SoftwareX 33 (2026) 102543

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

LabChain: Enabling reproducible and modular scientific experiments in Python

Manuel Couto [a,*], Javier Parapar [b], David E. Losada [a]

[a] Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela, Santiago de Compostela, Spain
[b] Centro de Investigación en Tecnologías de la Información y las Comunicaciones (CITIC), Universidade de A Coruña, A Coruña, Spain

ARTICLE INFO

Keywords:
Scientific workflows
Pipeline architecture
Hash-based caching
Reproducible research
Software engineering practices

ABSTRACT

Python's flexibility accelerates research prototyping but frequently results in unmaintainable code and duplicated computational effort. The absence of software engineering practices in academic development leads to fragile experiments where even minor modifications require rerunning expensive computations from scratch. LabChain addresses this through a pipeline-and-filter architecture with hash-based caching that automatically identifies and reuses intermediate results. When evaluating multiple classifiers on the same embeddings, the framework computes embeddings once—regardless of how many classifiers are tested. This automatic reuse extends across research teams: if another researcher applies different models to the same preprocessed data, LabChain detects existing results and eliminates redundant computation. Beyond efficiency, the framework's modular structure reduces technical debt that obscures experimental logic. Pipelines serialize to JSON for reproducibility and distributed execution across computational clusters. A mental health detection case study demonstrates dual impact: computational savings exceeding 12 hours per task with reduced CO_2 emissions, alongside substantial scientific improvements—performance gains up to 192.3% in some tasks. These improvements emerged from clearer experimental organization that exposed a critical preprocessing bug hidden in the original monolithic implementation. LabChain proves that software engineering discipline amplifies scientific discovery.

Metadata

Code metadata		
Nr.	Code metadata description	Metadata
C1	Current code version	v1.2.1
C2	Permanent link to code/repository	https://github.com/manucouto1/LabChain
C3	Permanent link to Reproducible Capsule	https://manucouto1.github.io/LabChain/examples/notebooks/optuna_optimizer_with_data_splitter_kfold/
C4	Legal Code License	AGPL-3.0
C5	Code versioning system used	Git
C6	Software languages, tools, and services	Python
C7	Compilation requirements, operating environments & dependencies	Python 3.11, typeguard 4.4.1, multimethod 1.12, pyspark 3.5.3, fastapi 0.115.5, pandas 2.2.3, torch 2.6.0, scipy 1.13.1, rich 13.9.4, boto3 1.35.73, scikit-learn 1.5.2, cloudpickle 3.1.0, tqdm 4.67.1, nltk 3.9.1, transformers 4.46.3, gensim 4.3.3, wandb 0.18.7, optuna 4.2.1, sentence-transformers 4.0.2
C8	Link to developer documentation/manual	https://manucouto1.github.io/LabChain/
C9	Support email for questions	manuel.couto.pintos@usc.es

* Corresponding author.
Email address: manuel.couto1@udc.es, manuel.couto.pintos@usc.es (M. Couto).

https://doi.org/10.1016/j.softx.2026.102543
Received 20 November 2025; Received in revised form 22 January 2026; Accepted 25 January 2026
Available online 5 February 2026

1. Motivation and significance

In the field of data science and machine learning, good practices and design principles play a key role in improving the efficiency, reproducibility, and scalability of experiments. Despite progress in standardizing tools for data processing and model building, the research community still faces challenges related to duplicated efforts, poor integration between different code components, and inefficient management of computational resources.

There are several open-source frameworks that attempt to address some of these issues, including scikit-learn, Kedro, Snakemake, PyCaret, and Luigi, among others. Each of these frameworks aims to facilitate the construction and execution of machine learning and data processing pipelines. For example, scikit-learn [1] is well-known for its support for building reusable models through pipelines and its broad collection of transformers and estimators, which simplify model reuse and validation. Kedro [2], in turn, specializes in building modular projects that enable large-scale workflow management and reproducibility, while Snakemake [3] provides a unified, transparent, and adaptable way to represent complete data analysis workflows, from raw data processing to final result visualization. PyCaret [4] offers a high-level solution that automates the modeling process without sacrificing modularity. Luigi [5], for its part, focuses on orchestrating complex workflows, particularly in distributed processing contexts.

Despite the advantages these frameworks offer [6], there are still important areas for improvement. Many of them do not efficiently address code duplication, the management of intermediate data, or the reuse of computational components across research teams working on similar tasks. For instance, in research settings involving multi-stage data processing pipelines, some steps are often computationally demanding and redundantly executed across various experiments. A typical example is the generation of embeddings using a pretrained neural model over a large textual corpus (e.g., applying BERT to obtain vectorial representations of documents). When embeddings are not shared across tasks, each experiment redundantly recomputes them, resulting in unnecessary computational overhead and substantial environmental impact.

Let us imagine the following scenario: two research teams are working on experiments that define different pipelines. These pipelines consist of various processing stages that we will refer to as filters. In the case of the first team, the pipeline is composed of filters A, B, and C, where steps A and B are extremely computationally expensive. Meanwhile, the second team is conducting a similar study, but their pipeline includes filters A, B, and D, among other possible variations.

Both teams could greatly benefit from a system that allows them to share filters (A, B) and avoid redundant computations. Our framework, LabChain, makes it possible to structure experiments in such a way that the computation of steps A and B is performed only once. LabChain handles unique identifiers that allow intermediate results to be stored and reused, thus eliminating the need to re-run the same transformations. This caching mechanism is illustrated in Fig. 1.

One of LabChain's most notable innovations is its ability to serialize and deserialize entire pipelines in JSON format. This allows packaging experiments as JSON files for transmission over standard protocols such as REST, GraphQL, gRPC, or event-based systems (Kafka, MQTT). This facilitates configuration exchange between users and enables remote compute nodes to receive serialized pipelines, deserialize them, and execute autonomously. As a result, it is possible to build distributed processing architectures where different nodes test experimental variations without manual intervention, fostering reproducibility and scalability essential to computational science.

1.1. Comparative analysis with related systems

LabChain occupies a unique niche among workflow frameworks by prioritizing automatic inter-team cache reuse and executable configuration over traditional workflow orchestration. While frameworks like scikit-learn, Kedro, Snakemake, and Luigi provide valuable pipeline abstractions, they do not address fundamental inefficiencies in collaborative research, such as the redundant computation of identical transformations across teams and the separation between workflow definition and execution logic.

