The IPOL Demo System: A Scalable Architecture of Microservices for Reproducible Research

Martín Arévalo[1], Carlos Escobar[2], Pascal Monasse[3], Nelson Monzón[4], and Miguel Colom[2](✉)

[1] Department of Biological Engineering, Universidad de la República, Montevideo, Uruguay
marevalo@cup.edu.uy

[2] CMLA, ENS Cachan, CNRS, Université Paris-Saclay, 94235 Cachan, France
carlos.escobar101@alu.ulpgc.es, colom@cmla.ens-cachan.fr

[3] LIGM, UMR 8049, École des Ponts, UPE, Champs-sur-Marne, France
pascal.monasse@enpc.fr

[4] CTIM, University of Las Palmas de Gran Canaria, Las Palmas, Spain
monzon@ctim.es

Abstract. We identified design problems related to the architecture, ergonomy, and performance in the previous version of the Image Processing on Line (IPOL) demonstration system. In order to correct them we moved to an architecture of microservices and performed many refactorings. This article first describes the state of the art in Reproducible Research platforms and explains IPOL in that context. The specific problems which were found are discussed, along with the solutions implemented in the new demo system, and the changes in its architecture with respect to the previous system. Finally, we expose the challenges of the system in the short term.

Keywords: IPOL · Reproducible research · Research · Journal · SOA · Microservices · Service-oriented · Platform · Continuous integration

1 Introduction

Image Processing on Line (IPOL) is a research journal started in 2010 on Reproducible Research in the field of Signal Processing (mainly Image Processing, but also video, sounds, and 3D data), giving a special emphasis on the role of mathematics in the design of the algorithms [1]. This article discusses the current system after the changes that were anticipated in [2], and towards which direction it plans to move in the future.

As pointed by Donoho et al. [3], there is a crisis of scientific credibility since in many published papers it is not possible for the readers to reproduce exactly

All authors contributed equally.

the same results given by the authors. The causes are many, including incomplete descriptions in the manuscripts, not releasing the source code, or that the published algorithm does not correspond to what actually is implemented. Each IPOL article has an online demo associated which allows users to run the algorithms with their own data; the reviewers of the IPOL articles must carefully check that both the description and the implementation match.

Since it started in 2010, the IPOL demo system has been continuously improved and according to usage statistics collected along these years, it has about 250 unique visitors per day. However, several problems of design and potential improvement actions were identified and, on February 2015, it was decided to build a second version of the system based on microservices [4]. Among these problems can be listed: the lack of modularity, tightly-coupled interfaces, difficulties to share the computational load along different machines, or complicated debugging of the system in case of malfunction.

The plan of the article follows. Section 2 discusses the state of the art in Reproducible Research and microservices platforms. Section 3 discusses the particularities of IPOL as a journal, and Sect. 4 presents the architecture of microservices of the new IPOL demo system. Section 5 reveals the software development methodologies in the software engineering process the IPOL team is applying internally. Section 6 refers to a particular tool we designed for the IPOL editors which allows them to manage the editorial process. Section 7 presents a very important novelty of the new system, which is the capability of quickly creating new demos from a textual description. Finally, Sect. 8 presents the conclusions.

2 State of the Art in Reproducible Research Platforms

Some very well-known platforms whose use is closely related to Reproducible Research exist nowadays. Some of them are domain-specific while others are more general.

In the case of Biology, the Galaxy project [5] is a platform for genomic research which makes available tools which can be used by non-expert users too. Galaxy defines a *workflow* as a reusable template which contains different algorithms applied to the input data. In order to achieve reproducibility the system stores: the input dataset, the tools and algorithms which were applied to the data within the chain, the parameters, and the output dataset. Thus, performing the same workflow with the same data ensures that the same results are obtained given that the version of all the elements is kept the same.

In the field of Document Image Analysis, the Document Analysis and Exploitation platform (DAE) was designed to share and distribute document image with algorithms. Created from 2012, the DAE platform also adds tools to exploit annotation and perform benchmarking [6].

