A Configuration-First Framework for Reproducible, Low-Code Localization

TIM STRNAD, Jožef Stefan Institute, Slovenia
BLAŽ BERTALANIČ, Jožef Stefan Institute, Slovenia
CAROLINA FORTUNA, Jožef Stefan Institute, Slovenia

Machine learning is increasingly permeating radio based localization services. To keep results credible and comparable, everyday workflows should make rigorous experiment specification and exact repeatability the default, without blocking advanced experimentation. However, in practice, researchers face a three-way gap that could be filled by a framework that offers (i) low coding effort for end-to-end studies, (ii) reproducibility by default including versioned code/data/configurations, controlled randomness, isolated runs, recorded artifacts, and (iii) built-in extensibility so new models, metrics, and stages can be added with minimal integration effort. Existing tools rarely deliver all three for machine learning in general and localization workflows in particular.

In this paper we introduce LOCALIZE, a low-code, configuration-first framework for radio localization in which 1) experiments are declared in human-readable configuration, 2) a workflow orchestrator runs standardized pipelines from data preparation to reporting, and 3) all artifacts, such as datasets, models, metrics, and reports are versioned. The pre-configured, versioned datasets reduce initial setup and boilerplate speeding up model development and evaluation. The design, with clear extension points, let experts add components without reworking the infrastructure. In a qualitative comparison and a head-to-head study against a plain Jupyter notebook baseline, we show that the framework reduces authoring effort while maintaining comparable runtime and memory behavior. Furthermore, using a Bluetooth low-energy dataset we show that scaling across training data (1× to 10×) keeps orchestration overheads bounded as data grows. Overall, the framework makes reproducible machine-learning-based localization experimentation practical, accessible, and extensible.

CCS Concepts: • **Software and its engineering** → *Application specific development environments.*

Additional Key Words and Phrases: localization, machine learning, reproducibility, workflow orchestration, low-code

ACM Reference Format:
Tim Strnad, Blaž Bertalanič, and Carolina Fortuna. 2025. A Configuration-First Framework for Reproducible, Low-Code Localization. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (October 2025), 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Background and motivation. Location-based services (LBS) underpin navigation, logistics, social applications, entertainment, and healthcare, and are currently largely powered by Global Navigation Satellite Systems (GNSS, 'GPS') [13]. Although recent advances have improved the performance of these systems, they still have clear limitations, such as in high-precision applications, complex outdoor environments, and indoor settings [23]. This motivates radio-based localization that can operate where GNSS degrades or is unavailable. Historically, localization relied on analytical models and signal geometry. With growing application demands and environmental complexity, localization research is

Authors' Contact Information: Tim Strnad, Jožef Stefan Institute, Department of Communication Systems, Ljubljana, Slovenia; Blaž Bertalanič, blaz.bertalanic@ijs.si, Jožef Stefan Institute, Department of Communication Systems,, Ljubljana, Slovenia; Carolina Fortuna, carolina.fortuna@ijs.si, Jožef Stefan Institute, Department of Communication Systems, Ljubljana, Slovenia.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

shifting from analytical, model-based methods to data-driven machine learning approaches [22] that learn mappings directly from data, fuse heterogeneous signals, and adapt to device and environment variation.

Gaps in current practice. Moving from analytical to machine learning approaches has shown promise in many settings, but raises practical challenges for everyday research. The complexity and heterogeneity of ML pipelines raise the barrier to entry. GUI/low-code platforms reduce coding effort, but can constrain advanced experimentation and fine-grained control [7, 25]. Conversely, code-centric workflows provide maximum flexibility but, absent dedicated tooling, place the burden of reproducibility, environment control, and consistent evaluation on the researcher [5, 10–12, 18]. In many localization studies, the primary novelty is a new model or training method evaluated on public datasets [6, 22]. Onboarding data and recreating baselines becomes setup overhead rather than the research focus. The broader ML literature shows that small changes to preprocessing, hyperparameters, software versions, or random seeds can alter conclusions and hinder fair comparison [11, 18]. Beyond statistical variance, exact repeatability is fragile: hidden global state, non-deterministic operations, data-order effects, and environment or hardware differences can change outputs even when seeds and settings are fixed [5, 12, 17]. Such issues are easy to miss and hard to diagnose.

This challenge is not unique to radio localization but represents a critical bottleneck in many scientific domains now adopting machine learning [11]. Fields such as computational biology and materials science face a similar gap where the transition from traditional models to complex, data-driven pipelines strains existing research practices [11]. The lack of standardized, reproducible, and extensible workflows leads to duplicated effort, fragile results, and a high barrier to entry for domain experts who are not also software engineers [24]. Therefore, the framework presented in this paper addresses a fundamental software engineering problem: how to build research-specific development environments that enforce scientific rigor by default without sacrificing the agility needed for rapid experimentation.

Proposed approach and contributions. To enable the development of high-quality generic models, the wireless localization research community could benefit from a framework enabling: (i) reproducibility by default, (ii) ease of use (low-code with minimal boilerplate), and (iii) flexibility that is built in, so advanced experimentation can be conducted without fighting the tool-chain. While many platforms address ease of use or extensibility, few make reproducibility the default alongside both. This missing combination is the focus of our work. This paper introduces LOCALIZE [1], a low-code, configuration-first framework that makes ML-based radio localization practical, accessible, and reproducible by reducing initial effort, enabling novice and expert use, and supporting expert extensibility while promoting sound experimental practices. The contributions of this paper presents are as follows:

- We introduce LOCALIZE, a low-code, configuration-first framework that bridges ease of use with scientific rigor by making reproducibility its default operating mode, integrating version control, execution isolation, and transparent artifact tracking.
- We design LOCALIZE to enforce consistent, comparable evaluation by applying a standardized set of metrics and reporting procedures across all methods and datasets, eliminating glue code and improving the credibility of results.
- We propose and validate a multi-tier workflow that lowers the entry barrier for novices, who can run experiments solely via configuration, while providing clear extension points for experts to add new models or stages with minimal integration overhead.

[1] https://github.com/sensorlab/localize

- Through a direct comparison with a Jupyter notebook baseline, we demonstrate that the framework substantially reduces authoring effort for common experimental changes, while performance benchmarks confirm it adds no significant runtime or memory overhead.

In the remainder of this paper, we review related work in Section 2, outline the problem and requirements in Section 3, describe the proposed framework in Section 4, present a reference implementation in Section 5, and evaluate it against established platforms in Section 7. Finally, Section 9 concludes the paper.

2 Related Work

In this section, we first discuss the well-documented challenge of reproducibility in machine learning, then analyze how general-purpose platforms address these three goals, and conclude by positioning our framework relative to other specialized efforts.

2.1 The Challenge of Reproducibility in Machine Learning

A consistent lesson from the literature is that ML results can be sensitive to seemingly minor choices, so reproducibility requires attention to both variability and exact repeatability. Bouthillier *et al.* [5] quantify sources of variability in benchmarks and show that initialization, data sampling and order, and hyperparameter settings all affect outcomes. They recommend practices such as repeated runs and clearer reporting to separate algorithmic gains from noise. Building on this empirical picture, Henderson *et al.* [12] demonstrate that non-determinism in benchmark environments and implementation details can materially change results, arguing for stricter methodology and documentation. Collectively, these studies establish that achieving computational reproducibility requires meticulous control over the entire experimental pipeline.

In response, subsequent work outlines reporting and release requirements. Heil *et al.* [11] introduce tiered levels of computational reproducibility (bronze/silver/gold) with concrete requirements covering code, data, captured environments, and automation. At the community level, Pineau *et al.* [18] describe the NeurIPS reproducibility program, checklists, a code policy, and a community challenge, which raised expectations for transparency across submissions.

Despite these advances, practical support in general ML platforms remains uneven. Gundersen *et al.* [10] report that reproducing published results often requires non-trivial user effort. This suggests a clear need for research workflows that embed controls, documentation, and repeatable execution as default, integral features rather than optional, post-hoc additions.