The inter-team reuse problem. Consider a research group where multiple members work on related experiments using the same preprocessing steps (e.g., BERT embeddings over a corpus). In traditional frameworks, each researcher either recomputes transformations (scikit-learn), manually coordinates file sharing (Kedro/Luigi), or references explicit file paths in workflow rules (Snakemake). LabChain utilizes content-addressable storage, which enables identical configurations and input data to automatically produce cache hits across researchers without requiring manual coordination. This works across local and cloud storage backends (S3), enabling seamless collaboration.

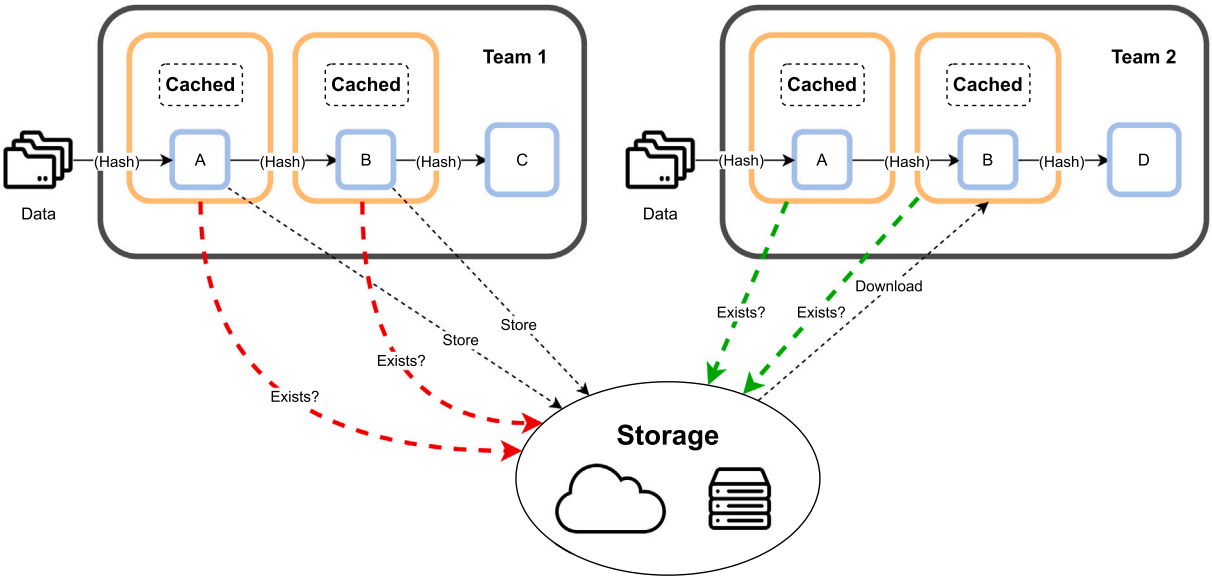

Fig. 1. LabChain: caching mechanism.

Table 1
Comparison of LabChain with existing workflow frameworks.

Feature	LabChain	scikit-learn	Kedro	Snakemake
Pipeline serialization	JSON (executable)	Pickle	YAML + code	YAML (rules)
Automatic caching	Hash-based	None	Manual	Timestamps
Inter-team reuse	Automatic	No	Manual	Manual
State management	Filter-internal	Pipeline objects	Catalog flow	File artifacts
Workflow definition	Code (filters)	Code	YAML DAG	YAML rules
Cloud storage	Native (S3)	No	Plugin	Limited
Setup overhead	Minimal	None	High	Medium
Target use case	Iterative research	Model building	Production	Bioinformatics

Executable configuration vs. declarative workflows. A key architectural distinction lies in how LabChain unifies configuration and execution logic. Kedro or Snakemake separate workflow definition (YAML/Python DAGs) from execution code. State (fitted models) flows through the pipeline alongside data, requiring explicit input/output declarations. In LabChain, configuration *is* executable code. A JSON-serialized pipeline can be directly instantiated and executed. Data flow is implicit in the filter sequence. State lives within filter instances as private attributes, not as pipeline data.

This design provides two practical advantages: (1) each JSON dump represents a complete, reproducible experiment that serves as an experimental backlog of explored configurations, and (2) researchers think in terms of sequential transformations rather than dependency graphs, with state management encapsulated within components.

Conceptual contribution. While individual components (pipelines, caching, dependency injection) are not novel, LabChain's contribution lies in their integration for collaborative experimental research with automatic result reuse. This differs from production workflow orchestration (Kedro, Luigi), bioinformatics workflows (Snakemake), and model building (scikit-learn), each of which targets distinct use cases with varying trade-offs. Table 1 provides a comparative analysis between LabChain and other existing workflow frameworks.

2. Software description

LabChain is built on established design principles and software patterns. Its architecture encapsulates each processing step as a *filter*, which can be combined into serializable pipelines. This approach facilitates reuse, composition, and distributed execution. The framework employs a container-based dependency injection system alongside a hash-based storage strategy reminiscent of content-addressable storage systems.

2.1. Filter architecture and serialization

LabChain follows the **Pipes and Filters** pattern. Each filter, derived from `BaseFilter`, encapsulates a specific data transformation. To overcome the limitations of standard JSON serialization when handling complex machine learning objects (such as tokenizers, vectorizers, or pretrained models), LabChain enforces a clear separation of attributes:

- **Public Attributes (Configuration):** These are the filter's configuration parameters that define its behavior. LabChain serializes these attributes and uses them as constructor arguments during filter reconstruction (passed as `**kwargs` to `__init__`). Consequently, all public attributes must be declared as `__init__` parameters. Only basic serializable types (strings, integers, floats, booleans, lists, dictionaries) are supported.
- **Private Attributes (Non-configuration State):** Private attributes (prefixed with an underscore) represent internal implementation details rather than configuration parameters. These include execution state, utility objects, and non-serializable components that do not define the filter's identity.

This design allows `item_dump()` to produce portable JSON representations of a pipeline "recipe" without the overhead of serializing arbitrary Python objects.

2.2. Cache-key specification and validation

Reproducibility in LabChain relies on an explicit and deterministic cache-key definition. Cache keys are computed as cryptographic hashes derived exclusively from user-visible and serialized information, ensuring that identical experimental configurations yield identical cache identifiers.

