



BerlinUP
Journals

Electronic Communications of the EASST

Volume 85 Year 2025

**deRSE25 - Selected Contributions of the 5th Conference for
Research Software Engineering in Germany**

*Edited by: René Caspart, Florian Goth, Oliver Karras, Jan Linxweiler, Florian Thiery,
Joachim Wuttke*

Research Software Lifecycles and Stages

Yo Yehudi, Mikaela Cashman, Michael Goedicke, Wilhelm Hasselbring, Daniel S. Katz,
Sebastian Müller, Carole Goble, Caroline Jay

DOI: 10.14279/eceasst.v85.2693

License: © ⓘ This article is licensed under a CC-BY 4.0 License.

Electronic Communications of the EASST (<https://eceasst.org>).

Published by **Berlin Universities Publishing**
(<https://www.berlin-universities-publishing.de/>)

Research Software Lifecycles and Stages


**Yo Yehudi¹, Mikaela Cashman², Michael Goedicke³, Wilhelm Hasselbring⁴,
Daniel S. Katz⁵, Sebastian Müller⁶, Carole Goble⁷, Caroline Jay⁸**

¹yo@we-are-ols.org, <https://yo-yehudi.com>

0000-0003-2705-1724 

Open Life Science

²mcashman@lbl.gov, <https://mikacashman.github.io/>

0000-0003-0620-7830 

Lawrence Berkeley National Laboratory

³michael.goedicke@paluno.uni-due.de, <https://s3.paluno.uni-due.de/team/michael-goedicke>

0009-0004-2383-6764 

Universität Duisburg-Essen

⁴hasselbring@email.uni-kiel.de, <https://www.cau-se.de/>

0000-0001-6625-4335 

Christian-Albrechts-Universität zu Kiel

⁵d.katz@ieee.org, <https://danielskatz.org/>

0000-0001-5934-7525 

University of Illinois Urbana-Champaign

⁶sebastian.mueller.1@uni-potsdam.de,
<https://www.uni-potsdam.de/en/cs-se/team/sebastian-mueller>

0000-0002-3057-1125 

Universität Potsdam

⁷carole.goble@manchester.ac.uk, <https://research.manchester.ac.uk/en/persons/carole.goble>

0000-0003-1219-2137 

The University of Manchester

⁸Caroline.Jay@manchester.ac.uk, <https://research.manchester.ac.uk/en/persons/caroline.jay>

0000-0002-6080-1382 

The University of Manchester

Abstract: Research software is usually developed by researchers themselves or by software developers working closely with researchers. It is typically developed to meet specific research needs. We develop and evaluate a comprehensive and flexible software lifecycle framework that reflects the unique challenges of research software. In review of related work on commercial and open-source software lifecycles, we observed limitations in addressing the heterogeneous, irregular development patterns, and longitudinal and cyclical nature of many research software projects. An initial model, which was based on case-study research of thirty-eight open-source projects was refined through collaborative discussion at a Dagstuhl seminar, introducing additional state transitions, and expanding details on blocked and active development stages for the specific context of research software. We highlight the overlap and extensions in the new model compared to prior literature and present the evaluation with the research software community. We derive the research software stages from our lifecycle model to categorise research software accordingly.

Keywords: Research Software Engineering, Software Lifecycle, Development Stage, Software Maturity

1 Introduction

The subset of computer code used by researchers, including computational and data scientists, to perform their research is known as “Research Software”, and may be used to produce prominent scientific breakthroughs such as images of a black hole [A⁺19], or Nobel-Prize winning 3D protein structure predictions [HBB16]. Research software is typically created either by researchers themselves or by software developers collaborating closely with them and is usually tailored to address specific research objectives. Research software engineering and the related research software engineer role has emerged as a job profile in its own right [CKB⁺21].

Despite this significance, research software (like all software) projects may collapse, i.e., cease to exist or fall out of maintenance, for one reason or another [Hin19]. Understanding the factors that underpin the failure or success of research software is an understudied field [FGG⁺25]: even the definition of software failure and success are unlikely to be agreed upon by software maintainers, software users, and researchers [CV17].

This paper builds on previous work on software longevity and lifecycles to create a framework and common nomenclature for understanding research software lifecycles. This gives research software funders, developers, maintainers, users, and researchers tools to discuss the progression, challenges, and risks inherent in research software development and longevity.

We start with a look at previous work in Section 2. Our initial research method is introduced in Section 3, and the framework for modern open-source projects that this yielded is then presented in Section 4. This initial framework was evaluated via case studies, as reported in Section 5. As presented in Section 6, we have extended the initial framework towards a framework for open-source *research* software. To evaluate this extended framework, we illustrate and discuss the relation with other published frameworks in Section 7. We derive the research software stages from our lifecycle model in Section 8 to categorise research software accordingly. Section 9 discusses our results and Section 10 concludes the paper with a look at future work.

2 Previous Work

Previous work includes stages for software lifecycles ([Subsection 2.1](#)) and software lifecycle frameworks ([Subsection 2.2](#)).

2.1 Stages for Software Lifecycles

Rajlich and Bennett describe the software lifecycle in which maintenance is actually a series of distinct stages [[RB00](#)]. As [Figure 1](#) shows, according to this model, the software lifecycle of commercial software consists of five distinct stages:

Initial development Engineers develop the system’s first functioning version.

Evolution Engineers extend the capabilities and functionality of the system to meet user needs, possibly in major ways.

Servicing Engineers make minor defect repairs and simple functional changes.

Phaseout The company decides not to undertake any more servicing, seeking to generate revenue from the system as long as possible.

Closedown The company withdraws the system from the market and directs users to a replacement system, if one exists.

For managing the evolution of long-living software, we need to take particular account of the stages after the initial development [[RGH⁺19](#)].

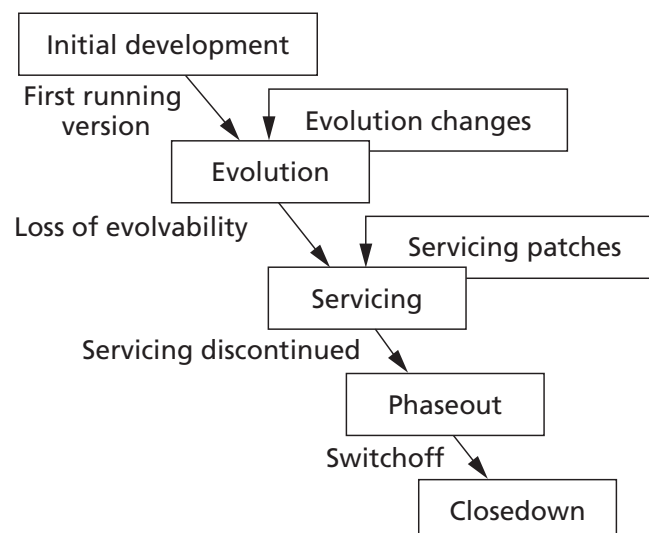


Figure 1: A staged model for the software lifecycle of commercial software [[RB00](#)].