Generic tools for Reproducible Research include the IPython tool and its notebooks. This mature tool created in 2001 allows to create reproducible articles by not only editing text in the notebook, but allowing code execution and creating figures *in situ*. This approach follows closely the definition of a "reproducible scientific publication" given by Claerbout and followed also by Buckheit and Donoho: *An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures* [7].

In 2014 the Jupyter project was started as a spin-off of IPython in order to separate the Python language part of IPython to all the other functionalities needed to run the notebooks, such as the notebook format, the web framework, or the message protocols. IPython turns then into just another computation kernel for Jupyter, which nowadays supports more than 40 languages that can be used as kernels[1].

There are also other generic tools which can be seen as *dissemination* platforms since their main objective is to make source code and data widely available to the public. In this category we find for example Research Compendia[2] focused on reaching reproducible research by storing data, code, in a form that is *accessible, traceable, and persistent*, MLOSS[3] for machine learning, datahub[4] to create, register, and share generic datasets, and RunMyCode[5] to associate code and data to scientific publications. Compared to these platforms, IPOL differs from them in the sense that it is a peer reviewed journal, and not only a dissemination platform.

Regarding the system architecture of IPOL, it is built as a Service-Oriented Architecture (SOA) made of microservices. This type of architecture allows IPOL to have simple units (called *modules* in its own terminology) which encapsulate isolated high-level functions (see in Sect. 4.1). Specifically, we use the CherryPy framework to provide the REST HTTP [8] services. Microservices in distributed system are specially useful for those system which need to serve millions of simultaneous requests. A good example of SOAs made of microservices is the Amazon AWS API Gateway[6] used by millions of users. Also, multimedia streaming services such as Netflix[7] which receives about two-billion daily requests or Spotify[8] are usually based on SOAs of microservices.

3 IPOL as a Peer-Reviewed Scientific Journal

IPOL is a scientific journal on mathematical signal processing algorithms (image, video, audio, 3D) which focuses on the importance of reproducibility. It differs from other classic journals in its editorial policy: each IPOL article must present a complete description of its mathematical details together with a precise explanation of its methods with pseudo-codes. These ones must describe exactly the

[1] https://github.com/ipython/ipython/wiki/IPython-kernels-for-other-languages.
[2] http://104.130.4.253/.
[3] http://mloss.org.
[4] https://datahub.io/.
[5] http://www.runmycode.org/.
[6] https://aws.amazon.com/api-gateway.
[7] https://media.netflix.com/en/company-blog/completing-the-netflix-cloud-migration.
[8] http://es.slideshare.net/kevingoldsmith/microservices-at-spotify.

implementation that achieves the results depicted in the paper. The idea is that readers with sufficient skills could implement their own version (in any programming language or environment) from the IPOL article. Furthermore, submitting an IPOL paper means to upload the manuscript coupled with the original source codes. Both are reviewed in depth by the referees to ensure the quality of the publication and that the pseudo-codes match exactly with the attached program, before the editor's decision. The publication process is divided in two stages: first, the reviewers evaluate the scientific interest, the experiments and the reproducibility of the work; secondly, if this evaluation is positive, the authors submit the original code and the online demo is published.

Each IPOL article contains [1]:

1. A description of *one algorithm* and its *source code*;
2. a PDF article associated with an *online demonstration*;
3. *archived experiments* run by users.

All these data, accessible through different tabs of the article webpage, make IPOL an open science journal in favor of reproducible research. The philosophy of the journal follows the guidelines on reproducible research topics, by obeying the standards of reproducible research [9,10]. This is meant as an answer to the credibility crisis in scientific computation pointed out by Donoho et al. [3].

IPOL publishes algorithms along with their implementation, but not compiled/binary software. Neither is it a software library, since each code must have minimal dependencies. The objective of IPOL is not simply to be a software or code diffusion platform. In this sense, the code must be as transparent to the reader as possible, not using implementation tricks unless they are described in the article. It should be seen as a reference implementation, always preferring clarity over run time optimization.