Deterministic hashing modes. LabChain implements a dual-mode hashing system adapted to filter characteristics:

- **Non-trainable filters (initialization-based):** For static transformations, the hash is computed at instantiation from the class name and all public attribute values.
- **Trainable filters (state-based):** For filters requiring a `fit()` method, the hash is finalized before training, combining the class name, public attributes, and the unique hash of the training data.

Formally, for each filter instance f, the filter hash H_f is computed as:

$$H_f = \texttt{hash}(\texttt{class\_name}(f), \texttt{pub\_attributes}(f), H_{\text{train}})$$

where H_{train} is included only for trainable filters and corresponds to the hash of the training dataset used during `fit()`. For non-trainable filters, H_{train} is omitted.

Data provenance chain. LabChain maintains pipeline integrity through a composition of hashes. When a filter processes an input dataset with hash H_{in}, the resulting output dataset hash is:

$$H_{\text{out}} = \texttt{hash}(H_f, H_{\text{in}})$$

This mechanism forms a *chain of hashes* that encodes the complete provenance of the data as it flows through the pipeline. The initial dataset hash, defined by the user, serves as the root of this provenance chain.

Included and excluded factors. By design, cache keys depend only on: the filter class identity, its public configuration parameters, and upstream data hashes (which themselves depend on data content or user-provided identifiers). Keys do not depend on source code versions, library versions, runtime environments, or hardware characteristics, unless such factors are explicitly encoded as public parameters.

Cache control and invalidation. To manage caching behavior, LabChain provides a `Cached` wrapper filter offering fine-grained control through three parameters:

- `store_data`: Toggles persistence of processed output datasets.
- `store_model`: Enables or disables caching of fitted filter states.
- `overwrite`: Forces recomputation even when a matching hash exists in cache. When enabled, if `store_data` and/or `store_model` are also active, the newly computed results will replace the existing cached entries for that hash.

This mechanism supports iterative development workflows where researchers may need to force recomputation due to external changes not captured by configuration parameters (e.g., bug fixes in dependencies, hardware-specific numerical behavior). To mitigate the risk of stale results following internal logic updates or bug fixes, we recommend that users either explicitly set the `overwrite` flag to `True` for a single run or include a dedicated versioning parameter within the filter's public configuration to force a hash change.

Non-determinism and stochasticity. Filters relying on stochastic operations are reproducible only if sources of randomness (e.g., random seeds) are explicitly exposed as public configuration parameters. LabChain guarantees cache consistency based on the defined key structure but does not enforce bit-wise numerical reproducibility for filters that do not control their random state.

Validation. The correctness of cache hits and misses is validated through dedicated tests in the LabChain suite. Atomic behavior is verified in `tests/unit`, ensuring hashing determinism and cache hits under controlled changes to filter parameters. Collective behavior within complex experimental workflows is validated in `tests/integration`, ensuring state consistency and the reliable reuse of cached results when combining multiple framework elements, such as sequential and parallel pipelines.

2.3. Storage backends

LabChain's caching system is storage-backend agnostic through the `BaseStorage` abstraction, which defines a uniform interface for storing and retrieving cached data and models. The framework provides two built-in implementations:

- **Local filesystem:** Stores cached results in the local file system, suitable for individual researchers or single-machine workflows.
- **Cloud storage (S3):** Persists cached results to Amazon S3 or S3-compatible object storage, enabling collaborative environments where multiple teams share cached intermediate results across geographical locations.

Users can implement custom storage backends by extending `BaseStorage` and implementing its interface methods (`upload_file`, `download_file`, `check_if_exists`, etc.). This extensibility allows integration with institutional storage systems, alternative cloud providers, or specialized distributed filesystems without modifying LabChain's core logic. The storage backend is configured at the pipeline or filter level, allowing researchers to mix local and cloud storage within the same workflow. For example, expensive preprocessing steps can be cached to shared cloud storage for team-wide reuse, while intermediate debugging outputs remain local.

2.4. Dependency management: container and IoC

At the core of LabChain is the Container, a class-name-based manager implementing Inversion of Control (IoC) and Dependency Injection (DI). Components are registered via the `@Container.bind()` decorator. This decouples components, which are assembled by the container rather than depending on each other directly. This design is particularly useful when reconstructing pipelines with `build_from_dump()`, as the container can re-instantiate private components based on the serialized public configuration. While it introduces a learning curve, it reduces technical debt by exposing structural errors that would remain hidden in monolithic scripts.

2.5. Software design

Fig. 2 illustrates the UML design of LabChain's core system. The architecture emphasizes extensibility, modularity, and component reuse. Several classical design patterns are applied:

- **Factory/Abstract Factory**: `BaseFactory` enables dynamic registration and instantiation of plugins from textual identifiers, decoupling component creation from specific implementations.
- **Plugin/Strategy**: Filters, metrics, and storage systems inherit from `BasePlugin`, allowing dynamic substitution and experimentation with alternative processing strategies.
- **Composite**: `BasePipeline` can compose multiple filters and metrics into hierarchical pipelines, treating entire workflows as reusable entities.
- **Template Method**: Base classes (`BaseFactory`, `BasePlugin`, `BaseFilter`) define the general workflow (`fit`, `predict`, `evaluate`, `item_dump`), while subclasses implement specific steps.
- **Adapter/Wrapper**: `BaseStorage` provides unified access to diverse storage backends (local, S3), which can be extended with custom implementations.