2.2 General-Purpose Experimentation Platforms

Several general-purpose platforms aim to simplify ML experimentation, but they typically fail to resolve the tension between usability, reproducibility, and extensibility, forcing researchers to make compromises. For example, graphical tools like WEKA [25] and visual programming environments like Orange3 [7] excel at lowering the barrier to entry in machine learning research in different domains. They allow for rapid prototyping without extensive coding. However, they externalize the burden of reproducibility, as rigorously capturing an end-to-end experiment, including precise library versions and sources of non-determinism, requires manual effort outside the core workflow.

Declarative, code-centric platforms like Ludwig [15] advance this goal by using configuration files to define deep learning experiments. This approach streamlines workflows and supports extensibility. While powerful for neural

networks, it is not primarily designed for the classical machine learning pipelines common in localization studies and, similar to graphical tools, leaves environment and data versioning as an external concern.

These platforms assist with parts of the problem—addressing low-code use or standardizing certain workflows, but they do not jointly provide a solution that is reproducible by default, accessible via a low-code interface, and flexible enough for advanced experimentation.

2.3 Other Related Efforts

Specialized tools also contribute valuable insights, typically focusing on specific domains or aspects of experimental management. Some efforts focus on automating workflows for a specific domain, such as CSI fingerprinting localization [21] or multivariate time-series forecasting [26], that package common data transforms and models. While these reduce engineering for their domains, they do not generalize readily across methods or guarantee environment capture and data versioning by default.

Libraries such as MLXP [1] focus on lightweight, code-centric management of configurations, runs, and results to reduce overhead and encourage replicable practice. However typically rely on external mechanisms for full environment capture or dataset and and provenance management versioning. Considering this, exact replay remains a user-assembled composition of tools.

As highlighted in related work, there is a recognized need for reproducibility, usability, and flexibility in research. Existing approaches make progress in specific areas—such as task, specific automation, task libraries, or experiment management, but they tend to address these aspects only in isolation. In contrast, our work is designed to integrate all three goals: reproducibility by default, ease of use, and flexibility for advanced experimentation.

3 Problem statement

This work aims to design and build a *reproducible, low-code experimentation suite* for radio localization without precluding advanced experimentation. Today, no widely used approach makes reproducibility the default while also keeping implementation effort low and preserving a clear path to customization. In Section 1, we set three goals: (i) reproducibility by default, (ii) ease of use (low-code with minimal boilerplate), and (iii) easy customization. Each goal carries concrete implications for how the system should behave.

First, *reproducibility by default* requires more than fixed seeds. Execution needs to be isolated so hidden state does not leak across stages. Code, configurations, data, and environments should be tracked together. Metrics and artifacts must be recorded automatically so the same inputs yield the same outputs. If conclusions are to be consistent across studies, evaluation must be repeatable as well, which calls for a common way to compute and report metrics.

Second, to achieve *ease of use*, complete experiments, data preparation, feature steps, models, training, and evaluation are driven by configuration rather than ad-hoc scripts. Routine changes should not require new code. Because much localization work evaluates new models on public datasets, reducing setup work matters.

Third, *easy customization* means the framework is built with extensibility in mind. Adding a model, metric, dataset, or stage should be straightforward and should automatically benefit from the same orchestration, tracking, and reporting as built-in components.

We formalize these implications as the following requirements:

R1: Low-code operation. Complete experiments (data preparation, model selection, training, evaluation) are defined in configuration and executed consistently, reducing boilerplate and lowering the entry barrier.

R2: Reproducibility by default. Under the same inputs and settings, runs repeat exactly. Code, configurations, data, and environments are versioned together. Execution is isolated. Metrics and artifacts are recorded automatically.

R3: Common evaluation layer. The same metrics, reports, and artifact capture are applied across methods and datasets so studies are directly comparable and conclusions remain consistent.

R4: Built-in extensibility. Unsupported models, metrics, datasets, or stages can be added with limited effort. Extensions are designed to plug in cleanly and inherit orchestration, tracking, and reporting without extra work.

R5: Fast setup and iteration. Time-to-first-run and time-to-modify should be short. The framework should minimize repetitive setup and make small changes cheap, while keeping choices explicit and reproducible.

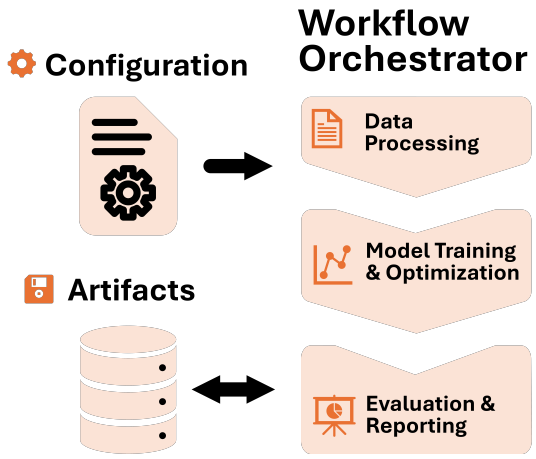

Fig. 1. Outline of the proposed architecture consisting of three components: configuration, artifacts, workflow orchestrator

4 Proposed framework

To address the problem stated in Section 3, we propose a low-code framework LOCALIZE for radio localization that makes reproducible experimentation the default while preserving a clear path to advanced use. The design maps to the requirements in Section 3: *R1: low-code operation*, *R2: reproducibility by default*, *R3: a common evaluation layer*, *R4: built-in extensibility*, and *R5: fast setup and iteration.*

4.1 Design overview

As shown in Fig. 1, the framework is organized around three cooperating parts. *Configuration* files describe experiments in human-readable form: datasets, stages, models, training settings, and evaluation criteria. This way users can run and iterate with minimal code, satisfying R1 and R5. A *workflow orchestrator* reads that configuration, resolves stage dependencies, and executes the pipeline from data preparation through reporting, applying the same procedures across methods and isolating runs fulfilling R2 and R3. An *artifacts* subsystem records per-run inputs, logs, and outputs at stage boundaries and exposes them for reuse and enabling exact reruns, fulfilling the R2 and R3. Together, these parts keep day-to-day work low code while retaining a clear path to extension, satisfying R4.

Configuration. Configuration is the primary interface to the framework. Users specify stages, datasets, models, search spaces, evaluation metrics *etc*. in a single place. This separation of specification from execution lowers boilerplate, increases modularity while allowing new components to be seamlessly integrated, makes changes easy to track and share, and ensures that the exact settings used for a result are captured alongside the code, satisfying requirements R1, R2, and R4 .

Workflow orchestrator. The workflow orchestrator executes stages according to the configurations. It resolves dependencies and determines the execution order, then runs each stage for every required combination of inputs and parameters in an isolated process to prevent hidden state and routes outputs to downstream stages. Working with the *Artifacts* component, it tracks the inputs, parameters, and outputs of each execution so that identical work is served from cache rather than recomputed. It remains stage-agnostic and therefore implements no stage logic and simply executes the configured plan, satisfying requirements R1 and R4. However, it requires each stage to guarantee deterministic behavior under fixed seeds and settings. System-level reproducibility follows from versioned configuration and code, end-to-end artifact tracking, per-execution isolation, and deterministic stage behavior, fulfilling the criteria R2.

Experiment pipeline design. For localization research, a lean pipeline is preferable to full MLOps stacks that include data collection, serving, and monitoring. Our goal is to support rapid, comparable experiments, not deployment. Splitting the workflow into *data preparation*, *model training and optimization*, and *evaluation and reporting* creates a clear boundary between dataset-specific logic and method-agnostic logic. This separation lowers routine effort fulfilling R1, keeps comparisons fair thus satisfying R3, and lets researchers change preprocessing, swap models or search spaces, or adjust reporting without entangling the other stages. The orchestrator reuses cached artifacts so only the changed stage and its dependents are recomputed (R4), while the orchestrator and artifacts together provide reproducible execution by default (R2).

