D7.2– First co-design repository
Version [1.0]

Document Information

Contract Number	824080
Project Website	www.pop-coe.eu
Contractual Deadline	M18
Dissemination Level	PU - Public
Nature	R
Authors	Xavier Teruel (BSC).
Contributors	Jesús Labarta (BSC), Christoph Niethammer (HLRS), Ramil Nabiev (HLRS), Martin Rose (HLRS), Lukas Maly (IT4I), Radim Vavrik (IT4I), Michael Knobloch (JSC), Jon Gibson (NAG), Jonathan Boyle (NAG), Wadud Miah (NAG), Federico Panichi (NAG), Fabian Orland (RWTH).
Reviewers	William Jalby (UVSQ), Julianna Anguelova (BSC).
Keywords	Co-design, kernels, patterns, best-practices.

Acknowledgements: The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement n°824080.

Change Log

Version	Author	Description of Change
v0.1	Xavier Teruel	Initial version of the document.
v0.7	Xavier Teruel	Populating sections in the document.
v0.9	Xavier Teruel	Including rest of the content. Also including Fabian Orland (RWTH) and Radim Vavřík (IT4I) feedback. Ready for internal review (requiring update conclusions with the final state).
v1.0	Xavier Teruel	Minor changes in the document before applying internal review. Including Willian Jalby, and Julianna Anguelova review comments. Updating conclusions, and including Executive summary.

Contents

Executive Summary

The deliverable consist of the *POP Co-design repository* filled with global data gathered from POP1 and POP2 reports until month 18 including detailed statistics on the relevance of the different efficiency factors in the POP methodology. It will also include a first set of micro-benchmarks characterizing behaviors found in real applications.

The repository has been finally published in the shape of a public web site, containing a mix of downloadable codes, pattern/best-practice descriptions, experimental results, and other classification criteria. All these items are linked to each other in order to facilitate the navigability among all of them. Reviewers may check the current status on-line in the following URL:

http://co-design.pop-coe.eu

In order to coordinate the work of the different partners, we decided to use an *issue tracking tool* (i.e., Gitlab issue module). This module allows to guarantee topic discussions on the regular work package technical teleconferences, arranged bi-weekly in the current period, as well as to gather feedback and discussion into a single place. The Gitlab issues follow a sequence of states, they must to go through, until completion: *(1) Open*, *(2) Triage or available*, *(3) Pending or Blocked*, *(4) Work in progress*, *(5) Review and discussion*, *(6) Execute or publish*, and *(7) Closed*.

Although the initial specification of the repository was made in a previous deliverable, in this document we also collect a series of changes made to the initial design. These changes are motivated by the needs that we have been encountering while working with the contents that were populating the co-design site. All these changes are described in the body of this document. The document also describes content grouped by kind: 1) *kernel related information*, 2) *patterns and best-practices*, and 3) *other classification criteria*. Reporting content do not imply it has been finally published in the site, but it contains a firm proposal by the corresponding author(s).

In the first 18 months of the project, we have published 10 kernels in the co-design site, as well as 6 patterns and 6 best-practices gathered from previous POP reports. In the same period, we have also included the descriptions of: 4 programming languages, 8 programming models, 10 disciplines, and 5 algorithms (as other classification criteria). The number of published kernels is above the 8 kernels required by *MS3 - First Methodology Milestone*.

1 Introduction

In this deliverable we will present the interim version of the POP co-design repository. The fundamentals of the repository were previously discussed in *D7.1 Co-design repository structure* [4], where we define the structure of the collections, and the relationships among them. A second goal of the deliverable is to report a progress in reaching the targeted metrics of POP2 Milestone M3: *First methodology milestone* [3], also related with this deliverable.

The repository has been finally published in the shape of a public web site, containing a mix of downloadable codes, pattern/best-practice descriptions, experimental results, and other classification criteria. All these items are linked to each other in order to facilitate the navigability among all of them. Although in this deliverable we are reporting a certain amount of items (i.e., kernels, patterns, best-practices, etc.), the site is in continuos development and reviewers may check the current status on-line in the following URL:

http://co-design.pop-coe.eu

The site is structured in different sections, each one listing the different items within the corresponding category. Figure 1 shows the co-design site structure: below the title, a navigable menu showing the different sections; below the menu, the page body that will include the actual contents of the selected item. The list of sections of the site is:

- *home*, the home page contains a brief introduction to the co-design site. It also links to the main components of the site.

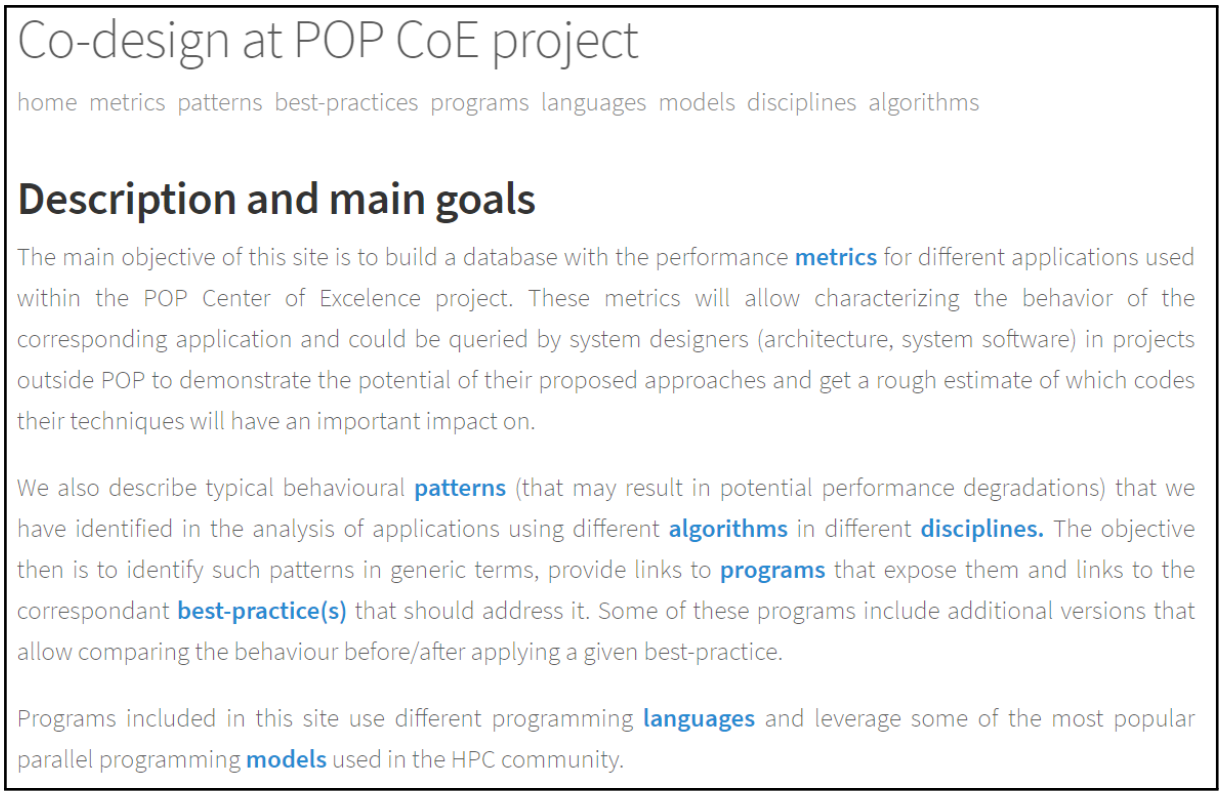

Figure 1: Co-design site structure: the home page.

