D7.1– Co-design repository structure
Version [1.0]

Document Information

Contract Number	824080
Project Website	www.pop-coe.eu
Contractual Deadline	M6
Dissemination Level	PU - Public
Nature	R
Authors	BSC
Contributors	NAG, JSC, HLRS, RWTH, IT4I
Reviewers	RWTH, BSC
Keywords	Co-design, kernels, repository, metrics, patterns, best-practices.

Notices: *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.*

©2019 POP Consortium Partners. All rights reserved.

Change Log

Version	Author	Description of Change
v0.1	Xavier Teruel	Initial version of the deliverable. Document structure's definition.
	Xavier Teruel	Initial contents for introduction, tools and user interface's sections.
v0.2	Victor Lopez	Software requirements.
v0.3	Xavier Teruel	Introduction: requirements, specification and design
v0.4	Xavier Teruel	User interface collections
	Xavier Teruel	Layout and program chapter. Contribution guidelines.
v0.6	Victor Lopez	Complete tools chapter. Repository integration.
v0.7	Xavier Teruel	Split introduction and analysis chapters.
	Xavier Teruel	Final (auto-) review. Some refactor.
	Xavier Teruel	Including RWTH review comments.
	Victor Lopez	Including tools chapter review comments.
v1.0	Xavier Teruel	Including BSC review comments.

Contents

Executive Summary **5**

1 Introduction **7**
- 1.1 POP methodology: a hierarchy of metrics . . . 7
- 1.2 Repository main goals and description . . . 8
- 1.3 Related work . . . 10
- 1.4 Glossary of terms . . . 11

2 Analysis and Design **12**
- 2.1 Analysis of requirements . . . 12
- 2.2 System specification . . . 14
- 2.3 Infrastructure design . . . 15

3 Supporting Tools and Technologies **18**
- 3.1 Jekyll introduction . . . 18
- 3.2 Markdown language introduction . . . 19
- 3.3 GitLab introduction . . . 21
- 3.4 Software management . . . 22

4 Building the User Interface **24**
- 4.1 User interface structure . . . 24
- 4.2 From repository entities to Jekyll collections . . . 25
- 4.3 List of user interface collections . . . 28
  - 4.3.1 Metrics (and efficiencies) . . . 28
  - 4.3.2 Patterns and behaviors . . . 29
  - 4.3.3 Best practices . . . 30
  - 4.3.4 Programming models . . . 30
  - 4.3.5 Base languages . . . 31
  - 4.3.6 Disciplines . . . 32
  - 4.3.7 Algorithms . . . 32
  - 4.3.8 Bottleneck boundaries . . . 33
  - 4.3.9 Systems . . . 34
- 4.4 Contribution guidelines . . . 34

5 Building the Program Repositories **36**
- 5.1 Kernel repository structure . . . 36
- 5.2 List of program collections . . . 37
  - 5.2.1 Programs . . . 37
  - 5.2.2 Versions . . . 38
  - 5.2.3 Experiments . . . 39
  - 5.2.4 Results . . . 40
- 5.3 Program organization guidelines . . . 41
- 5.4 Website integration . . . 41

Appendix A: The POP Methodology **44**

List of Acronyms and Abbreviations **46**

List of Figures **47**

References **48**

Executive Summary

The two main goals of the co-design repository are: 1) to build a database of performance metrics characterizing applications; and 2) to build a repository of kernels representative of fundamental behaviors identified in real applications (and linked to their importance in the application). This deliverable aims to establish the fundamental structure of the repository and, for this purpose, it will address the following topics: 1) Entities, relationships, and classifications; 2) Repositories to store data; and 3) Interface with users and collaborators.

POP has already defined a methodology for the analysis of parallel codes to provide a quantitative way of measuring relative impact of the different factors (i.e., performance metrics) inherent in parallelization. The repository should include these metrics as part of the final design of the project. It will also need to be able to incorpore any new metric as the result of this methodology extension (see Work Package 8).

The repository is defined around a set of collections and the relations among them. This information povides the structure of the database. The list of collections included in the codesing repository is:

- *Programs, versions, experiments, and results: are the core of the system.*
- *Language and programming models, algorithms and disciplines: are purely descriptive and allow to classify one of the main entities in the system, the program.*
- *Patterns and behaviors with best practices: are the indicators of problems and the corresponding solutions that users can find in their codes (program's versions).*
- *Bottleneck-boundaries, POP metrics and systems: indicate theoretical limits and performance results corroborating a certain observation. Systems offer additional information about the execution environment when reporting an experiment.*

The central component of the system is the program. All the relations between programs and languages, models, algorithms, and disciplines, have cardinality $0..N$*. A program can be related with multiple descriptive entries, and any of these descriptive entries can be related with multiple programs. There is a deterministic result-to-program path, i.e., results belong to one, and only one, experiment, an experiment uses one, and only one, version, and a version belongs to one, and only one, program. The relationships between the code and patterns/best-practices is done through program versions: a program could report multiple versions, one recommending a best-practice, another one implementing it. A program version can also report bottleneck boundary issues. This relationship allows to report the theoretical limits (memory, computation, transfer, etc.) of a given code version. Finally, experiments are done using a program version and they are executed on a single platform. Comparisons between the execution of a given program on two different architectures will produce two different experiment's entries. Experiments will include performance metric results obtained during the execution.*

In order to implement all these collections, the repository will use **Jekyll** *(a website generator). A collection's item will become a Jekyll file that will finally produce a HTML page. Jekyll uses* **Markdown** *to write the site content. Relationships between elements of a given collection with elements of another one will be specified through the Jekyll frontmatters and materialized by means of HTML links between the resulting pages.*

Managing code versions requires the use of a Control Version System. One of the most popular in recent times is **GitLab***, due its great versatility in terms of managing distributed repositories, as well as its ease and standardization of use. It offers an effective way in terms*

of managing groups, projects, issues (tickets), tags (named points in the program's history), permissions management (privacy and security), support for documentation using wiki, etc.

In order to implement the co-design repository database, we will distinguish among two different types of Jekyll collections:

- *Program's collections are related to a specific individual program. By design principle, they will live together with the code they are related to in different GitLab* **kernel repositories**. *The program collections are: programs, versions, experiments, and results.*

- *Shared collections are shared among all the programs in the repository. Kernel's programmers could leverage these common entities by relating them with their own program's collections (e.g., relate a program's description with a given algorithm). Shared collection will live in a single GitLab* **layout repository**.

Every time any of these GitLab repositories is updated, the system will automatically build the Jekyll HTML static pages and they will be immediately served on the public URL.

1 Introduction

This deliverable aims to establish the fundamental structure of the co-design repository. For this purpose, it will address the following topics:

- Entities relationships and classifications.
- Repositories to store data.
- Interface with users and collaborators.

Work Package 7 (WP7) within POP2 Center of Excellence aims to collect information from WP5 and WP6 about patterns that are usually found in parallel codes. This information is useful to develop co-design activities between hardware, parallel programming models and application developers. The co-design repository is the tool to store and query this information, we refer to this complete infrastructure as the POP Repository.

An important deliverable's aspect (as the result of Task 7.1 *Define repository structure* - months 1 to 6), is its character of *initial* structure definition. This is due the *dynamic* nature of the repository. After writing this document, the project will continue with two additional activities: one task populating the program repositories; another task reporting the metrics obtained by these codes. Both activities will last for the rest of the project life time (i.e., months 7 to 36), and therefore, it is expected that extensions and modifications will come up during this period. These changes will update the structure proposed in the deliverable in order to extend or adapt the initial structure definition with the new found requirements. The current state of the POP Repository is publihed in the following URL:

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

The rest of the introductory chapter focuses on completing the description of the co-design repository. First, it includes a section dedicated to the POP methodology and its metrics, which should take their place in the final structure of the repository. Second, Section 1.2 establishes the repository's scenario that will allow to carry out discussions of certain aspects of the repository structure (e.g., the interaction between a source-code repository and the end-user). Third, Section 1.3 discusses some related work useful to understand and complete the vision of the system. And finally, Section 1.4 includes some definitions that readers may consider of interest in order to understand the terminology used in the document.

The rest of the document contains the whole repository definition. It is structured as follows: Chapter 2 contains the description of the different development stages to produce an initial design of the POP repository. Chapter 3 presents an introduction to the tools and technologies used to implement the infrastructure. And finally, Chapters 4 and 5 present the details needed to implement the user interface and the program repositories respectively.

1.1 POP methodology: a hierarchy of metrics

As one of the fundamental goals in the design of the repository, we have to include the performance results obtained when executing the kernels on different architectures. Programs should report these results and allow to easily access them. On the other hand, POP has defined a methodology for the analysis of parallel codes to provide a quantitative way of measuring relative impact of the different factors inherent in parallelization. This section introduces these metrics, explains their meaning, and provides insight into the thinking behind them.

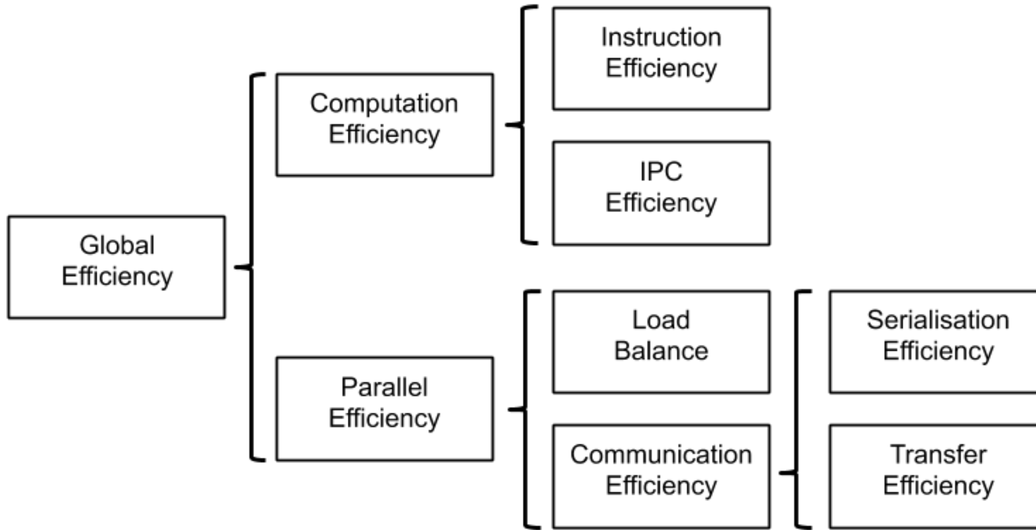

Figure 1: POP Methodology: a hierarchy of metrics and efficiencies.

POP metrics are the core of POP and therefore they must be considered as a requisite in the design.

A feature of the methodology is that it uses a hierarchy of metrics, each metric reflecting a common cause of inefficiency in parallel programs. These metrics then allow comparison of parallel performance (e.g., over a range of thread/process counts, across different machines, or at different stages of optimization and tuning) to identify which characteristics of the code contribute to inefficiency.

The metrics are calculated as efficiencies between 0 and 1, with higher numbers being better. In general, we regard efficiencies above 0.8 as acceptable, whereas lower values indicate performance issues that need to be explored in detail. The ultimate goal then for POP is rectifying these underlying issues.

Figure 1 shows the hierarchy of metrics. The reader can also find, in the appendices of this document, a description of each of the components of the POP methodology and how to compute their efficiency values.

The POP Repository should include these metrics as elements of one global entity (i.e., metrics), being part of the final design of the project. It will allow a second entity (i.e., experiments) to report the values obtained (i.e., efficiencies) for each of them. In addition, experiments should also relate a given program's version and the hardware description used to carried out the test. We consider these entities as system's pre-requisites.

1.2 Repository main goals and description

The two main goals of the co-design work package within the POP-2 CoE are: 1) to build a database of performance metrics characterizing applications, and 2) to build a repository of kernels representative of fundamental behaviors identified in real applications (and linked to their importance in the application). The objective of this repository is thus to gather, from any analysed code, the fundamental application's patterns, behaviours, and best practices that are globally representative of important issues to be considered in system design.

These kernels will be usable within the POP project as training material and to demonstrate, in dissemination activities, the benefits of the programming best-practices that we promote.

They will also be available for system designers (architecture, software, etc.) in projects outside POP to demonstrate the potential of their proposed approaches and get a rough estimation of which codes their techniques will have an important impact on.

