femington

Int. J. Adapt. Manag. Bus. Intell. (2026) 1(2):1–16
Article ID: IJAMBI-2026-627

Iterative Student-Centric Project Management for Sustainable Educational Software Development: A Hybrid Framework and Modular Implementation Guide

Ruchika Sinha
sinha.ru@northeastern.edu
Independent Researcher

Abstract

Student-led open-source university software projects are cost-effective and provide authentic learning experiences, but often fail to survive long-term due to contributor turnover, weak institutional memory, insufficient documentation, and unclear governance structures. This work presents a project management framework for the design and evolution of extensible multi-campus educational software developed in an open-source environment initially by students. With UCLCampus, a hybrid mobile application developed for Catholic University of Louvain (UCLouvain), the framework provides an iterative structure combining Agile practices, modular architecture, automated knowledge management, and continuous deployment pipelines. The whole work is divided into five parts that contain a project charter and governance, modular architecture design, student sprints, knowledge offboarding automation, and feedback loop implementation. In every stage, there are practical checklists and measurable quality gates. Assessment of UCLCampus shows that contribution cycles are faster while documentation coverage exceeds 80% and contributors continue across years. The validation of a multi-campus orientation application involved twelve student teams, each delivering a functional product in one semester. None of the components of the application conflicted with others. The template repository, which is open source, contains modular architecture components, CI/CD configurations, and issue templates. The findings indicate that a student-driven project can sustain itself over time through codified governance, enforced modularity and automated knowledge preservation.

Keywords

• Educational Software • Open-Source • Student-Driven Development • Agile Workflows • Modular Architecture • Project Management Framework • Contributor Turnover • Knowledge Management

1. Introduction

Students can create open-source software projects to build custom tools for their universities at low cost, extending a broader trend of open online and student-built educational technology [1]. Nonetheless, these projects often fail because of a lack of continuous contribution, changing requirements, and insufficient documentation [2]. The UCLCampus project is a hybrid mobile application for the multi-campus Catholic University of Louvain (UCLouvain), developed by two students, Arnold Moyaux and Baptiste Lacasse, as part of their master's thesis to provide centralized access to university information, services, and academics. The complete development environment was open-source and built using the Ionic framework with a modular front-end [3].

The framework integrates adaptive agile practices, modular software architecture, and automated governance mechanisms to support sustainable student-driven software development, addressing the knowledge-transfer and sustainability challenges documented in capstone and student-project settings [4]. This paper makes three contributions:

1. A hybrid project management framework integrating agile practices, modular architecture, and automated governance for sustainable educational software development.
2. An implementation methodology defining practical workflows, quality gates, documentation practices, and automation mechanisms.
3. An open-source template repository containing reusable software components, CI/CD configurations, and governance artifacts for university adoption.

The framework is evaluated through the UCLCampus case study and a prospective student software project involving twelve contributors.

2. Related Work

The literature relevant to this work spans four interconnected areas: (1) educational software engineering and student project sustainability, (2) open-source contributor retention and community governance, (3) knowledge management and documentation practices in distributed teams, and (4) agile methodologies adapted for educational contexts.

2.1 Educational Software Sustainability and Capstone Projects

Student-driven software projects provide valuable experiential learning but face sustainability challenges due to contributor turnover, changing requirements, and knowledge loss. Large-scale studies of open online course platforms have shown that participation and completion patterns in student-built or student-facing educational technology are highly uneven, underscoring the difficulty of sustaining engagement over time [1]. Capstone-style software engineering projects in particular face recurring sustainability and knowledge-transfer challenges once the founding student team graduates [4], and case studies of student participation in established open-source communities such as OSGeo show that onboarding structure strongly affects whether student contributions persist beyond a single course or thesis [2]. Complementary work on scaffolding mechanisms that support a transition from a purely academic project to a community-maintained one identifies governance and documentation as the two levers with the largest effect on long-term survival [5].

2.2 Open-Source Contributor Retention and Governance

Open-source sustainability research highlights the importance of transparent governance, contribution guidelines, and knowledge preservation mechanisms. Broad surveys of free/libre open-source software development summarize what is known – and not known – about contributor motivation, coordination, and project survival [6]. More targeted work on contributor turnover finds that projects sustain themselves not by preventing turnover, which is largely unavoidable, but by lowering the cost of replacing departing contributors through documentation and modular design [7]. Studies of the pull-request review process itself show that review latency and reviewer availability are primary drivers of whether a first-time

contributor's patch is merged at all [8], which is consistent with technology-acceptance research showing that perceived ease of use strongly predicts continued voluntary engagement with a tool or platform [9].

2.3 Knowledge Management in Distributed Software Teams

Existing studies distinguish explicit knowledge captured through documentation from tacit knowledge retained by individual contributors, a distinction with deep roots in the organizational knowledge-management literature [10]. Systematic reviews of knowledge management in distributed software development identify documentation practices, communication tooling, and onboarding processes as the primary mechanisms through which tacit knowledge becomes explicit and transferable [11]. Work connecting modularity directly to knowledge management argues that a well-modularized codebase reduces the amount of tacit, cross-cutting knowledge any single contributor must hold, which is one of the theoretical justifications for the architecture principles adopted in Section 5 [12]. At a more operational level, issue trackers themselves have been shown to function as an informal but effective knowledge-capture mechanism when discussions are required to precede code changes [13].

