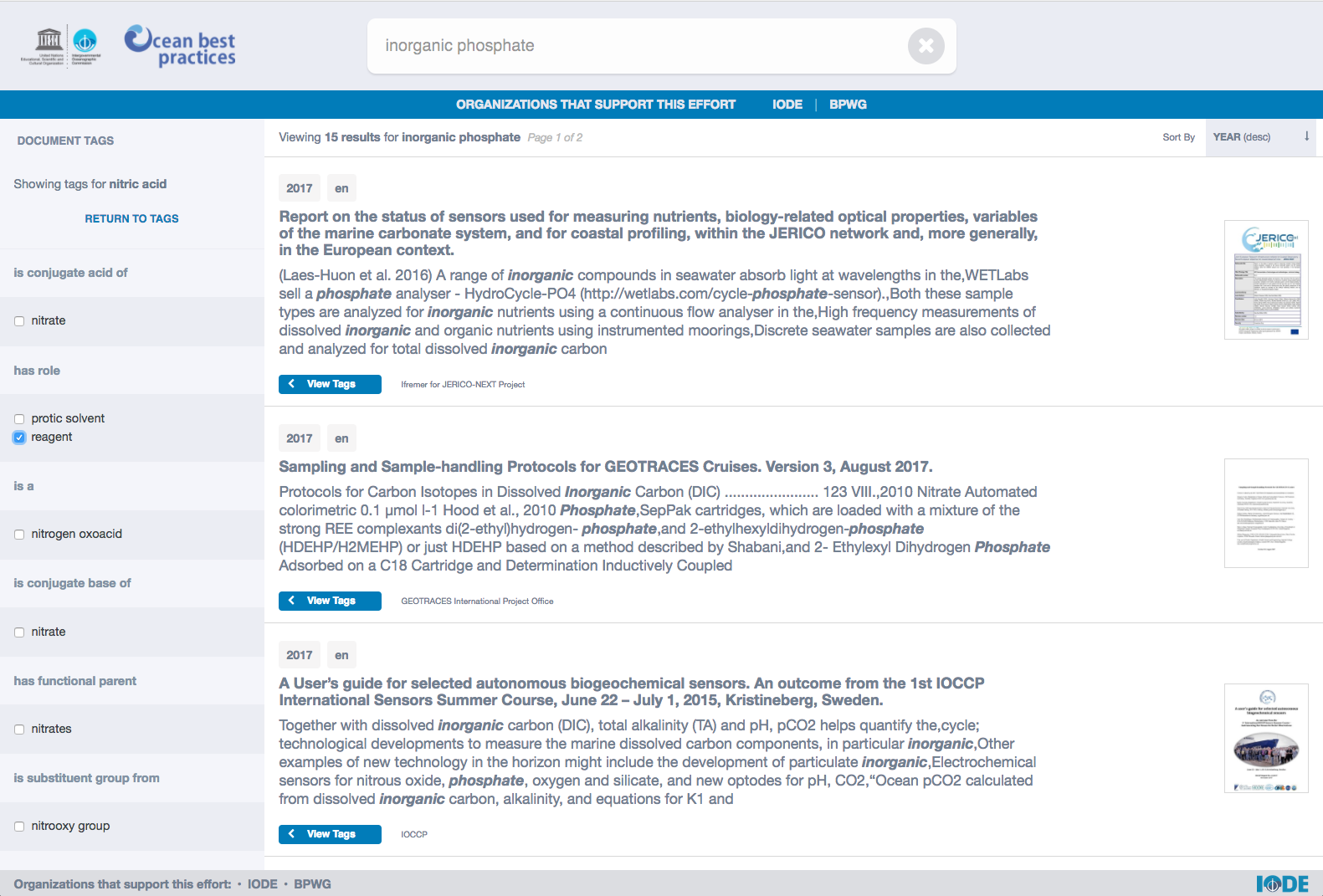
**Figure 7**: Search results page.

#### Document Tags and Relationships

Selecting the “View Tags” button in a search result will open the Document Tags panel. A subset of all related tags is presented to the user as well as an option to view all tags. Once a tag is selected, the user is presented with various relationships associated with that tag. Selecting a relationship will further constrict search results.