The current form of an IPOL article is illustrated in Fig. 1. The first tab (a) presents the links to the low- and high-resolution PDF manuscripts, and also to the reference source code. An embedded PDF viewer presents a preview of the manuscript. The second tab (b) is the interface of the demonstration system, proposing some illustrative input data. The user can also upload its own data from this page. Clicking on one proposed input dataset or uploading a new one brings to a page presenting a list of adjustable parameters of the algorithm, and possibly an image selection tool, used for example for cropping an image too big for real-time processing by the system (almost all demonstrations are expected to achieve their processing in at most 30 s). A click on the "Run" button brings to a waiting page, while the server runs the author's code, which finally updates into a webpage showing the results. At this stage, the user has the option to re-run on the same input data but modifying the parameters, or to change the input data. Running the algorithm on newly uploaded input data proposes to archive them and their results. The archived data of tab (c) in Fig. 1 have permanent URL. This facilitates online communication between distant collaborators working on an algorithm. The archived data allow to understand what usages are aimed at by visitors, can reveal failure cases not anticipated by the authors, etc. The amount of archived data can also serve as a crude measure of the interest an algorithm raises in the community, as a kind of substitute or complement to the number of citations in a standard journal. The most cited IPOL articles have also tens of thousands of archived online experiments.

Each IPOL demo downloads and compiles by itself the source code. This ensures that the users can reproduce exactly the results claimed by the authors. However, the authors of the demo can additionally add scripts or data, which is not peer-reviewed, but is needed to show the results in the website. This allows to avoid mixing the peer-reviewed source code of the submitted method with support extra codes needed only by the demo. This approach differs from classic publishing, where the method and some details about the implementation are usually described but it is not possible to reproduce and thus confirm the published results.

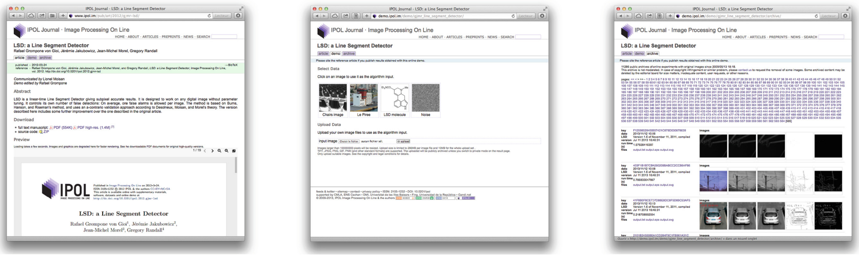


Fig. 1. The current form of an IPOL article with its three tabs: (a) article with manuscript, online viewer, and link to source code, (b) demonstration system, and (c) archived experiments.

Apart from this specific form of the articles, IPOL presents the same aspects as a classic scientific journal, with an editorial committee, contributors in the form of authors and editors, a reviewing process, an ISSN, a DOI, etc. Some special issues are proposed, for example some selected papers from the 16th IAPR International Conference on Discrete Geometry for Computer Imagery (DGCI) in 2011. There is also an agreement for publishing companion papers in SIAM Journal of Imaging Sciences (SIIMS) and IPOL, the first submission concentrating on the theory and general algorithm and the second one on practical implementation and algorithmic details. Note that originality of the algorithm is *not* a prerequisite for IPOL publication: the usefulness and efficiency of an algorithm are the decisive criteria. IPOL articles are indexed by all major indexers, such as Scirus, Google Scholar, DBLP, SHERPA/RoMEO, CVonline, etc.

The role of the reviewer is not restricted to the evaluation of the manuscript. The reviewers are also expected to test the online demonstration, check the algorithmic description in the article, and the source code. Most importantly, they must verify that the description of the algorithm and its implementation code match. An important requirement is that the code be readable and well documented.

4 The IPOL System Architecture

The architecture of the new IPOL demo system is an SOA based on microservices. This change was motivated by the problems found in the previous version of the demo system. First, it was designed as a monolithic program[9] which made it quite easy to deploy in the servers and to run it locally, but at the cost of many disadvantages. Given that it was a monolithic system, it was difficult to split it into different machines to share the computational load of the algorithms being executed. A simple solution would be to create specialized units to run the algorithms and to call them from the monolithic code, but this clearly evokes the first step to move to a microservices architecture. Indeed, this first step of *breaking the monolith* [4] can be iterated until all the functions of the system have been delegated in different modules. In the case of IPOL, we created specialized modules and removed the code from the monolith until the very monolith became a module itself: the Core. This Core module is in charge of all the system and delegates the operations to other modules. Figure 2 summarizes the IPOL modules and other components of the system.