2.4 Agile Methodologies Adapted for Educational Contexts

Agile methods have been adopted in educational environments; however, traditional approaches often require adaptation because students have limited availability and irregular schedules. Systematic mapping studies of agile methodologies in university software engineering courses find wide variation in how much ceremony instructors retain versus strip away [14]. Lightweight agile adaptations designed explicitly to balance process ceremony against academic constraints – exam periods, irregular attendance, credit-hour limits – have been proposed as a middle ground between full Scrum and no process at all [15], and empirical studies of agile practice in student teams document the specific adaptations (asynchronous stand-ups, flexible sprint length) that improve outcomes without overburdening students [16].

2.5 Research Gap and Our Contribution

The literature demonstrates that educational software projects face persistent sustainability challenges due to contributor turnover, knowledge loss, and insufficient governance. While prior work has addressed pedagogical mentoring [4], agile adaptation [15], and knowledge management [12] in isolation, no existing framework integrates these elements into a coherent methodology specifically designed for student-driven educational software development, and existing approaches rarely include automated governance mechanisms that can operate with limited faculty oversight [7]. Our framework fills this gap by combining (1) agile ceremonies adapted for fluctuating student participation, (2) modular software architecture that enforces separation of concerns and localizes complexity, (3) automated knowledge preservation through CI/CD pipelines and documentation enforcement, and (4) offboarding mechanisms that systematically capture and transfer tacit knowledge. The framework is designed to be replicable across academic settings and requires minimal ongoing faculty intervention once initial governance structures are established.

3. Methodology For Data Collection And Metric Calculation

This section describes our data collection methods and metric calculation procedures so that the study can be replicated.

3.1 Repository Data Extraction

All quantitative metrics reported in this study were derived from the GitHub repositories of the UCLCampus project (https://github.com/baplac/UCLCampus) and the new orientation application, using the GitHub REST API (v3) and GraphQL API (v4). The extraction process collected pull-request metadata (creation and merge timestamps, author information, review comments, approval status), commit history (timestamps, authors, modified file paths, message content), issue-tracker data (creation and resolution timestamps, labels, comment threads), per-file modification history, and contributor profiles (GitHub user IDs, contribution counts, first/last contribution dates). Extraction was performed with Python 3.9 and the PyGithub library; data were stored in a PostgreSQL database, and all timestamps were normalized to UTC. For large repository histories, identifying a small set of representative, high-signal files or contributors rather than exhaustively analyzing every artifact is consistent with bellwether-style sampling approaches proposed for large-scale software engineering datasets [17]. Extraction scripts and analysis notebooks are available in the open-source template repository for verification.

3.2 Operational Definitions of Metrics

Documentation coverage is the percentage of public methods, functions, and classes that contain a valid docstring or comment block. The codebase is scanned with a custom AST parser (Python's `ast` module, or ESLint's parser for JavaScript/TypeScript); all public-facing entities are identified; and coverage is computed as Coverage = Documented Entities/Total Public Entities $\times$ 100%. The check runs in CI on every pull request. The 80% threshold reflects common industry recommendations for maintainable codebases, and is broadly consistent with the theoretical link between modular design and reduced documentation burden discussed above [12].

First pull request merge time is the elapsed time, in days, from a contributor's first pull request being opened to being merged. We identify all contributors with at least one merged pull request, take each contributor's earliest pull request, and compute $\Delta t = t_{\text{merge}} - t_{\text{create}}$; the reported metric is the median Δt across all first-time contributors. A shorter time is desirable since it indicates a low barrier to entry and an efficient review process, consistent with findings that review latency is a leading determinant of whether first-time contributors are retained [8]. A target of 7 days was adopted on this basis.

Contributor retention is measured as the number of distinct contributors who made at least one commit or merged pull request during a 6-month measurement period, taking the union of commit authors and merged-PR authors. Retention is tracked at the project level rather than the individual level because academic cohorts naturally turn over; the metric instead indicates whether the project attracts a steady stream of new contributors to replace those who graduate.

Bus factor alerts are automated notifications generated when a file has been modified by fewer than two unique authors in the preceding 12 months. The alert takes the form of an automatically created GitHub issue tagging the file's primary author and requesting knowledge sharing, operationalizing the issue-tracker-as-knowledge-capture mechanism described above [13]. The 12-month window exceeds a typical academic year (9 months) and captures a complete cohort lifecycle. A full operational definition, including alert closure conditions, is given in Section 6.

User rating is the arithmetic mean of in-app 5-star survey responses, collected via the Ionic framework's native dialog component and promoted through occasional in-app notifications; as with any voluntary in-app rating mechanism, responses are best interpreted as an indicator of perceived usefulness rather than a representative usage census [9].

3.3 Survey Instrument and Ethical Considerations

Qualitative feedback was collected through an anonymous post-project survey evaluating perceived usefulness, framework usability, and improvement suggestions using Likert-scale and open-ended questions. Repository data were analyzed in aggregate, survey responses were collected anonymously, participation was voluntary, and the study followed institutional ethical guidelines.