	Phase: Initiation	Phase: Growth
Outcome: Indeterminate	Indeterminate, Initiation (II)	Indeterminate, Growth (IG)
Outcome: Tragedy	Tragedy, Initiation (TI)	Tragedy, Growth (TG)
Outcome: Success	Success, Initiation (SI)	Success, Growth (SG)

Table 1: Tragedy/success lifecycle framework [WC10]

2.2 Software Lifecycle Frameworks

Lehman’s Laws of Software Evolution divides software into categories, noting that most programs are intended for sufficiently complex tasks that they must continually evolve in order to remain satisfactory, based on iterative feedback and desired features. Lehman defines these evolving programs as “E programs”, and programs that can be fully defined by an initial specification as “S programs” [Leh84].

Avelino et al. note that most programs fall into the E program category, observing that occasionally mature software systems can be deemed feature-complete, but may still need bug fixes and security updates [ACVS19].

Champion and Hill define an open-source risk of underproduction: the scenario where a software package has an imbalance between its need for quality improvement, and the labour of people to implement these improvements [CH21].

Wynn looks at established business literature focusing on organisational structure, hierarchy, and sales growth/loss over time, conjecturing that downloads over time might be an appropriate measurement for open-source lifecycles [WJ04]. Thus, they adapted a linear business/sales lifecycle model, substituting “sales” for “downloads” of an open-source software. This model asserted that there were four primary lifecycle stages for a project: initial, growth, maturity, and either revival (returning to growth) or decline. A project could, in theory, move between Stages 2 (Growth), 3 (Maturity) and 4 (Revival/Decline) repeatedly, as shown in Figure 2. Projects may move between Growth, Maturity, and Decline/Revival repeatedly, depending on organisational circumstances.

Wiggins and Crowston [WC10] introduce a model similar to Wynn’s Introduction and Growth stages, and add three classes of outcome to each stage, resulting in projects that might be any of the following tragedy / growth / indeterminate states, as shown in Table 1. They defined a more matrix-like lifecycle model, which, similarly to Wynn, looks at project maturity, and begins with an initiation phase (I), followed by (potentially) a growth phase (G). This combines with outcomes of each phase, as “success” (S) or “tragedy” (T). A project might have a successful initiation phase (SI), but then fail in the growth phase, resulting in a final classification of TG – Tragedy, Growth. If project outcomes were not yet clear, it could also be classified as Indeterminate, resulting in Indeterminate, Initiation (II) and Indeterminate, Growth (IG). See Table 1.

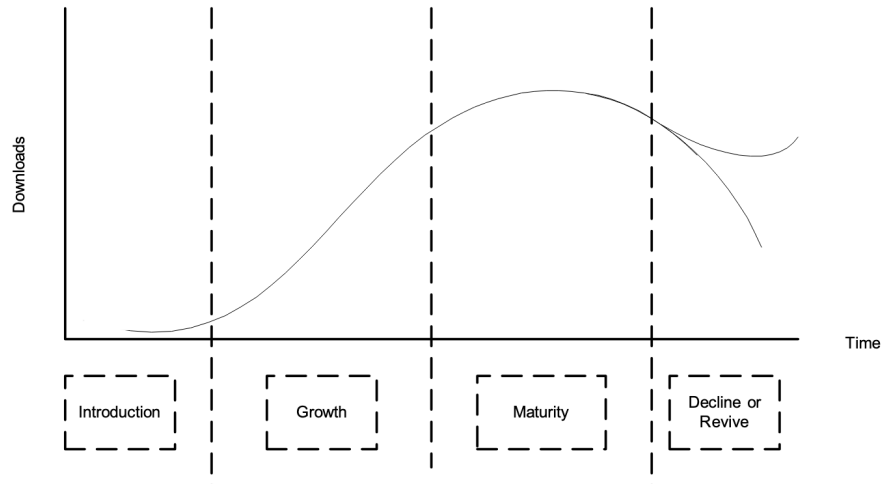


Figure 2: Wynn's four stage open-source project lifecycle [WJ04]

3 Our Initial Research Method

In order to investigate factors contributing to open-source community longevity, we drew on Yehudi's previous work, which monitored thirty-eight open-source projects over the period of a year, focusing primarily, but not exclusively, on open-science-related online code-oriented communities. This investigation resulted in a rich dataset of qualitative and quantitative software-repository-related indicators. The dataset spanned across three time points, with participants answering surveys at Months 0, 6, and 12 [Yeh24].

We also reviewed literature on open-source sustainability, where we encountered multiple open-source lifecycle frameworks, and it seemed natural to investigate whether or not they could be applied to the projects in our study, given the survey and activity data we had available [Yeh24].

Some frameworks we encountered were unsuitable for our study's set of projects. These were for project management, community maturity, or risk management and frameworks tied to a single point in time rather than a lifecycle. Others were for open-source software, but were insufficiently generic, for example, frameworks that are tied specifically to a programming language (such as Java [PPP⁺20]) or designed to be used among homogenous software packages within a specific environment (such as Debian and Debian packages [CH21]).

Two of the lifecycle frameworks we identified were designed for use with open-source software projects, and were sufficiently generic to be applied to a set of heterogeneous projects, or to stand-alone single open-source software projects: Wynn [WJ04], and Wiggins and Crowston [WC10], as introduced in Subsection 2.2.

The success/tragedy model was challenging to map to many of our participating projects, as it did not seem to cover later-stage project maturity and ongoing maintenance needs. It also had relatively strict criteria for each stage, based on downloads, release cycles, and project age. This framework appears to have been suitable for assessing the projects in the study at the time, but did

not seem to fit the heterogeneity of the repositories we were working with. This could be related to the source of the data for the success/tragedy studies, which came from SourceForge. Much like we speculate that GitHub has influenced the shape of modern repositories as a de-facto standards body, it seems plausible that before GitHub's rise in popularity, SourceForge itself might have been a significant influence on the shape and behaviour of open-source repositories.