- *metrics*, the metrics page contains a listing of the POP metrics. Each item in the list will define the metric entry and will link to kernel versions that may have issues with the specific metric. As the threshold, we have used the value 800‰, as defined in POP CoE main site.

- *patterns*, the patterns page contains a listing of the reported patterns. Each item in the list will define the pattern and it will link with programs reporting this behaviour. In addition it will include a list of best-practices recommended as the solution of the problem.

- *best-practices*, the best-practices page contains a listing of the recommended best-practices. Each item in the list will describe the best-practice and, in addition, a list of program versions which: 1) recommend the use of the best-practice; or 2) have already implemented this best-practice. A best-practice entry also links with its related pattern, in fact the pattern is summarized in the first paragraph introducing the best-practice (see Section 3.2).

- *programs*, the programs page contains a listing of currently included kernels. Each program entry will contain links to related versions, languages, models, disciplines, and algorithms. The program item is the entry link to the kernel description that will allow to navigate through all its elements. For instance, versions will contain links to download the specific source code, and a list of experiments (i.e., results), performed with the code.

- *languages*, the languages page contains a listing of registered programming languages. Each one of the list items links to programs using the corresponding programming language.

- *models*, the models page contains a listing of registered programming models. Each one of the list items links to programs using the corresponding programming model.

- *disciplines*, the disciplines page contains a listing of registered disciplines. Each one of the list items contains a description of the discipline and links to programs belonging to the corresponding category.

- *algorithms*, the algorithms page contains a listing of registered algorithms. Each one of the list items contains a description of the algorithm and links to programs containing the corresponding algorithm.

The main objective of the site is to offer easy navigability among its different components, being able to access the description of an element (e.g., the Fortran language in the programming language collection), and obtain a list of links to its related items (e.g., kernels programmed in Fortran). Therefore, all the elements that are part of the co-design site will be linked following the scheme proposed in *D7.1 Co-design repository structure* [4]: patterns with their corresponding best-practices, kernels belonging to a certain discipline, POP metrics and the corresponding kernels reporting a problem with it, etc. Usually these types of relationships are bidirectional (i.e., if an element contains a link to another element, the latter will also contain a link to the first element). In the future we also consider to allow certain *searches* within the site contents to ease the access to all these items.

1.1 Effort distribution

WP7 effort consists on: 1) contribution to the co-design site contents; 2) contribution to the kernel development; 3) contribution to the site extensions; and 4) contribution to the work package coordination. In addition, POP partners participating in this work package contribute at different levels of commitment:

- BSC (27 PM), 3 PM devoted to work package coordination, 6 PM devoted to site extensions (and layout content publication), and 18 PM devoted to kernels and contents;
- HLRS (18 PM), all the effort devoted to kernels and contents;
- NAG (18 PM) all the effort devoted to kernels and contents;
- RWTH (18 PM) all the effort devoted to kernels and contents;
- JSC (9 PM) all the effort devoted to kernels and contents; and
- IT4I (9 PM) all the effort devoted to kernels and contents.

This deliverable is directly related to *First Methodology Milestone (MS3)*, which commits to deliver 8 kernels with its corresponding results. We focus our effort on reaching this amount of kernels, but including a safety-margin to guarantee the milestone fulfillnes. We expect that each kernel will imply to report one associated pattern and its corresponding best-practice. We also expect to contribute with the description of other classification criteria (complementary items: languages, programming models, disciplines, and algorithms) proportional with the expected partner effort. Kernel repositories must include the pop-db branch (i.e., program and version descriptions, plus a preliminary experimental results directory). And finally, partners will also review other kernel repositories, in a peer review format, to guarantee the quality-check phase.

Component	BSC	HLRS	IT4I	JSC	NAG	RWTH	Total
Kernels (repositories)	2	2	1	1	2	2	10
Patterns & Best-Practices	7	4	2	2	4	4	23
Complementary items	7	4	2	2	4	4	23
Total	14	8	4	4	8	8	46

Table 1: Preliminary effort distribution per partner in the period M7–M18.

Table 1 shows a preliminary effort distribution per partner[1]. Taking this table as the baseline, we can still play with some minor adjustments considering the following criteria: 1) there will be patterns and best-practices that could apply to more than one kernel so, as far as all the kernels have their associated patterns and best-practices, we can migrate some effort from these two components to complementary items; 2) there will be patterns with more than one associated best-practice so, we can also migrate complementary items effort to patterns and best-practices components if required; and 3) patterns and best-practices may exist without the need of a kernel illustrating them so, partners are welcome to work on additional patterns and best-practices (reducing accordingly its complementary items effort).

For the sake of simplicity, we will consider the report of patterns, best-practices, and complementary items as *pages* (as far as they fulfill all the previously commented constraints). The aggregated target values on the current deliverable will consist on **10 kernels** and **46 pages**.

1.2 Work package methodology

In order to coordinate the work of the different partners, we decided to use an issue tracking tool (i.e., Gitlab [1] issue module). This tool will allow to guarantee topic discussions on regular WP technical teleconferences, arranged bi-weekly in the current period. The main idea consist of

[1] BSC effort includes an additional 75% of *patterns and best-practices*, and an additonal 75% of *complementary issues* with respect to an 18 PM partner contribution assignment.

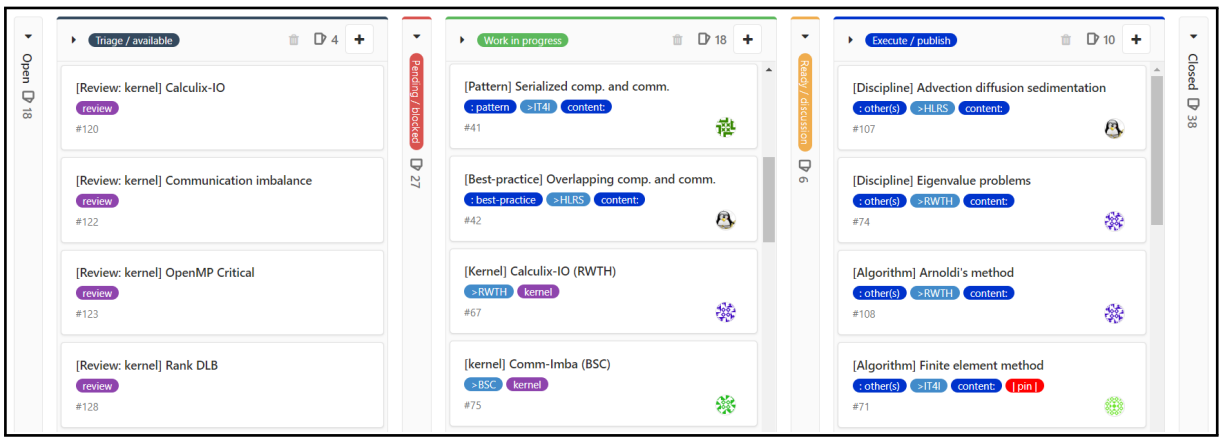

Figure 2: Layout repository issues: development board.

each partner describing the proposed text in the corresponding Gitlab issue, making it available to other partners to provide feedback, presenting the final proposal in one of the above mentioned teleconferences, and reaching consensus to validate the text.