The rest of this section will set up an initial structure of the platform, highlighting its main components as well as 1) the interaction between them and 2) the interaction between the whole system and the user. Its main purpose is to create a common base that will allow POP partners to express their expectations with respect the repository and collect them in Section 2.1: *Analysis of Requirements*.

Along this document we will use the term **repository** to refer the whole system, and we use the term component to refer to iternal subsystems. A component is defined as a delimited part of the system which is responsible for a specific activity. We distinguish among three different repository's components: 1) the **user interface** is the component that allows a more friendly interaction between the user and all the data; 2) the **program repositories**, composed by 2.a) the **source code repositories**, the component responsible to store kernels, benchmarks and applications (we will generically refer to all of them as programs), and 2.b) the **meta-data repository**, the component that keep track of all the information related-to or derived-from these programs; and finally 3) the **file repository** is an optional component responsible to store additional supporting files (i.e., input files, output files, simulation traces, etc.).

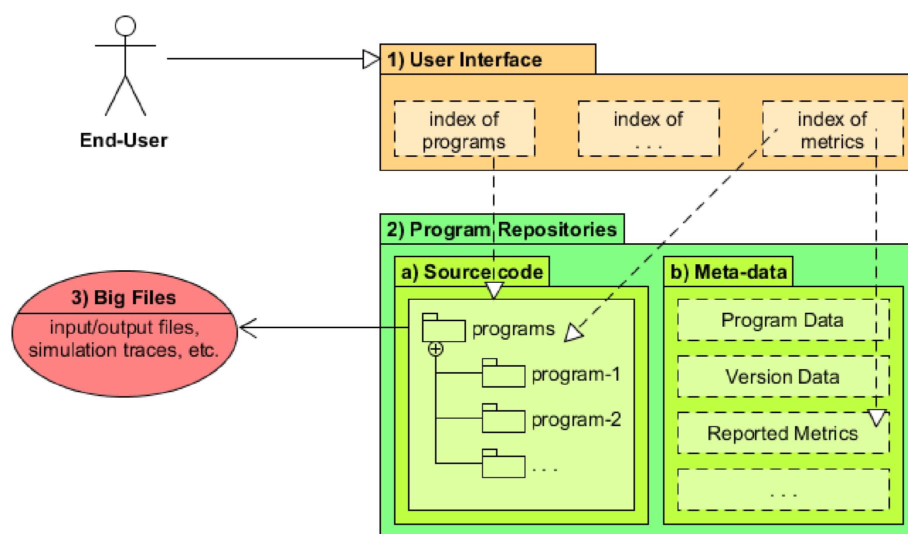

Figure 2: POP Repository structure: subsystems and components interaction.

Figure 2 shows a visual representation of the repository's components and how they interact with each other. On one hand, we have the **programs** and their associated **meta-data**. Programs are a set of source codes that can be downloaded by any potential user of the system. The meta-data describes different aspects of the program: i) the attributes of the program (i.e., field of research, programming languages, licenses, etc.); ii) specific details of a given program's version (i.e., data layout/structure used to implement it, programming models, parallel decomposition approach, etc.); and iii) the technical documents/reports, including performance metrics, that developers have reported (e.g., performance analysis or proof of concepts). Programs and meta-data are the core components of the system.

A second component, on top of the core system, is the **user interface** layer. This component allows navigating through all the repository contents in a structured manner. Users could apply different criteria that will allow them to look for the correspondig code packages and navigate

through their reported metrics. The user interface must allow navigating from a user "query" to the correspondent program version, its attributes, its technical reports, its guidelines, etc. The interaction degree could be as simple as listing all the kernels with the corresponding data allowing the user to navigate through the entire list, or a more complex interaction involving a *database-like* system to store the repository data and allowing quantitative queries.

As an additional component, we have introduced the support of a large file module which allows to store files whose size could be huge in comparison with other files or documents. Among these files we can find input files (problem set, configuration, etc.), and output files (results, execution traces, profile information, etc.).

In the final specification, components may exist by themselves or they can be combined in "bigger" entities, or also splited in "smaller" entities, according with the final design decisions. For example, the meta-data repository could also store big files merging both components in a single one.

1.3 Related work

Benchmarking allows to assess the performance of programs under certain circustances. The benchmark's results also allow to compare performance between different software's versions, parallelization approaches, compiler's quality of code, and systems. Conclusions over the obtained results can be used to take decisions about new design of architectures, to improve current techniques on application parallelization, or to propose new services on current programming models (or the design of new ones). There are plenty literature proposing benchmarking tools.

EPCC has developed a family of micro-benchmarks [5] allowing to measure programming model performance. Its main goal is to better understand the performance opportunities of the different HPC models, languages and architectures. Among the available benchmarks we can find the OpenMP microbenchmarks, measuring the overheads due to synchronisation, loop scheduling, array operations and task scheduling; OpenMP/MPI microbenchmarks, to measure performance in hybrid codes; the Fortran Coarray microbenchmarks, to measure a set of basic operation in this idiomm; the OpenACC benchmarks, including low-level benchmarks, kernels and small scale applications; etc.

In the other hand, synthetic benchmarking sometimes offers a biass vision on application performance. As they focus on measuring a given aspect of the performance (e.g., the overhead introduced by a MPI barrier), they deliverate ignore how it impacts on real application performance. For example,it is extremely useless to improve the overhed of creating a task in OpenMP if after that do not take into account any kind of data affinity when scheduling these tasks. So, it is good to characterize specific functionalities of a given programming model (e.g., synchronization at EPCC OpenMP microbenchmarks), but we also need to determine the role synchronization is playing in real application.

A second approach of benchmarking consists in characterizing the application behaviour by means of mini-apps (aka proxies). These "small" codes allow to isolate the main(s) component(s) of a given program, i.e., kernel(s), capturing the most important facets of the original application. The reduced size of the mini-app allows to easily test, analise and plan new software improvements that, in theory, will also impact on the genuine application.

Mantevo has developed a collection of mini-apps [8] mimetizing the most important computational operations of their corresponding original applications. As the mini-app captures the primary performance behavior it becomes the central element in the co-design activities based in the results obtained when profiling their execution. Among the set of mini-apps the Mantevo project includes in the site, we can find examples in multiple domains and algorithms: Implicit Finite Elements, Molecular Dynamics, Contact Detection, Electrical Circuits, etc..

1.4 Glossary of terms

POP Repository is the term we use to refer the whole system described in this document. It includes the layout site (an interface that simplifies the access to underlying shared data) and the program's repositories (where specific program's data lives).

Program is the generic term we use through this document to refer any code which can be referred in POP repository. In general, these codes will be applications or benchmarks, but we also refer as programs the extracted kernels.

Application is a program with useful engineering or scientific application, i.e., a weather-, particles- or fluid- simulator, data mining software, deep learning, or any other program within a HPC domain.

Benchmark is a program with the unique goal to measure certain performance metrics: speed-up, throughput, memory foot-print, architecture boundaries, etc.

Kernel is the core of a program which determines its behaviour. It caputures the essential pattern of operation in such a way that any improvement in the kernel will impact in a similar manner in the program. The repository can include *application*'s kernels or *benchmark*'s kernels.

Layout repository is the repository keeping track of the common site structure plus the shared collections.

Program repositories is the set of repositories keeping track of the program codes, including their versions and their associated meta-data (i.e.,*program collections*).

2 Analysis and Design

Based on the description made in the introductory chapter, we will now focus on achieving a more detailed description of the problem and understanding of the nature of the repository. In this stage, we will capture the expectations of the partners involved in the project as potential end-users of the infrastructure. The goal is to extract the initial requirements of the repository. Section 2.1 will be responsible for conducting the first discussion of these aspects.

Once the requirements have been established, we will proceed to the elaboration of the specification (Section 2.2), whose main objective is to start the definition of the proposed solution. In this phase, we will give a more structured format to the elements detected during the analysis of requirements. Specifically, it will allow: 1) to have a global vision of the different entities that are part of the repository, and 2) to establish the corresponding relationships between them.

We will continue refining the initial specification, and we will start to take into account those aspects related with the design of the whole infrastructure (Section 2.3). We will also provide a more detailed description of the entities coming from the previous phase, and we will decide which technologies and tools we need to use to implement them. Chapter 3 will later describe these tools.

2.1 Analysis of requirements

As we have seen in previous sections, one of the components that must appear in our initial design are the POP methodology **metrics** (see Section 1.1). These metrics are classified and described in the official web site of the POP Center of Excellence and they characterize the behavior of an application in order to carry out a first performance analysis. We consider the POP methodology metrics a pre-requisite, rather than a regular requisite, due the consideration of these metrics is given by the description of the problem, and not driven by an analysis process.

In addition to current metrics appearing in the performance analysis methodology, the Work Package 8 will define a new set of metrics extending the POP methodology. These new metrics will allow to characterize the application behavior in other scenarios not currently covered: use of accelerators, intra-process programming models, vectorization, etc. Then, among the list of non-functional requirements we must consider **extensibility** as one of them.

Other non-functional requirement of the repository will be related with its **availability**, i.e., it should be *public* in all its different components: code and metrics. This requirement is initially covered given that the platform will be published in the form of a website and embedded in the already deployed POP-2 online services: the POP main website and the corresponding form that allows requesting POP services to other European institutions. As a future consideration, we will need to pay special attention to the licenses of the codes included in the repository: first, with respect to the use of derivative works, but also with respect to the restrictions on the distribution of the resulting codes.

Given the low population that we target during the extend of the POP-2 project, the **simplicity** of the design and the technologies used will be prioritized over a design based on complex tools that, although they would allow *ad-hoc* searches of the contents, would hinder its initial design and the future maintainability of the system. This implies that the set of classifications criteria should be kept as reduced as possible, promoting the extensibility of the system when a new criteria becomes necessary due to the incorporation of one or more kernels that require it.

In order to gather the functional requirements of the POP-2 kernel repository, a questionnaire was circulated among the project partners (as potential system's end-users). These questions sought to find out which are the most outstanding points and the basic features that should be included in the initial design of the system. They were organized in different sections: starting

with more general questions about the system's expectations, potential users, metrics and alternative methods to classify the kernels; to much more specific questions about what kind of technologies should we use for the implementation of each of the components: layout's interface, code repository, repository of metrics and storage for auxiliary files. Important conclusions that could be inferred from these discussions are:

1. common agreement that the repository could be used by a wide part of the HPC community (including computing and data centers, HPC experts and consultants, application developers and responsibles, architecture experts, students and/or lecturers for training purposes, etc.). Within the POP project it could be also used as training and dissemination material;

2. the characteristics that seem to be clear in the design of the repository are: i) a web-based accessed platform, and ii) a version management system based on git (e.g., gitlab);

3. references to related work that should be taken as reference in the development of the infrastructure;

4. a considerable amount of classifications criteria (which will be discussed in the following paragraphs); and

5. the confirmation that system's *extensibility* is a critical objective in its design (especially taking into account the objective of extending the POP methodology).

The repository must have the ability/capacity to work with **programs** (i.e., to describe, identify and relate programs with the rest of the concepts discussed in this section), including all its different **versions**. A given program's version could report several **experiments** on the aforementioned performance **metrics**. Program's versions are the target of these experiments, due two versions of the same program can behave differently using the same input set and executing on the same architecture with the same amount of hardware resources).

With respect to the description of a program, we should consider the **programming language** (e.g., C++, Fortran, etc.) and the parallel **programming model** (e.g., MPI, OpenMP, etc.) as a very useful classification criteria. The system should allow to create listings of programs using any of these two classification basis.

It will be also interesting to obtain those programs that use a certain common **algorithm** (e.g., FFT, LU, etc.), and also those codes which belong to a certain scientific domain or **discipline** (e.g., particle simulation, computational fluid dynamics, etc.).

In relation to the program's version, it should be possible to report certain **patterns** of code or behaviors. We understand by code pattern a set of operations that are carried out in a certain order, for example a sequence of service calls. With respect to the behavior, it should be understood from a dynamic point of view, that is, as the result of its execution. These patterns and behaviors can be associated, whenever is possible, with one or more **best practices** (i.e., programming hints). Versions could also report if they are *already using* a certain best practice or if they *recommend* its use to solve a certain code pattern or behavior.

Additionally, a program's version may report algorithmic **boundary** bottlenecks which do not allow to achieve a perfect scalability behavior due to the structure of the code (e.g., memory bound, computational bound, etc.).