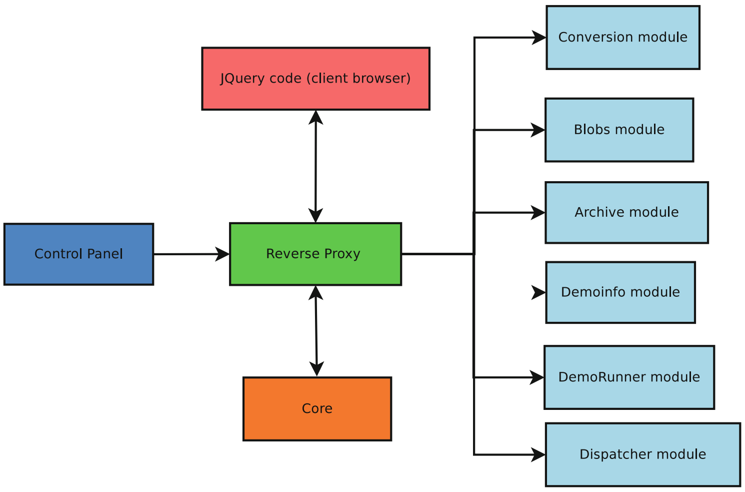


Fig. 2. IPOL as a modular system.

Other problems we had in the previous version of the demo system got solved when we moved to the microservices architecture. Since there is a loose coupling between the Core and the other modules, different members of the development team can work at the same time without worrying about the implementation details or data structures used in other parts of the system. Also, tracking down malfunctions is easier: since the Core centralizes all the operations, when a bug

[9] Of course, with a good separation of functionality among different classes.

shows it can only be generated either at the Core or at the involved module, but not at any other part of the system. In the old system a bug could be caused by complex conditions which depend on the global state of the program, making debugging a complex task. And as noted before, the fact that the architecture of the system is distributed and modular by design makes it very natural and simple to have mechanisms to share the computational load among several machines.

Hiding the internal implementation details behind the interfaces of the modules is an essential part of the system, and it is needed to provide loose coupling between its components. The internal architecture of the system is of course hidden from the users when they interact with the system, but it is also hidden *from the inside*. This means that any module (the Core included) does not need to know the location of the modules. Instead, all of them use a published API.

Once the API is defined, the routing to the modules is implemented by a reverse proxy[10]. It receives the requests from the clients according to this pattern: `/api/<module>/<service>` and redirects them to the corresponding module. Figure 3 shows how the API messages received by the proxy are routed to the corresponding modules, thus hiding the internal architecture of the system.

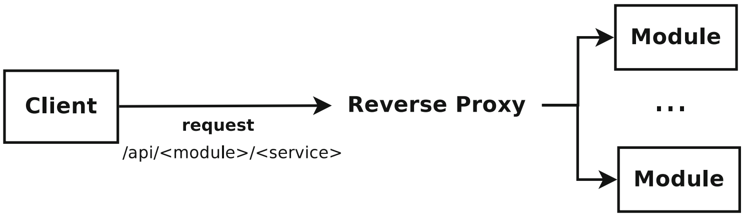


Fig. 3. The reverse proxy routes the API messages to the corresponding modules.

4.1 The IPOL Demo System Modules

The IPOL demo system is made of several standalone units used by the Core module to delegate specialized and well isolated functions. This section describes briefly these microservices modules.

Archive. The archive module stores all the experiments performed by the IPOL with their original data. The database stores the experiments and blobs, which are related with a junction table with a many-to-many relationship. It is worth noting that the system does not save file duplicates of the same blob, but detects them from their SHA1 hash.

This module offers several services, such as adding (or deleting) an experiment or deleting all the set of experiments related to a particular demo. The archive also has services to show particular experiments or several pages with all the experiments stored since the first use of the archive.