Data preparation produces an analysis-ready dataset from raw inputs by handling parsing, cleaning, optional feature construction, and the materialization of reusable splits. Grouping these operations in one stage keeps dataset logic local and creates a stable prepared dataset for downstream stages, making it easier to change out datasets and reducing boilerplate, fulfilling the R1 and R2 requirements.

Model training and optimization optimizes the models on their hyperparameter search spaces as declared in the configuration. Because this stage consumes the prepared data and splits, training is comparable across methods and easy to extend with additional models or hyperparameter combinations without altering the surrounding pipeline, satisfying the R3 and R4 requirements.

Evaluation and reporting aggregates predictions across folds and models, computes a standardized set of metrics, and generates the figures and reports used in the results. This ensures that all methods are scored and reported the same way and that reruns reproduce both numbers and artifacts, reinforcing consistency and comparability, thus ensuring compliance with R2 and R3.

When a stage and its inputs have not changed, the orchestrator reuses the recorded outputs of that stage. This cache-aware execution accelerates iteration while preserving traceability, thus fulfilling requirements R2, R4, and R5.

Artifacts. Artifacts capture the intermediate and final results of each experiment: prepared datasets and features, trained model versions, metrics, and reports. Artifacts are versioned and stored together with the configuration that produced them, so a run can be repeated with the same inputs and settings. This organization supports exact reruns and retrospective analysis without re-implementing earlier steps, satisfying the R2 and R3.

Preconfigured datasets. To address requirement *R5*, the framework provides a small set of versioned, preconfigured datasets and experiment templates that standardize data access and baseline pipelines, and enable testing new models with minimal setup and repetitive boilerplate.

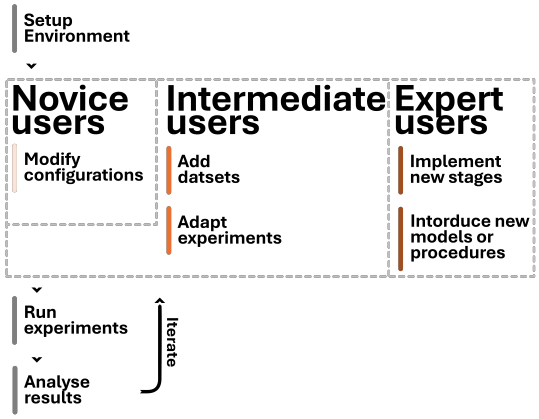

Fig. 2. User workflow supported by the framework, separated by experience level. Novice users modify configurations, intermediate users adapt experiments or add datasets, expert users implement new stages or models.

4.2 User types

As illustrated in Fig. 2, the framework accommodates users at different experience levels. All users follow a similar workflow and inherit reproducible-by-default operation. They differ in where changes are made.

Novice users run end-to-end experiments by editing configuration files and invoking the orchestrator, without writing any code. This includes switching datasets or splits, choosing models and hyperparameter search grids, and enabling or disabling metrics and reports entirely through configuration.

Intermediate users remain configuration-centric while introducing small code changes where needed. They may add additional datasets and write their corresponding preparation, featurization, and splitting code, or adjust experiments by modifying or adding new stages to fit their needs.

Expert users add new capabilities by writing new stages, custom models, new metrics, integrating external libraries, or additional report generation logic. This new functionality can be used by novice and intermediate users immediately through configuration.

These usage modes interoperate by design: experts contribute reusable components, intermediates compose them into experiments, and novices explore configurations safely. Because execution is isolated and deterministic, and artifacts are versioned for every run, ensuring everything remains reproducible and auditable. This aligns with our requirements defined in Section 3, specifically R1, R2, R3, and R4.

5 Reference implementation

This section details the implementation details of the LOCALIZE framework defined in Section 4 and the design alternatives we considered.

5.1 Backbone: orchestration, versioning, and configuration

First, we considered the approach to orchestration, versioning, and configuration as they are interdependent and dictate the design for the rest of the framework. Workflow schedulers such as Airflow, Prefect, and Luigi are great for production jobs, but they add servers, queues, and operational overhead we do not need for local research runs. General DAG engines like Snakemake and Kedro are strong at rule-based orchestration, yet their default models rely on timestamps or project scaffolding and require extra policy to get fresh-interpreter isolation and results caching, adding complexity. We therefore adopt DVC [2] paired with Git [3] for versioning. Each experiment is a self-contained directory that includes the configuration files. Running `dvc repro` executes the experiment end-to-end as declared in the configuration. DVC versions and resolves data dependencies, and if a stage's inputs and parameters are unchanged, it serves the cached outputs instead of recomputing them. Because each stage is executed as a shell command, it runs as a separate process, thereby enforcing isolation. Paired with Git for source and configuration history, this gives us exact reruns and clear diffs of both code and settings.

Each experiment directory contains a `dvc.yaml` that lists the stages and their dependencies, including the datasets, and a `params.yaml` that captures dataset splits, models or model architectures, hyperparameter search spaces, evaluation metrics, and other settings. This makes each experiment completely modular, making it easy for expert users to extend functionality.

5.2 Language and evnironment

We considered Python, R, Java, and MATLAB for the core runtime. Although MATLAB still appears in localization work [14], recent studies and toolchains lean heavily on Python [19, 20]. We chose **Python** for its mature ML ecosystem (NumPy/Pandas, scikit-learn, TensorFlow/Keras) and smaller coding footprint. The framework is distributed as a standard Python repository. Users clone it and create the environment from an `environment.yaml` file. Environments are managed with Conda (or Mamba), and dependencies are pinned in the environment file so installations are consistent across machines.

5.3 Implementing the three-stage design

Conceptually, the framework has three stages as depicted in Fig. 1: *data preparation*, *model training and optimization*, and *evaluation and reporting*. In the reference implementation, we split these into narrower executable units to keep change-scopes small, lower complexity, and match the libraries used in practice.

Data preparation. We implement this as three stages to keep concerns separate and caching effective. First, the *Prepare* stage parses raw inputs, cleans malformed records, and normalizes schema into a consistent tabular form. Keeping parsing independent prevents feature edits from re-triggering heavy I/O work. Second, the *Featurize* stage constructs features and transformations. Decoupling feature construction allows rapid iteration on feature sets without touching raw data handling, making caching more effective. Third, the *Split* stage materializes the cross-validation strategy as explicit fold indices that are reused across models. This keeps comparisons fair and makes reruns exact.

Model training and optimization. The design has one training stage, but classical and neural pipelines use different runtimes and produce different artifacts. We therefore provide two executable paths and standardize their outputs. First,

[2] https://dvc.org
[3] https://git-scm.com/

Grid-search is implemented with `scikit-learn` [4] using `GridSearchCV`. It offers a stable API across many algorithms, integrates naturally with NumPy/Pandas, accepts user-supplied fold indices, and behaves deterministically under fixed seeds. Explicit grids map cleanly to YAML, which keeps experiments low-code. Alternatives such as randomized search or Bayesian optimization (Optuna, scikit-optimize, Ray Tune) can be preferable for larger search spaces, but usually, for classical ML models, hyperparameter grids aren't prohibitively large. As a sensible default, and the de facto baseline, `scikit-learn` balances breadth, reproducibility, and authoring effort.

Second, *AutoML* is implemented with AutoKeras [5] on top of `Keras`/`TensorFlow`. It translates cleanly to configuration, and offers flexibility by allowing experts to define new blocks or expand search spaces using `Keras` and `KerasTuner`, with minimal integration effort. General AutoML systems like Auto-PyTorch, AutoGluon, and H2O don't offer the same level of simplicity and flexibility. AutoKeras, therefore, offers the best fit for a configuration-first neural workflow.

Both paths produce the same downstream artifacts: per-fold predictions, fold-wise and aggregate metrics, the selected model artifact (pickle or checkpoint), and a compact run summary. This keeps scoring and reporting identical across methods and lets users switch between classical and ML models without touching the reporting stage.