As we have said previously, program's versions are related to one or more performance reports that will describe the conditions on which the **experiment** was carried out. It is desirable that the experiment's description will favour its reproducibility. A certain experiment will be carried out as the result of its execution on a certain computer (more generically, **architecture**). Finally,

each experiment will include a set of reported **results** attached to it. These performance reports may be based, initially, on performance analysis, proof of concepts, and audit reports already presented within the context of the POP project (and therefore they will be related with their methodology metrics).

2.2 System specification

Taking as the starting point the list of requirements gathered in the previous section, we can elaborate a first Entity Relationship Diagram (ERD) in which we will consider each one of the entities as a container of multiple entries in our system. The entities that we have detected are:

- Programs, versions, experiments and results: are the core of the system.
- Programming languages and programming models; algorithms and disciplines: are purely descriptive and allow to classify one of the main entities in the system, the program.
- Patterns and behaviors with best practices: are the indicators of problems and the corresponding solutions that users can find in their codes (program's versions).
- Bottleneck-boundaries, POP metrics and architectures: indicate theoretical limits and performance results corroborating a certain observation. Architectures offer additional information about our execution environment when reporting an experiment.

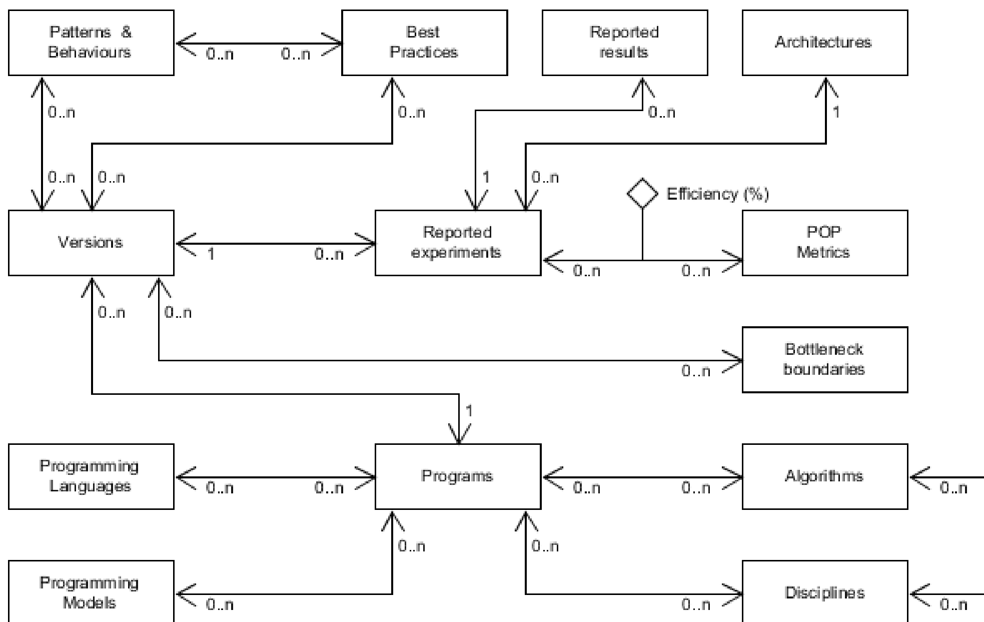

Figure 3: Entity Relationship Diagram (ERD) of the repository in the specification phase.

Figure 3 shows all these entities and their corresponding relationships. The lower side of the figure, focused on programs, is purely descriptive, while the upper side, focused on experiment-result reporting and it follows a more informative approach.

All the relations between programs and descriptive components (i.e., languages, models, algorithms, and disciplines), have cardinality $0..N$: a program can be related with multiple

descriptive entries in those entities, and any of these descriptive entries can be related with multiple programs. In addition, there is a relation between algorithms and disciplines allowing to relate these two concepts: *"algorithms commonly used in a given discipline"*.

With respect to the part of the system focus on experiments-results, it is worth mentioning:

- The deterministic *results-to-programs* path, i.e., results belong to an experiment, an experiment uses a certain version of a program, and a version belongs to one (and only one) program.
- The experiment-metric relationship contains the efficiency values of a certain POP Methodology metric.
- The results entity is an abstract entity which allows to store multiple reporting values/-metrics (and not only those related with the POP methodology).
- An experiment is executed exclusively on a single platform. Then, comparisons between the execution of a given program on two different architectures will produce two different experiment's entries.
- The relationships between program versions and patterns/best-practices allow the system to report problems/solutions of a certain code version.
- The program versions and bottleneck boundaries relationship allow the repository to report the theoretical limits (memory, computation, transfer, etc.) of a given code version.

2.3 Infrastructure design

In Section 1.2, Figure 2, we already saw the repository structure and its components. One of these components were the source code repository, which allows to keep different versions of the same program. Managing code versions or, in general, tracking the development history of a program requires the use of a *Control Version System* (CVS). One of the most popular in recent times is **Git**, due its great versatility in terms of managing distributed repositories, as well as its ease and standardization of use. Among the software which currently offers Git repositories, we find **GitLab** as one of the most popular. It offers an effective way in terms of managing groups, projects, issues (tickets), tags (named points in the program's history), permissions management (privacy and security), support for documentation using wiki, etc. In Section 3.3 we will find further information about this service.

The source code repository was closely related with the meta-data repository. In this component we found the program/version descriptions as well as the reported efficiencies (according to the POP methodology), plus other experimental results. These concepts correspond to programs-, versions-, experiments- (including efficiencies) and results- entities that we find in the specification section.

Although in this section is focused on system's design, we will start the introduction of some implementation guidelines in order to better understand all the concepts. The rest of implementation details will be fully developed in Chapters 4 and 5 , grouped into specific details about the layout layer, the program repositories, and the integration of these two components respectively (Section 5.4).

Among the different options that we have for the system implementation we could consider the use of a Database Generation System (DBGS). Using this approach, entities would become database tables and entity entries would become database records. These systems allow great flexibility accessing data and designing complex queries, but require a great design and

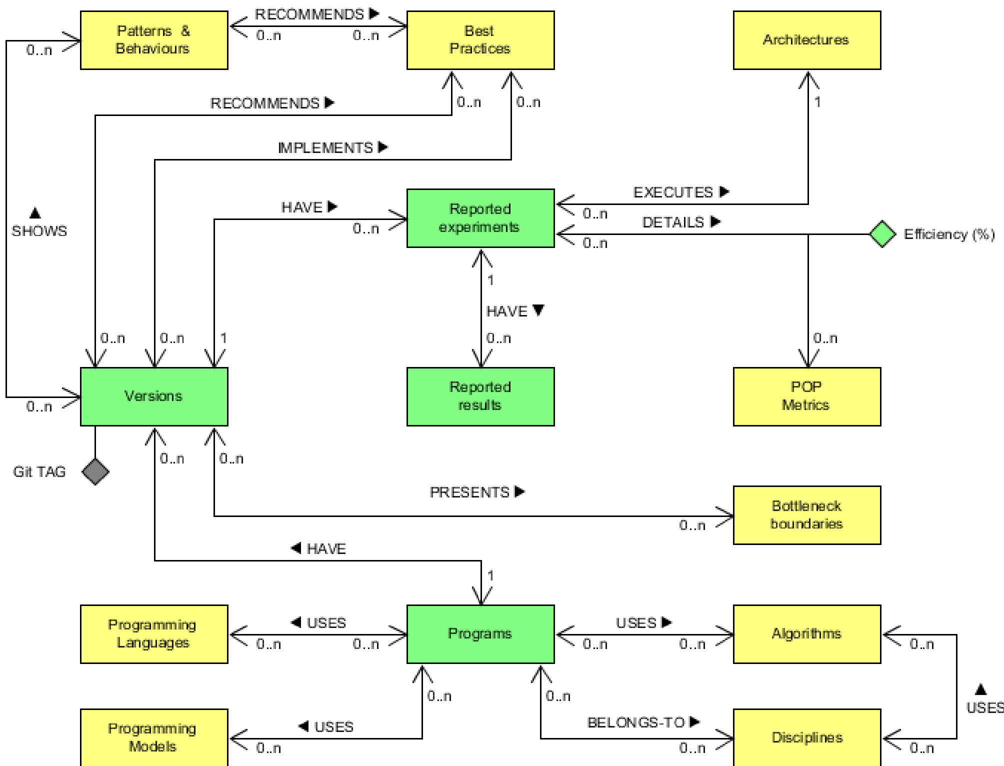

Figure 4: Entity Relationship Diagram (ERD) of the repository in the design phase.

implementation effort (database creation: tables, indexes, query design; interaction/interface: parametrizable queries, web forms, web based sql access, etc.) as well as a great maintenance effort (changes on tables, queries, web forms or related services accessing the system). These efforts exceed the expectations of this task and may lead to poor design decisions at early stages of the repository development.

Given one of the requirements appearing in the previous section was simplicity, due to the low expected kernel population at the end of the project, we also consider to include simplicity as one of the principles of design in the system. Therefore, we consider the use of a DBGS unnecessary, keeping in mind that it could be a complementary alternative once the current design and implementation have been validated employing the system through the life of the project.

As an alternative option to a DBGS we consider the use of the Jekyll web site generator. In particular, we are interested in the treatment of Jekyll's collections. A collection is a set of entries (pages) that can be traversed sequentially during the HTML code generation process. Then, an entity of our specification will become a Jekyll collection, and an entity's entry will become a Jekyll file that will finally produce an HTML page. In Jekyll jargon, an entity's entry would be a collection's item. Section 3.1 gives a more detailed description of Jekyll, and Section 4.2 will provide details about how the repository uses Jekyll collections. Markdown is the language used by this software to write the site content. It is described in Section 3.2 as a set of minimum guidelines to format collection's items. Relationships between elements of a given collection with elements of another one will be specified through the Jekyll frontmatters and materialized by means of HTML links between the resulting pages.

Thus, creation of a new entity in our system will consist in the creation of a new collection

in Jekyll; adding a new entry in an existing entity will consist in adding a new Markdown file in the corresponding Jekyll collection; and relating two entries is a matter of relating two file frontmatters (i.e., the “remote” frontmatter’s *identifier* variable with the corresponding “local” frontmatter’s *linker* variable). Jekyll will allow: 1) to iterate over all items in a repository collection, 2) to access their frontmatter’s variables and to compare them, and 3) once we detect a coincidence, to obtain the remote page URL and insert the link in the local page. Repository links will be automatically inserted using Jekyll’s layouts (i.e., template pages shared among all items within the same collection).

In order to simplify the structure and the process of adding new elements to the system, we distinguish between two fundamental components. The content of the **user interface** (see Figure 2 on page 9, labeled as *(1)* component) will live in a separate repository that will contain all the common collections used by the different programs (i.e., the Layout repository). Program **source codes** and **meta-data** (see Figure 2 on page 9, labeled as *(2.a)* and *(2.b)* components) will live in the corresponding program repositories. In the current design, program meta-data consist of the following collections: programs, versions, experiments, and results.

Figure 2.3 shows the repository entities in the design phase. We should highlight that the diagram shows in yellow those collections that will be stored in the Layout repository and in green those entities living in the program’s repositories. This decision is based on the convenience of having the information related with a specific program as close as possible to the code it describes. The main objective is minimizing inconsistency problems (data becomes outdated) between the code versions, its descriptions and the experiments performed with them.

Another aspect that is worth noting is the equivalence between a program version entity and the git tags. This *“external”* relationship allows to guarantee that version descriptions, experiments, and results have a related code regardless of the program history evolution. The content of a Jekyll version file must guarantee an univocal relationship with the git tag that represents its version. This guarantee must be ensured through naming conventions and/or the corresponding Jekyll frontmatter variable.

Figure 2.3 also labels the relationship between entities. Then, we can know that one element of an entity “has”, “uses”, “recommends”, etc. elements of another one. With respect to this type of relationships, it is also worth highlighting the double link between versions and best-practices in which in one case is “recommended”, while in the second is already “used”.

3 Supporting Tools and Technologies

3.1 Jekyll introduction

Jekyll [9] is a simple, powerful and extensible website generator. The system is able to transform a set of files written in Markdown language (see 3.2) into a set of HTML files properly linked to each other. The main difference between Jekyll and any other automatic site generators is the exclusively use of static features in the resulting web. By not producing any type of dynamic content, Jekyll does not require any additional software installed on the web server for its contents generation.