[10] We use Nginx as the reverse proxy.

Blobs. Each demo of IPOL offers the user a set of default blobs which can be tagged and linked to different demos. Thus, the users are not forced to supply their own files for the execution of the algorithms. This module introduces the concept of *templates*, which are sets of blobs which can be associated to a particular demo. For example, this allows all the demos of an specific type (e.g., denoising) to share the same images as default input data. Instead of editing each demo one by one, the editors can simply edit their template to make changes in all the demos, and then particular changes to each specific demo.

Core. This module is the centralized controller of the whole IPOL system. It delegates most of the tasks to the other modules, such as the execution of the demos, archiving experiments, or retrieving metadata, among others.

When an execution is requested, it obtains first the textual description of the corresponding demo by using the Demo Description Lines (DDL) from the DemoInfo module and it copies the blobs chosen by the users as the algorithm's input. Then, it asks for the workload of the different DemoRunners and gives this information to the Dispatcher module in order to pick the best DemoRunner according to the Dispatcher's selection policy. The Core asks the chosen DemoRunner to first ensure that the source codes are well compiled in the machine and then to run the algorithm with the parameters and inputs set by the user. The Core waits until the execution has finished or a timeout happens. Finally, it delegates into the Archive module to store the results of the experiment. In case of any failures, the Core terminates the execution and stores the errors in its log file. Eventually, it will send warning emails to the technical staff of IPOL (internal error) or to the IPOL editors of the article (compilation or execution failure).

Dispatcher. In order to distribute the computational load along different machines, this module is responsible of assigning a concrete DemoRunner according to a configurable policy. The policy takes into account the requirements of a demo and the workload of all the DemoRunners and returns the DemoRunner which best fits. The DemoRunners and their workloads are provided by the Core. Figure 4 shows the communication between the Core, Dispatcher, and the DemoRunner modules.

Currently the Dispatcher implements three policies:

- **random:** it assigns a random DemoRunner
- **sequential:** it iterates sequentially the list of DemoRunners;
- **lowest workload:** it chooses the DemoRunner with the lowest workload.

Any policy selects only the DemoRunners satisfying the requirements (for example, having MATLAB installed, or a particular version of openCV).

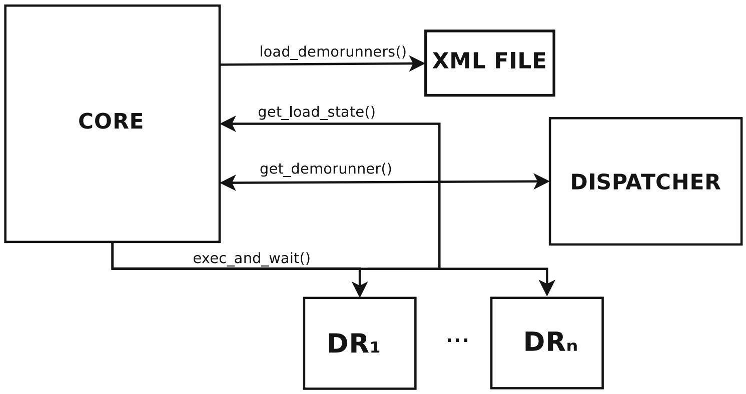


Fig. 4. Communication between the Core, Dispatcher, and the DemoRunner modules.

DemoInfo. The DemoInfo module stores the metadata of the demos. For example, the title, abstract, ID, or its authors, among others. It also stores the abstract textual description of the demo (DDL). All this information can be required by the Core when executing a demo or by the Control Panel when the demo is edited with its website interface.

It is possible that the demo requires non-reviewed support code to show results. In this case, the demo can use custom scripts to create result plots. Note that this only refers to scripts and data which is not peer-reviewed. In case they are important to reproduce the results or figures in the article, they need to be in the peer-reviewed source code package.

DemoRunner. This module controls the execution of the IPOL demos. The DemoRunner module is responsible of informing the Core about the load of the machine where it is running, of ensuring that the demo execution is done with the last source codes provided by the authors (it downloads and compiles these codes to maintain them updated), and of executing the algorithm with the parameters set by the users. It takes care of stopping the demo execution if a timeout is reached, and to inform the Core about the causes of a demo execution failure so the Core can take the best action in response.