At the initial stages, each partner proposed a set of kernels they wanted to work with during the current period. The number of kernels was determined by the effort distribution described in the previous section. Each kernel has an associated pattern and best-practice. All these generated issues are initially owned by the partner that has proposed them[2], and they become the initial set of issues assigned this partner.

During the period, more issues were created as the result of: 1) being discussed in a technical teleconference; 2) being proposed by mail by any of the partner; or 3) being required by any other issue (e.g., when a given kernel uses a well-known algorithm not yet described in the system). The progression of these issues is described by means of *labels*, which determine its current state. Figure 2 illustrates the classification of some of these issues. In the figure we can see the different states in which an issue should move on according with its current state (some of these states have been collapsed for the sake of clarity, but they are still present as a vertical *empty* column with the aggregated number of issues in such state, e.g., *Ready / discussion* with 6 issues).

The list of issue states is described as follows:

- Open, is the *wish-list* of topics we still have pending to start. They are not yet in our critical path (neither required by any other issue, nor prioritized by the project interests).
- Triage / available, the issue has been pre-selected. There is a requirement to work on this topic, but the issue has not been assigned to any partner. Once the issue is assigned to a partner it will become *Pending / blocked* if there are no plans to start working on it in the short-term, or *Work in progress* otherwise.
- Pending / blocked, issues has been already assigned, but no contents have been added yet.
- Work in progress, the issue owner has started to develop its contents, but is not ready to receive feedback yet.
- Review / discussion, there is a first version of the issue contents. Other partners may comment on the issue to provide feedback or to ask any question about its current contents.

[2]In some cases, two different kernels may have the same set of patterns and best-practices associated. In such situation, we decided to split the proposal development in between the involved partners

As the proposal evolves, we can have several versions of the corresponding item in its timeline (the last one being the current one).

- Execute / publish, there was an agreement in a previous technical teleconference to enact the contents of the corresponding item. The text is ready to become public in the co-design site, but it has not been published yet.
- Closed, the issue has been already published.

Other used labels in the Gitlab issue: owner institution (for accounting purposes), type of the issue (i.e., content, change[3], bug, format, kernel or review), sub-type of the issue (only for content or change types: pattern, best-practice, or others), and additional information (for management purposes: pin, multi, dismiss, or re-open).

Although the kernel issues do not imply any modification of the layout repository (kernel information is included directly in the corresponding source code repository), they have been included as a layout issue for tracking purposes. When a kernel issue in the layout repository becomes ready, it means the kernel repository contains a proper pop-db branch (see Section 2.1) and it has been made public on the co-design site.

1.3 Work package cross-references

Within the context of POP2 CoE, there are other WPs that are directly related to the co-design activities. Among these WPs we find the WP6 Proof-of-concept, and the WP8 Tools and Methodology.

In relation to **WP6 Proof-of-concepts**, we know some of its internal activities consist of proposing changes in the customer code improving a certain aspect of its performance behaviour. These changes may potentially represent a proposal for a best-practice. For this, one of the current activities proposed within the co-design context will consist of reviewing the conclusions of the deliverable *D6.1 First Report on Proof-of-Concept* [5]. In that deliverable they propose a set of lessons learned in relation to the reports they have written during this period. This activity will be complemented by reviewing the source/original reports (i.e., the Proof-of-concept report). From this activity we can derive: new relationships on the content already uploaded to the co-design site, new entries in the pattern and best-practice sections, and looking for new candidate codes that could potentialy synthesise in the shape of a new kernel.

With respect to the **WP8 Tools and methodology**, in this working group they are defining new metrics that could be also candidates to enrich the POP methodology. Among the proposals that this group is working on right now are: methodology for hybrid codes, vectorization analysis, and I/O activity analysis. So far all these proposals are still in process. Although some of them seem to have reached a very high degree of maturity (e.g., methodology for hybrid codes), there are also divergences in terms of the used metrics (i.e., additive model vs. multiplicative model, at *D8.1 First report on methodology development and tool improvement* [6]). From the co-design activities we will closely follow the evolution of these proposals. So far, it seems that the definition of these new methodologies will only impact the co-design activities in the inclusion of new set of metrics without changing any of the existing ones (besides formulation or having a slightly different semantics).

1.4 Glossary of terms

For the rest of the document we will make frequent use of the following terms:

[3]The main difference in betweeen content and change types is that content refers to a new entry in the co-design site, while change refers to the modification of an existent content.

- BSC: Barcelona Supercomputing Center
- CA: Consortium Agreement
- CAdv: Customer Advocate
- DoA: Description of Action (Annex 1 of the Grant Agreement)
- D: deliverable
- EC: European Commission
- GA: General Assembly / Grant Agreement
- HLRS: High Performance Computing Centre (University of Stuttgart)
- HPC: High Performance Computing
- HTML: HyperText Markup Language
- HTTP: HyperText Transfer Protocol
- IPR: Intellectual Property Right
- Juelich: Forschungszentrum Juelich GmbH
- JSC: Juelich Supercomputing Center
- KPI: Key Performance Indicator
- M: Month
- MS: Milestones
- PEB: Project Executive Board
- PM: Person month / Project manager
- POP: Performance Optimization and Productivity
- R: Risk
- RV: Review
- RWTH Aachen: Rheinisch-Westfaelische Technische Hochschule Aachen
- USTUTT (HLRS): University of Stuttgart
- WP: Work Package
- WPL: Work Package Leader

2 Extensions, suppressions and changes in the repository structure

Although the initial specification of the repository was made in a previous deliverable, in this section we will collect a series of changes made to the initial design. These changes are motivated by the needs that we have been encountering while working with the contents that were populating the co-design site.

2.1 Changing kernel documentation methodology

One of the first proposed changes is related to the way in which kernels interact with the centralized repository. Initially, we established a directory structure in which the information related to POP was located in the "doc/pop" directory of the master branch. However, when implementing the first use cases we realized the inconsistency of reporting information related to any of the branches of the repository in the main repository branch (e.g., reporting the description of an already optimized version of the code in the original version branch).

In our new approach, all kernel repositories will have an additional branch to report all the information related to the rest of the source code branches. Following our definition of namespaces, the branch in question will have the name of pop-db and will only contain the information related to the co-design repository. Thus, the automated process of collecting information related to kernels will be in charge of cloning all the public repositories of the Gitlab Kernels group, do a checkout of the pop-db branch, and copying all existing files of the corresponding Jekyll [7] collections in the build directory.