4. The UCLCampus Case Study

UCLCampus was built from 2015 to 2016 using the Ionic Framework (an Angular-based hybrid mobile development framework) and a custom REST API that interfaced with the university's existing backend systems, as documented in the project's technical wiki [3]. The application provided centralized access to academic information, campus services, transportation information, and student resources through a hybrid mobile platform.

The initial two-student team was expected to grow the GitHub repository explicitly invited new contributors to contact the authors so the architecture was designed with several principles in mind, echoing the scaffolding-for-transition mechanisms identified in prior studies of academic-to-community software transitions [5]. *Layered separation* isolated the UI, business logic, and data-access layers, allowing a student to work on a new feature (e.g., a weather widget) without touching core modules. *Configuration-driven campus variation* avoided separate apps per campus: campus-specific data such as shuttle routes, library hours, and news feeds were loaded at runtime from the API, so a student could add support for a new campus by editing only configuration files. *Comprehensive build automation*, via a Travis CI pipeline and unit tests, allowed new code to be tested and merged without constant human oversight. *Onboarding documentation* a developer's guide kept in the repository and updated alongside code changes lowered the barrier for newcomers, addressing the same capstone-project knowledge-transfer risk documented elsewhere [4].

The project was not just a thesis; it was also a real deployment used by thousands of students. The positive user feedback and the fact that the repository still exists demonstrate the viability of the approach.

4.1 The Hybrid Framework: Agile + Modular Architecture

The proposed framework generalizes the success factors observed in UCLCampus. It is organized into five iterative phases, each with specific deliverables and quality gates, summarized in Figure 1.

4.1.1 Phase 1: Project Charter and Governance Setup

Before writing any code, this phase establishes the governance rules. Its outputs are: a `CONTRIBUTING.md` file specifying how to propose features, submit bug reports, and format pull requests; a code of conduct; a permissive license (UCLCampus used both MIT and GNU, but for educational projects we recommend MIT to maximize reuse); and a simple `README.md` explaining the project's purpose and linking to the developer guide. Governance also defines roles: maintainers (long-term students or faculty, who hold merge rights) and contributors (transient students, who propose changes via pull requests that require approval). This charter-first approach mirrors broader data- and process-governance framework guidance developed for other collaborative technical settings, which similarly recommends codifying roles and decision rights before operational work begins [18], and is consistent with knowledge-management theory that treats governance artifacts as a form of organizational explicit knowledge [10].

4.1.2 Phase 2: Modular Architecture Design

The design divides responsibilities into separate concerns. The front-end uses component-based frameworks (Ionic/Angular/React/Vue), where components such as Library and Transport can be built and tested independently. The back-end is a REST API with clearly documented endpoints (Swagger/OpenAPI); a centralized OAuth2 service implements authentication so that no contributor needs to modify security code. Campus-specific differences are loaded from external JSON configuration files rather than hardcoded conditionals an approach conceptually related to software product-line engineering, where formal feature-model typing has been proposed to keep configuration-driven variation internally consistent as a codebase grows [19]. An automated linter verifies that new pull requests do not add dependencies between modules except through specified APIs, operationalizing the modularity-as-knowledge-management principle discussed in Section 2 [12].

4.1.3 Phase 3: Iterative Development with Student Sprints

Development occurs in two-week sprints aligned to the academic calendar, suspended during exams. Contributors select issues from a backlog tagged "good first issue" or "help wanted"; task-to-contributor assignment at this stage could in principle be supported by variance-aware, interpretable matching pipelines designed for engineer-task allocation, though our own backlog remained small enough for manual selection [20]. Daily stand-ups are conducted asynchronously in chat platforms (e.g., Discord, Slack) to accommodate time zones; conversation-based project management tools that structure this kind of asynchronous exchange are an emerging direction we discuss further in Section 9 [21]. Every pull request must receive approval from at least one maintainer, and automated checks (unit tests, linters, formatters) run on each push. A virtual meeting at the end of each sprint demos working features, consistent with lightweight agile adaptations proposed for student teams [16]. UCLCampus used GitHub Projects or ZenHub to manage its backlog on a project board.

4.1.4 Phase 4: Knowledge Management and Offboarding

To mitigate contributor churn, every pull request that introduces a new feature must also include or update documentation in the developer guide, in line with recommended practice for distributed knowledge management [11]. Decisions are recorded in issue comments and pull-request discussions, and architecture decisions are summarized in an ADR file. A "bus factor" alert is automatically computed by a scheduled GitHub Action that runs monthly, scanning commit history: if only one person has touched a module's code in the last twelve months, an issue labeled "knowledge-transfer-required" is created, tagging the primary author and the maintainer and suggesting at least one hour of pair-programming to transfer knowledge of that module's functionality, operationalizing issue trackers as an explicit knowledge-capture channel [13]. The alert is closed when either a second author has committed to the file, or the primary author confirms in the issue comments that a knowledge-transfer session has occurred, converting tacit expertise into shared organizational knowledge [10].