Evaluation and reporting. Evaluation runs inside each training backend because data structures and APIs differ. Outputs are then normalized to a shared schema. A separate *Report* stage aggregates results across runs and methods into tables and figures. Reporting never re-trains models and uses only recorded artifacts, which makes modifying reports fast and reliable.

5.4 Reproducibility

To support *reproducibility by default*, Git is used for source and configuration versioning, and DVC for end-to-end artifact tracking and isolation for each run of every stage by starting a new interpreter. Each stage is implemented to produce deterministic results, and any new stages added are required to do the same.

5.5 Preconfigured datasets

The repository includes 5 preconfigured datasets and their corresponding dataset-specific stages (Prepare, Featurize, and Split with several different cross-validation strategies) and experiment templates: Lumos5G [16] and UMU datasets for cellular based localization [3], LOG-a-TEC dataset for Bluetooth low-energy based localization [4], CTW 2019 [2] and CTW 2020 [8] datasets for Wifi based localization . For experimenting with any of these datasets, a user can copy a template, edit `params.yaml`, and run a study without writing glue code or configuration boilerplate.

6 Evaluation methodology

In this section we elaborate on the methodology to evaluate the proposed LOCALIZE framework introduced in Section 4 along the following four aspects: (i) cross-tool capabilities using a qualitative comparison, (ii) developer effort by comparing our framework with a plain Jupyter notebook under controlled edits, (iii) runtime and resource usage by pipeline stage, and (iv) scalability as dataset size increases.

[4] https://scikit-learn.org
[5] https://autokeras.com

6.1 Cross–platform qualitative comparison

As the first step in the evaluation of LOCALIZE, we identify the closest platforms that are strong candidates to enable similar functionality as the proposed framework and dive deeper to understand specific differences. We identify WEKA [25], Orange3 [7], Ludwig [15], and Jupyter notebooks in JupyterLab [9] as the closest relevant platforms.

Table 1. Requirement definitions (R1–R5) used in the qualitative comparison. Wording is broadened from Section 3 to facilitate a fairer cross-platform comparison

Item	Description
Primary focus	Main application domain or use case of each platform. Descriptive only; not scored.
R1: Low-code operation	Barrier to entry for running and iterating on end-to-end localization experiments via configuration (or GUI) rather than code; minimal setup and boilerplate. *"Very high"* = little/no coding; *"Very low"* = substantial coding.
R2: Reproducibility by default	Ability to reproduce identical results *by default*: versioned code/config/data, captured environments, controlled randomness, isolated execution, and automatic artifact/metrics recording. *"Very high"* = full reproducibility guaranteed without extra effort; *"Very low"*= reproducibility depends entirely on manual intervention.
R3: Common evaluation layer	Built-in, reusable evaluation across methods and datasets (metrics, reporting, artifact schema) without bespoke glue code. *"Very high"* = rich, standardized evaluation layer that works across models/datasets out of the box; *"Very low"* = no shared evaluation.
R4: Built-in extensibility	Effort to add or replace models, metrics, datasets, or stages through configuration or small adapters without modifying core internals. *"Very high"* = seamless extensibility via lightweight configuration; *"Very low"* = extensions require invasive modifications to the framework's internals.
R5: Fast setup and iteration	The effort required for initial setup and to iterate on experiments. *"Very high"* = ready-to-use environment with minimal dependencies; *"Very low"* = heavy installation burden, long configuration times, and slow iteration cycles.

We evaluate each platform on five requirement-oriented criteria (R1–R5) plus a descriptive *Primary focus*, using the definitions in Table 1. To facilitate a fair cross-platform assessment (including GUI-first tools), the wording in Table 1 is broadened relative to Section 3. For each item, we assign an ordinal grade from *"Very low"* to *"Very high"*, based on official documentation and hands-on use. When a capability is not available out of the box but can be reasonably achieved via supported extensions or common external tools, the grade is capped at *"Medium"*. This comparison is intended to situate LOCALIZE among existing solutions and to assess, at a high level, how well each platform aligns with the requirements that motivate our framework.

6.2 Materials

All the quantitative experiments were carried out partly on the UMU cellular localization [4] dataset and primarily on the LOG-a-TEC Bluetooh localization dataset [3].

All experiments ran on a Kubernetes cluster with two AMD EPYC 75F3 CPUs (128 hardware threads in total) and 1 TB RAM. Each user executed inside an isolated JupyterLab container with access to the full CPU and memory resources. The software stack was: Python 3.11.13, scikit-learn 1.4.2, TensorFlow 2.16.2, Keras 3.5.0, AutoKeras 2.0.0, DVC 3.62.0, Git, Conda, and `psutil` 5.9.0. All computations used CPU only.

6.3 Lines of code study

We implemented functionally equivalent baseline pipelines in both environments. Both produced identical outputs and were initially composed of the stages *Prepare*, *Featurize*, *Split*, and *Grid-search* for the UMU dataset, implemented as they were described in Section 5. The pipelines initially used the `LinearRegression` model from scikit-learn, a two-value hyperparameter grid for `fit_intercept`, evaluated with 5-fold `KFold` and `rmse`. We then introduced four incremental changes:

- Change 1: replace the estimator and add a second one,
- Change 2: switch the dataset from UMU to LOG-a-TEC,
- Change 3: add cross-validation strategy and an extra evaluation metric,
- Change 4: append an AutoML stage using AutoKeras with 10 trials (default 1000 epochs).

For each change, we counted *lines of code (LOC) added* and *LOC removed* separately (two bars per platform), excluding blank lines, comments, and lines containing only structural tokens (*e.g.*, brackets/braces/commas).c

6.4 Runtime and resource profiling

We profiled the final pipelines from the LOC study to determine whether the framework introduces measurable overhead. Specifically, we evaluated how many resources both frameworks utilise during the execution of 5 different stages of model development: Data preparation, Data Feature extraction, data train-test split, model hyperparameter grid search, and AutoML. Executions used five worker CPU cores plus one dedicated logging core. Memory and CPU usage were sampled at 10 Hz with `psutil`. CPU time was then obtained by integrating the timeseries CPU usage data via trapezoidal integration of the samples. Wall time was measured with a high-resolution performance counter and averaged. The data was collected for five identical runs. For memory usage, we calculated the mean, max, and the first and third quartiles across all runs combined. While CPU and wall time were averaged over the five runs.

6.5 Scalability on LOG-a-TEC

To assess scalability, we reused the final pipelines from the LOC study and increased the dataset size by concatenation after the preparation stage to obtain 1×, 5×, and 10× the base LOG-a-TEC size. Statistics for memory usage and CPU times were computed with the same procedure as above.

6.6 Reproducibility measures

Configurations, code, and data versions were tracked. Each run used the same environment, inputs, and fixed seeds on both platforms and produced identical outputs. The framework executes each stage in a fresh Python interpreter (DVC behavior), which prevents hidden state from leaking across runs. In the notebook baseline, we executed each run in a fresh child process (via multiprocessing) solely to isolate interpreter state; no algorithmic parallelism or extra workers were used. Despite standard measures (fixing seeds, re-instantiating models, resetting Keras/TensorFlow sessions, and forcing garbage collection), single-interpreter execution yielded training order-dependent outputs.

Table 2. Qualitative comparison of platforms along evaluation features (definitions in Table 1).

Feature/Platform	LOCALIZE	WEKA [25]	Orange3 [7]	Jupyter notebook [9]	Ludwig [15]
Primary focus	Localization	General ML	Data mining & visualization	Experimentation & scripting	Deep learning
R1: Low-code operation	High	Very high	Very high	Very low	High
R2: Reproducibility by default	High	Low	Medium	Low	Medium
R3: Common evaluation layer	Very high	Low	Medium	Low	Medium
R4: Built-in extensibility	Very high	High	Medium	Very high	Very high
R5: Fast setup and iteration	Very high	High	Medium	High	High

7 Evaluation

In this section, we discuss the evaluation result of the LOCALIZE framework introduced in Section 4 according to the methodology elaborated in Section 6. First, we focus on the qualitative cross–platform comparison of five solutions, followed by a quantitative study against a Jupyter Notebook baseline, and concluding with a benchmark on the LOG-a-TEC dataset scaled to different sizes.