We attempted to map projects to Wynn's four-stage framework using the study data, to assess its suitability with real-world projects on GitHub. We use indicators for the three survey time points *Initial*, *Growth*, *Maturity*, and *Revival/Decline*. In order to test the applicability of the frameworks to a heterogeneous set of open-source projects, we reviewed each project's initial, midpoint, and final surveys, identifying changes in survey answers that might indicate whether a project was in initial stages, growing, mature/stable, or at risk of decline. This four-stage framework allowed us to identify the indicators as possible ways to map the framework to the data we had about our projects.

Identifying projects in a *growth* or *revival* phase was straightforward to define, when over the period of a year their user base or employee base went up noticeably. The opposite also appeared to be true for *decline*: staff numbers or user numbers going down seemed to be easy to spot. Other stages were harder to identify: projects that were small but stable/mature had indicators that were very similar to projects that were in their initial phase.

The bulk of the projects seemed to fit best in the "decline" phase, but when organized thematically, they had clearly identifiable subgroups. Some project leaders reported plans for low activity "maintenance mode" possibly followed by a graceful slow discontinuation; others reported a lack of resources, but a desire to continue. Project A in the case-study evaluation, for instance, appeared to have entered a decline phase, but exited it again and returned to growth (see [Section 5](#)). They reported having funding and staff at the start of the project, that it had ended in Month 6, but by Month 12 were reporting that while the staff were no longer paid to work on this project, a new grant would be starting soon.

4 A Framework for Modern Open-Source Projects

Many of the software lifecycle frameworks we reviewed seemed to partially fit the projects in different ways. By combining aspects of several of them, we were able to illustrate some of the project examples above more effectively than any of the previous frameworks standing alone.

Below, our initially-proposed lifecycle framework elements are highlighted in-text in **bold**, and references to lifecycle elements from previous literature are underlined. [Figure 3](#) presents our initial open-source project lifecycle framework.

Given projects that illustrated multiple lifecycle phases over the course of a year (such as Project A in the subsequent [Section 5](#)), we intentionally created the framework to be cyclical. You can return to most phases, except the initial phase. A very simple and clearly specified S-program (using Lehman's rules of software evolution [[Leh84](#)]) might move directly from the Initial stage to **Complete**, but we would expect most other projects to be E-programs, and evolve over the course of the project's lifetime, through a series of **Growth events**, **Stability** (analogous to maturity), and **Challenge events**.

Projects that are stable – perhaps with a guaranteed revenue stream, via grant funding or cor-

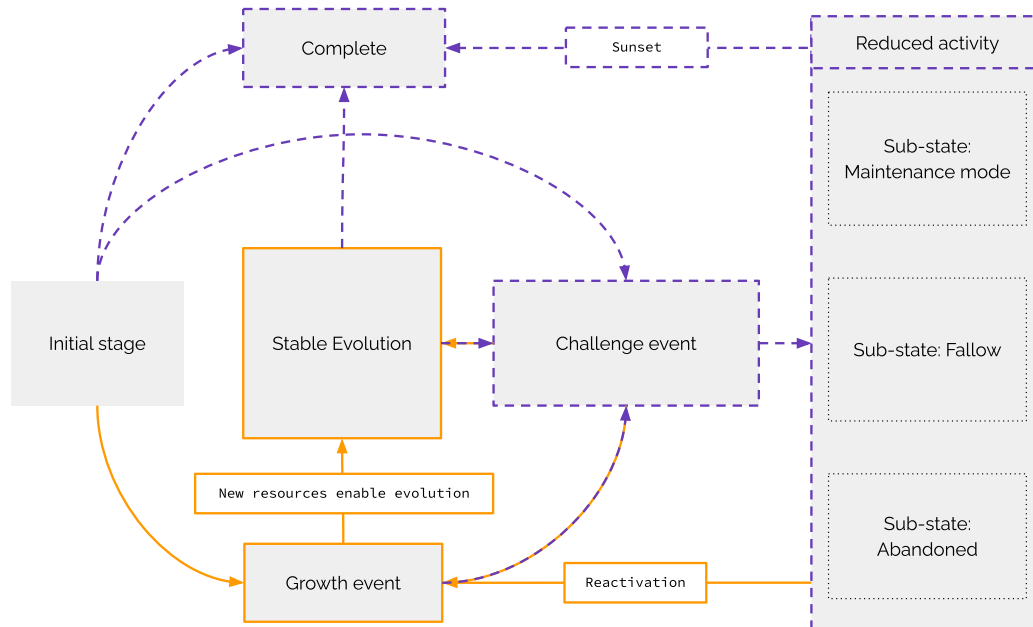


Figure 3: Our open-source project lifecycle framework – dashed purple lines indicate reduction of activity, solid yellow lines indicate stability or growth.

porate backing – eventually face challenges of some kind. This could be funding running out, a corporate takeover changing organizational priorities, as happened with SourceForge [TBPK20], or one or more core developers leaving – a Truck Factor Developer Detachment (TFDD) event [ACVS19].

Coming out of the challenge event successfully, projects might return directly to the stable phase, or go through another growth event. But similar to the tragedy/success model [WC10] and Wynn’s four-stage model [WJ04], outcomes might not end up returning to growth or stability for the project. The tragedy-analogous pathway heads to reduced activity. Reasons to head towards the reduced activity state might be one of the reasons for open-source project failure (project, environmental, and team factors) [CV17].

We define three **Reduced activity** substates (on the right in Figure 3), depending on why the project is in the reduced activity state.

If a project is aware that it is heading towards a final endpoint, and does not seek to change it, we dub this **Maintenance mode**. Project 15 of the case studies in [Yeh24] is one example of this. The project leader notes that Perl is not a growing language, and that many of the functionalities of the project have been picked up by other projects, which presumably are in programming languages with higher modern adoption levels. They foresee the project gracefully sunsetting, if the other projects pick up any remaining unique functionality that is currently only offered by this project. During the study period, there were only two commits to the project over the whole year [Yeh24].

Some projects may desire returning to active and stable states. The second reduced activity project state we observed in our data was when a project had a desire to continue, and the leader

believed there was a genuine niche for their work, but they no longer had sufficient resources to continue working on the project. We refer to this as a **Fallow** state, as it may resume stable activity in the future, if resources permit.