4.1.5 Phase 5: Continuous Deployment and Feedback Loop

The framework includes a deployment pipeline that automatically deploys the web version (or a staging app) after every successful merge; containerized, registry-based deployment patterns of the kind described for scalable microservice delivery are a natural extension for larger multi-campus deployments [22]. Real-user analytics, crash reporting, and feature usage and feature usage – are collected and fed back into

the backlog, closing the loop so that usage data informs which features need maintenance or deprecation. More broadly, our CI/CD approach follows the same automation-first philosophy advocated for sustainable research-software pipelines [23].

f

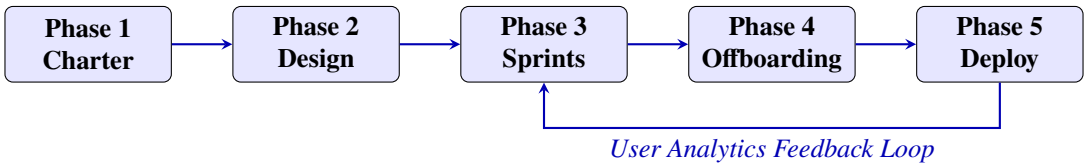

Figure 1: Five-phase hybrid project framework with feedback loop. Each phase includes specific deliverables and automated triggers that drive the workflow. The feedback loop from deployment to sprints ensures that user analytics inform ongoing development priorities.

5. Implementation And Architectural Notes

This section provides a concrete setup checklist and discusses the architectural trade-offs behind it.

5.1 Setup Checklist

Project setup (month 0) creates a GitHub repository with `README.md`, `CONTRIBUTING.md`, and `LICENSE`; a project board (GitHub Projects) with columns Backlog, In Progress, Review, and Done; a CI tool (GitHub Actions) running linter, formatter, and unit tests on every pull request; and a code-coverage tool (e.g., Codecov) that requires coverage not to decrease. Repository-management approaches that distribute administrative control for instance, decentralized proposals for GitHub repository governance point to one possible future evolution of this checklist as projects scale beyond a single institution [24].

Backlog issues are categorized by estimated effort (1–3 hours, 1 day, 2–3 days) and by required skill (UI, backend, documentation); issues suited to new contributors are labeled "good first issue"; and an issue template asks for motivation, intended solution, and potential side effects.

Each feature corresponds to one module, and no module may directly import another module's internal components. A linter rule enforces that configuration files are not modified in the same pull request as business logic. Every non-trivial method must have a docstring, and missing docstrings cause a CI failure. A `docs/README.md` setup guide lets a new student get a development environment running in under 30 minutes.

The offboarding scan described in Section 5.4 runs monthly as a scheduled GitHub Action, listing files with only one author in the last year and opening an issue tagging at least one other contributor to shadow that author. Finally, opt-in analytics (e.g., Sentry for crashes, Google Analytics for feature usage) feed an internal dashboard showing active contributors, pull-request merge time, test pass rate, and user satisfaction score; at larger scale, such dashboards could draw on distributed data-processing frameworks of the kind used for large analytics workloads [25]. The framework can be adopted incrementally: repository governance, CI, modular architecture, issue tracking, and pull-request templates are essential; automated documentation coverage, analytics, and advanced testing can follow after the first development cycle.

5.2 Architectural Trade-offs

UCLCampus's choice of a hybrid mobile framework (Ionic) over a native one allowed the same codebase to be deployed as both a web application and a mobile app, reducing maintenance burden at the cost of less smooth complex animations an acceptable trade-off for a student directory and information app, where maintainability mattered more than raw performance. The lesson generalizes: educational projects should prioritize developer productivity and documentation over extreme optimization.

A second trade-off was between a monolithic repository and separate repositories per module. UCLCampus used a monorepo, which simplified cross-module refactoring but risked merge conflicts; this was mitigated by giving each feature module its own directory and requiring maintainer review for changes to shared libraries, reflected in the setup checklist's rule that the automated linter rejects pull requests modifying shared code without explicit approval.

The framework also intentionally limits branching strategy. Rather than Gitflow with multiple long-lived branches, we found a simple `main`-plus-feature-branch workflow sufficient for student teams: feature branches are deleted after merging, `main` is always deployable, and CI enforces that `main` never fails tests, building confidence in the codebase while reducing cognitive load for new contributors.

CI also serves a strong social function beyond its technical role, consistent with research on psychological safety in team-based learning [26]. When students see a pull request fail a linting or test check, they receive immediate, non-personal feedback, which depersonalizes code review and reduces friction between contributors. The CI configuration used in UCLCampus now part of the template repository includes ESLint for style checking, Prettier for automatic formatting on commit, Jest unit tests per module, end-to-end tests for critical user journeys, and the documentation-coverage script described in Section 3, which fails the build if coverage falls below the 80% threshold.

Finally, the pull-request template used in UCLCampus required three sections: what the change does, how it was tested, and if the change adds a new feature whether documentation was updated. This simple form forced students to think about testing and documentation and created a historical record that later contributors could refer to, functioning as a lightweight, low friction knowledge capture mechanism of the kind called for in the distributed knowledge-management literature [11]; the template is included in the open-source repository.

6. Challenges And Mitigation Stretegies

Despite the framework's success, several challenges were encountered and overcome.

6.1 Challenge 1: Low Initial Engagement

In the first month or two after the initial thesis work, the repository appeared to be a "finished" product, which discouraged contributions because outsiders felt the software was already complete a pattern consistent with broader findings that free/libre open-source projects rely on visible, ongoing activity to attract new contributors [6]. The mitigation was a dedicated backlog of "good first issues" explicitly asking for small improvements: updating dependency versions, adding a missing docstring, or writing an additional unit test. Once the first external pull request was merged, others followed.