As a static web, it does not require any additional step for its visualization and pages are processed directly on the computer of the user accessing the page. This considerably limits the amount of languages that can be used (i.e., HTML, CSS and/or JavaScript). A dynamic web, on the contrary, is generated at the moment of its visualization, so it needs an interpreter of the chosen language that allows its generation (e.g., PHP). The treatment performed during the visualization of dynamically generated pages can significantly increase its response time.

Jekyll, as a site generator system, allows to completely decouple the site's contents from its final formatting. For this purpose, it uses a powerful mechanism based on layouts that allows to process and adapt multiple pages simultaneously. Jekyll reads files written in Markdown and interprets their content using the corresponding layout to produce HTML annotation. Thus, the variation of a single layout will affect all the entries based on itself.

Jekyll Pages and Posts Jekyll's most basic kind of content are pages and posts. *Pages* are documents that do not have a relationship with other pages and can live anywhere within the site's source directory. *Posts* are articles that are organized chronologically, usually in the form of blog entries or news

Jekyll pages and posts can be written in HTML or Markdown. Any file under the base source directory that Jekyll recognizes as an accepted filetype will be processed to generate a final HTML file. Either HTML or Markdown files may contain a YAML front matter set between triple-dashed symbols (i.e, "---"). This header's purpose, written in YAML syntax, is to set predefined or even custom variables that might be used later as Liquid tags in the page's content. A simple example of a front matter could be:

```
---
layout: program
title: Cholesky benchmark
language: Fortran
---
```

This YAML front matter is declaring in this case three different variables:

- predefined variable `layout` with value *"program"*
- predefined variable `title` with value *"The Cholesky benchmark"*
- custom variable `language` with value *"Fortran"*

Anything after the optional YAML front matter block is considered by Jekyll as page content, that is the actual content to be displayed when accessing the page. Whether it is HTML or Markdown, Liquid tags may be used to access the page's variables. This is an example contiuing the YAML front matter presented above:

```
---
layout: program
title: Cholesky benchmark
language: Fortran
---
<h1>{{page.title}}</h1>
<p>The {{page.title}} is written in the {{page.language}} language.</p>
```

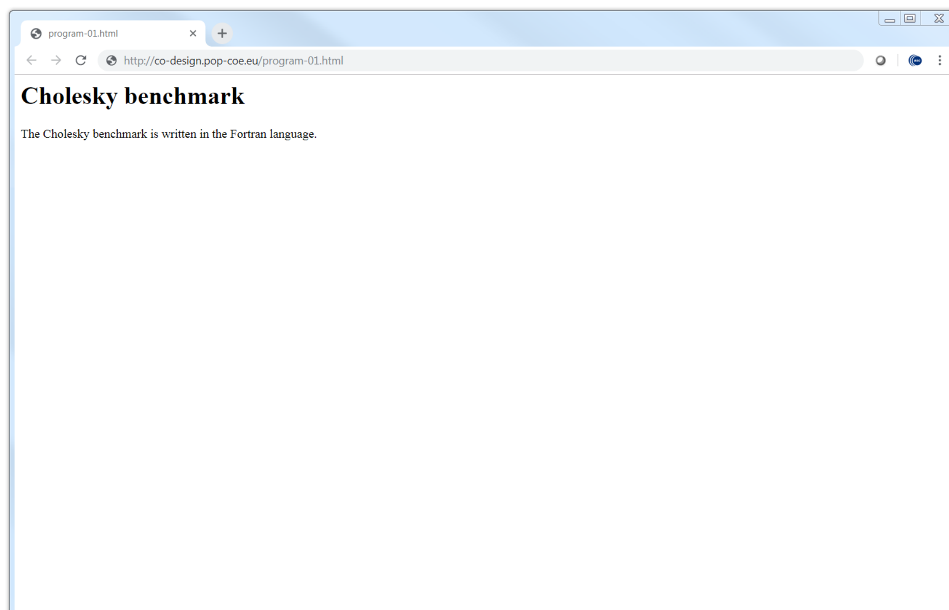

Figure 5: Generated HTML page using local Liquid tags substitutions.

The previous code, after being processed by Jekyll, will generate the HTML page shown in Figure 3.1. In this case Jekyll will substitute the `page.title` and `page.language` variables for the values they have assigned in the page front matter.

Jekyll Collections Additionally, *Collections* provide a mechanism to group related content that do not fit in the Pages and Posts categories explained at the beginning of this section. Collections are made of files placed in the same directory that might share some properties, like the layout, and additionally Jekyll is able to index each collection items into a data structure to allow listing or iterating over it. Collections usage is explained in section 4.2.

3.2 Markdown language introduction

Markdown [7] is a lightweight markup language designed to emphasize its readability in plain text. It was originally designed to write easy-to-read and easy-to-write documents in text format that will then be converted to HTML files.

The Markdown specification only covers some basic syntax and over the years several re-implementations have appeared always extending the basic specification with additional functionalities, such as tables, footnotes, etc. Jekyll's default Markdown renderer is `Kramdown`, wich supports some extra extended options but in this document will only cover the basic Markdown style needed to start contributing to the website content.

Headers Headers can be defined with the # symbol at the beggining of the line, and the number of symbols determines the header level.

```
# This is an H1

## This is an H2

### This is an H3
```

Alternatively, headers can also be specified with an underline-ish style:

```
This is an H1
=============

This is an H2
-------------
```

Paragraphs and line breaks A paragraph is formed by a sequence of non-blank lines. The paragraph's raw content is formed by concatenating the lines and removing initial and final whitespace. Multiple blank lines between paragraphs are ignored.

A line break is achieved with two or more spaces at the end of the line.

```
This is a paragraph
with no line breaks.

This is another paragraph␣␣
with a line break.
```

Emphasis and strong emphasis Emphasis, aka italics, is represented surrounding the text with *asterisks* or *underscores*. Strong emphasis, aka bold, is represented with *two asterisks* or *two underscores*. Both emphasis can be combined.

```
This is _italics_
This is **bold**
This is **_italics and bold_**
```

Code Inline code can be indicated with backtick quotes (`) around it. Blocks of multilple lines of code is fenced by lines with three backticks (```).

````
One can talk about the function `dgemm()` in a paragraph.

But code blocks are represented like this:
```
subroutine dgemm (...)
    ! subroutine description
end subroutine
```
````

Lists Markdown supports ordered (numbered) and unordered (bulleted) lists. Ordered lists can be used by starting each list item with a number followed by a period. Any number is valid, the numbered list does not need consecutive numbers.

```
1. Introduction
1. Development
1. Conclusions
```

Unordered lists can be generated by starting each list item with asterisks, pluses, or hyphens indistinctly.

```
* Red
* Blue
* Green
```

Links A link in Mardown, as in HTML, is formed by link text (the text that is visible), and the link destination (the link URL). Markdown supports two kinds of links. Inline links are the ones that the destination is given immediately after the link text. In reference links the destination is given elsewhere in the document, and the link itself only contains a reference to the real URL.

```
This is [an example](http://example.com/) inline link.

This [an example][id] reference-style link.
[id]: http://example.com/
```

Images Markdown uses an image syntax that is intended to resemble the syntax for links. It allows also inline and reference style images in the same way as links. An image in Markdown is formed by an alt text, also known as image description, and the image URL. All other rules are the same as for the link, except that an image starts with an exclamation mark (!).

```
![Alt text](/path/to/img.jpg)

![Alt text][id]
[id]: /path/to/img.jpg
```

3.3 GitLab introduction

GitLab [6] is a web-based tool that provides an entire software development lifecycle including project planning, source code management, wiki, issue tracking and CI/CD pipeline features. It has an open source version called GitLab Community Edition, with an MIT Expat license, which can be hosted on a private server.

In GitLab, users can create projects for hosting their codes. The projects can be created under a personal namespace or a group path indistinctly as long as the user has the appropriate permissions. Each project contains many DevOps tools needed to manage an application, such as:

- *Repositories:* As part of the source code management, GitLab hosts the application code using the Git version control system. Its interface is highly integrated with all Git features and fully supports filetree views, file editing, branch and tag management and history graphs.
- *Issue tracker:* GitLab implements a system where the user can create an issue and fill the appropriate fields to keep track of any matter related to the project. Generally, the issue may serve to report a program bug or to propose a discussion. Depending on the type of issue, it may have an assignee, a due date, a set of tags, a milestone, etc. in order to help managing a project team.
- *Merge Requests:* Depending on the contribution guidelines or the membership policy for a project, a user may not be able to contribute directly to the code. Merge requests is a mechanism to propose changes to the code that the maintainers will then be able to review and decide whether to accept them.

- *Wiki:* The documentation of the project can be published in the Wiki section. The Wiki is built on top of another Git repository, but it can also be edited from the same web interface. Wikis support a variety of common formats like Markdown, RDoc or AsciiDoc.
- *CI/CD:* GitLab's built-in Continuous Integration and Deployment. It enables the automatic code testing and deployment of the application.

GitLab Continuous Methodologies GitLab provides a built-in tool for software development using continuous methodologies. These methodologies are based on automating the execution of scripts to build, test and deploy each time a code change is pushed to the project repository. These are Continuous Integration (CI) and Continuous Deployment (CD).

Continuous Integration is the methodology to automatically test the source code of a project for every change submitted to the version control repository. The script for Continuous Integration must generally build and test the application to ensure that the introduced changes pass all tests and guidelines established for the application.

Continuous Deployment is a step beyond Continuous Integration. The application is also built and tested at every code change submitted to the code, but additionaly it is also deployed automatically.

The scripts are part of the source code repository. Each time that some change is pushed to the code, the scripts are automatically executed by a daemon process called GitLab Runner. This daemon process may have been configured to execute the scripts locally as a shell session, although we do recommend to configure it using a Docker executor. There are two main reasons; the first one is that the script will be executed in an isolated environment and, thus, there is an extra layer of security that prevents any malicious script to interfere in the server where the daemon is running; and the second reason is that the script may specify the Docker base image to use for the CI scripts, this way the script will run under a known environment with all the tools needed to execute it.

The GitLab CI/CD steps are described in a YAML file named `.gitlab-ci.yml` located in the root path of the source code repository. This file may contain the definition of the required steps, dependencies or commands to run.

The scripts are grouped into jobs, and together they compose a pipeline. A minimalist example of `.gitlab-ci.yml` file could contain:

```
build:
  stage: build
  script:
    - make
```

The pipeline is executed on every push for every branch if the file is present, but it can also be triggered manually using the GitLab web interface, or through an HTTP request to the GitLab API if the project is configured to do so. If this option is enabled, GitLab generates a secret token which is included in the URL to allow triggering the pipeline without login credentials.

GitLab also implements webhooks to trigger other pipelines or other external CI services if needed. A webhook is an action that triggers an HTTP request to some URL provided by the user. The action that triggers the webhook can be selected among different predefined events, such as push events, merge request events, issues, etc.

3.4 Software management

The technologies and tools explained in this section serve the purpose of providing the mechanisms for publishing the POP repository. The software listed in this section has been installed on a dedicated server in order to offer a public access.

Docker Docker [1] is a software and a platform that allows to run isolated environments inside a host system. Docker consists of many components but the two key concepts used for this project are: *containers*, which are encapsulated environments that run applications, and *images*, which are read-only templates to build containers. Docker has been installed using the package manager and it does not need further configuration.

GitLab GitLab has been installed using a *Docker GitLab image* as described in its documentation [2]. Docker images run all the necessary services on a single container. This solution greatly simplifies the installation procedure as every depedency is already pre-installed in the Docker image. Furthermore, it grants another layer of security by keeping the container environment and internal services isolated from the rest of the services.

GitLab Runner The GitLab Runner has been installed as a pre-compiled binary following the instructions described in its documentation [3]. The GitLab Runner daemon has been registered into the GitLab instance to provide a Docker executor for the CI scripts of any source code repository hosted in the GitLab instance. This means that any CI script will be executed inside an isolated Docker container, based on a Docker image specified from the same script.

This method has numerous advantages, like the extra layer of security that the Docker container provides, or that the job software dependencies are described in the CI script and automatically downloaded before the CI script execution. For this very same reason, there is no need to install Jekyll on the server, since all the build scripts for the website generation will be executed inside a Docker container and, thus, the GitLab Runner will download the required images as needed.

NGINX NGINX [4] is an open source webserver. NGINX is widely used and it is usually available as pre-compiled package in the package manager of the most common GNU/Linux distributions, so it is easily installed. NGINX has been configured to serve two different virtual servers: `https://gitlab.pop-coe.eu` is the public URL where the POP GitLab instance listens, and it is available for all the POP partners once they request a user account. The other virtual server is configured with the public URL `https://kernels.pop-coe.eu`, and contains the static website built with Jekyll.