None of the projects in this study were abandoned, but abandoned projects are well-evidenced in literature. We define **Abandoned** as no longer receiving updates from previously core maintainers, but note that community members and users of the project might still desire updates to the project. This is different from the fallow state, as in fallow projects the project leadership still intends to work actively on the project.

An example of project abandonment where the owner no longer updates the repository, but that is still desired by the community, might be Mozilla’s jschannel library [Moz13]. At the time of writing, the original repository has received no active updates since 2013 – over twelve years ago. Issues in the repository show that users had been asking for the project to be published on JavaScript package managers such as bower or npm, without response from Mozilla. Eventually, the conversations in the issues show that users forked the project and released the project on these package managers themselves. As of 2025, nine years after the last forked project update, the community-published version of jschannel still receives over 2,600 downloads a week, and has projects depending on it such as PeerTube [Moz23], an open-source YouTube alternative, which has over 13,000 stars on GitHub, over 1,600 forks, and hosts more than 600,000 videos across a federated network [Cho25].

Avelino et al. define a new Truck Factor developer picking the project up as “reactivation” [ACVS19]. This is one reason a project might reactivate, but another might be if a project receives a new injection of resources in some other form, perhaps grant funding. This is illustrated in Project A, which we mapped to both growth and decline phases, when testing Wynn’s four-stage model.

5 Initial Framework Case Study Evaluation

In the following, we present four case studies drawn from projects studied by Yehudi while developing the initial framework from the previous Section 4 [Yeh24]. The full case-study research with thirty-eight open-source projects is presented by Yehudi [Yeh24].

Project A (Yehudi’s Project 95 [Yeh24]) was founded five years before the study began, reported 100-1000 users or contributors (collected through a survey) and had 191 commits (gathered via source control) over the period of the study. The project’s co-founder and PI initially reported that it was grant funded, but noted that grant funding ceased in Month 6 (M06). This was evident in the staff count, which dropped from five paid staff in Month 0 (M0) to four staff in Month 6, and only one in Month 12 (M12), see Figure 4 and Table 2.

Project B (Yehudi’s Project 23 [Yeh24]) was around a year old at the start of the study, was funded, and had a small leadership team of two. It was unable to secure further funding after the initial grant, and ultimately it entered a reduced activity “fallow” phase, see Table 3.

The participant offered a narrative around the circumstances of their project, explaining their sustainability and funding situation:

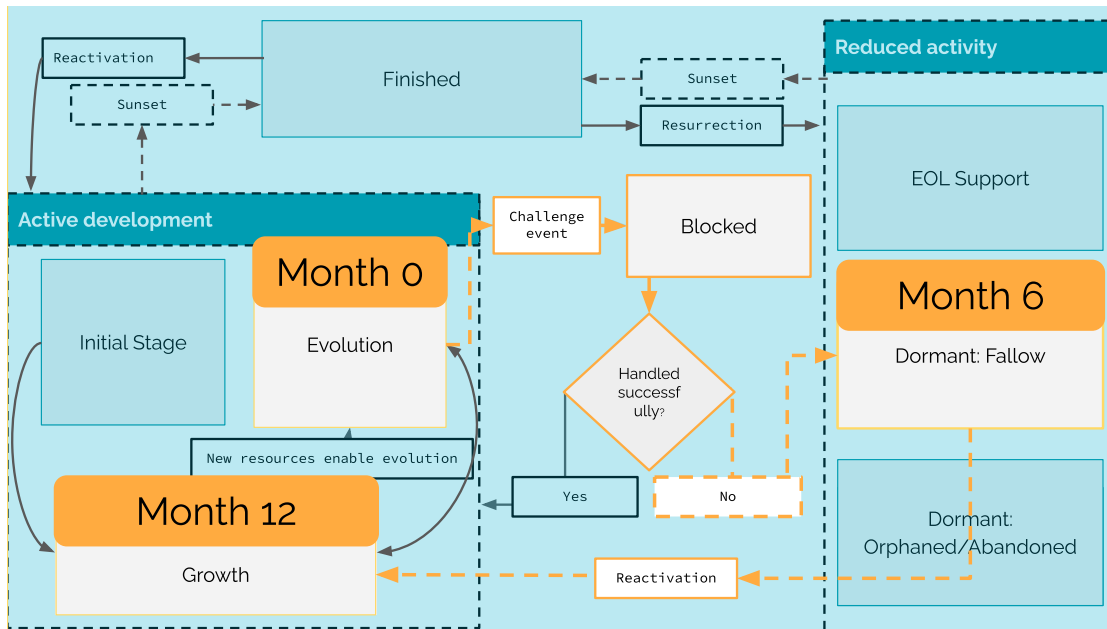


Figure 4: Project A's reported states, mapped to our project lifecycle framework.

	M0	M06	M12
Lifecycle stage	Stable evolution	Challenge event, Fallow	Reactivation, Growth event
Paid Staff	5	4	1
Funding	Yes	No	New grant starting soon "we have 1 who is very active, and a total of 50-100 over time"
Users / Contributors	100-1000	100-1000	

Table 2: Indicators for Project A as a framework case study.

	M0	M06	M12
Lifecycle stage	Initial, or Stable evolution	Challenge event, Fallow	Fallow
Paid Staff	3	0	0
Funding	Yes	No	No
Users / Contributors	20-50	20-50	1-10

Table 3: Lifecycle stages for Project B, based on interpretation of survey data.

	M0	M06	M12
Lifecycle stage	Initial	Growth event, Stable evolution	Complete
Paid Staff	1	6	0
Funding	Yes	Yes	No
Users / Contributors	1-10	1-10	1-10

Table 4: Indicators for Project C as a framework case study.

M0 “We are a young project and I’m not aware of anyone who would be able to maintain it now, without funding, if I wasn’t able. If I became unable now, I would offer it to anyone to take over and aim to provide input when I could. I hope we can move towards being maintainable by others.”

M06 “I am not currently funded to work on this project. We are committed to Project B, still believe it addresses an otherwise unmet opportunity, but are struggling to keep it funded.”

M12 “As part of a funded project we tried to create enough to demonstrate utility and need and start a community. We hoped to use this to attract more funding. At this stage we have not succeeded in those objectives.”