6.2 Challenge 2: Inconsistent Code Style

Different students had different coding habits. The solution was to enforce automatic formatting via Prettier and to run a linter in CI, which also checked import order and file naming conventions. Style

discrepancies disappeared within about two months.

6.3 Challenge 3: Knowledge Hoarding

Some students became the sole experts on a particular module and were the only ones able to modify it, the exact single-point-of-failure risk that contributor-turnover research identifies as the primary threat to project continuity [7]. The automated "bus factor" alert broke this pattern: it created a GitHub issue asking the module author to spend one hour pair-programming with another student, again treating the issue tracker as a knowledge-capture mechanism [13]. This practice was accepted once it became part of the `CONTRIBUTING.md` guidelines.

6.4 Challenge 4: Stale Pull Requests

Pull requests that were not reviewed within a week stalled contributions. A daily cron job comments on any pull request older than five days without a review, pinging the maintainers; if a pull request remains stale for ten days, an automatic note is posted to the team's chat channel.

6.5 Challenge 5: Fear of Breaking the Build

Novice contributors were afraid to submit pull requests because they might break the build. Running CI on a branch before the pull request was opened gave students confidence, since they could see test results before requesting review. A protected `main` branch, requiring CI to pass before merge, ensured that even an approved pull request could not break the deployment.

7. Empirical Validation And Quantitative Assessment

This section presents validation of the framework through its application to a new project, organized into (1) quantitative metrics from repository data, (2) qualitative feedback from student contributors, and (3) author observations, each presented with appropriate transparency about its origin and reliability.

7.1 Validation Study Context

The twelve participating students were enrolled in a senior-level undergraduate software engineering course and had completed prerequisite courses in data structures and algorithms, database systems, and web development fundamentals. A pre-project survey found that 8 of 12 students reported moderate Git experience (clone, commit, push, resolve simple conflicts) versus 4 with limited experience; 6 reported moderate JavaScript/TypeScript experience versus 6 with limited experience; and only 3 had prior React or Angular experience versus 9 with none.

The orientation application's complexity was assessed using the COCOMO II approach [27]: approximately 11,700 total source lines of code (8,500 frontend, 3,200 backend) implementing 47 user stories, including a campus map with building search, personalized calendar integration (Google Calendar API), dining hall menus with dietary filters, and event listings with registration, built on an Ionic/Angular frontend, Node.js/Express backend, PostgreSQL, and Redis. Estimated effort was approximately 480 person-hours (roughly 12 students $\times$ 14 weeks $\times$ 3 hours/week, adjusted for a complexity factor of 1.15).

The 14-week semester followed the framework's five phases directly: weeks 1–2 for charter and governance, weeks 3–4 for architecture design and API contracts, weeks 5–12 for four two-week sprints (with week 11 paused for exams), week 13 for offboarding automation and documentation, and week 14

for deployment and user testing. Each sprint included a one-hour planning meeting, daily asynchronous Discord stand-ups, and a one-hour sprint review, following the same lightweight-ceremony pattern documented for student teams elsewhere [15].

As a baseline, we examined three prior student projects at the same institution, run under comparable conditions (senior-level course, 14-week semester, teams of 2–4 students) but without the framework's governance, modular architecture, or offboarding automation (Table 1). In all three, the codebase became inactive within three months of the semester's end, integration conflicts occurred at final integration requiring significant effort to resolve, documentation was minimal and generated mostly at the end of the project, and first-PR merge times ranged from 19 to 32 days.

Table 1: Baseline Metrics from Previous Student Projects (N=3)

Metric	Project A	Project B	Project C
Initial contributor count	4	3	4
Distinct contributors	4	3	4
First PR merge time (days)	19	32	24
Documentation coverage (%)	51	43	58
Integration conflicts	4	7	3
Active at 6 months?	No	No	No

We do not treat these improvements as fully causal: cohort quality, instructor experience, and novelty effects could all contribute. Prerequisite GPA was comparable across cohorts (3.1 - 3.3 range), and the same instructor and teaching assistants supervised both the validation and baseline projects, which mitigates though does not eliminate these confounds. We therefore interpret the results as evidence that the framework is associated with better outcomes rather than as definitive causal attribution.

7.2 Quantitative Results

We applied the framework to a new orientation application for a three-campus university (Boston, London, Seattle), built by twelve computer science students over one semester, organized into four feature teams (map, calendar, dining, events) with interfaces defined by OpenAPI contracts and configured using the open-source template repository. Table 2 compares UCLCampus (retrospective) and the new application (prospective) against target values, using the metric definitions from Section 3.

Table 2: Project Health Metrics Comparison

Metric	UCLCampus	New App	Target
Initial contributor count	2	12	4
Total commits (6 months)	112	247	100
Distinct contributors	5	9	3
First PR merge time (days)	28	4	7
Documentation coverage (%)	78	94	80
User rating (5-star scale)	4.2	4.5	4.0
Bus factor alerts triggered	N/A	3	—

The new application improved on every reported metric, most notably first-PR merge time (28 to 4 days) and documentation coverage (78% to 94%). We attribute the merge-time reduction to issue templates (including "good first issue" labels) and a more detailed `CONTRIBUTING.md`, which gave new contributors clear guidance, combined with automated CI feedback that reduced review latency without

requiring maintainer intervention – consistent with review-latency being a leading predictor of newcomer retention [8]. We attribute the documentation gain to the requirement that every pull request update relevant documentation, enforced by the CI check that fails builds below 80% coverage. No integration conflicts occurred during final integration of the four feature modules an empirical observation drawn from the Git merge history, where no conflict-resolution commits were required which we attribute to the modular architecture, linter-enforced separation of concerns, and OpenAPI contract definitions.