To create this new branch, a guideline document with the necessary Git commands has been made available to all partners, as well as the naming conventions required for the correct execution of the system: branch name, file names, fields identifiers in the Jekyll header, etc.

2.2 Removing the boundary collection

Another major change from the initial repository specification in *D7.1: Co-design repository structure* is related with the *Boundaries* collection. Initially, this Jekyll collection allowed to classify kernel versions according to boundary criteria (e.g., memory bound applications). As these metrics are completely orthogonal with POP methodology metrics, we decided to move them away from the co-design repository and focus exclusively on the POP metrics. This also makes it possible to simplify the structure of the site and to ease navigability.

This elimination is carried out without significant loss of information, given that the limiting factors do not provide any additional value with respect to the POP metrics. Thus, for instance, the performance bottleneck due to the interconnection network (i.e., network bound) will have a direct effect on the POP *Transfer Efficiency* metric. The memory bound behaviour will impact the POP *Computational Efficiency* metric, etc.

2.3 Adding the pattern excerpt in the best-practice entry

This change is more related to the co-design formatting than to the real content of the repository. The main idea behind this change is to give best-practice a self-contained form. A pattern specifies a behavior or code structure that presents some performance problem, so the patterns within the co-design repository are content independent. Instead, a best-practice represents a possible solution to a problem, so it is incomplete to present a solution without first mentioning the problem. Despite the fact that the elements are related to each other by means of HTTP links, it is more understandable to start the presentation of the best-practice by first including a brief introduction to the problem.

When generating the page resulting from a best-practice, the Jekyll build system will include the *excerpt* component associated with the related pattern. The code used to generate the best-practice introduction is the following (included in the best-practice template):

```
{% for pattern in site.patterns %}
   {% if page.patterns contains pattern.keyword %}
   <span class="solid␣paragraph">
      <span class="title">{{ pattern.title}}:</span>
      {{pattern.excerpt | remove: '<p>' | remove: '</p>'}}
      <a href="{{␣pattern.url␣|␣prepend:␣site.baseurl␣}}">(more...)</a>
   </span>
   {% endif %}
{% endfor %}
```

As the result, the page will contain a header where the main idea of the related pattern will be introduced before the best-practice description (see Figure 3, introductory shadowed text).

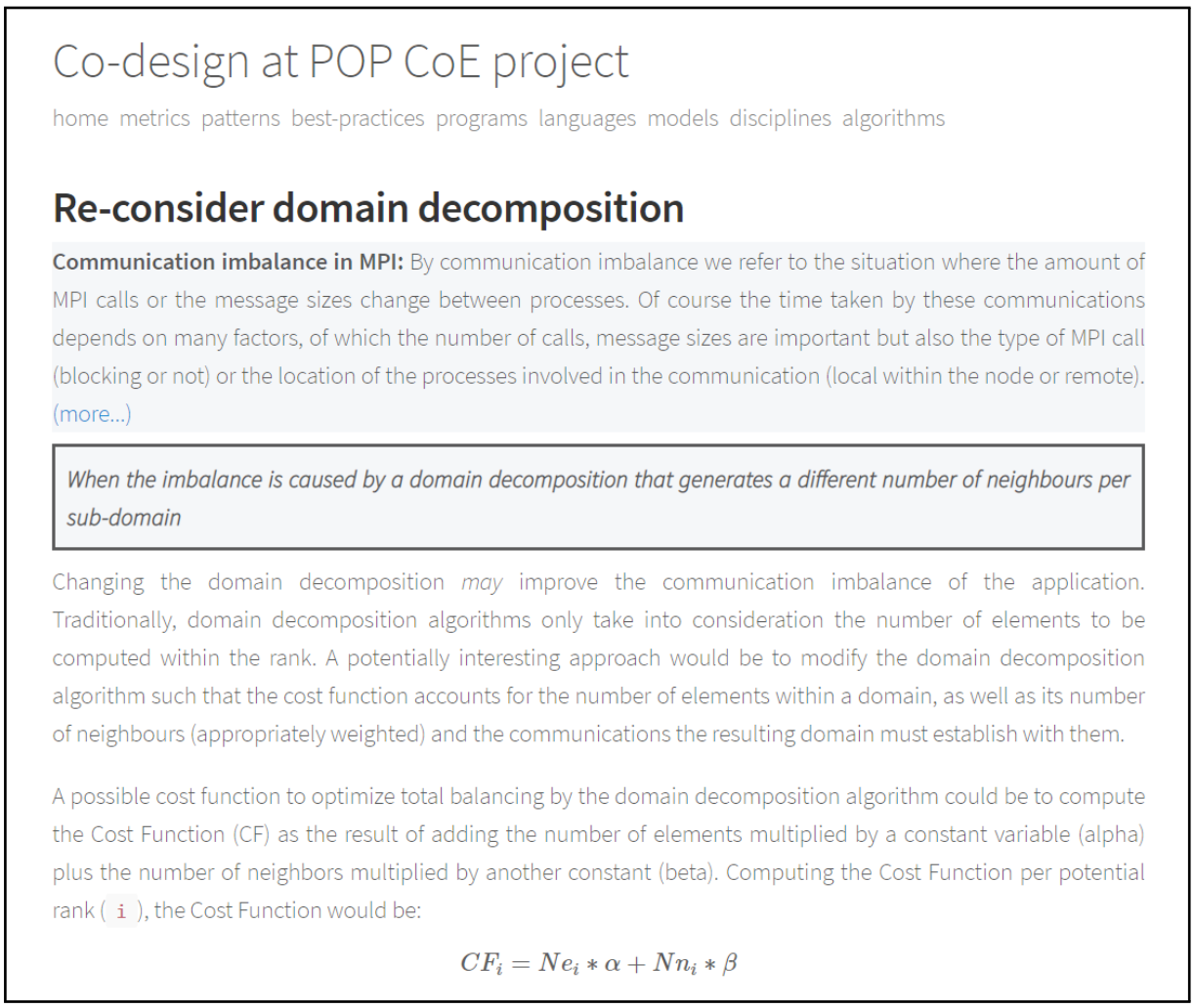

Figure 3: Best-practice example at the co-design site: *Re-consider domain decomposition*.

2.4 Adding condition to best-practices

This modification is also related to best-practices formatting in the co-design site. When showing a best-practice, before we introduce its related pattern by means of an embedded *excerpt* (see previous Section). Right below this pattern introduction, the system will also add any condition required by the best-practice in order to make possible its applicability. The *condition* then, could be considered as a best-practice enabler, and we can read it in the following manner: *"If a given **pattern** is present, and the **condition** is true, then we could apply the following **best-practice**".*

The code used to show the best-practice condition is the following (included in the best-practice template):

```
{% if page.condition and page.condition != "" %}
<span class="solid␣boxed␣paragraph">
<span class="head">{{ page.condition }}</span>
</span>
{% endif %}
```

As the result, the system will introduce the required condition (which needs to be true) in between the pattern excerpt and the best-practice description (see Figure 3, boxed and shadowed

text).