Project C (Yehudi’s Project 85 [Yeh24]) was one of the newest participating projects, founded the same year as the study was conducted, and always had a small scope throughout the study. It had a total of 27 commits on GitHub during the study period, never anticipated more than 10 users, and had ceased accepting contributions at the end of the study. This closure was planned, consistent with the respondent’s assertion in Month 0 that they expected the project would be wrapped up in one year’s time. As such, it did not appear to move through any challenge events or reduced activity periods. Table 4 shows the indicators for Project C.

Project D (Yehudi’s Project 73 [Yeh24]) was the largest of our studied projects, one of the oldest (founded in 2003), and appeared to be in the growth phase, but on the edge of a potential challenge phase the whole time, with a number of paid staff that was lower than many of our smaller projects. Free-text responses from the project lead notes that they were the third lead developer of the project, suggesting the project may have survived a TFDD event in the past. Table 5 shows the indicators for Project D.

6 Extending our Lifecycle Framework to Research Software

Through a collaborative discussion at a Dagstuhl seminar,¹ we extended the open-source lifecycle framework to address typical phases in research software development. Figure 5 presents the extended lifecycle model for open-source research software [HCH⁺20]. Notable changes

¹ <https://www.dagstuhl.de/24161>

	M0	M06	M12
Lifecycle stage	Stable evolution	Growth event	Growth event
Paid Staff	2	2	1-3 (1 full time, several people are able to spend work hours on it)
Funding	Yes	Yes	Partial salary support from [Private philanthropy] and soon from [Government agency]
Users / Contributors	150,000	1M+ users, few hundred contributors a year, 15ish regular contributors	1M-5M

Table 5: Indicators for Project D as a framework case study.

to [Figure 3](#) are several new state transitions: While the original model does not allow leaving the *Complete* state, the new proposed model now has a state transition *Reactivation* (allowing to move from *Finished* back to *Active development*) and a state transition *Resurrection* (allowing to move from *Finished* back to *Reduced activity*).

To better reflect the dynamic nature of research software development, the *Active Development* state group introduces a development loop. This loop acknowledges that within the two stages *Growth* and *Evolution*, typical software development processes often revisit previous phases when new resources become available. These program iterations can arise with new requirements, changes in the research context, or evolving software objectives. An explicit inclusion of such a loop in our proposed model underscores this non-linear progression typical for research software.

Moreover, the new proposed model includes a *Blocked* state. Here, the *Challenge event* of the previous model can be resolved both successfully or unsuccessfully: depending on the outcome of how the *Challenge event* is dealt with, the new model now provides state transitions to both *Active development* (successful handling of the challenge) and *Reduced activity* (unsuccessful handling of the challenge).

Finally, the previously ungrouped states *Initial Stage*, *Growth event*, and *Stable evolution* are now grouped under *Active development* to indicate that a *Challenge event* may interrupt the active development at any stage within this new state group.

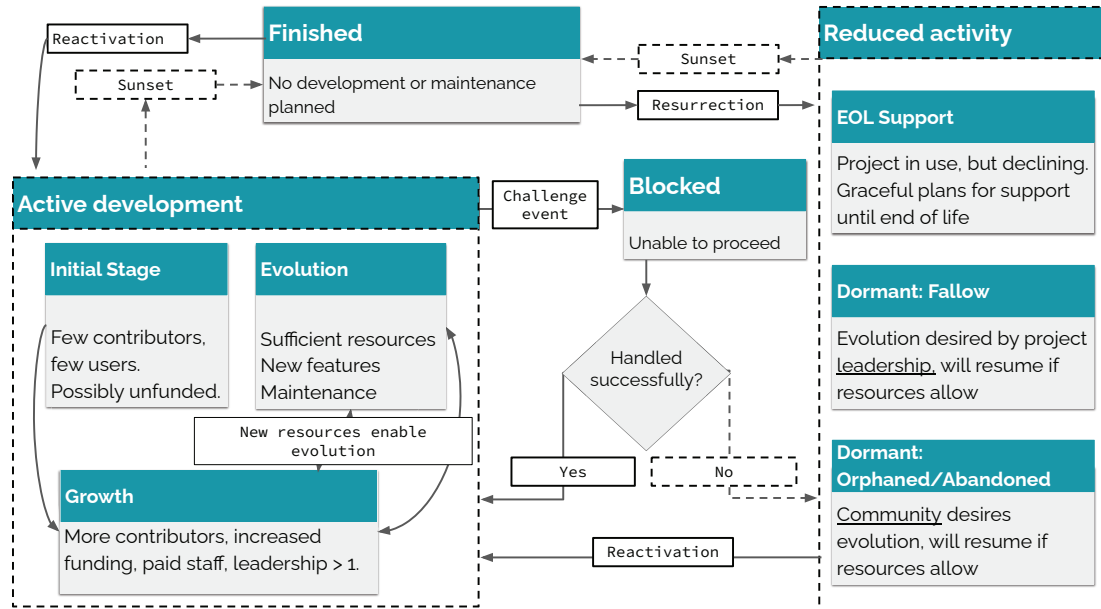


Figure 5: Research software lifecycle.

7 Framework Evaluation

To evaluate our extended framework, we illustrate and discuss the relation with other published frameworks. We map our proposed open-source community lifecycle framework to related frameworks in the literature. The following figures show how our proposed lifecycle framework takes other lifecycle frameworks into account. Figure 6 overlays Wynn’s four-stage open-source project lifecycle model over our model. Figure 7 overlays the reasons why open-source projects fail, and Truck Factor Developer Detach events over our model. Figure 8 overlays the Success/Tragedy lifecycle model over our model. Figure 9 overlays the underproduction and stable production over our model.