All the installed software will be maintained and updated regularly with security patches in order to keep the system as stable as possible and protected from software vulnerabilities.

The system is also protected against data loses by applying a daily backup policy. In case anything happens to the server, all data is backed up every night in another physical location so the whole system can be restored in a few hours.

4 Building the User Interface

During the phase of analysis of requirements, all partners agree that the repository should be available as an internet's web site and should provide a structured and straightforward way to navigate among its different elements. In the ERD presented in Section 2.3 we distinguish among three different elements: 1) entities, 2) entity's records, and 3) relationships between specific entity's records. In the final website design the entity's records will become individual web pages describing such concept. The entities will become logical groups of entity's records (of the same type). The relationships between specific entity's records will become HTML hyperlinks relating both items.

At the end of the design phase, we also presented some clues about the recommended software to produce such web site. Jekyll is an ideal tool to define HTML patterns that should be applied over a set of content-based files annotated with a light-weight markup language like Markdown. Taking into account the Jekyll introduction, in Section 3.1, we can infer that an entity record will become just a Markdown file, placed in the right directory, that will produce an individual web page (with its own linkable URL, which will allow expressing the relationships among two different entity records). We can also see that an entity can easily be transformed into a Jekyll collection (allowing to group entity's records of the same type and iterate over all them).

We distinguish among two different types of Jekyll collections:

- Program's collections are related to a specific individual program. By design principle, they will live together with the code they are related with.
- Shared collections are, as its own name says, shared among all the programs in the repository. This is, a collection's item in a shared collection could relate with different programs.

In this Chapter, we will focus on shared collections. Kernel's programmers could leverage these common entities by relating them with their own program's collections (e.g., relate a program's description with a given algorithm). We will develop program's collections in Chapter 5.

In the following sections, we will define the mechanisms to convert these shared collection's items into a set of web pages interrelated through HTML hyperlinks. First, in Section 4.1 we will define the structure of the layout site. Second, in Section 4.2 we will establish some guidelines to transform a repository's entity to a Jekyll's collection. Finally, in Section 4.3 and its internal subsections we will specify the particularities of each collection, including how to generate a navigable list of collection's items, and how to present each item interrelated with others repository's collections using templates (i.e., layouts in Jekyll jargon).

4.1 User interface structure

The design of the layout web site should allow a straightforward way to navigate through all the included repository's collections and items. Figure 4.1 shows a common web site structure based on three main components: header, contents, and footer.

The web's header contains the site's main title plus the navigation bar. This navigation bar contains an entry for the home page, an entry for each shared collection (i.e., algorithms, disciplines, languages, etc.), and an entry for all included programs (listing them). The rest of the programs collection will be navigable only through other pages: 1) program's versions will be navigable from programs, 2) experiments will be navigabre from versions, and 3) results will be navigable from experiments.

The central component of the web site shows the actual contents of the current selected option. Usually contents will be splitted in two different levels: 1) a list of collection's items

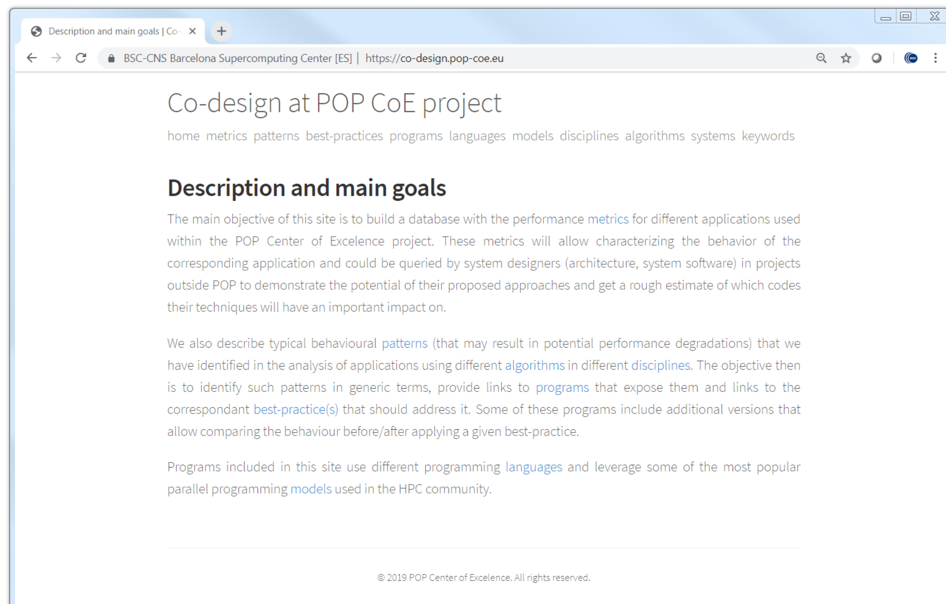

Figure 6: POP Repository interface: main site structure.

(e.g., the list of algorithms); and 2) an individual entry of a collection's items (e.g., the description of an algorithm). In the following sections we will define the templates for these two levels per collection basis.

The footer will contain the copyright information of the POP Kernel Repository and any other necessary warnings (e.g., privacy policies with respect cookies).

4.2 From repository entities to Jekyll collections

Adding a new collection in the repository will require different steps:

- Register a new collection in the _config.yml file.
- Create the collection directory to include collection's items.
- Create the corresponding collection layout file (or use an existent one).
- Create the corresponding directory to index collection's items (optional).

Registering a new collection will require to add a new entry in the `collections` section of the `_config.yml` file. A collection could have several attributes specified by means of variables. The `output` attribute will determine if Jekyll should create an individual page for each item in the collection or not. Individual pages have, by default, an URL composed by the site baseurl + collection's name + collection file name. If we want to change this default behaviour we will need to include the `permalink` attribute to specificy a different one.

In the following example we will include program, pattern and best-practice collections into the repository structure. We will assume that patterns and best-practices are directly related by a relationship defined by *"A best-practice is recomended to address a given pattern"*.

Adding these three collections will require to add three entries in the collections section. These three collections must generate output pages (i.e.,`output` attribute equals `true`) for all their collection's items. In addition, the program collection will employ a different url format using the item's name as the directory name, rather than as the file name. In the `permalink`

attribute we will specify the collection's name using the `:collection` variable, then the item's name using the `:name` variable, and finally we will use the literal `'index'` plus the output extension defined by the `:output_ext` variable. We use the URL *slash* (/) symbol as separator.

> Using the previous definition: if we have an item called `'cholesky'` within the `'programs'` collection, the resulting permanent link will be the *base url* plus `'/programs/cholesky/index.html'` instead of the default collection file-naming convention, which is `'/programs/cholesky.html'`.

The following listing is the result of adding these three collections and their respective attributes in the `_config.yml` file:

```
# List of Collections
collections:
   programs:
     output: true
     permalink: /:collection/:name/index:output_ext
   patterns:
     output: true
   best-practices:
     output: true
```

In order to create the collection directories we should name them as their respective entries in the configuration file adding an underscore symbol (i.e.,`_`) preffixing the name of the collection. In our case we will create `_programs`, `_patterns`, and `_best-practices` directories in the Jekyll's project root directory. These three directories will contain all the corresponding collection's items we want to register in the system within these three categories.

Collection's items must fullfil with all the restrictions specified in the corresponding collection's section (see Sections 4.3 and 5.2 for further details). As an example, we can include the following best-practice collection's item within the `_best-practices` collection:

```
---
layout: best-practices
title: Use Dynamic Load Balance tool
keyword: dlb
patterns: [load-imbalance]
---
Dynamic Load Balance (DLB) is a tool which mitigate load imbalance...
```

If the `output` collection variable is set to true, for each collection's item encountered in the collection directory, Jekyll will generate a new HTML file following the layout pattern specified in the frontmatter (in the previous example it is `best-practices`). A good practice will consist on using a different layout for each collection allowing to change its behaviour individually.

Following the previous example, we will create a new file named `best-practices.html` in the Jekyll's `_layouts` directory:

```
---
layout: default
---
{{ content }}

The following patterns recommend the use of this best-practice:
<ul>
{% for patterns in site.patterns %}
   {% if page.patterns contains pattern.keyword %}
      <li><a href="{{␣pattern.url␣|␣prepend:␣site.baseurl␣}}">
          <span class="pattern">{{ pattern.title }}</span></a>
      </li>
```

```
    {% endif %}
{% endfor %}
</ul>
```

A layout could also be based on a different layout, defining a hierarchical structure of layouts. In the previous example the `best-practice` layout is based on the `default` layout which in turn could also be based in a third layout structure. The idea is to include common elements in higher level layouts and specific details in the lower level ones. A common element is the title. All the pages will contain a title, so the title's formatting will be defined in a higher level layout (i.e., `default.html`). Specific elements are those exclusively related with a set of pages. For instace, only best-practice collection's items will include a list of patterns which recommend them. Then, the list of patterns formatting will be defined in a lower level layout (i.e., `best-practices.html`, as in the previous example).

Combining both files, description and layout, Jekyll will include the best-practice's content/description (i.e.,*"Dynamic Load Balance (DLB) is a tool which mitigate load imabalance..."* when substituting the `{{content}}` placeholder of the layout file. In addition, Jekyll will include the list of patterns which recommend the use of this best practice: for each item in the `patterns` collection, it will only print these ones which `keyword` is contained in the current page *patterns* variable. In addition, for each printed element, Jekyll will create a link to the corresponding pattern's main page.

Once we have registered the collection, we have populated the collection's directory and we have created the collection's item layout, we can create an index page for this collection and its correspondant entry in the main menu. To register a new menu's option we should include a new entry in the *nav* configuration variable in the `_config.yml` file and define its URL attribute.

```
nav:
  - name:          "Home"
    url:           "/index.html"
  - name:          "Patterns"
    url:           "/patterns/index.html"
  - name:          "Best-practices"
    url:           "/best-practices/index.html"
  - name:          "Programs"
    url:           "/programs/index.html"
```

After declaring the new navigation bar entries, we should define the HMTL templates for these new options. This template file will be the landing page when the user click on the navigation bar's option. In the previous example we should define the `index.html` file in `patterns`, `best-practices` and `programs` directories. This file will show a navigable list of the collection (allowing the site to access them). The following listing produces a navigable list of patterns:

```
---
layout: default
title: List of patterns
---
{% for pattern in site.patterns %}
    <span class="section">
        <a href="{{ pattern.url | prepend: site.baseurl }}">
            <span class="title">{{pattern.title}}</span>
        </a>
        <span class="excerpt">{{ pattern.excerpt }}</span>
        <span class="text">List of recommended best-practices:</span>
        {% for bp in site.best-practices %}
            {% if bp.patterns contains pattern.keyword %}
            <a href="{{ bp.url | prepend: site.baseurl }}">
            <span class="bp">{{ bp.title }}</span>
            </a>
```

```
        {{site.LS}}
        {% endif %}
      {% endfor %}
   </span>
{% endfor %}
```

The listing page is based in the default layout (it will just apply the site common style options), and it will show the title variable as the header of the content. The description contains the iterative construct over the pattern collection and for each element the listing will show: 1) a navigable link to the collection item page (using pattern's title as description), 2) an excerpt of the pattern (i.e., the introductory paragraph), and 3) a navigable compact list of related best practices.

4.3 List of user interface collections

Following sections will describe each shared collection in the POP repository. Each one will contain the relationships with other repository's collections, a reference Markdown file used to define a new collection's item, the conceptual description of the collection, and the content's description of related web pages: list of items template and individual item template.

4.3.1 Metrics (and efficiencies)

The metrics collection is directly related to the experiments collection. The *metric to experiment* relationship cardinality is defined as 0..N to 0..N (i.e., one metric appears in zero or more experiments, and one experiment can report efficiencies to zero or more metrics. This relationship is only defined in the experiment instance (i.e., a metric instance will not include a list of experiments with reported efficiency). For each metric to experiment relation there is an associated numerical value (efficiency) that can be interpreted as "this is the efficiency ratio for a specific metric in the current experiment".

The following code shows the frontmatter structure of a metric entry:

```
---
layout: metrics
title: <Metric's title>
keyword: <metric-id>
order: <order-val>
---
<Metric's description>
```

As metric's elements are ordered according with the POP methodology hiearchy, the Jekyll collection must be processed by a sorting criteria. The "order" variable in the frontmatter will help in creating a sorted list of metrics.