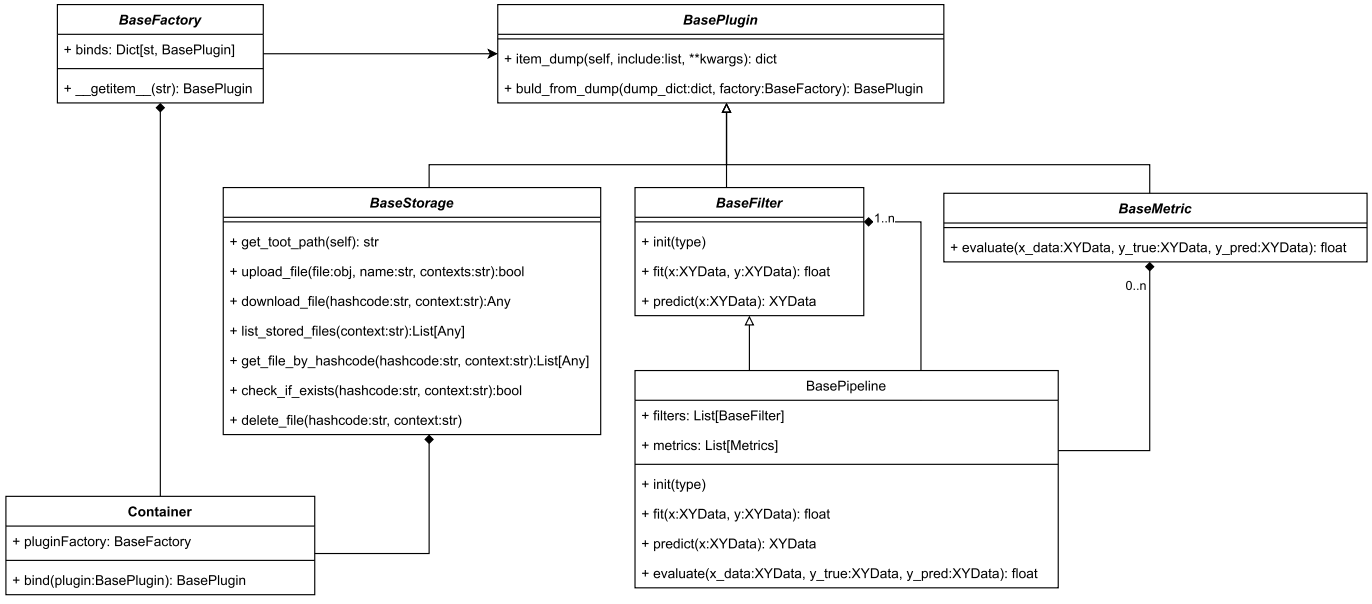

Fig. 2. General architecture of LabChain. Filters are organized into pipelines managed by a container. Each filter generates unique hashes that are stored in a content-addressable caching system.

This design allows independent testing, modular extensions, and portable reconstruction of pipelines using `build_from_dump`.

3. Illustrative examples

To illustrate the flexibility and power of the `LabChain` framework, we present a practical example focused on the early detection of signs of mental illness (based on temporal linguistic patterns). This example highlights three key capabilities: modular definition of filters, experiment serialization, and distributed execution of pipelines.

3.1. Pipeline construction and execution

As a use case, we implemented a pipeline inspired by research on temporal language analysis in social media [7]. The goal is to detect early signs of mental disorders based on users' posts (from Reddit). The data consist of sequences of messages, grouped by user, and organized into two classes (users diagnosed with depression vs control users).

The pipeline consists of two main phases:

- **Temporal feature extraction.** We use the Temporal Word Embeddings with a Compass model (`TWEC`), a variant of `Word2Vec`, to generate word embeddings that evolve over time [8]. By comparing these dynamic embeddings with their static versions, we obtain measures of individual semantic change for each word and user. The computation of measures of change is supported by different distance metrics (e.g., cosine, Euclidean, Chebyshev).
- **Classification and optimization.** The resulting representations score the evolution of the words in the vocabulary at different times. These features are fed into different classifiers – Support Vector Machines (`SVM`), Random Forest Classifiers (`RFC`), and K Nearest Neighbors (`KNN`)– which are automatically optimized using `GridSearch` with cross-validation. This allows prediction of signs of mental health based on the semantic evolution of language.

The output of the pipeline is a binary prediction of the user's mental health status, generated for each set of user's texts (the user's posts are organized in temporal *chunks*).

Custom filters. One of the advantages of `LabChain` is the ease of creating reusable filters. In this example, semantic distance matrices between embeddings (referred to as deltas) are precomputed and cached. These distances represent the movement of each word in each chunk relative to a canonical representation of that word (i.e., the degree to which the word shifts from its general usage). The following filter allows the selection at runtime of the distance metric to be applied:

```
@Container.bind()
class DeltaSelector(BaseFilter):
    def __init__(
        self,
        deltas_f: Literal[
            "cosine",
            "euclidean",
            "chebyshev",
            "jensen_shannon",
            "wasserstein",
            "manhattan",
            "minkowski",
        ] = "cosine",
    ):
        self.deltas_f = deltas_f
    def fit(self, x: XYData, y: XYData | None):
        pass
    def predict(self, x: XYData) -> XYData:
        stored_dok = x.value[self.deltas_f]
        if not isinstance(stored_dok, csr_matrix)
    :
            selected_matrix = stored_dok.tocsr()
        return XYData.mock(selected_matrix)
```

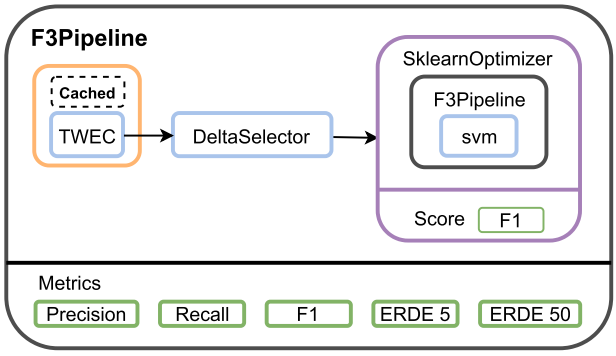

Fig. 3. Conceptual representation of the pipeline.

This filter acts as a view selector, allowing the user to easily switch the analysis metric without recalculating the underlying embeddings.

Pipeline design. Fig. 3 schematically represents the processing flow for this use case. All embeddings and deltas are generated in a single pass, cached, and then reused.

Advantages of the approach. This design demonstrates several key principles of the framework:

- **Efficiency.** The design's flexibility allows temporal embeddings and deltas to be generated in the initial filter, followed by a selector filter. This approach inherently reduces computation time. Furthermore, caching can significantly reduce the final computation time when using multiple classifiers, as would be the case when replicating the original paper's methodology.
- **Modularity.** Components are implemented as reusable and interchangeable filters. This architecture ensures easy adaptation to new datasets, evaluation metrics, or models without modifying the core structure.
- **Scalability.** The pipeline can be easily parallelized and executed in a distributed manner, regardless of data size or the number of experimental configurations.
- **Rich Evaluation.** The system supports both internal evaluation metrics (e.g., `F1` during optimization) and task-specific external metrics (e.g., `ERDE`, an error measure that penalizes late decisions about risk). Additionally, the caching mechanism enables detailed inspection of intermediate data, allowing for deeper and more informative evaluations at each pipeline stage.