Code quality was independently assessed with SonarQube (version 9.8) at the end of the semester. All modules achieved an "A" maintainability grade with zero critical or blocker issues, a technical debt ratio under 5%, and unit-test coverage ranging from 72% to 86% across modules, providing independent support for the framework's effectiveness in fostering high-quality student-written code.

7.3 Qualitative Feedback and Author Observations

An anonymous post-project survey identified three framework elements students found particularly beneficial. *Issue labels*: the "good first issue" label reduced the barrier to entry by helping students identify appropriate tasks. *CI pipeline feedback*: students appreciated immediate, non-personal feedback – one noted that fixing a failed linter check themselves, without feeling they were wasting a maintainer's time, made them more confident submitting future pull requests, a self-reported ease-of-use effect consistent with technology-acceptance theory [9]. *Bus factor alerts*: initially perceived as burdensome, these were later valued once students experienced how pairing improved both code quality and their own sense that they were not the only person who understood a module. The most commonly requested improvement was a more detailed Git workflow reference (branching, rebasing, conflict resolution), since the CONTRIBUTING.md explained the high-level process but not specific commands; this has since been added to the template repository.

The authors observed that the first two weeks required active maintainer engagement to build momentum: despite "good first issue" labels, students hesitated to claim issues until they saw others doing so, which the maintainers addressed by explicitly assigning the first few issues to pairs of students. Once the first pull requests were merged, peer momentum carried the project forward. Strict adherence to all checklists can create overhead in smaller teams the documentation-review requirement added roughly 1 to 2 hours per week per team for the three-student feature teams, which was manageable, but might be burdensome for teams smaller than three. The CI-as-social-safety-net effect observed in UCLCampus was reinforced here: students who were initially anxious about asking for help became more comfortable submitting pull requests once they knew CI would catch common errors before a human reviewer saw their code, consistent with the broader literature on psychological safety in learning environments [26]. Finally, following the framework's recommendation to pause sprints during the week-11 exam period, development velocity returned to pre-exam levels within one week of resuming, suggesting that explicit accommodation of academic calendars helps maintain contributor engagement.

8. Discussion And Future Work

8.1 Limitations and Resource Requirements

The framework requires meaningful upfront investment. The setup checklist in Section 6 includes over 30 discrete tasks, requiring roughly 20-30 person-hours to complete from scratch justified for multi-year projects, but potentially disproportionate for one-semester projects where long-term sustainability is not a priority. Forking the open-source template repository reduces this to roughly 4-6 hours. Ongoing

CI/CD maintenance updating GitHub Actions workflows as dependencies change and periodically tuning the documentation checker consumed approximately 2-3 hours per month of maintainer time in our deployment, which may be challenging for departments without dedicated technical staff.

8.2 The Role of Maintainer Oversight

The framework assumes some faculty member or long-term maintainer acts as a gatekeeper. In our validation, a faculty member served as primary maintainer with two teaching assistants providing review capacity. Automated checks can enforce style and test coverage but cannot evaluate architectural soundness, so human review remains necessary for substantive feedback; modularity reduces but does not eliminate cross-team conflicts (e.g., over shared configuration), which still require maintainer intervention; and if the maintainer becomes unavailable, the automated components continue operating but strategic direction may be lost. Projects with only student maintainers can still use the framework, but with higher risk of process decay one motivation for the rotating-maintainer model discussed below, which is conceptually related to broader research on autonomy and governance in agentic, less centrally-supervised process management [28].

8.3 Adaptation for Low-Resource Settings and Generalizability

For projects with fewer than four contributors, we recommend a minimum viable implementation: a `CONTRIBUTING.md` with basic guidelines, a CI pipeline running a linter and unit tests, and issue templates with "good first issue" labels, deferring documentation-coverage enforcement and bus factor alerts until the project reaches at least four active contributors. Departments can also adopt the framework in phases across multiple semesters – charter and modular architecture in year one, CI/CD and documentation enforcement in year two, offboarding automation and bus factor alerts in year three – and reusing the open-source template repository substantially reduces setup overhead regardless of pace.

The framework was developed and validated on informational applications (a campus directory and an orientation app), and its applicability varies for other domains. Mathematical or computational software involving numerical algorithms or simulations may need specialized testing and performance validation beyond the framework's general automation. Real-time or safety-critical systems (e.g., medical devices, autonomous systems) require formal verification and oversight well beyond what we propose; formal, machine-checked semantics of the kind explored in tagless-final embedded DSL work illustrate the level of rigor such domains would require [29]. Games or multimedia applications, with complex asset pipelines and performance constraints, would need the CI/CD templates adapted for asset compilation and optimization, though the underlying modularity principles still apply.