7.1 Cross–platform qualitative comparison

We evaluate five platforms, LOCALIZE, WEKA, Orange3, Jupyter notebooks, and Ludwig, against the five requirements (R1–R5) defined in Table 1, using the rubric in Section 6.1. Table 2 lists platforms across columns and requirements down the rows. Below, we explain the grades and the main takeaways for each requirement.

Primary focus (descriptive). As per first row in Table 2, LOCALIZE targets end-to-end localization workflows. WEKA emphasizes general classical ML, Orange3 focuses on visual data mining, Jupyter emphasizes code-centric experimentation, and Ludwig targets deep learning with configuration files. This establishes scope and is not scored.

R1: Low-code operation. As can be seen in second row of Table 2, both WEKA and Orange3 have a *"Very high"* score for R1, as users can assemble pipelines in a GUI with minimal coding. Jupyter has a score of *"Very low"* since routine edits require writing and maintaining scripts. LOCALIZE and Ludwig achieve a score of *"High"*, since both support configuration-first editing for common changes while allowing custom code when needed.

R2: Reproducibility by default. Analysing the third row of Table 2, it can be seen that LOCALIZE has a *"High"* score due to configuration, environments, execution isolation, and artifacts being handled as part of the default workflow. In comparison, WEKA score is *"Low"*, as it only provides limited reproducibility support, because GUI projects are not inherently versioned, and environment capture is manual. Orange3 sits in the middle at score *"Medium"*, as workflows can be saved, but data and environment versioning require external practice. Jupyter notebooks are rated as *"Low"* due to offering no built-in reproducibility support, so they depend on user discipline and external tooling for reproducibility. However, even with extensions, achieving end-to-end replicability typically requires a multi-step setup and careful consideration. Finally, Ludwig's YAML promotes consistent experiment specs, but environment and dataset versioning remain external concerns, so it achieves score rate of *"Medium"*.

R3: Common evaluation layer. LOCALIZE scores *"Very high"* due to uniform metrics, reporting, and artifact schema applied across datasets and methods. Orange3 and Ludwig are *"Medium"*. Orange3 provides evaluation widgets, yet consistent reuse across heterogeneous methods and datasets often needs add-ons or custom wiring. Ludwig supplies a strong evaluation for its deep-learning workflows, but does not generalize to classical pipelines out of the box. WEKA

and Jupyter are *"Low"* because comparable, reusable evaluation generally requires bespoke code or plug-ins. The takeaway is that LOCALIZE uniquely enforces like-for-like scoring and reports without extra glue.

R4: Built-in extensibility. Jupyter and Ludwig are *"Very high"* since they are designed for heavy customization. LOCALIZE also scores *"Very high"* by exposing clear extension points that can be referenced from configuration, so additions plug into the same orchestration and reporting. WEKA is *"High"* because it is extensible through Java APIs and plug-ins, although GUI integration can be involved. Orange3 is *"Medium"* due to the overhead of developing and maintaining PyQt widgets.

R5: Fast setup and iteration. LOCALIZE is rated *"Very high"*, as it combines ready-to-run datasets and templates with cached, stage-wise execution, so common edits avoid full reruns. Jupyter and Ludwig iterate quickly once code patterns and environments are in place and are therefore rated *"High"*. WEKA is rate *"High"* due to GUIs support of quick initial setup, although complex studies add manual steps. Orange3 is *"Medium"* since visual assembly scales less smoothly when experiments branch or require custom tooling.

Table 2 highlights a consistent trade-off. GUI-first tools make initial use easy but rely on user practice for exact repeatability and uniform scoring. Code-first tools enable deep customization and rapid iteration, but place reproducibility and evaluation consistency on the user. LOCALIZE is designed to close these gaps by making reproducibility the default, while preserving low-code operation and extensibility. In the context of localization studies, this combination makes LOCALIZE the most appropriate choice among the compared platforms.

7.2 Comparison against notebooks

The qualitative comparison covers five platforms, but the quantitative studies use a Jupyter notebook as the sole baseline. Two factors motivate this choice. First, metrics such as "lines of code added/removed" are not meaningfully comparable to GUI-first tools, such as WEKA and Orange3. Second, while Ludwig is otherwise a strong configuration-first reference point, it targets deep-learning workflows and does not provide classical machine-learning pipelines out of the box, so a fair, like-for-like comparison would either exclude classical ML or require custom adapters, undermining comparability. Within machine learning community, notebooks remain the workflow of choice for advanced experimentation because they offer maximal flexibility with minimal imposed structure. Using Jupyter as the baseline, therefore, yields a fair, code-level comparison against current practice, while the cross-platform comparison captures the broader landscape.

Lines of code comparison. Figure 3 compares edit effort for four incremental changes in LOCALIZE versus a Jupyter baseline. The x-axis enumerates Changes 1–4; the y-axis reports lines of code, counted as described in Section 6.3. For each platform and change, the upward bar shows *lines added* and the downward bar shows *lines removed*, both have their corresponding value for lines of code added/removed written on them. Two bars are plotted per platform, so additions and deletions are visible separately.

In our example looking at Figure 3, Change 1, swapping and adding an estimator, is identical on both sides with adding **+16** and removing **-4** LOC, because it involved only modifying values in dictionaries, meaning YAML and Python edits map one-to-one. Change 2, switching datasets, diverges between our framework and Notebooks. LOCALIZE mostly just involves changing the configuration references to the new preparation scripts resulting in **+29** added and **-27** removed LOC, whereas the notebook rewrites dataset-specific steps resulting in **+55** additional and **-38** removed LOC. Change 3, add a CV strategy and a metric, involves only adding the options to the configuration for LOCALIZE resulting in **+2** added and **-2** removed LOC, but the notebook needs additional control logic to handle combinations thus

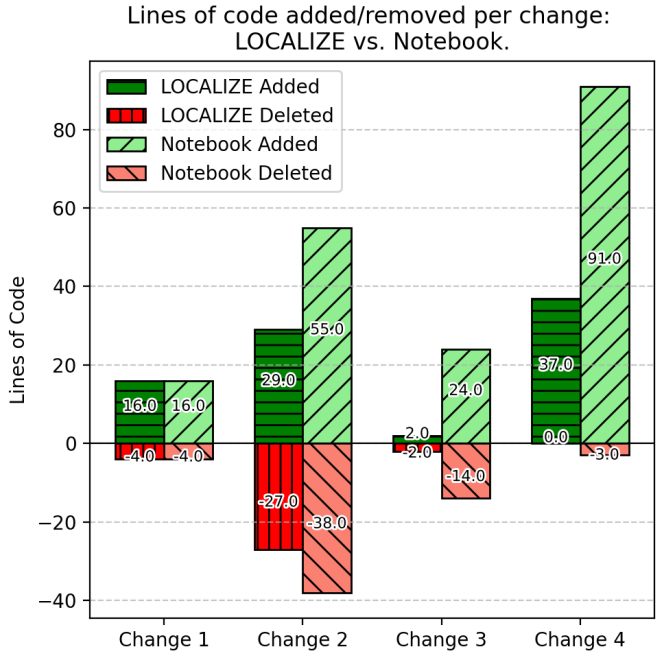

Fig. 3. Lines of code added (positive) and removed (negative) when introducing successive changes in the Framework and Notebook implementations.

resulting in **+24** additional LOC and **-14** removed. Finally, in Change 4, adding AutoML, requires moderate configuration in LOCALIZE by adding **+37** LOC, while the notebook implements a full new stage resulting in **+91** additional and **-3** removed LOC. Overall, the main pattern is clear: small edits take similar effort in both setups, but as the scope grows, the notebook requires substantially more code, while LOCALIZE remains configuration-centric.