Pipeline serialization. Pipelines can be easily exported to `JSON` using the `item_dump()` function, as shown below:

```
dumped_pipeline = pipeline.item_dump()
```

The output is a nested dictionary representing the pipeline structure and the public parameters of each component. This representation is easily serializable and shareable, enabling experiment storage, reproducibility, and distributed execution. Currently, only basic data types (e.g., str, int, list) are supported for filter parameters, excluding arbitrary objects.

3.2. Reproducibility and collaboration

The use of declarative pipeline representations enables seamless sharing and reconstruction of experiments. Pipelines serialize to JSON format, facilitating collaboration among team members and ensuring that computational workflows can be stored alongside datasets. This approach transforms pipeline definitions into executable documentation that captures not just the results, but the entire experimental process.

Reconstructing a pipeline is straightforward, a single function call restores the complete computational workflow. This capability enables researchers to systematically compare pipeline variants defined at different times, test alternative approaches, or apply identical processing to new datasets without risk of introducing subtle implementation differences.

```
from labchain.base import BasePipeline,
    BasePlugin
from typing import cast

reconstructed_pipeline: BasePipeline = cast(
    BasePipeline,
    BasePlugin.build_from_dump(
        dumped_pipeline, Container.pif
    )
)
reconstructed_pipeline.fit(X, y)
```

3.3. Distributed execution: exploring multiple pipelines in parallel

Classical distributed computing frameworks like Hadoop and Spark parallelize data processing by replicating the same computational logic across different data fragments. LabChain proposes a complementary strategy: parallelizing entire pipeline variants rather than data. This design proves particularly valuable in automated experimentation and systematic configuration exploration.

Consider a common research scenario: evaluating multiple classifiers (such as `SVM`, `RFC`, and `KNN`) on the same feature representations. With LabChain's distributed execution, different classifier configurations can be distributed across cluster nodes while sharing preprocessed data through the caching system. This eliminates redundant computation of expensive preprocessing steps and focuses computational resources on exploring classification and optimization strategies.

The following example demonstrates distributed pipeline execution. Each pipeline variant receives serialized configuration, executes independently on cluster workers, and the system automatically collects results into a unified output structure:

```
from labchain.pipeline import HPCPipeline
from labchain.plugins.filters import Knn,
    RandomForest, SVM

 f_experiment = lambda classifiers: F3Pipeline(
    filters=[
        TWEC(),
        DeltaSelector(),
        *classifiers
    ],
    metrics=[F1(), Precision(), Recall()]
)
pipeline = HPCPipeline(
    filters=[
        f_experiment([SVM()]),
        f_experiment([RandomForest()]),
        f_experiment([Knn()]),
    ],
    app_name="classifier_comparison",
    master="192.168.0.100:7077"
)
pipeline.fit(X, y)
predictions = pipeline.predict(X)
```

In this configuration, each branch of the `HPCPipeline` represents a complete experimental variant with a different classifier. The distributed system handles the complexity of scheduling, execution, and result aggregation, while cached intermediate results (embeddings and distance computations) are automatically shared across variants. Fig. 4 illustrates how these pipeline variants are distributed across cluster workers.

3.4. Hyperparameter optimization

Modern machine learning experiments require systematic exploration of hyperparameter spaces. This challenge has been central to automated machine learning research [9,10], where evaluating multiple pipeline configurations is considered essential for identifying robust and reproducible solutions. LabChain provides modular integration of multiple optimization strategies through a unified interface, supporting approaches ranging from exhaustive grid search to sophisticated Bayesian optimization, as well as visual tracking tools for monitoring experimental progress.

GridOptimizer. For problems with discrete hyperparameter spaces, LabChain includes a grid search optimizer that systematically evaluates all combinations of parameters defined through the `.grid()` method. This exhaustive approach works well when the search space is bounded and computational resources allow thorough exploration.[1]

OptunaOptimizer. When dealing with large or continuous hyperparameter spaces, Bayesian optimization offers a more efficient alternative. Through integration with Optuna [11], LabChain enables intelligent search strategies that learn from previous evaluations to guide exploration toward promising regions of the parameter space.[2]

WandbOptimizer. Systematic experimentation benefits from visual tracking and comparison of different configurations. Integration with Weights & Biases [12] provides real-time monitoring of optimization progress, comparison of multiple experimental runs, and support for remote collaboration among research team members.[3]

```
pipeline = F3Pipeline(
    filters=[
        StandardScaler(),
        Knn().grid({
            "n_neighbors": [2, 3, 4, 5, 6]
        })
    ],
    metrics=[F1(), Precision(), Recall()]
)
.splitter(
    KFoldSplitter(n_splits=5, shuffle=True)
)
.optimizer(GridOptimizer(scorer=F1()))

pipeline.fit(X, y)
```

[1] Tutorial: GridOptimizer example.
[2] Tutorial: OptunaOptimizer example.
[3] Tutorial: WandbOptimizer.

```
pipeline = F3Pipeline(
    filters=[
        PCA().grid({
            "n_components": [10, 100]
        }),
        SVM(kernel="rbf").grid({
            "C": [0.1, 10.0],
            "gamma": [0.001, 0.01]
        })
    ],
    metrics=[F1(), Precision(), Recall()]
)
.splitter(
    KFoldSplitter(n_splits=5, shuffle=True)
)
.optimizer(
    OptunaOptimizer(
        n_trials=50,
        direction="maximize",
        study_name="svm_optimization",
        load_if_exists=True,
        storage="sqlite:///optuna_studies.db"
    )
)

pipeline.fit(X, y)
```

```
pipeline = F3Pipeline(
    filters=[
        StandardScaler(),
        Knn().grid({
            "n_neighbors": [2, 3, 4, 5, 6]
        })
    ],
    metrics=[F1(), Precision(), Recall()]
)
.splitter(
    KFoldSplitter(n_splits=5, shuffle=True)
)
.optimizer(
    WandbOptimizer(
        project="mental_health_detection",
        sweep_id=None,
        scorer=F1()
    )
)

pipeline.fit(X, y)
```

SklearnOptimizer. For researchers already working within the scikit-learn [1] ecosystem, LabChain provides an optimizer that maintains compatibility with scikit-learn's validation infrastructure. This enables seamless integration with existing workflows while benefiting from LabChain's caching and pipeline management capabilities. The case study presented in this article demonstrates this optimizer in practice.