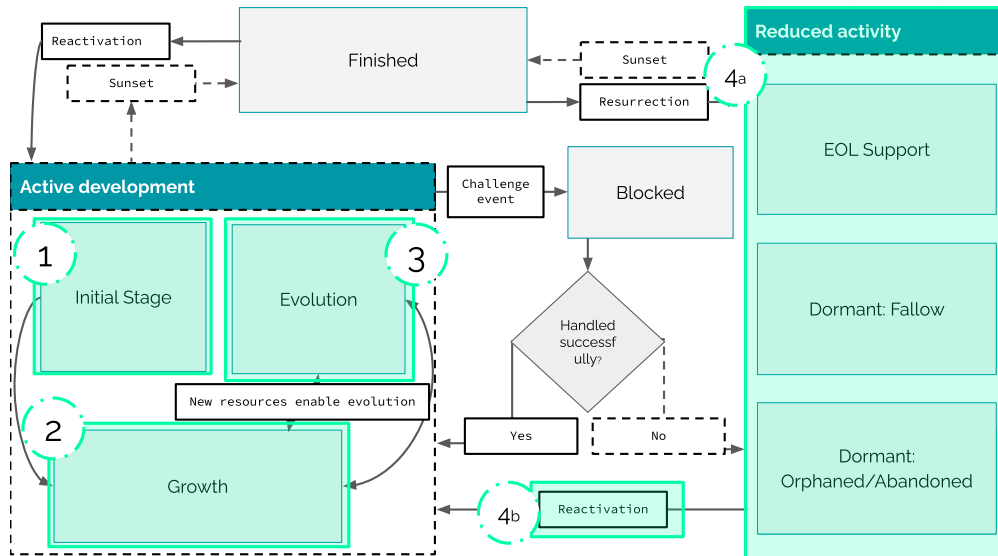


Figure 6: Wynn's four-stage open-source project lifecycle model, overlaid over our proposed model. 1: Introduction, 2: Growth, 3: Maturity, 4a: Decline, 4b: Revival.

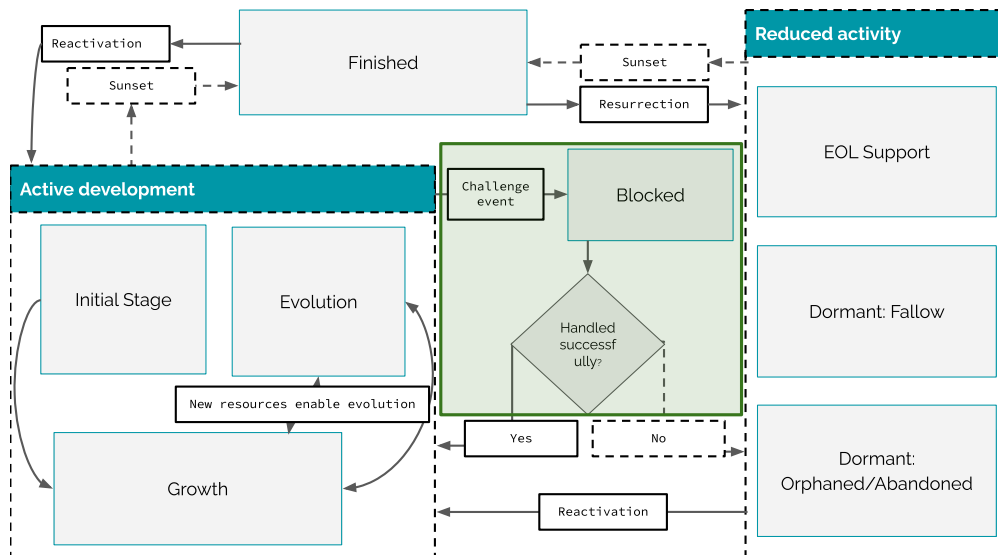


Figure 7: Reasons why open-source projects fail, and Truck Factor Developer Detach events overlaid over our proposed model.

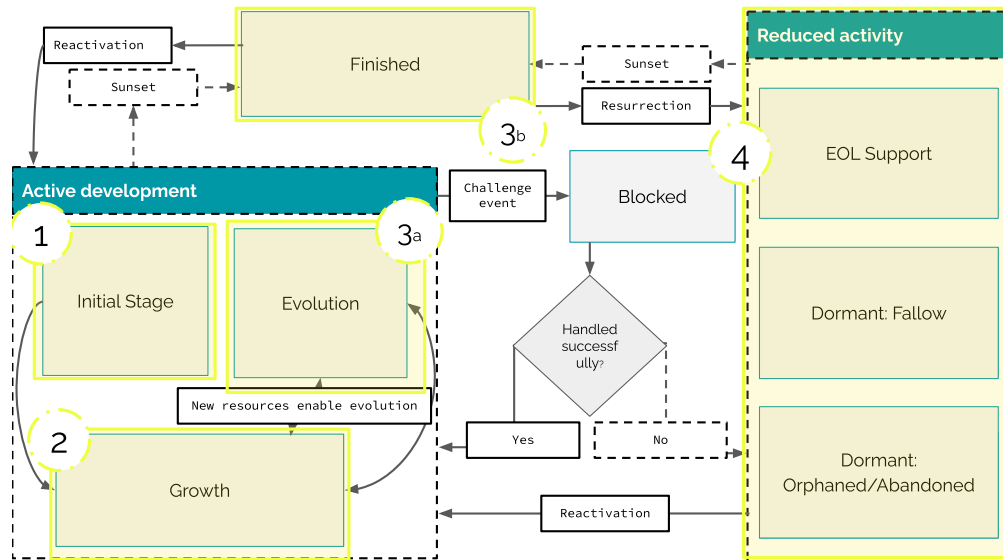


Figure 8: Success/Tragedy lifecycle model, overlaid over our proposed model. 1: Initiation, 2: Growth, 3a and 3b: Success, 4: Tragedy.

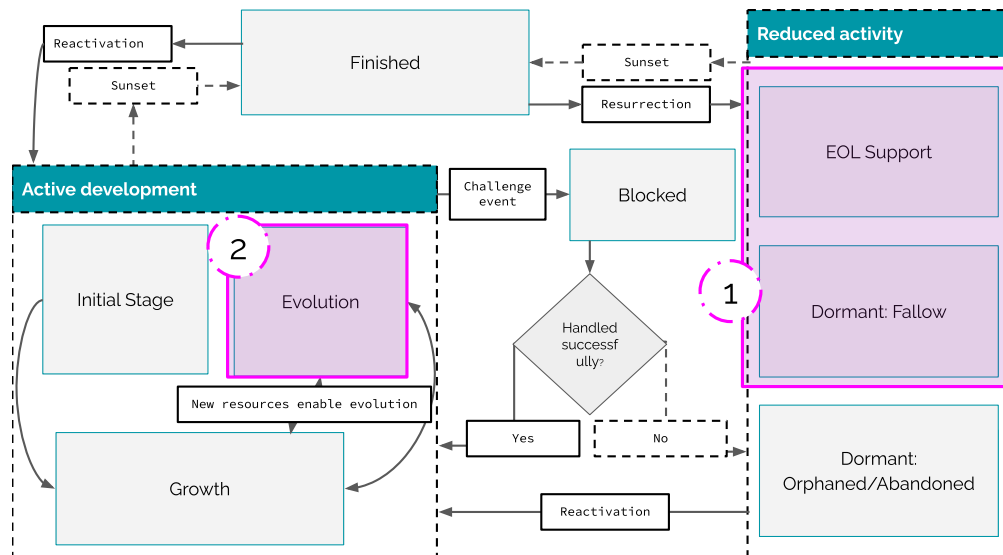


Figure 9: Underproduction and stable production, overlaid over our proposed model. 1: Underproduction, 2: Stable production.