List of metric's template The metric listing page is the default page when selecting the menu's option *metrics*. It will include a list of all the metrics in the repository (ordered according to the $order - val$, from lowest to highest). For each metric, the page will include: 1) the title, 2) the excerpt, and 3) a list of programs with any experiment reporting an efficiency value below the site's default threshold (800 *per thousand*).

A description of the metric concept, within the context of the repository, precedes the list:

> POP has defined a methodology for analysis of parallel codes to provide a quantitative way of measuring relative impact of the different factors inherent in parallelisation. The methodology uses a hierarchy of metrics each one reflecting a common

cause of inefficiency in parallel programs. These metrics then allow comparison of parallel performance (e.g., over a range of thread/process counts, across different machines, or at different stages of optimisation and tuning) to identify which characteristics of the code contribute to inefficiency.

The metrics are then calculated as efficiencies between 0 and 1000 (*per thousand* values), with higher numbers being better. In general, we regard efficiencies above 800 as acceptable, whereas lower values indicate performance issues that need to be explored in detail.

Metric's template Metric's pages describe a single element in the metric's collection. They are indexed through the list of metrics page, and they contain its description followed by the list of programs with any efficiency value below the site's default threshold (800 *per thousand*).

4.3.2 Patterns and behaviors

The patterns collection is directly related to versions and best-practices collections. Pattern to version relationship cardinality is 0..N to 0..N (i.e., a pattern can appear in zero or more program's versions, and a program's version can report zero or more code's patterns). Pattern to best-practice relationship cardinality is 0..N to 0..N (i.e., a pattern can recommend zero or more best-practices, and a best-practice can *solve* zero or more code's patterns). Pattern's reference (if any) will appear in program's version items and in best-practice items. No relationship variable will appear in the pattern's frontmatter section.

The following code shows the frontmatter structure of a pattern entry:

```
---
layout: patterns
title: <Pattern's title>
keyword: <pattern-id>
---
<Pattern's description>
```

List of pattern's template The pattern listing page is the default page when selecting the menu's option *patterns*. It will include a list of all the patterns and behaviours stored in the repository. For each pattern, the list will include: 1) the title, 2) the excerpt, and 3) a list of best-practices related to this specific pattern.

A description of the pattern concept, within the context of the repository, precedes the list:

> In this section we describe typical behavioural patterns that we have identified in the analysis of applications in different domains. By behavioural pattern we understand typical sequences of operations, memory accesses, communications and or synchronizations that perform general algorithmic steps appearing in many different programs. These patterns may result in potential performance degradations.
>
> The objective is to identify such patterns in generic terms, provide links to applications that expose them and links to best practices how we consider they should be addressed. Although we tried to group the different patterns by relationship between the issues, the list of patterns is somewhat unstructured. We suggest looking at the global list and its introductory description to identify the topics that may be relevant for your co-design target.

Pattern's template Pattern's pages describe a single element in the pattern's collection. They are indexed through the pattern listing page and they contain their description followed by the list of related best-practices.

4.3.3 Best practices

The best-practice collection is directly related to versions collection twice (*recommended* and *implemented*) and it is also related to the patterns collection. Best-practice to version relationship cardinality is 0..N to 0..N for both cases, recommended and implemented, (i.e., a best-practice {can be used | is already used} in zero or more program's versions, and a program's version {can use | already uses} zero or more best-practices). Best-practice to pattern relationship cardinality is 0..N to 0..N (i.e., a best-practice is recommended in zero or more patterns, and a pattern can recommend zero or more best-practices). Pattern's references (if any) will appear in correspondant best-practice. Best-practice's references will appear (if any) in the program's version collection item.

The following code shows the frontmatter structure of a best-practice entry:

```
---
layout: best-practices
title: <Best-practice's title>
keyword: <best-practice-id>
patterns: [<pattern-id1>, <pattern-id2>,...]
---
<Best-practice's description>
```

List of best-practice's template The best-practice listing page is the default page when selecting the menu's option $best - practices$. It will include a list of all the best-practices stored in the repository. For each one, the page will include: 1) the title, 2) the excerpt, and 3) a list of programs including/recommending the best-practice.

A description of the best-practice concept, within the context of the repository, precedes the list:

> These recommendations will address the holistic space form application refactoring to using or proposing new features in the system software or hardware architecture. Our holistic vision of co-design aims at addressing the issues at the level (or split between levels) that maximizes the gain at the minimal cost.
>
> The following entries will suggest basic directions and provide links to code or raw data that can be used to further explore the options and quantify the benefits.

Best-practice's template Best-practice's pages describe a single element in the best-practice's collection. They are indexed through the best-practice listing page and they contain the item's description followed by a list of related patterns and a list of programs using or recommending its use.

4.3.4 Programming models

The programming models collection is related directly to the programs collection. The programming model to program relationship cardinality is 0..N to 0..N (i.e., a programming model is used in zero or more programs, and a program can use zero or more programming models). The relation is defined only in program instances.

The following code shows the frontmatter structure of a programming model entry:

```
---
layout: models
title: <Programming model's title>
keyword: <model-id>
---
<Programming model's description>
```

List of programming model's template The programming model listing page is the default page when selecting the menu's option *models*. It will include a list of all the programming models registered in the repository. For each entry, the list will include: 1) the title, 2) the excerpt, and 3) a list of programs using this programming model.

A description of the programming model concept, within the context of the repository, precedes the list:

> In this page we gather a set of programming models usually employed in parallel application's development. By programming model we mean a set of services providing support to high-level programming languages and offering and abstraction of the underlying architecture or hardware resources.
>
> Programming models can offer its support by means of certain library calls, language extensions, compiler/pre-processor directives, etc.

Programming model's template Programming model's pages describe a single element in the programming model's collection. They are indexed through the programming model listing page and they contain its description followed by the list of programs using this model.

4.3.5 Base languages

The programming languages collection is related directly to programs collection. The language to program relationship cardinality is 0..N to 0..N (i.e., a programming language is used in zero or more programs, and a program uses zero or more programming models). The relation is only defined in the program instance.

The following code shows the frontmatter structure of a programming language entry:

```
---
layout: languages
title: <Programming language's name>
keyword: <language-id>
---
<Programming language's description>
```

List of programming language's template The programming language listing page is the default page when selecting the menu's option *language*. It will include a list of all the programming languages registered in the repository. For each language, the list will include: 1) the title, 2) the excerpt, and 3) a list of programs using this language.

A description of the programming language concept, within the context of the repository, precedes the list:

> In this page we list the set of programming languages used to implement the POP Repository's programs. The main objective is to allow repository's users to easily navigate among the set of programs included in the repository according with the used programming language criteria.

Programming language's template Programming language's pages describe a single element in the language's collection. They are indexed through the programming language listing page and they contain its description followed by the list of related programs.

4.3.6 Disciplines

The disciplines collection is related directly to programs collection. The discipline to program relationship cardinality is 0..N to 0..N (i.e., a discipline is related with zero or more programs, and a program is related with zero or more disciplines). The relationship is only defined in the program instance.

The following code shows the frontmatter structure of a discipline entry:

```
---
layout: disciplines
title: <Discipline's name>
keyword: <discipline-id>
---
<Discipline's description>
```

List of discipline's template The discipline listing page is the default page when selecting the menu's option *discipline*. It will include a list of all the disciplines registered in the repository. For each one, the list will include: 1) the title, 2) the excerpt, and 3) a list of programs classified under this grouping critera.

A description of the programming language concept, within the context of the repository, precedes the list:

> This page gathers the different disciplines registered in the POP repository. By discipline we mean a branch of knowledge within the High-Performance Computing world, and typically group a set of programs with common applications. The main goal is to allow repository's users to easily navigate among the set of programs included in the repository according to this grouping criteria.

Discipline's template Discipline's pages describe a single element in the discipline's collection. They are indexed through the discipline listing page and they contain its description followed by the list of related programs.

4.3.7 Algorithms

The algorithms collection is directly related to programs and disciplines collections. The algorithm to program relationship cardinality is 0..N to 0..N (i.e., an algorithm is used in zero or more programs, and a program contains zero or more algorithms). The algorithm to program relationship is only defined in the program instance. The algorithm to discipline relationship cardinality is 0..N to 0..N (i.e., an algorithms is used in zero or more disciplines, and a discipline can use zero or more algorithms). The algorithm to discipline relationship is only defined in the algorithm instance.

The following code shows the frontmatter structure of an algorithm entry:

```
---
layout: algorithms
title: <Algorithm's name>
keyword: <algorithm-id>
disciplines: [<discipline-id1>, <discipline-id2>,...]
---
<Algorithm's description>
```

List of algorithm's template The algorithm listing page is the default page when selecting the menu's option *algorithms*. It will include a list of all the named algorithms registered in the repository. For each algorithm, the list will include: 1) the title, 2) the excerpt, and 3) a list of programs using this algorithm.

A description of the algorithm concept, within the context of the repository, precedes the list:

> In this page we list the set of algorithms included in the implementation of some of the POP Repository's programs. The main objective is to allow repository's users to easily navigate among the set of programs included in the repository according with the implemented algorithm(s) criteria.

Algorithm's template Algorithm's pages describe a single element in the algorithm's collection. They are indexed through the algorihtm listing page and they contain its description followed by a list of related programs.

4.3.8 Bottleneck boundaries

The bottleneck boundaries collection is directly related to versions collection. The bottleneck to version relationship cardinality is 0..N to 0..N (i.e., a bottleneck boundary can be reported, as a performance issue, in zero or more program's versions, and a program's version can report zero or more bottleneck boundaries). The relationship is only defined in the version instance.

The following code shows the frontmatter structure of an algorithm entry:

```
---
layout: bottlenecks
title: <Bottleneck boundary's name>
keyword: <bottleneck-id>
---
<Bottleneck boundary's description>
```

List of bottleneck boundary's template The bottleneck listing page is the default page when selecting the menu's option *bottlenecks*. It will include a list of all the bottleneck boundaries registered as common source of problems in the repository. For each bottleneck, the listing will include: 1) the title, 2) the excerpt, and 3) a list of program's versions that have already reported this problem.

A description of the bottleneck concept, within the context of the repository, precedes the list:

> In this page we list the set of bottleneck boundaries registered in the repository. The main objective is to allow repository's users to easily navigate among the set of programs included in the repository according with a reported bottleneck boundary criteria.

Bottleneck boundary's template Bottleneck's pages describe a single element in the bottleneck boundary's collection. They are indexed through the bottleneck listing page and they contain its description followed by a list of related program's verions.

4.3.9 Systems

The systems collection is related directly to experiments collection. The system to experiment relationship cardinality is 0..1 to 0..N (i.e., a system is used in zero or more reported experiments, and a experiment uses zero or one system). Experiments with no related system means the experiment was carried out in no registered system within the POP repository. The relationship is only defined in the experiment instance.

The following code shows the frontmatter structure of a system entry:

```
---
layout: systems
title: <System's name>
keyword: <system-id>
---
<System's description>
```

List of system's template The system listing page is the default page when selecting the menu's option *systems*. It will include a list of all the computer systems registered in the repository. For each entry, the list will include: 1) the title, 2) the excerpt, and 3) a list of experiments carried out in this system. As the number of experiments may grown during the project lifetime, it will be interesting to consider limiting the amount of experiments linked with a computer system.

A description of the system concept, within the context of the repository, precedes the list:

> In this page we list the set of computer systems used to execute some of the POP Repository's experiments. The main objective is to allow the repository's users to easily navigate among the set of experiments carried out in a given computer system, and also have a common and formal description of the characteristics of a given system which is commonly used when reporting POP's experiments.

System's template Computer system's pages describe a single element in the computer system's collection. They are indexed through the computer system listing page and they contain its description followed by the list of related experiments.

4.4 Contribution guidelines

In this section we will explain the protocols to mantain and extend the POP Repository infrastructure. Firstly we will focus on contributions on the Layout repository. Layout contributions could be done directly by modifying the source code repository or by means of GitLab issues and relying on layout maintainers to apply these changes in the main repository. Secondly, we will mention some guidelines about how contributions should be done in the program repositories.

Layout direct contribution guidelines These guidelines are recommended for all kind of modifications in the Layout repository. In order to directly contribute in the layout repository contributors will need to follow the next steps:

1. *Fork the main layout's repository into your personal GitLab namespace.* You should open the main layout's repository (i.e., *Documents* > *co−design* > *layout* at https://gitlab.pop-coe.eu/documents/co-design/layout) and click the "Fork" button. Once you click on that button you will be asked to select where you want to fork the project. You should select your personal namespace, usually your name, among the available option(s).