4. Impact

LabChain transforms how research teams develop, execute, and standardize computational experiments. Beyond technical improvements, it addresses fundamental challenges in scientific software: reproducibility, sustainability, and the psychological burden of experimental iteration.

Substantial improvement in experimental outcomes. To demonstrate practical utility, we re-implemented the experimental pipeline from prior work on early detection of mental health indicators [7]. The pipeline comprises three stages: (1) computing temporal representations using `TWEC` with multiple distance metrics, (2) selecting metrics via `DeltaSelector`, and (3) optimizing an `SVM` classifier. The initial stage alone requires one to several hours depending on dataset size.

Traditional implementations follow computational complexity N×(A+B+C), where N represents experimental variants and A, B, C represent pipeline stages. LabChain's caching reduces this to A+N×(B+C) by automatically reusing intermediate results. When exploring multiple classifiers on the same embeddings, this difference becomes transformative, expensive preprocessing executes once rather than repeatedly.

Beyond efficiency, caching provides psychological liberation. Research with dynamically typed languages like Python involves numerous subtle errors, type mismatches, configuration mistakes, and module incompatibilities, that often surface only after hours of computation. Automatic caching creates checkpoints that preserve progress, enabling researchers to explore alternative approaches without fear of losing work. This reduces both mental and computational costs of experimentation, encouraging the creative, iterative mindset essential to scientific discovery.

Preserved intermediate data also enables analytical depth. Researchers can study how different pipeline stages contribute to final performance without manually implementing inspection mechanisms. By embedding this capability into the framework, LabChain reduces technical debt while enhancing scientific focus.

The re-implementation using LabChain produced results demonstrably superior to those originally published, with some achieving state-of-the-art performance. This empirically confirms that well-designed tools not only improve development efficiency but can directly enhance scientific outcomes.

Environmental sustainability. Computational efficiency translates to environmental impact. Delta computation via `TWEC` on the depression dataset requires approximately three hours. The original study evaluated five models, without caching, this totals 15 hours per task. Extending to four tasks (depression, anorexia, self-harm, gambling) with multiple classifiers amounts to at least 48 hours of redundant computation.

Using conservative estimates (0.052 kg CO_2/hour), caching saves approximately 2.50 kg CO_2 across these experiments. Under higher emission assumptions (0.234 kg/hour), savings exceed 11 kg CO_2. These figures underscore both efficiency gains and contributions to sustainable research practices.

Empirical evidence. LabChain does not modify model architectures or algorithms, its impact lies in how experiments are structured. By reducing technical debt through modular organization, the framework creates mental space for deeper analytical thinking. When researchers are freed from managing ad-hoc scripts, tracking intermediate results manually, or reconstructing preprocessing pipelines, they can focus cognitive resources on identifying improvements, testing hypotheses, and refining experimental design.

This cognitive benefit manifested in tangible outcomes. During re-implementation of temporal language analysis for early risk prediction [7] using LabChain,[4] the modular structure immediately exposed a critical bug in the original corpus preparation. The preprocessing step failed to flatten the hierarchical data structure before feeding it to the word embedding model (TWEC). Instead of processing individual posts as separate documents, the entire corpus was concatenated into a single massive sequence. This structural error corrupted the training process: the model lost document boundaries, distorting word co-occurrence statistics and context windows essential for learning meaningful embeddings. The bug remained undetected across multiple experiments because monolithic preprocessing code obscured the data flow. LabChain's encapsulation of preprocessing as an explicit, testable filter made the structural corruption immediately visible during component validation.

Correcting this single structural error, combined with the systematic exploration enabled by LabChain's caching system, yielded the substantial improvements shown in Table 2. The results demonstrate that

[4] github.com/manucouto1/Temporal-Word-Embeddings-for-Early-Detection.

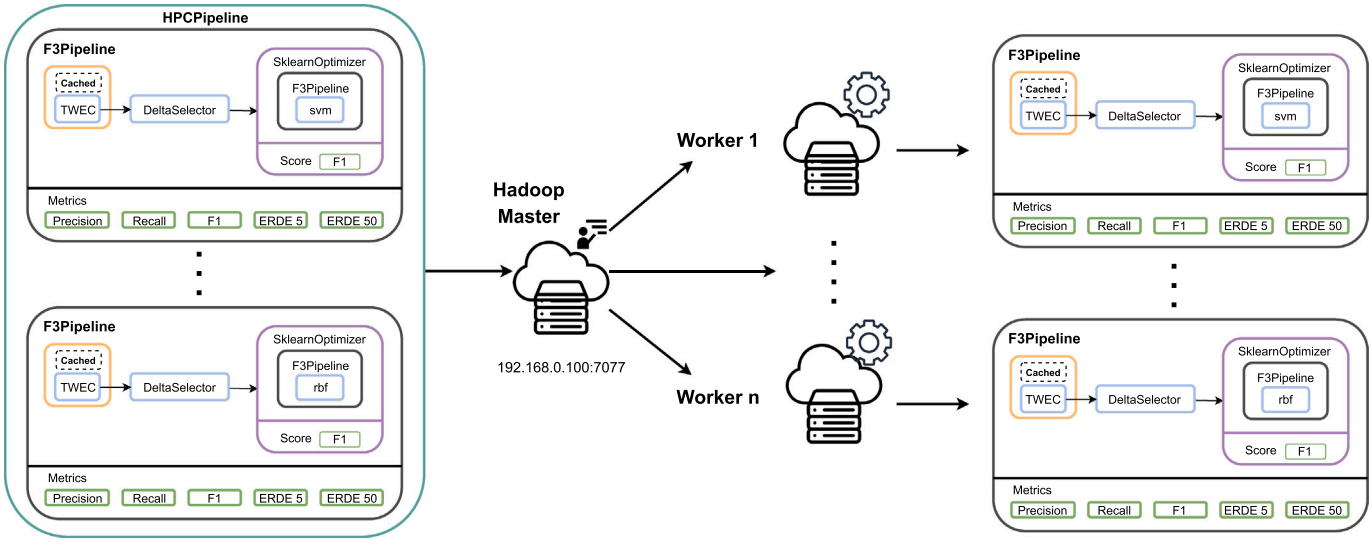

Fig. 4. Distribution of pipeline jobs in a Hadoop cluster. Each worker executes a complete pipeline variant, sharing cached preprocessing results.