2.5 Allowing mathematical formulation

Another important aspect for the proper description of some elements in the co-design site is related to the mathematical notation. This feature mostly affects the algorithm and discipline items (where a certain method can be exemplified algorithmically), but it can also used in pattern and best-practice descriptions.

The use of mathematical notation is left open in the Markdown [2] syntax, delegating its correct rendering to external plugins (along with the Markdown render in itself, in our case Kramdown [9]). First, we are going to exploit the feature included in the Kramdown render for mathematical notation (i.e., Math Blocks), which allows the mathematical notation to be included in the scope delimited by double dollar (i.e., '$$'). Kramdown will offload the content of this block to the mathematical rendering engine that we configure on our site.

For mathematical rendering, we have chosen the Javascript library MathJax [8] (as the Kramdown documentation recommends [10]). The use of this Javascript library is enabled by including the script file in all of the web pages in the web site. Script is imported in the first three lines of the following code, the rest of the snipped just configure the MathJax extensions enabled in the site. This code is included in the `base.html` file, from which all the generated pages in the site inherit the structure:

```
<script type="text/javascript" async
  src="https://cdnjs.cloudflare.com/ajax/libs/mathjax/2.7.5/MathJax.js">
</script>
<script type="text/x-mathjax-config">
  MathJax.Hub.Config({
    extensions: [
      "MathMenu.js",
      "MathZoom.js",
      "AssistiveMML.js",
      "a11y/accessibility-menu.js"
    ],
    jax: ["input/TeX", "output/CommonHTML"],
    TeX: {
      extensions: [
        "AMSmath.js",
        "AMSsymbols.js",
        "noErrors.js",
        "noUndefined.js",
      ]
    }
  });
</script>
```

Once our site is configured, any page that uses the Kramdown Math Blocks syntax will render the content using the Javascript library MathJax, and generating an entry in HTML code for each element of the formula.

2.6 Creating internal/management pages

Although not directly related to the structure of the co-design site, we have also included internal/management pages (they are not directly linked on the site) to simplify the development of new content. A page that can be useful for writing new entries to the site is the *keywords* page. This page lists all the keys used by the elements already registered. Thus, if we need to find which key corresponds to a certain best-practice, which we are linking to a version of our

kernel, we should only search on this page to check the related identifier. This page would be accessible at:

https://co-design.pop-coe.eu/keywords

Another page that is considered of interest for the internal discussions in the co-design group is the list of published patterns and best-practices, so that we can generate a complete report (*c-report*) in a simple way. This page would be accessible at:

https://co-design.pop-coe.eu/c-report

All these pages help in the development of co-design activities, and are built on purpose when a certain need for management arises.

3 Repository contents

In this section, we briefly describe the contents, grouped by kind, that we have included within the current milestone period. All the contents described in this section do not imply they have been finally published in the site, but they contain a firm proposal by the corresponding author(s). Aggregated values of the current state (at the delivery of this document) could be found in Section 4.

The selection criteria of each reported item can be motivated by three different aspects: 1) *Interest*, a POP report have demonstrated the topic matters, so we include this kernel, pattern, and best-practices into our agenda (e.g., the analysis of the FFTXlib kernel was carried out during the POP1 project, resulting in a performance assessment and a proof of concept with promising results); 2) *Strategy*, once we have included an item, we decide to add other items which may complement it (e.g., we include the GMRES algorithm due it is used in the BEM4I kernel and the Calculix solver); and 3) *Affinity*, a work package member may propose a topic that corresponds to an area of its interest. If the topic can be potentially used by any future element, we also decided to include it (e.g., some applications recently studied include Python as the programming language).

3.1 Kernel related information

1. *FFTXlib* is the stand-alone kernel that represents the Fast Fourier Transformation (FFT) algorithm used in the Quantum ESPRESSO application, one of the most used plane-wave Density Functional Theory (DFT) codes in the community of material science.

2. *Comm Imbalance* is a synthetic program which reproduces a communication pattern in between several MPI processes.

3. *DuMuX/DUNE.* This kernel code implements alltoall type communication for sparse data sets with data structure update. The kernel is inspired by a POP study investigating the scalability of the DumUX/Dune code. DuMuX is a simulator for flow and transport processes in porous media. It is based on the Distributed and Unified Numerics Environment (DUNE). One of the main computational parts of DuMuX is a differential system solve step with a sparse matrix. The solver step requires the exchange of data between neighbouring processes and the updating of the data structures based on the exchanged data. The amount of data to be exchanged varies for the different processes and the update of the data structure can be expensive.

4. *RankDLB* This kernel demonstrates the dynamic re-distribution of work packages among MPI ranks. Initially, all ranks have the same amount of work packages but, at some point in time, one rank starts to produce more work items and therefore requires more time.

5. *BEM4I* is a library of parallel boundary element based solvers developed at IT4Innovations National Supercomputing Center. It supports solutions of the Laplace, Helmholtz, Lame, and wave equations. The library implements OpenMP and hybrid OpenMP/MPI parallelization.

6. *JuPedSim* is an open source framework for simulating, analyzing and visualizing pedestrian dynamics in complex geometries, with the possibility for several exits and obstacles.

7. *Parallel File I/O* is a naive approach to file I/O in parallel software for one process to sequentially read/write ASCII data to/from a single file (e.g., using the C fscanf and fprintf commands) with point to point communications to share the data with all other processes.

8. *OpenMP Critical* is a miniapp recreated from an oil & gas code that represents the OpenMP critical section pattern and the computational aspects of the original code. This application solves the 3D wave equation using the pseudospectral method.

9. *Calculix Solver* is a kernel that solves a non-symmetric system of linear equations using a GMRES solver. It is extracted from the CalculiX code which calculates the laminar flow of air through a bent pipe. At the heart of this calculation in each timestep an iterative scheme is employed, which in each iteration solves multiple of these non-symmetric systems of linear equations using GMRES.

10. *Calculix I/O* writes simulation results to a file in frd-format with a user-defined frequency. The data is structured in certain blocks. Each block contains line-by-line the information stored at the nodes of the simulation mesh. This information includes the node positions, the topology information how nodes are connected to form finite-elements and user-specific data like (viscous) stresses, heat fluxes or other physical quantities.

3.2 Patterns and best-practices

1. *Communication imbalance in MPI.* By communication imbalance we refer to the situation where the amount of MPI calls or the message sizes change between processes. Of course the time taken by these communications depends on many factors, of which the number of calls, message sizes are important but also the type of MPI call (blocking or not) or the location of the processes involved in the communication (local within the node or remote).

   (a) *Re-consider domain decomposition.* Changing the domain decomposition may improve the communication imbalance of the application. Traditionally, domain decomposition algorithms only take into consideration the number of elements to be computed within the rank. A potentially interesting approach would be to modify the domain decomposition algorithm such that the cost function accounts for the number of elements within a domain, as well as its number of neighbours (appropriately weighted) and the communications the resulting domain must establish with them.