2. Once you have forked the main repository you will have an exact copy of the repository in your personal namespace. You should clone and *work with that copy of the repository*: git -add, -commit, -push, -pull, etc. Before uploading something to your own repository make sure the site compiles (see "Building the Layout" section in this document) and shows all your changes as they are expected:

```
git clone https://gitlab.pop-coe.eu/jsmith/layout.git
<edits/updates here>
jekyll build
<verify jekyll output>
git commit -m ''Your commit messages''
git push
```

   Once you consider your updates are ready, you should create a new merge request (*see step 4*)

3. *[Optional]*: Sometimes it is useful to merge other changes happened in the main layout's repository into your personal layout's repository while you were working on your updates. In order to do that, you should add the main layout's repository as a new remote into your working repository and *fetch & merge changes from main repository to your working copy*. Once you have synchronized both histories you should upload changes to your personal layout's repository and create a merge request (*see step 4*).

```
git remote add upstream https://gitlab.pop-coe.eu/documents/co-design/layout.git
git pull upstream master --no-ff
jekyll build
<check jekyll output>
git push origin master
```

4. Once you have updated your personal layout's repository you will want to *merge these changes to the main layout's repository*. In order to create a merge request you need to click the "Merge Requests" option in your left-hand GitLab menu from your personal layout's repository then, click the green "New Merge Request" button and follow the instruction about creating the merge request (filling the 2 form-pages):

   - In the first one, select the source branch and target branch (a branch is defined by the name of a repository plus the name of the branch in this specific repository); then press **Compare branches and continue**.
   - In the second form page, write a meaningful *title* and *description* fields, verify branch's names and the list of commits which will be included in this merge (at the end of the page); then press **Submit merge request**.

Layout alternative contribution guidelines (by means of GitLab issues). These guidelines are recommended for *"adding new content"* updates (i.e., they are not recomended to change site structure or to change specific template pages). In order to contribute with new contents in the layout repository contributors will need to create a new issue and use its main description or a comment section to write the corresponding Markdown text. The issue should include all the Jekyll's text that will become part of the repository as a new file (i.e., the front matter plus the item's description).

5 Building the Program Repositories

As described in Chapter 2.3, we will use GitLab as the control version system for the implementation of the code repositories. Each program will have their own entry in the repository and will implement as many versions as needed. The boundary between a new version and a new entry in the repository will depend on programmers criteria. As a rule of thumb, a GitLab entry will use versions for incremental updates, improvements or approaches, while a refactor will consider a new entry in GitLab.

We will identify repository program's version using git tags. Naming conventions should be adapted to each particular case, but we will try (if reasonable) to keep the original version as part of the POP Version (e.g., A POP version of LULESH based on LULESH 2.0 could be LULESH 2.0-pop-1.0).

POP kernel repositories will be located on a GitLab group in order to allow a better permission management and to restrict the use of specific contigous methodologies (see Section 3.3 as reference to this mechanism).

5.1 Kernel repository structure

Individual kernel's repositories must fulfil a series of requirements that guarantee a minimum of homogeneity among all of them. These requirements are related with how to structure the files and directories, how to build and execute the kernel and some guidelines about how to document it.

All kernels must have a README.md file in the root directory that briefly introduces the program and summarizes its main characteristics. This file will also contain the following sections (or the correspondant links to separate files for):

- Compilation guidelines: software requirements and instructions to build the program. It will include any configuration parameter to enable/disable program's features and the description of any conditional compilation the kernel could implement.
- Execution guidelines: list of command line options, environment variables, input files availabilty and any other detail needed to run the program.
- Developer guidelines: specific directory structure (if needed), file and algorithm descriptions, parallelisation description, or any other data related with coding.

The README.md file should also allow to browse through any other documentation file included in the repository. The location of all these files is determined by the recommended directory structure (see below).

Although kernel distribution is not forced to use any specific automatic building mechanism, it is highly recommeded to use a simple way to allow users easily create the correspondant binaries. Kernel programmers may decide to use a compilation script, a naked Makefile, or any CMAKE or Autotools support to compile their programs, as long as they provide a clear recipe in the compilation guidelines section (or file). Building default options should be completely agnostic of the computer used (directories, compiler versions, programming model versions, etc.).

With respect to directory structure, programmers may decide how to organize source and generated files. It is highly recommended to locate all the documentation in a “doc” directory created in the repository's root. Program collection's items stored within the code repository

(i.e., programs, versions and experiments) must be located in the “pop” subdirectory within the “doc” directory.

Different program’s versions will be tagged accordingly using Git tags, and the repository collection will be aligned with them. Repository’s program version identifier will coincide with the Git tag name (allowing to directly download this version using appropiate URL pattern, by means of string manipulations).

Code diversion will be also tagged but it will also use a different Git branch in order to allow both version to progress independenly.

When multiple approaches coexist in the same program’s version the build or execution guidelines will clearly explain how to use any of them. If approaches could be decided at runtime, execution guidelines will define any parameter, configuration file option, environment variable, etc. required to use this approach. If approaches are decided at build time, compilation guidelines will clearly define compile option (e.g., used for conditional compilation), or source files (involved in a given approach compilation) used to produce final executables. For the purpose of clarity, different source files are recommended.

5.2 List of program collections

Following sections will describe the program collections included in the POP repository. Each unit will contain the description of the relationships with other repository collections (as well as the description of other jekyll variables used in the collection’s item frontmatter); a reference Markdown file, which can be used to register a new collection’s item; and the descriptions of the related web page’s contents (i.e., the list of items template, if applies, and the individual item template).

5.2.1 Programs

The programs collection is directly related with programming languages, programming models, algorithms, disciplines and program’s versions collections. The program-language relationship cardinality is 0..N to 0..N (i.e., a program is written in zero or more programming languages, and a programming language is used in zero or more programs). The program-model relationship cardinality is 0..N to 0..N (i.e., a programs uses zero or more programming models, and a programming model is used in zero or more programs). The program-algorithm relationship cardinality is 0..N to 0..N (i.e., a program includes zero or more algorithms, and an algorithm is used in zero or more programs). The program-discipline relationship cardinality is 0..N to 0..N (i.e., a program belongs to zero or more disciplines, and a discipline has zero or more programs). The program-version relationship cardinality is 1 to 0..N (i.e., a program has zero or more versions, but a version belogs to one, and only one, program). All relationships, but the program-version one, are defined in the program instance. The program-version relationship is defined in the version instance.

All version relationships are indirectly related with the program collection due the deterministic path between versions and programs.

The following code shows the frontmatter structure of a program entry:

```
---
layout: programs
title: <Program's name>
keyword: <program-id>
languages: [<language-id1>, <language-id2>,...]
models: [<model-id1>, <model-id2>,...]
algorithms: [<algorithm-id1>,<algorithm-id2>,...]
disciplines:  [<discipline-id1>,<discipline-id2>,...]
```

```
---
<Program's description>
```

List of program's template The program listing page is the default page when selecting the menu's option *programs*. It will include a list of all the programs registered in the repository. For each program entry in the list, the page will include: 1) the title, 2) the excerpt, and 3) a list of program's available version(s).

Program's template Program's pages describe a single element in the program's collection. They are indexed through the program listing page. A program's page contains a formatted header followed by the program's description. The formatted header contains: the program's name, a list of available version(s), links to the associated code repository (GitLab) plus direct links to the latest downloadable source code, and finally, links to related programming language(s), programming model(s), algorithm(s) and discipline(s). The formatted header allows to navigate to all repository's collections directly related with a program instance.

5.2.2 Versions

The version collection is directly related with programs, boundaries, patterns, best-practices and experiments collections. This collection have a composed identifier as the result of the combination of the program (i.e., program-id) and the keyword (i.e., version-id) frontmatter's variables. So, the keyword variable does not require to be universaly unique among all the versions of different programs but, the keyword variable must be unique among versions of the same program (i.e., with the same program-id). The version-program relationship cardinality is 0..N to 1 (i.e., a version belongs to one, and only one, program, but a program could have zero or more versions). The version-boundary relationship cardinality is 0..N to 0..N (i.e., a version might be related with zero or more bottleneck boundaries, and a bottleneck boundary could have associated zero or more versions). The version-pattern relationship cardinality is 0..N to 0..N (i.e., a version contains zero or more patterns, and a pattern could appear in zero or more versions). The version-best-practice relationship cardinalities are two-folded. On one hand we have the "recommends" relationship with cardinality 0..N to 0..N (i.e., a version recommends zero or more best-practices, and best-practice is recommended in zero or more versions). On the other hand we have the "implements" relationship with cardinality 0..N to 0..N (i.e., a version already implements zero or more best-practices, and a best-practice is implemented in zero or more versions). The version-experiment relationship cardinality is 1 to 0..N (i.e., one, and only one, version includes zero or more patterns). This relationship implies that we can determine which specific program's version has been involved in a given experiment.

The following code shows the frontmatter structure of a version entry:

```
---
layout: versions
title: <Versions's name>
program: <program-id>
keyword: <version-id>
boundaries: [<boundary-id1>, <boundary-id2>,...]
patterns: [<pattern-id1>, <pattern-id2>,...]
recommended-bp: [<bp-id1>,<bp-id2>,...]
implemented-bp: [<bp-id1>,<bp-id2>,...]
---
<Version's description>
```

The program's version collection does not have any explicit listing page (directly hanging from the web site menu), but program pages contain a list of registered program's versions (defined in its Jekyll's layout).

Version's template Version's pages describe a single element in the version's collection. They are indexed in their respective program's page. A version's page contains a formatted header followed by the version's description and a list of experiments carried out (and reported) with that specific program's versions. The formatted header contains: the program and version names, a link to the program's web page, direct links to the GitLab tagged downloadable source code, and finally, links to related bottleneck boundaries, pattern(s), and recommended/implemented best-practice(s). The formatted header allows to navigate to all repository's collections directly related with a version instance. After the version's description a list of experiments carried out with that specific program's version will allow to navigate through all this data.

5.2.3 Experiments

The experiments collection is directly related with program's versions, metrics, systems, and results collections. In addition, the collection is indirectly related with the programs collection due to the deterministic path between experiments and programs. Experiments have a composed identifier as the result of the combination of the program (i.e., program-id), the version (i.e., version-id), and the keyword (i.e., experiment-id) frontmatter's variables. So, the keyword variable does not require to be universaly unique among all the experiments of different program's versions but, the keyword variable must be unique among all experiments of the same program's versions (i.e., with the same program-id and version-id).

The experiment-version relationship cardinality is 0..N to 1 (i.e., an experiment belongs to one, and only one, version, and a version could report zero or more experiments). As the version-program relationship is also 0..N to 1 (a version belongs to one, and only one, program), the repository could navigate from an experiment to its related program (deterministic path). The experiment-metric relationship cardinality is 0..N to 0..N (i.e., an experiment can report efficiency values with respect to zero or more metrics, and a metric could appear in zero or more experiments). The experiment-sytem relationship cardinality is 0..N to 0..1 (i.e., an experiment is executed in zero or one system, where no associated system means the system is unknow or not regitered, and a system could report zero or more experiments, of the same or different program/version). The experiment-result relationship cardinality is 1 to 0..N (i.e., an experiment contains zero or more results, and a result entry is related with one, and only one, experiment).

The experiment entry will contain a list of metrics and the correspondant efficiency values for a given core count (i.e., *MCC* or *Maximum Core Count*). These two lists establish a one-to-one correspondence between the n-element in the metric list with a value stored as the n-element in the efficiency list. The relationship should be interpreted then as: "this is the efficiency ratio (per thousand value) for a specific metric in the current experiment". Finally, the version and system relations are defined in the experiment instance. The result relation is defined in the result instance.

The following code shows the frontmatter structure of an experiment entry:

```
---
layout: experiments
title: <experiment's title>
date: <date>
program: <program-id>
version: <version-id>
keyword: <experiment-id>
```

```
system: <system-id>
metrics: [<metric-id-1>, <metric-id-2> ...]
efficiencies: [<efficiency-val-1>, <efficiency-val-2>, ...]
mcc: <nn>
---
```

The experiment collection does not have any explicit listing page (directly hanging from the web site menu), but program's version pages contain a list of registered experiments (defined in its Jekyll's layout).