8 Research Software Stages Categorization

Research software may be categorized into multiple dimensions. In the realm of RSE research, that is, research on research software engineering (RSE) [FGG⁺25], such categorizations provide a framework for classifying research objects, supporting software corpus analyses, and enhancing our understanding of the different types of research software and their properties. This structured approach may help organize and interpret the vast landscape of research software, contributing to advancements in RSE methodologies and practices. [KJB25]

Hasselbring et al. present a multidimensional categorization of research software along the dimensions of roles, readiness, developers, and dissemination [HDB⁺25]. Here, we add another dimension in Figure 10: research software *stages*, which are derived from our research software lifecycle model in Figure 5. A major difference between commercial software lifecycle models (Figure 1 on Page 3) and our research software lifecycle models is the highlighted *Blocked* stage in Figure 10, which is caused by *Challenge events* (see Section 6). Please note that commercial software could also be blocked by Challenge events, such as staff leaving, budget cuts, changing consumer habits, competing products, changing legislation, etc. However, these Challenge events are not yet included in commercial software lifecycle models.

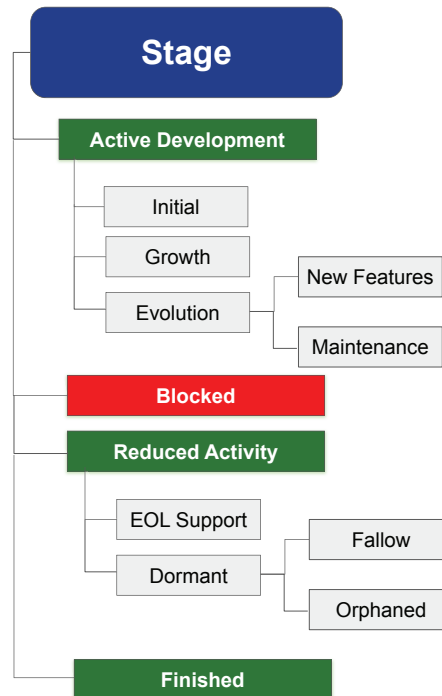


Figure 10: Research software stages, derived from Figure 5.

Research software has been categorized in different contexts to serve different aims. Hinsen, for instance, categorizes along the research software stack, from non-scientific infrastructure, scientific infrastructure, discipline-specific software, up to project-specific software [Hin19]. Van Nieuwpoort and Katz [NK24] present a role-based categorization.

More related to our stage dimension are categorizations of the *maturity* of research software. In their National Agenda for Research Software [HT21], the Australian Research Data Commons describe a three-level maturity categorization of research software:

1. *Research Data Processes* captured as software. The result is analysis code that captures research processes and methodology: the steps taken for tasks like data generation, preparation, analysis, and visualization.
2. *Novel Methods and Models* captured as software. The results are prototype tools that demonstrate a new idea, method, or model for research.
3. *Accepted Methods and Models* captured as software. The result can become research software infrastructure that captures more broadly accepted and used ideas, methods, and models for research.

Each category faces specific challenges with regard to recognition, from making research practice transparent, to creating impact through quality software and safeguarding longer-term maintenance.

Institutional guidelines typically define so-called application classes for research software, which require appropriate quality properties, and, thus software engineering methods [SMH18]:

- For software in Application Class 0, the focus is on personal use in conjunction with a small scope.
- For software in Application Class 1, it should be possible, for those not involved in the development, to use it to the extent specified and to continue its development.
- For software in Application Class 2, it is intended to ensure long-term development and maintainability. It is the basis for a transition to product status.
- For software in Application Class 3, it is essential to avoid errors and to reduce risks. This applies in particular to critical software.

Maturity is related to these application classes, and also to technology readiness [HDB⁺25]. The European Virtual Institute for Research Software Excellence (EVERSE) has released a Research Software Quality Kit (RSQkit) that includes a three-tier view [EVE25]. This categorization is based on the ARDC model, consisting of analysis code, prototype tools, and research software infrastructure, though this also roughly corresponds to DLR's class 0, 1–2, and 2–3, respectively.

Beyond such basic categorizations of research software maturity, more comprehensive frameworks have recently been proposed. RSMM, for instance, offers a structured pathway for evaluating and refining research software *project management* by categorizing 79 best practices into 17 capabilities across 4 focus areas [DBM⁺24]. The Helmholtz Association develops a quality indicator for research data and research software *publications*, which is based on the COBIT Maturity Model [CDJ⁺24].

The European Open Science Cloud (EOSC) aims to create a virtual environment for sharing and accessing research data across borders and scientific disciplines. Subgroup 1 “On the Software Lifecycle” of the EOSC Task Force “Infrastructure for quality research software” provides a categorization for software in the research lifecycle [CFG⁺23]:

1. Individual creating research software for own use (e.g., a PhD student).
2. A research team creating an application or workflow for use within the team.
3. A team / community developing (possibly broadly applicable) open-source research software.
4. A team or community creating a research service.

These are stage categories in the developer dimension, according to Hasselbring et al. [HDB⁺25].

9 Discussion

In this paper, we present several (graphical) models for research software lifecycles and stages.

The often cited statement “All models are wrong, but some are useful” by George Box [Box76] emphasizes the value of abstraction in order to gain insight. Although abstraction inherently involves some degree of misrepresentation, it enables the explicit definition of model assumptions and the interpretation of results within those boundaries. In interdisciplinary research, the modelling process itself—along with the dialogue surrounding its assumptions—is often seen as especially valuable. This process fosters a structured way of thinking about complex systems and the mechanisms underlying observed phenomena.

A challenge is that software in general, but apparently research software in particular, can be in so many states that a comprehensive framework seems hard to define. So, our models for research software lifecycles and stages should be viewed under this assumption, whereby we consider them helpful in practice and in understanding research software projects.

Thus, to elicit feedback from the wider RSE community, our extended lifecycle model for research software (Figure 5) was presented at both US-RSE’24 [YCF⁺24] and deRSE25 [YCF⁺25].