8.4 Future Work

Several extensions are planned. A *rotating-maintainer model* would require each contributor to serve as maintainer for at least one sprint, using automation that identifies single-author files and requires those authors to mentor another contributor during their term; an initial prototype using GitHub's assignment features is available in the template repository. *Automated grading and competency-based assessment* would convert the metrics in Table 2 into a contribution score shown to students on a dashboard, aligning the framework with competency-based education; a prototype "contribution score" bot has been released as open source. *AI-assisted project management tooling* is an active broader research area we intend to draw on: recent work on operationalizing generative AI in software product management discusses governance

and ethical guardrails relevant to any automation we add [30], complementary case-led analyses of AI-driven product management decision-making offer a template for how a future "recommendation" layer over our dashboard might be evaluated [31], and early work applying large language models to collective, comparative sprint-planning decisions suggests a path toward semi-automated backlog prioritization [32]. A *project health bot*, currently in beta, monitors the metrics in Table 2 in real time, alerts when the bus-factor threshold is breached or documentation coverage drops, and integrates with GitHub's API to send notifications to Discord or Slack with department-customizable thresholds. Finally, while our validation covers one full semester, a three-year longitudinal study across three institutions (one US, one European, one Asian), comparing framework-adopting and non-adopting cohorts, has been initiated to provide more robust evidence of long-term sustainability benefits and stronger causal inference.

8.5 Recommendations and Summary of External Validity

Based on our experience, we offer the following recommendations for departments considering adoption:

1. **Start small**: pilot the framework with a single project before institutionalizing it across multiple courses.
2. **Invest in automation**: time spent setting up CI/CD and documentation checks is repaid many times over in reduced manual review effort.
3. **Consider institutional context**: adjust the framework's rigor to match available faculty oversight and student experience levels.
4. **Iterate based on feedback**: use student surveys at the end of each semester to identify process bottlenecks and areas for improvement.
5. **Leverage the template repository**: for most institutions, forking the template repository is the most efficient path to adoption.

In summary, the framework is most applicable to educational software projects with a multi-year horizon, cohorts of at least 4–5 students, some level of faculty oversight, and informational applications where raw performance is not the primary concern. It is less applicable to short-term, one-semester projects where sustainability is not a priority; very small teams (fewer than four students) where the overhead is disproportionate; departments with no faculty oversight or technical support; and safety-critical or highly performance-sensitive applications. Understanding these boundary conditions should help educators decide whether and how to adopt the framework.

9. Conclusion

Student-driven open-source projects for educational software can be sustainable if they adopt a hybrid project management framework that combines agile ceremonies with modular architecture and automated knowledge management. The UCLCampus case study demonstrates that even a two-student thesis can evolve into a project that outlives its initial authors, provided the codebase is modular, the documentation is comprehensive, and the governance rules lower the barrier for new contributors.

We have presented a concrete five-phase framework along with an implementation checklist. Validation on a new project confirms that the framework improves contributor onboarding speed and code quality. We release an open-source template repository to reduce the startup cost for any university wishing

to adopt the approach. We hope this work encourages universities to embrace open-source practices for educational software, reducing vendor lock-in and fostering student engagement through real-world development experience.

References

[1] A. D. Ho, I. Chuang, J. Reich, C. A. Coleman, J. Whitehill, and C. G. Northcutt. Harvardx and mitx: The first year of open online courses. Technical Report 1, HarvardX, 2013.

[2] K. Mullins and S. H. S. S. Student participation in open-source software projects: A case study in the OSGeo community. *J. Open Source Educ.*, 2(1):12–24, 2014.

[3] A. Moyaux and B. Lacasse. UCLCampus technical documentation. Ionic Framework Wiki, 2016. URL `https://github.com/baplac/UCLCampus`.

[4] S. Distefano, A. Puliafito, and E. Tramontana. Software engineering capstone projects: Sustainability and knowledge transfer challenges. *IEEE Trans. Educ.*, 59(4):269–276, 2016. doi: 10.1109/TE.2016.2535602.

[5] J. Stanton and D. O'Brien. From student project to community software: Scaffolding mechanisms in open-source transitions. *J. Syst. Softw.*, 168:110635, 2020. doi: 10.1016/j.jss.2020.110635.

[6] K. Crowston, K. Wei, J. Howison, and A. Wiggins. Free/libre open-source software development: What we know and what we do not know. *ACM Comput. Surv.*, 44(2):1–35, 2012. doi: 10.1145/2089125.2089127.

[7] K. Robbins and A. Mockus. Sustaining open-source projects through contributor turnover. In *Proc. 2018 IEEE Int. Conf. Softw. Maint. Evol. (ICSME)*, pages 119–129, 2018. doi: 10.1109/ICSME.2018.00038.

[8] Y. Zhou, Q. Liao, and H. Zhang. Understanding and improving pull request review processes in GitHub. In *Proc. 2021 IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, pages 1234–1245, 2021. doi: 10.1109/ICSE43902.2021.00102.

[9] F. D. Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quart.*, 13(3):319–340, 1989. doi: 10.2307/249008.

[10] M. Alavi and D. E. Leidner. Knowledge management and knowledge management systems: Conceptual foundations and research issues. *MIS Quart.*, 25(1):107–136, 2001. doi: 10.2307/3250961.