5 Software Engineering in the IPOL Demo System

The current IPOL project tries to follow the best practices in software engineering. Specifically, for this kind of project we found that Continuous Integration was a good choice in order to achieve fast delivery of results and ensuring quality. Continuous Integration is a methodology for software development proposed by Martin Fowler, which consists of making automatic integrations of each increment achieved in a project as often as possible in order to detect failures as soon as possible. This integration includes the compilation and software testing of the entire project.

It is a set of policies that, together with continuous deployment, ensures that the code can be put to work quickly. It involves automatic testing in both integration and production environments. In this sense, each contribution in the IPOL system is quickly submitted and several automatic test are performed. If any of these tests fail the system sends an email indicating the causes. Another advice of Continuous Integration is minimal branching. We use two. On one hand, master is the default branch and where all the contributions are committed. It is used for the development, testing and this continuous integration; on the other hand, the prod branch is used only in the production servers. It is merged with master regularly. We use two different environments: integration and production. The integration server is where the master branch is pulled after each commit. The prod branch is used for the production servers and the code in this branch is assumed to be stable. However, the code in the integration server is also assumed to be stable and theoretically the code in the master branch could be promoted to production at any time once it has been deployed to the integration server and checked that is fully functional and without errors.

Quality is perhaps the most important requirement in the software guidelines of the IPOL development team. The code of the modules must be readable and the use of reusable solution is advised [11]. The modules must be simple, well tested and documented, with loose interface coupling, and with proper error logging. Note that it is not possible to ensure that any part of the IPOL will not fail, but in case of a failure we need to limit the propagation of the problem through the system and to end up with diagnostic information which allows to determine the causes afterwards. Refactoring [12] is performed regularly and documentation is as important as the source code. In fact, any discrepancy between the source code and the documentation is considered as a bug.

5.1 Tools

The IPOL development team has created so far three main tools for the system administrators, developers, and editors to interact with the IPOL system. Some of their capabilities might be duplicated or overlapping with the Control Panel (for example, reading and modifying the DDL of the demos is a function implemented in the DDL tool and in the Control Panel, but they are still useful to perform massive changes or to automatize tasks).

Terminal. The Terminal is a small Python standalone application intended for system administrators which allows to start, stop, and query the status of each module. The current list of commands is: **start:** launches the specified module; **ping:** checks if the module is up; **shutdown:** stops the specified module;

info: prints the list of available commands of the module; **modules:** displays the list of the modules of the system.

Control Panel. The Control Panel web application offers a unified interface to configure and manage the platform. It provides a navigation menu option for each module, which allows the editors to edit the demos or the modules directly (say, the add or remove images of a demo, or to delete experiments from the Archive upon request). Look at Sect. 6 for more information on the Control Panel.

DDL Tool. This tool is a standalone Python script which allows to read and write the DDLs of each demo. The main justification for this tool is to perform massive changes in the DDLs and automatize some needed tasks. It admits the following list of commands. **Read:** downloads the DDLs of the specified demos; **Read all:** downloads the DDLs of all the demos known by the system; **Write:** uploads the content of the file with the DDLs of the specified demos.

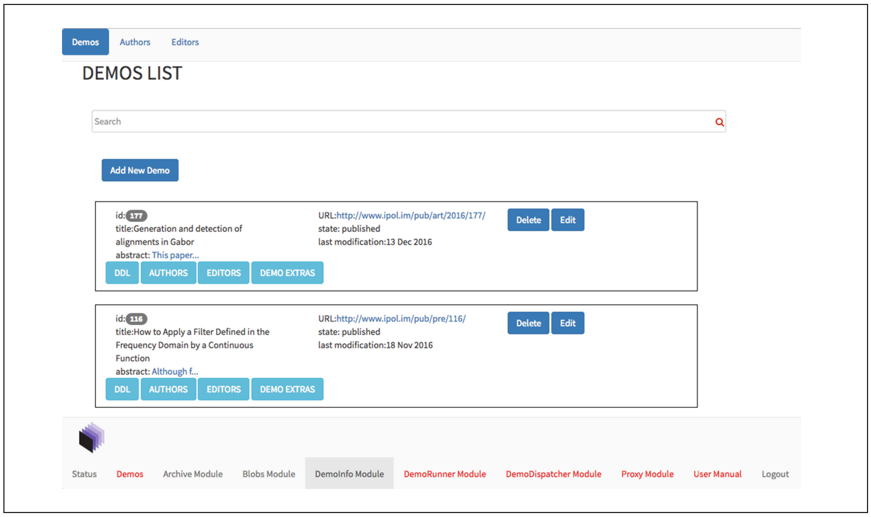
6 Editorial Management: The Control Panel