Memory by stage. Figure 4 plots mean memory per stage, interquartile ranges (solid error bars), and the observed maxima (dotted whiskers with values). LOCALIZE uses 332, 190, and 187 MB in Prepare, Featurize, and Split, lower than the notebook's 367, 267, and 269 MB. Grid-search averages 2310 MB in LOCALIZE versus 1558 MB in the notebook due to some inefficient imports, causing fixed overhead. AutoML averages 1078 MB in LOCALIZE and 1675 MB in the notebook. The notebook's higher footprint reflects retained state across cells, whereas LOCALIZE starts AutoML in a fresh interpreter. The maxima are similar, 2597 MB for LOCALIZE during Grid-search and 2615 MB for the notebook during AutoML, so peak demand is essentially the same. Overall, LOCALIZE adds overhead in Grid-search but avoids carry-over into AutoML, resulting in no material memory overhead.

CPU and wall time by stage. Figure 5 contains five panels, one per stage. Each panel shows CPU time in core-seconds and wall time in seconds as adjacent bars. Prepare is effectively identical on both platforms at about 18.5 core-seconds and 18.5 seconds wall. Featurize and Split are sub-second with negligible differences that represent constant overhead. During Grid-search, LOCALIZE expends slightly more work, about 3218 versus 3054 core-seconds and 688 versus 675 seconds wall, because it saves all models, which the notebook environment cannot do. In AutoML, LOCALIZE

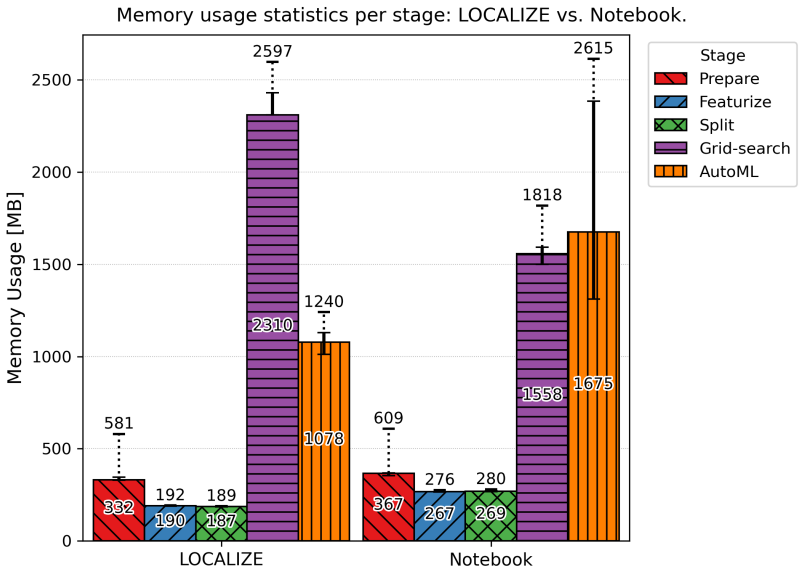

Fig. 4. Memory usage statistics per pipeline stage for LOCALIZE and the Notebook implementation. Bars show means with first/third quartiles as error bars.

completes faster in wall time at about 774 seconds compared with 823 seconds for the notebook, while CPU time remains essentially the same near 1200 core-seconds. Overall, timings are broadly comparable and we do not observe a meaningful performance penalty for LOCALIZE.

7.3 Scalability on LOG-a-TEC

Per-stage memory usage. Fig.6 shows per-stage memory usage for LOCALIZE at three dataset sizes (1×, 5×, 10×). The x-axis lists stages (*Prepare*, *Featurize*, *Split*, *Grid-search*, *AutoML*). For each stage, there are three bars, one per size, with the mean memory usage printed on each bar and first and third quartiles as error bars. The y-axis reports MB. Memory growth is stage-dependent. *Prepare* is roughly constant because scaling was done by concatenation after preparation. *Featurize* and *Split* grow near-linearly with data. *Grid-search* and *AutoML* rise with data but also reflect fixed imports and overhead, so their increases are modest relative to compute time growth.

Per-stage CPU and wall time. Fig. 7 comprises of five plots, one per stage. Within each plot, results are grouped by dataset size: 1×, 5×, 10×. For each size there are two vertical bars: CPU time (core-seconds) and wall time (seconds), computed with the same procedure as described in Section 6.4. Times increase with data, but growth is sublinear overall due to bounded orchestration costs and other overhead. Stages with little overhead (*Featurize*, *Split*) show near-linear scaling, while training stages absorb most of the growth.

Aggregate CPU and wall time. Table 3 aggregates CPU time (core-seconds) and wall time (seconds) across all stages for 1×, 5×, and 10× LOG-a-TEC. Scaling from 1× to 5× increases CPU time by 3.82× and wall time by 3.28×. At 10× the factors are 6.09× for CPU and 4.99× for wall, confirming sublinear growth. Sublinearity indicates that fixed orchestration overheads remain bounded as data scale increases, while compute-heavy training dominates the incremental cost.

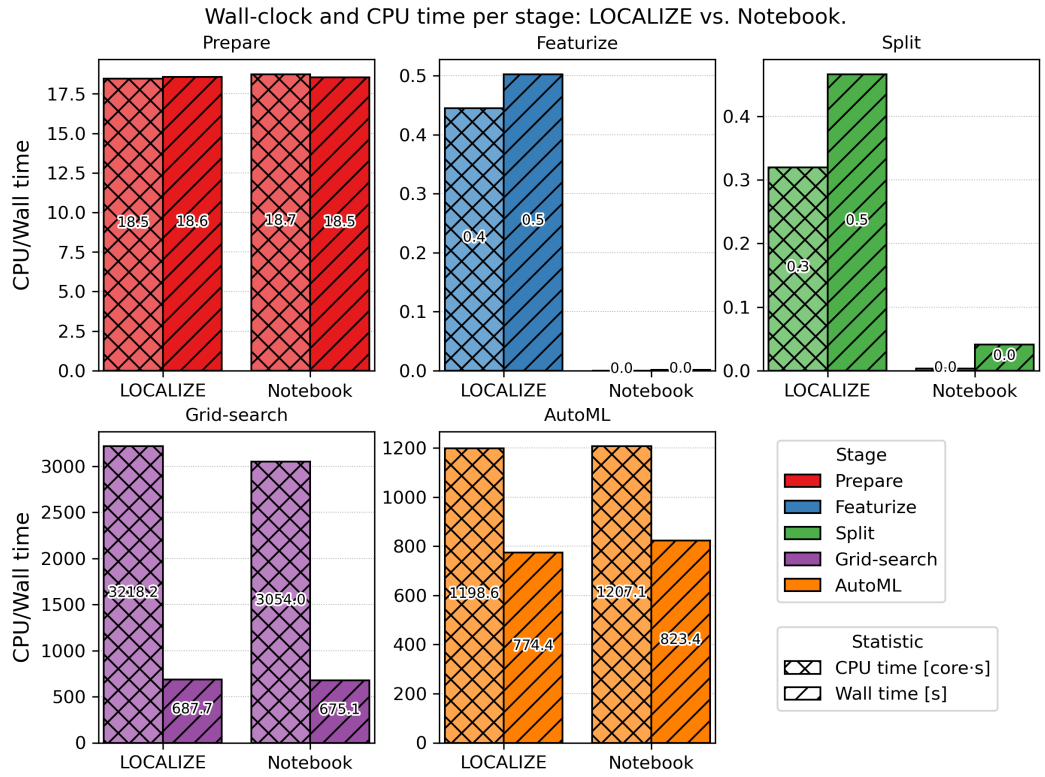

Fig. 5. Wall-clock and CPU time per pipeline stage for LOCALIZE and the Notebook implementation.

Table 3. CPU and wall time for multiples of the LOG-a-TEC dataset (1×, 5×, 10× the base size).

Metric	1× base	5× base	10× base
CPU time [cores·s]	4224	16147	25711
Wall time [s]	1347	4414	6714

Because scaling was performed by concatenating after *Prepare*, that stage's contribution stays effectively constant across sizes.