Table 2
Performance comparison between original and LabChain implementations. Bold indicates best performance per dataset (lower ERDE values are better).

Dataset	Model	$ERDE_5$	$ERDE_{50}$	F1
Depression				
2017	LabChain	**12.44**	**7.66**	**0.50**
	Original	12.56	10.21	0.25
2018	LabChain	9.36	**5.56**	**0.46**
	Original	**9.09**	7.84	0.21
2022	LabChain	**6.57**	**3.81**	**0.46**
	Original	6.69	5.82	0.17
Anorexia				
2018	LabChain	**18.35**	**12.52**	**0.26**
	Original	21.40	15.23	0.22
2019	LabChain	**7.95**	**2.60**	**0.63**
	Original	8.17	6.30	0.27
Self-harm				
2020	LabChain	**19.96**	9.17	**0.62**
	Original	18.79	**16.37**	0.41
2021	LabChain	9.80	**5.42**	**0.33**
	Original	**9.02**	7.21	0.26
Gambling				
2022	LabChain	**3.35**	**0.88**	**0.38**
	Original	3.76	3.31	0.13
2023	LabChain	2.96	**0.41**	**0.88**
	Original	**2.92**	1.10	0.74

software engineering practices directly influence research outcomes: modularity makes errors visible that would otherwise remain buried in unstructured code.

Standardization and collaboration. Contemporary computational science lacks standardization in experimental development, impeding collaboration, traceability, and reproducibility. LabChain provides consistent infrastructure for defining, executing, and documenting experiments within a unified framework. This aligns with Baggen et al.'s [6] demonstration that code quality standards improve maintainability and development efficiency. By adopting modular design and separation of concerns, LabChain enhances collaboration among researchers with varying technical expertise.

The framework addresses growing demand for standardized methodologies in research software. Wainer et al. [13] emphasize that absent standards hamper tool interoperability, while Katz et al. [14] argue that community organizations transform research software culture toward sustainable, reproducible practices.

Adoption potential and future directions. LabChain's component-oriented design evokes established tools like LONI Pipeline [15], which accelerated reproducible neuroimaging research through visual interfaces and modular architecture. Similarly, it shares MLPro's [16] spirit of orchestrating complex workflows through consistent, extensible interfaces. The framework positions itself as a potential de facto standard for scientific experiments in AI, machine learning, and computational science.

Standardization through LabChain enables new research questions in scientific software engineering: how do coding styles, architectural quality, or code smells influence result reliability? Jerzyk et al. [17] show that development practices affect library security, while Kim et al. [18] demonstrate that poor design introduces vulnerabilities. Garlan et al. [19,20] argue that architectural patterns enable robustness evaluation. LabChain captures experimental architecture as versionable artifacts, facilitating such analyses.

Following Wright and Druta [21], LabChain serves as nexus between community and standardization. Effective standardization emerges from active communities sharing tools, conventions, and documentation. The framework enables groups to adopt common structures while maintaining flexibility, with reusable modules and versioned pipelines evolving into shared forms. This promotes participatory governance of computational knowledge, aligning with open science principles.

Current development focuses on graphical interfaces for users without programming expertise, similar to LONI's approach [15]. Future work includes federation of execution servers through actor-based architectures, enabling pipeline components to run across physical locations. Breivold et al. [22] advocate this architecture for large-scale service-oriented systems, LabChain could become the technological foundation for virtual laboratories and federated experimentation environments.

At a deeper technical level, future work will explore tighter integration with PyTorch, allowing neural network architectures to be expressed as cacheable filter pipelines. This would enable layer-wise pretraining with reuse of intermediate representations and selective parameter freezing for downstream tasks, extending LabChain's caching model to internal neural network components. In addition, ongoing work investigates mechanisms for coordinated execution over shared storage backends to mitigate race conditions under concurrent pipeline

usage. Finally, future releases are expected to incorporate versioned and persistent registries for plugins and serialized classes stored in shared backends, enabling remote pipeline execution without requiring full project source code at runtime. These directions remain subject to further validation across diverse execution environments before being considered for production use.

5. Conclusions

Scientific computing faces a software crisis that undermines reproducibility and wastes resources. LabChain addresses this through principled engineering: modular filters, hash-based caching, and JSON-serializable pipelines transform fragile research scripts into robust, reusable infrastructure. The framework demonstrates that software quality is not orthogonal to scientific quality, they are fundamentally linked.

The mental health detection case study validates this connection. Restructuring experiments with LabChain yielded both computational savings (12+ hours per task, reduced CO_2 emissions) and superior scientific outcomes, with performance improvements reaching 192.3% in some tasks. These gains emerged not from algorithmic innovation but from reduced technical debt: when researchers spend less cognitive energy managing code complexity, they invest more in hypothesis refinement and experimental insight.

This work challenges an implicit assumption in research software development: rapid prototyping requires sacrificing maintainability and exploration demands technical chaos. LabChain proves the opposite. Modularity, encapsulation, and automatic caching do not constrain creativity, they amplify it by removing friction from the experimental cycle. The framework's caching system acts as a psychological safety net, encouraging researchers to explore risky ideas without fear of losing progress.

Maturity and roadmap. LabChain is being actively maintained and employed daily in production research workflows at our research center (CiTIUS). Core components (filters, pipelines, Container, caching, data splitting, WandB optimizer) are production-ready. Experimental components (Sklearn/Optuna optimizers, distributed execution) are functional but have not been extensively tested. The API is stabilizing, with significant changes documented in GitHub releases. As an open-source project with limited resources, we strongly encourage community contributions.

The framework is publicly available with complete documentation at https://manucouto1.github.io/LabChain/. We invite the community not only to use LabChain, but also to embrace its underlying proposition: that applying software engineering principles to computational science is not a constraint on research, but an accelerant. The code we write shapes not only what we can compute, but what we can discover.