The Control Panel is a Django web application which offers a unified interface to configure and manage the platform. Its graphical interface gives to the editors a menu with options to access the different modules available on the system. It provides many editing options to the users. The first option is the Status, that shows a list of the modules with summarized information about them, allowing the user to monitor if they are currently running. In second place there is an Archive Module option to provide a list of the demos with stored experiments, as a result of an execution with original data. It allows the editor to remove specific experiments upon request (e.g., inappropriate images). There is also a Blobs Module option, which allows to add and remove blobs for a particular demo.

Additionally, the DemoInfo Module option permits the user to access information about the demos, authors and editors stored on the IPOL demo system, organized in three sections. The Demos section is the option selected by default, and makes it possible to edit the demo metadata, such as its ID, title, or the source code URL, the assigned editors, or its support scripts, among others. Figure 5 shows a screen capture of the Control Panel application as shown in the browser.

7 Automatic Demo Generation

In the previous version of the IPOL demo system the demo editors had to write Python code to create a new demo. Specifically, to override some methods in a base demo class in order to configure its input parameters, to call the program



id: 177
title:Generation and detection of alignments in Gabor
abstract: This paper...
URL:http://www.ipol.im/pub/art/2016/177/
state: published
last modification:13 Dec 2016

id: 116
title:How to Apply a Filter Defined in the Frequency Domain by a Continuous Function
abstract: Although f...
URL:http://www.ipol.im/pub/pre/116/
state: published
last modification:18 Nov 2016

Fig. 5. List of demos in DemoInfo Module option of the Control Panel.

implementing the algorithm, and also to design Mako HTML templates for the results page.

This approach does not really solve anything, since it simply moves the inability to generate demos from a simple description from the system to the demo editors. Since the Python code is written by the demo editors, it is prone to bugs which are specific to each demo. Moreover, fixing a bug in a demo does not prevent the others the have similar problems, with different implementations.

In fact, it is evident that this is a bad design, since to completely define a demo all that is needed is: (1) The title of the demo; (2) the URL where the system should download the source code of the demo; (3) the compilation instructions; (4) a description of the input parameters; (5) a description of what needs to be shown as results.

This information is not tied to any particular language or visualization technique, but it can be a simple abstract textual description. In IPOL we called this abstract textual description the Demo Description Lines (DDL). The IPOL editors only need to write such a description and the system takes care of making available the demo according to it. This not only avoids any coding problems (since there is nothing to code, but writing the short DDL), but also allows IPOL to have non-expert demo editors, and makes it possible to edit and publish demos quickly.

As an example, the following DDL listing is from a published IPOL demo:

```
{ "archive": {
        "files": {
            "derivative1.png":" Derivative 1",
            "sinc3.png":"Sinc 3"
        },
        "params": [
            "a1","a2","variable","orientation","sigma"
        ]
    },
    "build": {
        "build1": {
             "url":"http://www.ipol.im/pub/art/2016/116/filtering_1.00.zip",
            "construct":"cmake filtering_1.00 && make -C filtering_1.00",
            "move":"main_comparison"
         }
    },
    "general": {
        "demo_title":"How to Apply a Filter Defined in the Frequency Domain by a Continuous Function",
        "xlink_article":"http://www.ipol.im/pub/art/2016/116/"
    },
    "inputs": [
        {
            "description":"input",
            "dtype":"3x8i","ext":".png",
            "max_pixels":"700*700","max_weight":"10*1024*1024",
            "type":"image"
        }
    ],
    "params": [
        {
            "id":"a1",
            "label":"x-component shifting",
            "type":"range",
            "values": {
                "default": 0.25,"max": 1,"min": 0,"step": 0.05
             }
        },
     (...)
    ],
    "results": [
        {
            "contents": {
                "Derivative 1":"derivative1.png",
                "Sinc 3":"sinc3.png"
             },
            "label":"Filtered and difference images",
            "type":"gallery"
        },
    ],
    "run":"main_comparison input_0.png -a $a1 -b $a2 -V $variable -g $sigma -Q $orientation -e png"
    }
```