In summary, the scalability evaluation demonstrates that the framework is well-suited for larger datasets. Aggregate results show that both CPU and wall times scale sublinearly as the dataset size increases from 1x to 10x. This efficiency stems from the framework's design, where fixed orchestration overheads remain bounded, ensuring their relative impact diminishes as data volumes grow. Consequently, performance is dictated by the computationally intensive training stages rather than by limitations of the orchestration logic, confirming the framework's robust scalability.

8 Limitations and Future Work

While LOCALIZE successfully meets its design goals, certain limitations present clear opportunities for future development. The framework is primarily designed and optimized for tabular data. Although its configuration system can

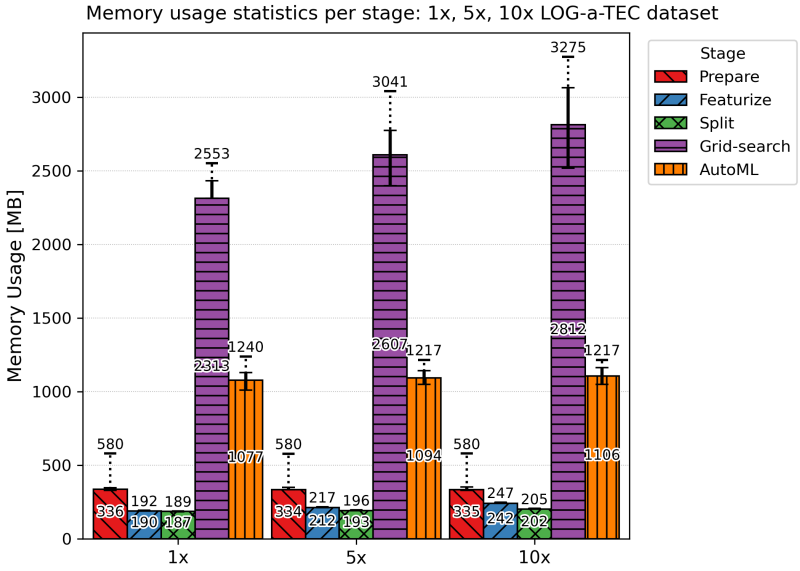

Fig. 6. Memory usage per stage for LOCALIZE on 1×, 5×, and 10× LOG-a-TEC.

reference other data types, such as the tensors in the CTW 2019 dataset, the internal pipeline components are not guaranteed to be compatible. Future work should focus on implementing modular, modality-specific data handlers to natively support a wider range of inputs. Furthermore, the system's automation is currently limited to hyperparameter optimization. While AutoKeras can tune a model's hyperparameters, the neural network architecture itself remains static. Integrating Neural Architecture Search (NAS) techniques would be a significant extension, enabling the automatic discovery of novel model topologies.

From a software architecture perspective, the implementation is tightly coupled to its current machine learning backends, specifically scikit-learn with GridSearchCV and TensorFlow via AutoKeras. This design makes it challenging to incorporate alternative libraries, such as PyTorch, or more flexible optimization frameworks such as Optuna and Ray Tune, which were considered during development. A key direction for future work would involve refactoring the training and evaluation logic into more granular, interchangeable components. This would allow users to seamlessly plug in different backends and optimizers, greatly expanding the framework's flexibility and long-term utility.

9 Conclusion

We set out to reduce the practical complexity of ML-based radio localization by making reproducibility the default, lowering routine coding effort, and supporting easy extensibility. LOCALIZE delivers this with a configuration-first design: experiments are declared in human-readable files, a workflow orchestrator enforces a consistent execution, and an artifact layer captures everything needed for exact reruns and fair comparison. Ready-to-run datasets and templates further reduce non-central setup so iteration stays fast.

Across a cross-platform comparison, LOCALIZE is the only option that consistently meets the combined needs of low-code use, default reproducibility, a common evaluation layer, and built-in extensibility. In a head-to-head study versus a plain Jupyter notebook workflow, small edits (e.g., swapping a model or dataset) required similar effort, while

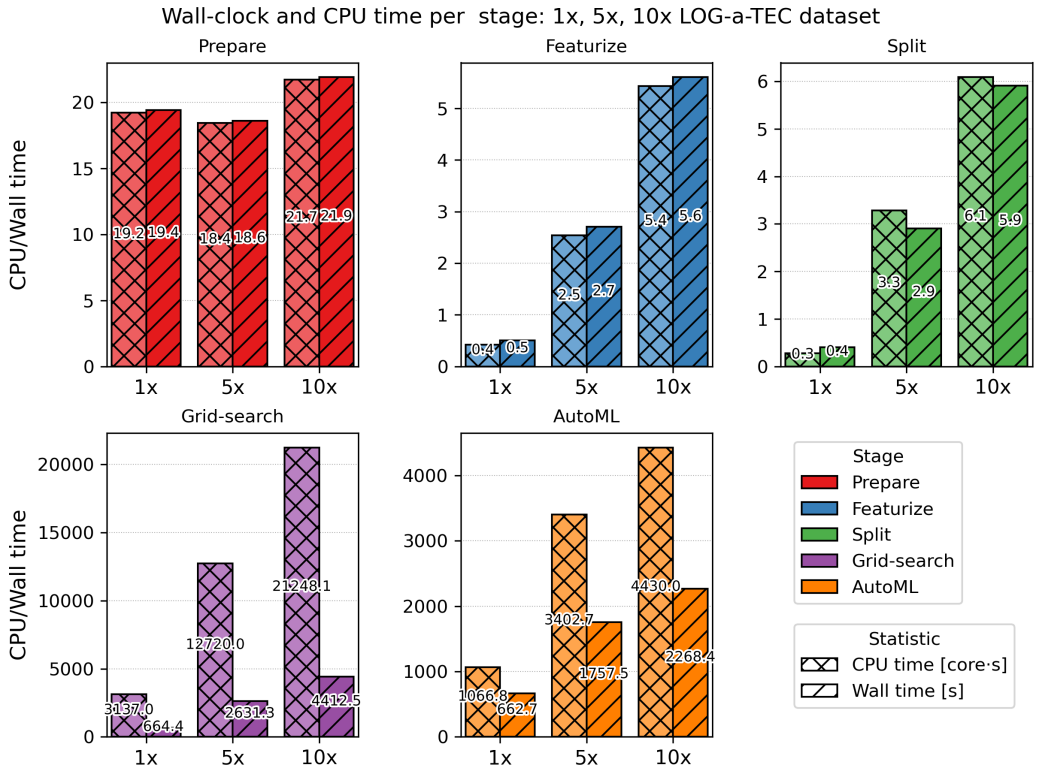

Fig. 7. Wall-clock and CPU time per stage for LOCALIZE on 1×, 5×, and 10× LOG-a-TEC.

broader changes (adding validation strategies/metrics or AutoML) demanded substantially fewer edits with LOCALIZE. Runtime results show no material penalty: stage-wise memory and time closely match the notebook baseline, and scaling LOG-a-TEC from 1× to 5× and 10×increased aggregate CPU time by 3.82×and 6.09× and wall-clock by 3.28×and 4.99× respectively, showing that fixed orchestration overheads remain bounded and diminish in relative impact as data grow.

While LOCALIZE is implemented for radio localization, its core principles offer a generalizable blueprint for building reproducible, low-code research frameworks in other scientific fields. Our configuration-first design, strict separation of concerns between experimental stages, and built-in artifact versioning are software engineering patterns that can be adapted to any domain struggling with the complexity of ML pipelines. This work can be seen as a case study in creating specialized, 'reproducible-by-design' development environments that bridge the gap between ease of use and the rigorous methodology demanded by modern computational science.

Overall, the configuration-first approach makes localization experimentation practical, accessible, and reproducible without giving up advanced use. Future work will aim to lower any performance overheads, expand the set of preconfigured datasets, further improve and standardize extensibility, and enrich reporting and diagnostics.

Acknowledgments

This work was supported by the Slovenian Research Agency under grant P2-0016. We would like to thank Dr. Gregor Cerar for helping with the development of the early version of the software prototype.