Description guidelines: Besides the relationship frontmatter's variables, the experiment entry will also contain the descriptive's variables: title (naming the experiment), date (when the experiment was carried out), and mcc (Maximum Core Count i.e., the maximum number of cores used in the experiment). The reported metrics indicate efficiencies using this number of cores.

The experiment's description (after the frontmatter), contains a free formatted text describing the results obtained while executing the experiment. It can contain highlighted remarks, conclusions, other non-formatted results, summary tables, figures, etc. which allow to illustrate the behaviour of the program when carrying out the experiment.

In order to guarantee a certain degree of reproducibility the description should also include a complete environment's description. This description includes all the software stack used during the program's compilation and link phases (compiler and linker versions plus their respective options), the software used during program's execution (supporting libraries, e.g., MKL version, cluster execution options, e.g., mpirun options), and any additional option used during the execution (e.g., environment variables, program's parameters, etc.). If the system is not specified in the frontmatter it will be required an overall system description including processor's family/-model, memory description, network architecture/driver, operating system version/distribution, etc. All in all, the underlying idea is to provide enough information about the experiment conditions which allows to reproduce the experiment in the future.

5.2.4 Results

The results collection is directly related with the experiments collection. In addition, it is indirectly related with version and program collections due to the deterministic path between results and programs. Results have a composed identifier as the result of the combination of the program (i.e., program-id), the version (i.e., version-id), the experiment (i.e., experiment-id) and the keyword (i.e., result-id) frontmatter's variables. So, the keyword variable does not require to be universaly unique among all the results in the collection, but the keyword variable must be unique among all the results of the same experiment (i.e., with the same program-id, version-id, and experiment-id).

The result-experiment relationship cardinality is 0..N to 1 (i.e., a result belongs to one, and only one, experiment, and an experiment could have zero or more results). As the experiment to version plus version to program relationships are also 0..N to 1 (an experiment belongs to one, and only one, version; and a version belongs to one, and only one, program), the repository could navigate from a result to its related program (deterministic path).

The following code shows the frontmatter structure of an result entry:

```
---
layout: results
formatting: <formatting-val>
title: <result's title>
date: <date>
```

```
program: <program-id>
version: <version-id>
experiment: <experiment-id>
keyword: <result-id>
---
‘‘‘
<Result’s CSV Content>
’’’
```

The results collection does not have any explicit listing page (directly hanging from the web site menu), but experiment pages contain a list of registered results defined in the Jekyll's layout.

Description guidelines: In the results collection, the formatting variable defined in the front-matter determines the structure of the *<Result's CSV Content>*. Therefore, the description must follow a *Comma Separated Values* (CSV) format in which the number columns and their semantics will fit the formatting variable value. The current supported format is:

- **model-factors**: number of cores, reported metric and reported efficiency.

5.3 Program organization guidelines

Each kernel repository will have a user with the maintainer *role permission* on the list of their repository members. The maintainer access level will be assigned by the GitLab administrator by request of the partner actually proposing the kernel. The repository maintainer is the final responsible to assign the developer role permission to other members of their respective institution and/or to request the creation of a new user to GitLab administrator.

The repository maintainer will have the following responsabilitites:

- Establish source code organisation in directories.
- Manage Git branches and tags. Coordinate tags with reported program's versions.
- Guarantee the proper documentation degree of the current repository (i.e., build, execute, and develop).
- Establish the repository contribution guidelines as part of the program documentation. Maintainers could adapt the layout contribution guidelines (see Section 4.4) to each individual kernel repository.
- Guarantee meta-data correctness with respect to all the collections associated with a program (see Section 5.2.1).

5.4 Website integration

Every time the git repositories are updated, GitLab automatically builds the Jekyll HTML static pages and are immediately served on the public URL.

As mentioned in the previous section, the layout site is an independent project that contains the basic structure of the website while each kernel will have its own entry in GitLab. This section describes the procedures to combine the data from all the repositories and how to deploy the website. The sequence of steps are depicted in Figure 5.4 and the following sections describe, step by step, each one of these phases.

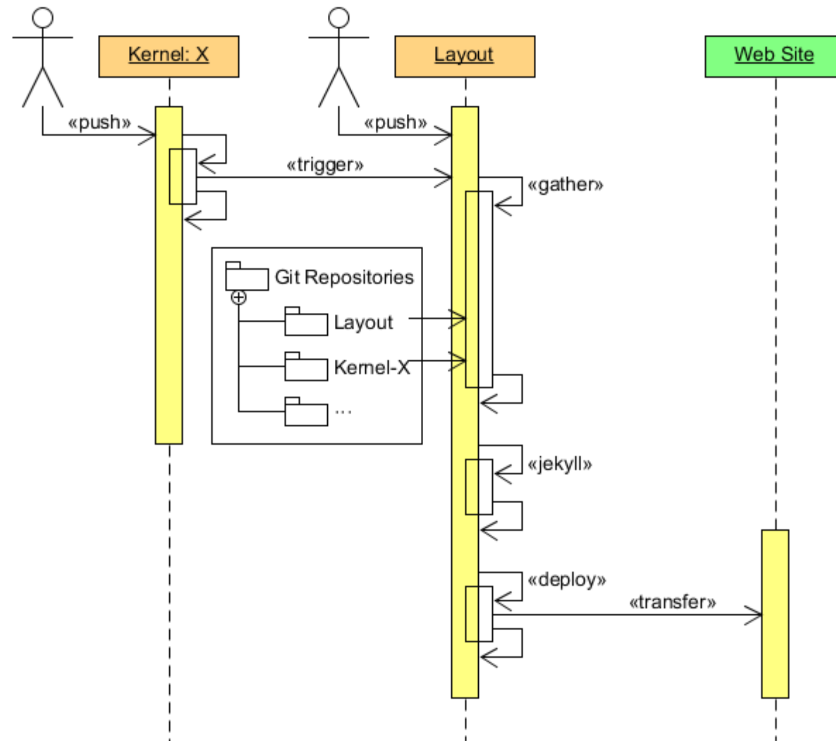

Figure 7: Repository automatic website generation: trigger, gather, build, and deploy.

Triggering the layout build GitLab's CI/CD infrastructure implements some features that allow the automatic deployment of a software project even if it is based in multiple components or has specific dependencies. We will use triggered pipelines and webhooks explained in Section 3.3.

The layout repository only contains the basic structure of the website and depends on the contents of the kernel repositories. Therefore, the CI scripts that run after each repository push are not enough to keep the website updated. These scripts need to be executed also when new commits are pushed to any kernel repository.

To achive that, the first step is to configure the layout project to accept pipeline trigger events through the GitLab API. After this configuration, GitLab provides a URL with a secret token to trigger the pipeline of that specific project.

The next step is to configure each of the kernel repositories to trigger a webhook on each push event. This means that any push on any kernel repository will trigger the pipeline of the layout repository and will update the website with the new content.

Gathering program's collections As mentioned, the layout repository depends on all of the kernel repositories to add content to the website. On each pipeline execution, the first job is to gather the contents of all the kernels.

This is done invoking the script `.gitlab-ci/gather_kernels.sh`, located in the same layout repository.

Using the GitLab API, this script performs an HTTP query to obtain all the public repositories inside the group `kernels`. This HTTP query returns the information in JSON format. This string is then parsed to obtain the name and the URL of each project needed to fill the contents of the website. After that, the script only needs to iterate all the parsed data of ker-

nel repositories, clone its contents and copy the necessary files into the accepted components: *programs, versions, experiments or results.*

Layout build and deployment After gathering all the kernels contents, the only steps left are to invoke `jekyll build` and to install the built files to the path where NGINX can serve its contents.

This last installation job is explicitly filtered in the `.gitlab-ci.yml` file to be executed only on the master branch of the main project, and not by any fork since it needs a specific GitLab Runner with access to the path used by NGINX, outside of the isolated job environment.

Appendix A: The POP Methodology

At the top of the hierarchy is Global Efficiency (GE), which we use to judge overall quality of parallelisation.

Typically, inefficiencies in parallel code have two main sources:

- Overheads imposed by the parallel nature of a code.
- Poor scaling of computation with increasing numbers of processes.

And to reflect this behaviour we define two sub-metrics to measure these two inefficiencies. These are Computation Efficiency (CompE) and Parallel Efficiency (ParE), and our top-level metric is the product of these two sub-metrics:

```
GE = ParE * CompE
```

Computation Efficiency (CompE) are ratios of total time in useful computation summed over all processes. For strong scaling (i.e., problem size is constant) it is the ratio of total time in useful computation for a reference case (e.g., on 1 process or 1 compute node) to the total time as the number of processes (or nodes) is increased. For CompE to have a value of 1, this time must remain constant regardless of the number of processes.

Insight into possible causes of poor computation scaling can be investigated using metrics devised from processor hardware counter data. Two causes of poor computational scaling are:

- Dividing work over additional processes increases the total computation required.
- Using additional processes leads to contention for shared resources.

We investigate these causes using Instruction Efficiency (IE) and Instructions Per Cycle (IPC) Efficiency.

```
CompE = IE * IPCE
```

Instruction Efficiency is the ratio of total number of useful instructions for a reference case (e.g., 1 processor) compared to values when increasing the numbers of processes. A decrease in Instruction Efficiency corresponds to an increase in the total number of instructions required to solve a computational problem.

IPC Efficiency compares IPC to the reference, where lower values indicate that rate of computation has decreased. Typical causes for this include decreasing cache hit rate and exhaustion of memory bandwidth, these can leave processes stalled and waiting for data.

Parallel Efficiency (PE) reveals the inefficiency in splitting computation over processes and then communicating data between processes. PE is a compound metric whose components reflects two important factors in achieving good parallel performance in code:

- Ensuring even distribution of computational work across processes.
- Minimising time communicating data between processes.

These are measured with Load Balance (LB) Efficiency and Communication Efficiency (CommE), and PE is defined as the product of these two sub-metrics:

```
PE = LB * CommE
```

Load Balance (LB) is computed as the ratio between average useful computation time (across all processes) and maximum useful computation time (also across all processes):

```
LB = average computation time / maximum computation time
```

Communication Efficiency (CommE) is the maximum across all processes of the ratio between useful computation time and total runtime:

```
CommE = maximum computation time / total runtime
```

Serialisation Efficiency (SerE) measures inefficiency due to idle time within communications (i.e., time where no data is transferred) and is expressed as:

```
SerE = maximum computation time on ideal network / total runtime on ideal network
```

Transfer Efficiency (TE) measures inefficiencies due to time in data transfer:

```
TE = total runtime on ideal network / total runtime on real network
```

List of Acronyms and Abbreviations

- BSC: Barcelona Supercomputing Center
- CA: Consortium Agreement
- CAdv: Customer Advocate
- CD: Continuous Deployment
- CI: Continuous Integration
- CVS: Control Version System
- DBGS: Database Generation System
- DoA: Description of Action (Annex 1 of the Grant Agreement)
- D: Deliverable
- EC: European Commission
- ERD: Entity Relationship Diagram
- 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
- 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
- URL: Uniform Resource Locator
- USTUTT (HLRS): University of Stuttgart
- WP: Work Package
- WPL: Work Package Leader
- YAML: YAML Ain't Markup Language

List of Figures

1 POP Methodology: a hierarchy of metrics and efficiencies. 8
2 POP Repository structure: subsystems and components interaction. 9
3 Entity Relationship Diagram (ERD) of the repository in the specification phase. . 14
4 Entity Relationship Diagram (ERD) of the repository in the design phase. 16
5 Generated HTML page using local Liquid tags substitutions. 19
6 POP Repository interface: main site structure. 25
7 Repository automatic website generation: trigger, gather, build, and deploy. . . . 42

References

[1] Docker: https://www.docker.com, Accessed: May 2019.

[2] Gitlab docker images: https://docs.gitlab.com/omnibus/docker, Accessed: May 2019.

[3] Install gitlab runner: https://docs.gitlab.com/runner/install, Accessed: May 2019.

[4] Nginx: https://www.nginx.com, Accessed: May 2019.

[5] EPCC. Performance characterisation and benchmarking: https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking, Accessed: May 2019.

[6] GitLab. Gitlab documentation: https://docs.gitlab.com/ce, Accessed: March 2019.

[7] Gruber, J. Markdown: https://daringfireball.net/projects/markdown, Accessed: May 2019.

[8] Laboratories, S. N. Mantevo project: https://mantevo.org, Accessed: May 2019.

[9] The Jekyll Team. Jekyll on-line documentation: https://jekyllrb.com/docs, Accessed: February 2019.