2. *MPI endpoint contention.* MPI processes often have to communicate with a list of neighbours. Depending on the order of send and recieve calls it may happen that many processes

get "synchronized" in that all of them try to send at the same time to the same given destination, resulting in the limited incoming bamdwidth at the destination becoming a limiter for the overall communication performance.

   (a) *Re-schedule communications.* A simple way to address the issue would be to sort the list in ways that avoid such endpoint contention. Optimal communication schedules can be computed, but in practice, just starting each list by the first neighbor with rang higher that the sender and proceeding circularly to the lower ranked neighbor when the number of processes in the communicator is reached will probably reduce the endpoint contention effect.

3. *Sequential communications in hybrid programming.* A frequent practice in hybrid programming is to only parallelize with OpenMP the main computational regions. The communication phases are left as in the original MPI program and thus execute in order in the main thread while other threads are idling. This may limit the scalability of hybrid programs and often results in the hybrid code being slower than an equivalent pure MPI code using the same total number of cores.

   (a) *Parallelize packing and unpacking regions.* We consider it is a good practice to taskify these operations allowing them to execute concurrently and far before the actual MPI call in the case of packs or far after in the case of unpacks. These operations are typically not very large and very memory bandwidth bound. For that reason we think it is a good practice not to parallelize each of them with fork join *parallel do* for each of them as granularity will be very fine and the overhead will have a significant impact.

4. *Multiple independent operations requiring global communication.* It is frequent to find codes where multiple independent operations are executed in sequence. If each operation has communication phases constituting a high percentage of its total execution time, the overal performance will be low compared with the real potential of the application.

   (a) *Coarse grain taskification (comm + comp) with dependencies* This best-practice involves the creation of coarse grain tasks (including communication and computation), using task dependencies to drive work-flow execution. Parallelism is then exposed at the coarser grain level, whenever it is possible.

5. *Wait for non-blocking send operations preventing computational progress.* MPI programmers often use non-blocking calls to overlap communication and computation. In such codes, the MPI process communicates with its neighbors through a sequence of stages: 1) an optional pack of data (if needed); 2) a set of send/receive non-blocking operations, which potentially could overlap one to each other; 3) wait for communications (potentially splitting for send and receive requests; and 4) the computational phase.

   (a) *Postpone the execution of MPI wait operations.* The recommended best-practice will consist on postponing the execution of the waits on sending buffers as much as possible, in order to potentially overlap the send operations with the Computational phase.

6. *Load imbalance due to computational complexity unknown a priori.* In some algorithms it is possible to divide the whole computation into smaller, independent subparts. These

subparts may then be executed in parallel by different workers. Even though the data, which is worked on in each subpart, might be well balanced in terms of memory requirements there may be a load imbalance of the workloads. This imbalance may occur if the computational complexity of each subpart depends on the actual data and cannot be estimated prior to execution.

(a) *Dynamic loop scheduling.* When parallelizing a loop which independent iterations may have different execution time, the number of iterations is large (compared to the number of cores), and the granularity of each iteration may be small, one can use a dynamic schedule to distribute the iterations across work units as provided by the OpenMP paradigm, for example.

(b) *Conditional nested tasks within an unbalanced phase.* In other cases N may be small (close or equal to the number of cores) and the granularity of individual sub-computations may be large. This happens for example in the Calculix application. In that case, the individual sub-computations correspond to the solution of independent systems of equations with iterative methods that may have different convergence properties for each of the systems.

(c) *Conditional nested parallel region within an unbalanced phase.* In cases where the number of iterations (N) is equal to the number of workers (W) nested parallelism inside the straggling workers can be used to decrease the load imbalance. The idea is to utilize the unused computational resources of workers already finished and thus idling.

7. *Inefficient file I/O due to many unbuffered write operations.* In a naive implementation I/O operations are most likely implemented in serial. Data is read from and written to disk on demand whenever it is required to do so. However this might lead to a significant performance decrease if the amount of data transferred to or from file is very small in a single operation and many of these operations happen.

   (a) *Using buffered write operations.* Modern HPC file systems are designed to handle writing a large amount of data to a file. However, if the application performs a lot of write operations that write data in very small chunks this leads to an inefficient use of the file system's capabilities. Thus it is recommended to use fewer write operations that write large chunks of data to a file.

8. *Sequential ASCII file I/O.* In this pattern data held on all processes is read or written to an ASCII file by a single process. This is inefficient for several reasons: 1) Data must be communicated to the file I/O process from all other processes, one at a time; 2) It is more expensive to read/write ASCII data compared to binary files; or 3)It does not make use of any available parallel file system.

   (a) *Parallel library file I/O.* This best practice uses a parallel library for file I/O to write to a single binary file. A parallel library can make optimal use of any underlying parallel file system, and will give better performance than serial file I/O. Additionally, reading and writing binary is more efficient than writing the equivalent ASCII data.

   (b) *Parallel multifile I/O.* This best practice uses one file per process for reading and writing. This approach may be appropriate when a single file isn't required, e.g. when writing checkpoint data for restarting on the same number of processes, or where it is optimal to aggregate multiple files at the end.

9. *OpenMP critical section.* The OpenMP standard provides a critical section construct, which only allows one thread to execute the block of code within the construct. This feature allows blocks of code to be protected from race conditions, for example with write accesses into a shared array or incrementing a shared counter. However, usage of this construct, especially within parallel loops, can severely reduce performance. This is due to serialisation of the execution causing threads to "queue" to enter the critical region, as well as introducing large lock-management overheads required to manage the critical region.

   (a) *Remove critical section block.* This best practice recommends removing the critical statement from the parallel region. This can be achieved by moving the critical region outside of the loop, often by caching per-thread results in some way, and finalising the results in a single or master only region. This trades the serialisation and lock-management overheads for some additional serial execution but will often lead to overall performance improvement due to performance gains in the parallel region.

   (b) *Replacing critical section with reduction.* This best practice recommends that if the critical block is performing a reduction operation, this be replaced by the OpenMP reduction clause which has a much lower overhead than a critical section.

10. *Serialized computation and communication.* Many parallel algorithms on distributed memory systems contain a certain pattern, where a computational phase is immediately followed (serially) by a collective communication to share the computed results.

    (a) *Overlapping computation and communication.* To overlap the communication with the computation, the communication has to be initiated before the computation and one waits for the completion of the computation later on.

11. *Manual loop unrolling.* Most of the program execution time is spent on cycles. Consider the loop optimization method as a manual loop unrolling. This optimization is performed when the cycle body is not complex. It is necessary to use executable blocks more efficiently at each iteration, duplicating the loop body many times depending on the number of executing blocks.

    (a) *Effective auto-vectorization.* A number of methods of automatic generation of vector instructions have been developed for optimizing compilers. They allow vectorizing code without control branches or with some template branches.

12. *Dynamic balance problems in MPI.* A performance analysis reveals a significant time spent in wait state and an imbalance in the computational load, i.e. the number of work packages, per MPI rank. It is important to verify that the load imbalance is present and that the wait time is not cause by the limited speed of the network.