References

[1] Michael Arbel and Alexander Zouaoui. 2024. MLXP: A framework for conducting replicable experiments in Python. In *Proceedings of the 2nd ACM Conference on Reproducibility and Replicability (ACM REP '24)*. Association for Computing Machinery, New York, NY, USA, 134–144. doi:10.1145/3641525.3663648

[2] Maximilian Arnold, Jakob Hoydis, and Stephan ten Brink. 2019. Novel Massive MIMO Channel Sounding Data applied to Deep Learning-based Indoor Positioning. In *SCC 2019; 12th International ITG Conference on Systems, Communications and Coding*. IEEE, Hamburg, Germany, 1–6. doi:10.30420/454862021

[3] Rodrigo Asensio-Garriga, Anthony Joel Pogo Medina, Gonzalo Alarcon-Hellin, Luis Bernal Escobedo, Ramon Sanchez-Iborra, and Antonio Skarmeta Gómez. 2025. University of Murcia 5G Dataset 1. https://zenodo.org/records/15516876

[4] Blaž Bertalanič, Grega Morano, and Gregor Cerar. 2022. LOG-a-TEC Testbed outdoor localization using BLE beacons. In *2022 International Balkan Conference on Communications and Networking (BalkanCom)*. IEEE, 115–119.

[5] Xavier Bouthillier, Pierre Delaunay, Mirko Bronzi, Assya Trofimov, Brennan Nichyporuk, Justin Szeto, Nazanin Mohammadi Sepahvand, Edward Raff, Kanika Madan, Vikram Voleti, Samira Ebrahimi Kahou, Vincent Michalski, Tal Arbel, Chris Pal, Gael Varoquaux, and Pascal Vincent. 2021. Accounting for Variance in Machine Learning Benchmarks. *Proceedings of Machine Learning and Systems* 3 (March 2021), 747–769. https://proceedings.mlsys.org/paper_files/paper/2021/hash/0184b0cd3cfb185989f858a1d9f5c1eb-Abstract.html?utm_source=chatgpt.com

[6] Daoud Burghal, Ashwin T. Ravi, Varun Rao, Abdullah A. Alghafis, and Andreas F. Molisch. 2020. A Comprehensive Survey of Machine Learning Based Localization with Wireless Signals. *ArXiv* (Dec. 2020). arXiv:2012.11171 doi:10.48550/arXiv.2012.11171 arXiv:2012.11171 [eess].

[7] Janez Dems ˇar, Janez Demsar, Toma ˇz Curk, Tomaz Curk, Ales ˇ Erjavec, and Ales Erjave. 2013. Orange: Data Mining Toolbox in Python. *Journal of Machine Learning Research 14* (2013).

[8] Marc Gauger, Maximilian Arnold, and Stephan ten Brink. 2020. Massive MIMO Channel Measurements and Achievable Rates in a Residential Area. In *WSA 2020; 24th International ITG Workshop on Smart Antennas*. IEEE, Hamburg, Germany, 1–6. https://ieeexplore.ieee.org/abstract/document/9097159

[9] Brian E. Granger and Fernando Pérez. 2021. Jupyter: Thinking and Storytelling With Code and Data. *Computing in Science & Engineering* 23, 2 (March 2021), 7–14. doi:10.1109/MCSE.2021.3059263

[10] Odd Erik Gundersen, Saeid Shamsaliei, and Richard Juul Isdahl. 202. Do machine learning platforms provide out-of-the-box reproducibility? *Future Generation Computer Systems* 126 (202), 34–47.

[11] Benjamin J Heil, Michael M Hoffman, Florian Markowetz, Su-In Lee, Casey S Greene, and Stephanie C Hicks. 2021. Reproducibility standards for machine learning in the life sciences. *Nature Methods* 18, 10 (2021), 1132–1135.

[12] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2019. Deep Reinforcement Learning that Matters. *ArXiv* (Jan. 2019). doi:10.48550/arXiv.1709.06560 arXiv:1709.06560 [cs].

[13] Haosheng Huang and Song Gao. 2018. Location-Based Services. In *The Geographic Information Science & Technology Body of Knowledge*, John P. Wilson (Ed.). Vol. 2018(Q1). The Geographic Information Science & Technology Body of Knowledge. doi:10.22224/gistbok/2018.1.14

[14] Katarzyna Kosek-Szott, Szymon Szott, Wojciech Ciezobka, Maksymilian Wojnar, Krzysztof Rusek, and Jonathan Segev. 2025. Indoor Positioning with Wi-Fi Location: A Survey of IEEE 802.11mc/az/bk Fine Timing Measurement Research. *arXiv preprint arXiv:2509.03901* (2025). https://arxiv.org/abs/2509.03901 Survey includes Matlab 802.11az implementation and MATLAB 2024a positioning toolbox evaluation.

[15] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. 2019. Ludwig: a type-based declarative deep learning toolbox. doi:10.48550/arXiv.1909.07930 arXiv:1909.07930 [cs].

[16] Arvind Narayanan, Eman Ramadan, Rishabh Mehta, Xinyue Hu, Qingxu Liu, Rostand A. K. Fezeu, Udhaya Kumar Dayalan, Saurabh Verma, Peiqi Ji, Tao Li, Feng Qian, and Zhi-Li Zhang. 2020. Lumos5G: Mapping and Predicting Commercial mmWave 5G Throughput. In *Proceedings of the ACM Internet Measurement Conference*. ACM, Virtual Event USA, 176–193. doi:10.1145/3419394.3423629

[17] Hung Viet Pham. 2020. Problems and Opportunities in Training Deep Learning Software Systems: An Analysis of Variance. *IEEE/ACM* (2020).

[18] Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Lariviere, Alina Beygelzimer, Florence d'Alche Buc, Emily Fox, and Hugo Larochelle. 2021. Improving Reproducibility in Machine Learning Research. *Journal of Machine Learning Research* 22 (2021), 1–20.

[19] Sebastian Raschka, Joshua Patterson, and Corey Nolet. 2020. Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence. *Information* 11, 4 (2020), 193. doi:10.3390/info11040193

[20] Priya Roy and Chandreyee Chowdhury. 2021. A Survey of Machine Learning Techniques for Indoor Localization and Navigation Systems. *Journal of Intelligent & Robotic Systems* 101, 3 (2021), 63.

[21] Xinyu Tong, Yang Wan, Qianru Li, Xiaohua Tian, and Xinbing Wang. 2021. CSI Fingerprinting Localization With Low Human Efforts. *IEEE/ACM Transactions on Networking* 29, 1 (2021), 442–455.

[22] Stylianos E. Trevlakis, Alexandros-Apostolos A. Boulogeorgos, Dimitrios Pliatsios, Jorge Querol, Konstantinos Ntontin, Panagiotis Sarigiannidis, Symeon Chatzinotas, and Marco Di Renzo. 2023. Localization as a Key Enabler of 6G Wireless Systems: A Comprehensive Survey and an Outlook.

IEEE Open Journal of the Communications Society 4 (2023), 2733–2801. doi:10.1109/OJCOMS.2023.3324952 Conference Name: IEEE Open Journal of the Communications Society.

[23] Martin Werner. 2014. *Indoor Location-Based Services: Prerequisites and Foundations*. Springer International Publishing. https://link.springer.com/book/10.1007/978-3-319-10699-1

[24] Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K Teal. 2017. Good enough practices in scientific computing. *PLoS computational biology* 13, 6 (2017), e1005510.

[25] Ian H. Witten, Eibe Frank, Mark A. Hall, Christopher J. Pal, and James Foulds. 2025. *Data Mining: Practical Machine Learning Tools and Techniques*. Elsevier. Google-Books-ID: u90eEQAAQBAJ.

[26] Junchen Ye, Weimiao Li, Zhixin Zhang, Tongyu Zhu, Leilei Sun, and Bowen Du. 2024. MvTS-library: An open library for deep multivariate time series forecasting. *Knowledge-Based Systems* 283 (2024), 111170.