**Figure 8**: Document tags panel on left.



**Figure 9**: Constricting results based on tag relationships.

### Technical Approach

The user interface is a React.js single page application (SPA) that utilizes Redux to store the application’s various states and Router.js to load, show, and hide "pages" dynamically when URLs are requested without having to refresh the page.

One of React.js’ main strengths is its focus on creating reusable, composable, and stateful components. For example, each search result returned (Fig. 7), is a ‘SearchResult’ component, and is populated for each document returned from a query. Additionally, each component’s output is updated when the component’s state changes, allowing for a reactive and dynamic application.

Redux stores the states of the application’s components. For instance, the application stores the state of the search bar, the document tags selected, and the search filter. The components of the application can ‘listen’ to the Redux store to see if any of the application’s states have updated, and then update itself accordingly. If a user were to change the search filter, the search results would update according to the filter.

In terms of styling, the SPA is using Block Element Modifier (BEM) based Sassy CSS (SCSS) that helps create reusable components and encourages code sharing in front-end development. All of which are employed by using BEM’s naming conventions. Again, the application was created with modularity at the forefront of development.

The decision to go with React.js (and the accompanying technology)  
resulted in a componentized application that did not rely on the completion of the other modules.

# Continuation of Development

The next phase of development will include improvements to tagging, search, user interface, and system architecture.

## Improved Semantic Tagging of OBP-R Documents

The pilot was designed to only ingest PDF documents. We will expand this to include Microsoft Word (DOCX) documents and improve text extraction to include document structure, so semantic tags can be linked to sections within the document. Furthermore, tagging routines will be improved to tag text based on partial label matches and to use exact synonyms, alternative labels, and related annotation properties that are stored in the system’s ontologies.

Additional ocean and marine science ontologies will be imported into the application. All ontologies will be automatically updated periodically by the application; and the document corpus will be re-tagged as appropriate.

The OBP-R corpus will be analyzed to identify commonly used terminology to generate new, informal classes for tagging. This will create a broader base of terms that users would expect to see when searching for documents within the OBP-R.

## Semantically Enhanced Search and Browse Capabilities

Search will be improved to include frequently occuring n-grams within the OBP-R document corpus, as well as to allow partial matching of ontology class labels and annotation properties. Search result boosting will be tuned such that hits on tags in document titles will be presented as more relevant.

Metadata fields will also be made searchable, and users will be able to further refine search results by specifying specific metadata they wish to explore (e.g. restrict results to those published by the World Meteorological Organization).

Users will be able to view the short record display on the search results page by hovering over a result; removing the need to click and navigate to a new browser tab to learn more about the result. Users will also be presented with a means to navigate search hits within a document.

We will add an advanced search capability that will help users create more complex queries (with examples) and provide a SPARQL interface to facilitate advanced searching by power users.

## User Interface Improvements

Feedback from the pilot effort will be used to improve the user interface and user experience, including: the current query will be better displayed, such that the user  
can see the exact query that was submitted (and any modifications), tool tips and search examples will be added throughout to help users use the system effectively, and support for viewing on mobile devices.

An interactive, graphical interface showing search results and their “neighborhood”  
of related tags will be presented to users, allowing them to further expand or restrict their search.

A news feed will be added to the home page and a simple content management system will be created to allow administrators to make changes to the interface text.

## Architecture and System Improvements

A number of system-level improvements have been identified for future development.

Architecturally, the system will be adapted such that each module (ingest, search, UI) is self-contained and can be migrated to an on-premises IODE environment and for long-term updating and maintenance. A backup and disaster recovery plan will be established to ensure system resiliency.

Application analytics will be enabled to track user search terms, behaviors, and interactions. Data gathered from these analytics will inform the team where further improvements in tagging, search, and user interface are needed, as well as help the team gain insight into the types of users and popular topics.

To further improve the user experience, we’ll track user search history and boost results that have tags matching their previous searches. We will also enable the export of results to users in JSON and CSV formats.

# Appendix 1: System Architecture