CRediT authorship contribution statement

Manuel Couto: Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Javier Parapar:** Writing – review & editing, Supervision, Project administration, Methodology, Funding acquisition, Conceptualization. **David E. Losada:** Writing – review & editing, Supervision, Project administration, Methodology, Funding acquisition, Conceptualization.

Declaration of generative AI and AI-assisted technologies in the manuscript preparation process

During the preparation of this work, the authors used Claude (Anthropic) to assist with improving language clarity, manuscript organization, and refining technical explanations. The tool was used to enhance readability and ensure consistent technical terminology throughout the document. After using this tool, the authors carefully reviewed, edited, and validated all content to ensure accuracy and alignment with the research objectives. The authors take full responsibility for the content of the published article.

Declaration of competing interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

MC and DEL thank the financial support provided by MICIU/AEI/10.13039/501100011033 (PID2022-137061OB-C22, supported by ERDF) and Xunta de Galicia-Consellería de Cultura, Educación, Formación Profesional e Universidades (ED431G 2023/04, ED431C 2022/19, supported by ERDF).

JP has received support from project PID2022-137061OB-C21 (MCIU/AEI/10.13039/5011000 11033, Ministerio de Ciencia e Innovación). He also thanks the financial support provided by the Consellería de Educación, Universidade e Formación Profesional, Spain (grant number ED481A-2024–079 and GRC ED431C 2025/49); and the European Regional Development Fund, which supports the CITIC Research Center.

Data availability

The LabChain framework is publicly available at https://github.com/manucouto1/LabChain. The reference implementation for this article is **v1.2.1**.

The mental health detection case study uses publicly available datasets from the eRisk shared tasks: Depression (2017, 2018, 2022), Anorexia (2018, 2019), Self-harm (2020, 2021), and Gambling (2022, 2023). These datasets can be requested from the eRisk organizers at https://erisk.irlab.org/.

The complete implementation of the case study, including all pipeline configurations and preprocessing code, is available at https://github.com/manucouto1/Temporal-Word-Embeddings-for-Early-Detection....

No new data were generated or analyzed in support of this research.

References

[1] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, et al. Scikit-learn: machine learning in Python. J Mach Learn Res 2011;12:2825–30.

[2] QuantumBlack AMC. Kedro: an open-source Python framework for creating reproducible, maintainable and modular data science code, 2019. [Accessed: 2025–05–23]; https://kedro.org.

[3] Mölder F, Jablonski KP, Letcher B, Hall MB, Tomkins-Tinch CH, Sochat V, Forster J, Lee S, Twardziok SO, Kanitz A, et al. Sustainable data analysis with snakemake. F1000Research 2021;10:33.

[4] Ali M. PyCaret: an open source, low-code machine learning library in Python, pyCaret version 1.0 Apr 2020. https://www.pycaret.org.

[5] Rieger M, Erdmann M, Fischer B, Fischer R. Design and execution of make-like, distributed analyses based on Spotify's pipelining package Luigi. arXiv preprint arXiv:1706.00955, 2017.

[6] Baggen R, Correia JP, Schill K, Visser J. Standardized code quality benchmarking for improving software maintainability. Softw Qual J 2012;20(2):287–307.

[7] Couto M, Perez A, Parapar J, Losada DE. Temporal word embeddings for early detection of psychological disorders on social media. J Healthc Inform Res 2025:1–30.

[8] Di Carlo V, Bianchi F, Palmonari M. Training temporal word embeddings with a compass. In: Proceedings of the AAAI conference on artificial intelligence, vol. 33. 2019. p. 6326–34.

[9] Feurer M, Klein A, Eggensperger K, Springenberg J, Blum M, Hutter F. Efficient and robust automated machine learning. Adv Neural Inf Process Syst 2015; 28.

[10] Olson RS, Bartley N, Urbanowicz RJ, Moore JH. Evaluation of a tree-based pipeline optimization tool for automating data science. In: Proceedings of the genetic and evolutionary computation conference 2016; 2016. p. 485–92.

[11] Akiba T, Sano S, Yanase T, Ohta T, Koyama M. Optuna: a next-generation hyperparameter optimization framework. In: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery &. data mining; 2019. p. 2623–31. https://doi.org/10.1145/3292500.3330701

[12] Biewald L. Experiment tracking with weights and biases, software available from wandb.com; 2020. https://www.wandb.com/.

[13] Wainer GA, Al-Zoubi K, Hill DR, Mittal S, Martín JLR, Sarjoughian H, Touraille L, Traoré MK, Zeigler BP. An introduction to DEVS standardization. In: Discrete-Event Modeling and Simulation, CRC Press; 2018. pp. 393–425.

[14] Katz DS, McInnes LC, Bernholdt DE, Mayes AC, Hong NPC, Duckles J, Gesing S, Heroux MA, Hettrick S, Jimenez RC, et al. Community organizations: changing the culture in which research software is developed and sustained. Comput Sci Eng 2018;21(2):8–24.
[15] Rex DE, Ma JQ, Toga AW. The LONI pipeline processing environment. Neuroimage 2003;19(3):1033–48.
[16] Arend D, Diprasetya MR, Yuwono S, Schwung A. MLPro: an integrative middleware framework for standardized machine learning tasks in Python. Softw Impacts 2022;14:100421.
[17] Jerzyk M, Madeyski L. Code smells: a comprehensive online catalog and taxonomy. In: Developments in information and knowledge management systems for business applications, vol. 7: Springer; 2023. pp. 543–76.
[18] Kim S, Lee H. Software systems at risk: an empirical study of cloned vulnerabilities in practice. Comput & Secur 2018;77:720–36.
[19] Garlan D, Shaw M. An introduction to software architecture. In: Advances in software engineering and knowledge engineering. World Scientific; 1993. pp. 1–39.
[20] Garlan D. Software architecture: a travelogue. In: Future of software engineering proceedings; 2014. p. 29–39.
[21] Wright SA, Druta D. Open source and standards: the role of open source in the dialogue between research and standardization. In: 2014 IEEE globecom workshops (GC Wkshps). IEEE; 2014. p. 650–5.
[22] Breivold HP, Crnkovic I, Larsson M. A systematic review of software architecture evolution research. Inf Softw Technol 2012;54(1):16–40.