    (a) *Solving dynamic load balance problems.* In order to dynamically balance the computational work among MPI ranks, a small set of routines must be implemented that perform the following tasks: 1) measure computational load for each MPI rank; 2) remove and add work packages from a single MPI rank; and 3) transfer work packages among MPI ranks.

13. *Loop iterations manually distributed.* Some OpenMP application developers manually distribute the iterations of loops in parallel regions instead of using OpenMP's worksharing constructs. This can prevent the runtime from using optimized schedules.

(a) *Leveraging OpenMP worksharing constructs.* When the individual iterations of an for loop have significant differences in runtime, a manual distribution or the naive OpenMP for loop might lead to sub-optimal results. However, OpenMP provides several advanced worksharing constructs that help the developer to mitigate this situation.

3.3 Other classification criteria

In this section we report the complementary items used as the additional classification criteria beyond the patterns and best-practices described in the previous section. We group these complementary items in 4 different categories: *programming languages*, *programming models*, *algorithms*, and *disciplines*. For each of the new entries in the co-design site we provide a brief description.

We have included 4 **programming languages**:

1. *C* is an imperative procedural language widely used in system programming due to its huge portability (C compilers are available through a great quantity of platforms) and its ability to allow programmers to write very efficient implementations. The language provides a complete support of memory management and its language constructs maps efficiently to the underlying Instruction Set Architecture (i.e., C is a low-level programming language).

2. *C++* is a general-purpose programming language with object-oriented programming features. It is available in a great quantity of platforms which allow C++ programs to be portable across different architectures. C++ shares with the C language most of the syntax and its memory management features (i.e., static-, dynamic- and automatic- storage duration objects plus the thread local storage objects).

3. *Fortran* is a general-purpose programming language still quite used in scientific computing and very popular in high-performance computing. Its name derives from FORmula TRANslation, which gives an idea of the initial target of the language. Although multiple variants of this language exist, their main features can be summarized as: i) modularity and i)) intrinsic support of array operations

4. *Python* is an interpreted, high-level, general-purpose programming language. It is dynamically typed and garbage-collected. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Python's design philosophy emphasizes code readability, and aims to help programmers write clear, logical code for small and large-scale projects. CPython is the open-source reference implementation of the Python programming language.

We have included 8 **programming models**:

1. *CUDA* is a general purpose parallel computing platform and scalable programming model for NVIDIA graphics processing units (GPUs). It allows C/C++ and Fortran developers to design specific device functions called kernels that are executed in parallel by groups of threads and thus efficiently utilize a large number of CUDA cores available on current GPUs.

2. *MPI* is a standardized and portable message-passing communication protocol for programming parallel computers. MPI provides communicators, point-to-point communication, collective communication, derived datatypes, and some modern concepts as one-sided communication, dynamic process management, and I/O.

3. *OmpSs* extends OpenMP with compiler directives for asynchronous parallelism and heterogeneous architectures (i.e., GPUs, FPGAs, accelerators). Also, it can be understood as an extension of accelerator-based APIs like CUDA or OpenCL. A detailed description can be found at https://pm.bsc.es/ompss.

4. *OpenACC* is a directive-based high-level programming model similar to the OpenMP but intended for accelerators. The parallel regions are decorated with compiler directives that enable portability of the code to a wide range of accelerators.

5. *OpenCL* is a standard for programming heterogeneous systems, e.g. CPU and GPU, and supports data and task parallelism. It defines abstract platform, execution, memory and programming models that describe features and behaviour of the target system.

6. *OpenMP* is an implementation of multithreading, where a master thread creates a specific number of child threads (workers). The system splits a master's computational task into smaller ones and distributes them to threads. These threads then compute concurrently. Each thread is executed on a different processor. Parts of code that should run in parallel (parallel regions) are distinguished by specific compiler directives inserted into the code.

7. *PGAS* assumes a global memory address space that is logically partitioned and a portion of it is local to each process, thread, or processing element. This can facilitate the development of productive programming languages that can reduce the time to solution, i.e. both development time and execution time. Languages based on PGAS are Unified Parallel C, Co-Array Fortran, Titanium, X-10, Chapel and others.

8. *POSIX Threads* defines an Application Programming Interface (API) for thread programming. Implementations of this interface exist for a large number of UNIX-like operating systems (GNU/Linux, Solaris, FreeBSD, OpenBSD, NetBSD, OS X), as well as for Microsoft Windows and others.

We have included 5 **algorithms**:

1. *The Arnoldi Method* is an algorithm that was originally used to reduce an arbitrary matrix to Hessenberg form. Later it was discovered that it can incorporated into solving eigenvalue problems because eigenvalues of a matrix in Hessenberg form can be computed more easily compared to a matrix of general form.

2. *The Finite Element Method* is a general numerical method for solving physical problems in engineering analysis and design. It is used for modelling of a continuum in steady-state, propagation (transient), and eigenvalue problems in many domains, e.g. structural mechanics, CFD, thermodynamics, electromagnetics, etc. It is a robust universal method able to solve large complex systems for the price of relatively high computational demands.

3. *The Link Cell Algorithm* comes from the domain of Molecular Dynamics Simulations. Here, for a set of molecules, pairwise interactions have to be computed under the constraint of a limited interaction range.

4. *The Generalized Minimal RESidual Method* is an iterative method for the numerical solution of a nonsymmetric system of linear equations. The method approximates the solution by the vector in a Krylov subspace with minimal residual.

5. *The Fast Fourier Transform* is an algorithm that computes the Discrete Fourier transform (DFT) of a sequence, or its inverse (IDFT). The algorithm can be applied to both real and complex input, and can result in both a real or a complex output.

We have included 10 **disciplines**:

1. *Density Functional Theory* is a computational quantum mechanical modelling method to investigate the electronic structure of many-body systems, in particular atoms, molecules, and the condensed phases. The name Density Functional Theory comes from the use of functionals (functions that take functions as argument or return them) of the electron density.

2. *Advection diffusion sedimentation.* The people living their lives near active volcanoes are under the grave threat due to the fallout of the volcanic ash. It can disrupt the air-traffic, harm the agro-based activities, destroy the residential structures due to overloading, et cetera. It is essential to understand the evolution of the fallout of the volcanic ash using attested computational methods.

3. *Volcanic plume simulation.* Volcanic eruptions are complex phenomena that involve the dynamics of magma transfer, starting of volcanic unrest, interaction with the volcanic edifice, eruption of magma and gases, and finally the atmospheric transport of volcanic ash and aerosols. The violent activities involved in the volcanic eruption classify it as a natural hazard.

4. *Eigenvalue problems* are a well known problem from linear algebra that can be found in many scientific fields. In structural engineering vibration analysis involves solving eigenvalue problems to obtain natural frequencies of the system. For the stability analysis of dynamical systems the sign of the eigenvalues indicate if a system will converge to a stable point or not.

5. *Astronomical simulation* is an active research field of Astrophysics. HPC is tightly connected with Astronomical simulation due to the large scale and complexity of Astrophysical problems. Cosmological N-body, star-formation Magnetohydrodynamical (MHD) simulations, data analysis and data modelling are a few examples of Astronomical simulation.