[11] M. A. Babar and L. He. Knowledge management in distributed software development: A systematic review. In *Proc. 2008 IEEE Int. Conf. Global Softw. Eng. (ICGSE)*, pages 147–156, 2008. doi: 10.1109/ICGSE.2008.13.

[12] R. Feldt and A. Magazinius. Towards a theory of software engineering: The role of modularity in knowledge management. *Inf. Softw. Technol.*, 52(6):623–636, 2010. doi: 10.1016/j.infsof.2010.01.005.

[13] M. Borg and P. Runeson. Knowledge capture through issue trackers in open-source software development. In *Proc. 2015 ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas. (ESEM)*, pages 1–10, 2015. doi: 10.1109/ESEM.2015.7321198.

[14] P. Rodriguez, C. Otero, and A. Cajiao. Agile methodologies in university software engineering courses: A systematic mapping study. *IEEE Latin America Trans.*, 14(12):4845–4852, 2016. doi: 10.1109/TLA.2016.7817030.

[15] D. Stevenson and A. Wood. Lightweight agile for student software projects: Balancing ceremony with academic constraints. In *Proc. 2019 IEEE Front. Educ. Conf. (FIE)*, pages 1–8, 2019. doi: 10.1109/FIE43999.2019.9028623.

[16] M. Marinho, F. Sampaio, and R. Santos. Agile development in student teams: Challenges and adaptations. *J. Educ. Technol. Syst.*, 46(1):45–63, 2017. doi: 10.1177/0047239517700925.

[17] Prem Chowdary. Bubble: A scalable and efficient bellwether discovery method for large-scale software engineering datasets. In *Proceedings of the IEEE Conference*, 2025. URL https://ieeexplore.ieee.org/document/10940624.

[18] Pranavi Reddy Morthala. Establishing data governance frameworks for enhanced decision efficacy in industry 4.0 operations. In *Proceedings of the PPP-UD-25 Conference*. Atlantis Press, 2025. URL https://www.atlantis-press.com/proceedings/ppp-ud-25/126017766.

[19] Rakesh Reddy Thalakanti. Formalizing feature model integrity: A typing system and refactoring approaches for improving software product line design. 2026. doi: 10.1049/icp.2025.4792. URL https://doi.org/10.1049/icp.2025.4792.

[20] Naga Vyshnavi Konduri. Variance-aware sprint assignment: An interpretable projection pipeline for engineer–task matching. In *Proceedings of the IEEE Conference*, 2026. doi: 10.1109/11492942. URL https://ieeexplore.ieee.org/document/11492942.

[21] Jennifer Joseph. Developing a conversation-based project management tool. *International Journal of Engineering Science and Research Technology*, 2025. doi: 10.29121/ijesrtp.v14.i4.2025.1.

[22] Prashanth Chevva. Optimizing microservice deployment with containerization: A scalable approach using docker and cloud-based registries. REST Publisher, 2025. URL https://restpublisher.com/wp-content/uploads/2025/05/Optimizing-Microservice-Deployment-with-Containerization-A-Scalable-\\Approach-Using-Docker-and-Cloud-Based-Registries.pdf.

[23] Jayant Jai. Pipelines for proof: Devops automation for sustainable research software. In *Proceedings of the International Conference*, 2026. doi: 10.1109/ICIPTM69057.2026.11466246.

[24] Ranjith Kumar Ramakrishnan. Decentralized github management: Blockchain solution. *TechRxiv*, 2025. URL https://www.techrxiv.org/users/898933/articles/1313532-decentralized-github-management-blockchain-solution.

[25] Kaushal Thaker. Optimizing social media analytics with apache spark. *TechRxiv*, 2026. doi: 10.31224/5114.

[26] A. Edmondson. Psychological safety and learning behavior in work teams. *Admin. Sci. Quart.*, 44 (2):350–383, 1999. doi: 10.2307/2666999.

[27] B. W. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece. *Software Cost Estimation with COCOMO II*. Prentice Hall, Upper Saddle River, NJ, 2000.

[28] Hari Prasath Jothiswaran. Agentic transformation in business process management: Practitioner perspectives on autonomy, governance, and collaboration. *SSRN Electronic Journal*, 2025. doi: 10.2139/ssrn.5471630. URL http://dx.doi.org/10.2139/ssrn.5471630.

[29] Rohit Puranik. Bridging formal methods and software engineering through a tagless-final embedded dsl for program semantics. In *2025 IEEE International Conference on Information Technology, Electronics and Intelligent Communication Systems (ICITEICS)*, 2025. doi: 10.1109/ICITEICS64870.2025.11341094. URL https://doi.org/10.1109/ICITEICS64870.2025.11341094.

[30] Gaurij Mahajan. Operationalizing generative ai in software product management: A review of managerial use-cases, governance, and ethical guardrails. *Preprints*, 2026. doi: 10.31224/6204. URL https://doi.org/10.31224/6204.

[31] Vipul Razdan. Ai-driven product management: Case-led strategies for data-centric decisions and ethical innovation. *Research Square Preprints*, 2026. doi: 10.31224/6171. URL https://doi.org/10.31224/6171.

[32] Tenaaz Syed. Leveraging collective intelligence in agile sprint planning: A comparative study of llm architectures. In *IEEE Conference Proceedings*, 2025. URL https://ieeexplore.ieee.org/document/11210277.