8 Conclusions

The first version of the IPOL demo system has been working since the first article was published in 2010, with a total of 1434 citations and h- and i10-indexes of 20 and 36 respectively; its demo system is receiving about 250 unique visitors per day. While it is clear that the system is functional, some problems were detected: the system was difficult to debug to track down malfunctions, it suffered from tightly coupled interfaces, it was complicated to distribute the computational load among different machines, and the editors needed to write Python code to create and edit demos. These problems compromised the durability of the system at the same time they started to create a bottleneck that prevented to create and edit demos quickly.

The new system moved to a distributed architecture of microservices which solved many of these problems. It introduced however the typical problems of moving the complexity from the monolithic code to the boundaries of the microservices, but in general the balance has been quite positive. The system is made now of simple parts and the development team has gained flexibility due to the isolation of the microservices. Also, the editors are able now to quickly create and edit demos thanks to the abstract syntax of the DDL.

The next challenges for the very short term are to integrate new data types such as video, audio, and 3D, and the development team is quite optimistic about that, since the system is able to manage generic types (even if we refer to *images*, for the system they are simply *blobs*) and it comes down to a visualization problem in the website interface.

In conclusion, we managed to fix many of the problems found in the previous system by redesign and refactoring and now the system is ready to be expanded again, with a solid architecture and codebase.

Acknowledgments. This work was partly funded by the European Research Council (advanced grant Twelve Labours), the Office of Naval Research (ONR grand N00014-14-1-0023), and the ANR-DGA project ANR-12-ASTR-0035.

References

1. Limare, N.: Reproducible research, software quality, online interfaces and publishing for image processing. Ph.D. thesis, École normale supérieure de Cachan-ENS Cachan (2012)
2. Colom, M., Kerautret, B., Limare, N., Monasse, P., Morel, J.-M.: IPOL: a new journal for fully reproducible research; analysis of four years development. In: 2015 7th International Conference on New Technologies, Mobility and Security (NTMS), pp. 1–5. IEEE (2015)
3. Donoho, D.L., Maleki, A., Rahman, I.U., Shahram, M., Stodden, V.: Reproducible research in computational harmonic analysis. Comput. Sci. Eng. **11**(1), 8–18 (2009)
4. Neuman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly Media (2015)
5. Goecks, J., Nekrutenko, A., Taylor, J.: Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biol. **11**(8), 1 (2010)
6. Lamiroy, B., Lopresti, D.: The DAE platform: a framework for reproducible research in document image analysis. In: Kerautret, B., Colom, M., Monasse, P. (eds.) RRPR 2016. LNCS, vol. 10214, pp. 17–29. Springer, Cham (2017)
7. Buckheit, J.B., Donoho, D.L.: Wavelab and reproducible research. In: Antoniadis, A., Oppenheim, G. (eds.) Wavelets and Statistics. Springer, New York (1995)
8. Berners-Lee, T., Fielding, R., Frystyk, H.: Hypertext transfer protocol-HTTP/1.0, RFC 1945, RFC Editor (1996)
9. Stodden, V.: The legal framework for reproducible scientific research: licensing and copyright. Comput. Sci. Eng. **11**(1), 35–40 (2009)
10. Stodden, V.: Enabling reproducible research: open licensing for scientific innovation. Int. J. Commun. Law Policy **13**, 1–25 (2009)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (2008)
12. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Programs. Addison-Wesley, Reading (1999)