6. *Molecular Dynamics* is one of the oldest applications of computers in science and up to today one of the base pillars in many computational science areas like Biology, Chemistry and Material Sciences. MD simulates the movement of atoms and molecules based on time integration of Newton's equation of motion.

7. *Computational Fluid Dynamics* is a discipline that deals with the numerical solution of partial differential equations (PDEs) governing the flow of fluids. The equations under consideration are the Navier-Stokes equations, describing the momentum transport in the fluid.

8. *Pedestrian dynamics.* Pedestrian dynamics is a field in the area of computational civil safety research. The goal is to model and simulate the flow of pedestrians in emergencies. This helps to design better emergency exits in large buildings like stadiums or convention centers and plan safety concepts for mass events like concerts or demonstrations.

9. *Neuroscience* is a branch of neuroscience which employs mathematical models, theoretical analysis and abstractions of the brain to understand the principles that govern the development, structure, physiology and cognitive abilities of the nervous system. It employs computational simulations to validate and solve the mathematical models that - are in most cases - too complex to be solved analytically.

10. *Seismic Data Processing*. Seismic profiling is the technique of using sound waves to image underground rock strata, either on land or under the sea. It is widely used in oil exploration. Once the raw data has been acquired, Seismic Data Processing is then used to convert it into a meainingful form, such as a 2D picture of the cross-section being imaged, and then to enhance that image for greater clarity. This processing consists of a number of stages: filtering, deconvolution, stacking (summing separate seismic traces from the same depth) and seismic migration (the process of geometrically relocating seismic events in space or time).

4 Conclusions

In this section we will discuss the overall success of the *MS3 - First Methodology Milestone* (at M18), we will present a snapshot of the current state of the repository pages, we will decompose such evaluation per partner (crossing results with Section 1.1), and we will conclude the evaluation with qualitative appraisals over these results. In addition, we will hightlight some future work we plan to include in the next milestone.

In Section 1.1, we presented our expected contribution up to M18 (summarized in 10 kernels and 46 pages). During this period we have started to work on, 10 kernel repositories (see Section 3.1) and 57 pages (see Sections 3.2 and 3.3). Besides, we have also included in our agenda, as possible candidates, 8 additional kernels and 26 additional pages. Nevertheless, in the following paragraphs we will just focus on reporting the state of the content described in Section 3.

We have published, up to the delivery of this document[4], 10 out of 10 kernels. Taking into the account the defined target in the *MS3 - First Methodology Milestone* [3] was to hit 8 kernels, we should conclude that we have succeeded in reaching the expected amount of kernels.

Table 2 summarizes the current state of the reported pages. A total of 39 out of 57 pages has been published in the co-design site, representing a 68% of them. In addition, 28% are on-review, 4% in-progress and *none* of them are pending to start.

Component	Pending	In progress	On Review	Published	Total
Patterns		1	6	6	13
Best-practices		1	10	6	17
Programmming models				8	8
Programmming languages				4	4
Disciplines				10	10
Algorithms				5	5
Total		2	16	39	57

Table 2: Current state of the MS3 repository pages at the delivery of this document.

Taking into the account the kernels which have been made already public (despite the *peer* review process of these may still be pending), and the pages with a firm proposal (i.e., those already published, or being reviewed), the contribution per partner is summarized in Table 3. The table gathers the number of kernels, pages and/or extensions each partner has contributed with. It also details how the pages are distributed in collections, and the kind of extensions created by the partners (if applicable). Some partners also declare additional items which has

[4]The last update of this deliverable was done on May 28th, 2020

Institution	Contribution
BSC	2 out of 2 kernels. 14 out of 14 pages: 5 patterns, 5 best-practices, 3 languages, and 1 algorithm. 5 site extensions: 2 internal pages, 2 site formatting issues, and 1 feature extension.
HLRS	2 out of 2 kernels. 8 out of 8 pages: 1 pattern, and 2 best-practices, 3 disciplines, 1 programming model, and 1 algorithm. In addition: 1 pattern, and 1 best-practice (both pending).
IT4I	1 out of 1 kernels. 4 out of 4 pages: 1 pattern, 2 algorithms, and 1 programming language. In addition: 7 programming models (all published).
JSC	1 out of 1 kernels. 4 out of 4 pages: 1 pattern, 1 best-practice, and 2 disciplines.
NAG	2 out of 2 kernels. 8 out of 8 pages: 2 patterns, 4 best-practices, and 2 disciplines.
RWTH	2 out of 2 kernels. 8 out of 8 pages: 1 pattern, 3 best-practices, 3 disciplines, and 1 algorithm. In addition: 1 pattern, and 1 best-practice (both in-progress).

Table 3: Actual contribution per partner for MS3 at the delivery of this document.

been reported in this document, but do not correspond with the initial expected effort. In such cases, we have included an additional entry in the description prefixed with *In addition:*.

Overall, we can conclude that all the partners have contributed according with the expected effort described in Section 1.1.

The current contributions allow to figure out the type of contents that will populate a site of this nature. For instance, the publication of the currently found patterns and best-practices will help to understand the relationships that may exist among them. That will allow to consider further classification criteria, merging or breaking down such categories (e.g., patterns), or the creation of a taxonomy of these components. So we expect to revisit the already published pages in the short-/mid- term to reformulate them, if needed.

As future work, we plan to continue with more extensions that will allow to better exploit the information gathered in the site. That will include, as previously commented, to consider the creation of a taxonomy of patterns that will allow to navigate through all these items from a generic concept to more specific ones. Other possible extensions may involve presenting results by means of third party tools (e.g., Grafana), although we will consider the required effort to carry out this extension with respect to dedicate it to improve other aspects in the site. Another possible extension may involve to include public POP reports (coming from WP4 or WP5) within the co-design site. POP reports will become, in such a way, a collection items that could be referenced by other co-design components (e.g., reference a POP assessment from a pattern).

References

[1] GitLab. Gitlab documentation. https://docs.gitlab.com/ce. [On-line, accessed: May 2020].

[2] Gruber, J. Markdown. https://daringfireball.net/projects/markdown. [On-line, accessed: May 2020].

[3] POP Consortium. Performance Optimisation and Productivity: A Centre of Excellence in Computing Applications, 2018.

[4] POP Consortium. D7.1 - Co-design repository structure, 2019.

[5] POP Consortium. D6.1 - First Report on Proof-of-Concept, 2020.

[6] POP Consortium. D8.1 - First report on methodology development and tool improvement, 2020.

[7] The Jekyll Team. Jekyll on-line documentation. https://jekyllrb.com/docs. [On-line, accessed: May 2020].

[8] The MathJax team. MathJax. https://www.mathjax.org. [On-line, accessed: May 2020].

[9] Thomas Leitner. kramdown. https://kramdown.gettalong.org/index.html. [On-line, accessed: May 2020].

[10] Thomas Leitner. Math Blocks in: kramdown. https://kramdown.gettalong.org/syntax.html#math-blocks. [On-line, accessed: May 2020].