Feedback Received at US-RSE’24: The feedback on Figure 5, elicited at the US-RSE conference, was mostly positive and indicating a need to use the overview during a project’s life:

- “This will be useful to have as a base to define work planning and billing.”
- “This diagram is useful to help us keep better (more detailed) track of our software lifecycles and think about user/stakeholder needs.”
- One person suggested that another state should be added: “Stable functioning, mostly user support, no big changes, only bug fixes”. This state could be extracted as a substate from the state group *EOL Support*.

Feedback Received at deRSE25: The additional feedback elicited at the deRSE conference, addressed mainly proposed additions to the model and extended descriptions of the model:

- In terms of the motivation questions at the top, an additional one for developers might be “How easy is it?”
- This overall effort should be compared with the EOSC research software lifecycle, which isn’t focused on software but does include it as part of research.

- The Active Development box could also include “and operations”.
- The Resurrection box between Finished and Reduced Activity might need an example. The poster viewer and presenter were not able to describe a situation where this would happen. This might be seen as a more general comment: that all boxes and arrows would benefit from explanations and in some cases, examples.
- Regarding the Growth box, a viewer asked “Is leadership > 1 needed?”, implying that having a single leader might be acceptable in some situations.

A comparison with the EOSC Research Software Lifecycle, for instance, may be found in the previous [Section 8](#). In our future work, we will consider this feedback, and plan for a systematic evaluation of the extended framework.

10 Conclusions and Future Work

In this work, we developed and evaluated a comprehensive and flexible software lifecycle framework that reflects the unique challenges of research software. In review of related work on commercial and open-source software lifecycles, we noted limitations in addressing the heterogeneous, irregular development patterns, and longitudinal and cyclical nature of many research software projects.

Extending on proposed themes from related work, we developed an initial model informed by case studies of thirty-eight open-source projects evaluated across several factors. This expanded model demonstrates the ability to track projects following cyclical development, subject to different types of interrupted progress, and distinct categories of reduced activity. We demonstrated this new framework across four representative projects to illustrate its utility.

This model was then refined through collaborative discussion at a Dagstuhl seminar, introducing additional state transitions, and expanding details on blocked and active development stages. We highlight the overlap and extensions in the new model compared to prior literature and present evaluation from the research software community.

As highlighted by our community feedback, this research software oriented lifecycle model could be a useful foundation for work planning, billing, and tracking software progress. Its relevance extends across a wide range of stakeholders, including software users, developers, principal investigators, funders, and other decision makers in the research software ecosystem.

We proposed this framework with the goal that others might find it useful in explaining events and flow throughout an open-source project lifecycle. In future work, more projects could be observed across multiple time points to evaluate the framework, beyond the illustrative case studies we have provided here, and iterated upon where needed.

Lu et al. analysed open-source software repositories to gain a comprehensive understanding of how open-source software projects and companies operate [LKK⁺25]. A similar analysis of open-source *research* software could be a topic for future research.

In addition, the use of the Essence Framework [JLN⁺19], [OMG24] might be useful for the specific profiles of the various types of workflows. This framework is focussed on defining maturity related indicators which are used to guide the iterative, sometimes meandering paths through a research software development project.

Acknowledgements: Part of the work was funded by the NFDIxCs project of the German NFDI initiative by the DFG project number 501930651.

Bibliography

- [A⁺19] K. Akiyama et al. First M87 Event Horizon Telescope Results. I. The Shadow of the Supermassive Black Hole. *The Astrophysical Journal Letters* 875(1), Apr. 2019.
[doi:10.3847/2041-8213/ab0ec7](https://doi.org/10.3847/2041-8213/ab0ec7)
- [ACVS19] G. Avelino, E. Constantinou, M. T. Valente, A. Serebrenik. On the abandonment and survival of open source projects: An empirical investigation. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Pp. 1–12. 2019.
[doi:10.1109/ESEM.2019.8870181](https://doi.org/10.1109/ESEM.2019.8870181)
- [Box76] G. E. Box. Science and Statistics. *Journal of the American Statistical Association* 71(356):791–799, Dec. 1976.
[doi:10.1080/01621459.1976.10480949](https://doi.org/10.1080/01621459.1976.10480949)
- [CDJ⁺24] W. zu Castell, D. Dransch, G. Juckeland, M. Meistring, B. Fritzsche, R. Gey, B. Höpfner, M. Köhler, C. Meeßen, H. Mehrrens, F. Mühlbauer, S. Schindler, T. Schnicke, R. Bertelmann. Towards a Quality Indicator for Research Data publications and Research Software publications – A vision from the Helmholtz Association. *arXiv*, 2024.
[doi:10.48550/arXiv.2401.08804](https://doi.org/10.48550/arXiv.2401.08804)
- [CFG⁺23] G. Courbebaisse, B. Flemisch, K. Graf, U. Konrad, J. Maassen, R. Ritz. Research Software Lifecycle. *Zenodo*, Sept. 2023.
[doi:10.5281/zenodo.8324828](https://doi.org/10.5281/zenodo.8324828)
- [CH21] K. Champion, B. M. Hill. Underproduction: An approach for measuring risk in open source software. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Pp. 388–399. 2021.
[doi:10.1109/SANER50967.2021.00043](https://doi.org/10.1109/SANER50967.2021.00043)
- [Cho25] Chocobozzz/PeerTube. ActivityPub-federated video streaming platform using P2P directly in your web browser. *GitHub*, 2025.
<https://github.com/Chocobozzz/PeerTube>
- [CKB⁺21] J. Cohen, D. S. Katz, M. Barker, N. Chue Hong, R. Haines, C. Jay. The Four Pillars of Research Software Engineering. *IEEE Software* 38(1):97–105, Jan. 2021.
[doi:10.1109/ms.2020.2973362](https://doi.org/10.1109/ms.2020.2973362)
- [CV17] J. Coelho, M. T. Valente. Why modern open source projects fail. In *Proceedings of the 2017 11th Joint meeting on foundations of software engineering*. Pp. 186–196.

2017.
[doi:10.1145/3106237.3106246](https://doi.org/10.1145/3106237.3106246)
- [DBM⁺24] Deekshitha, R. Bakhshi, J. Maassen, C. M. Ortiz, R. van Nieuwpoort, S. Jansen. RSMM: A Framework to Assess Maturity of Research Software Project. *arXiv*, 2024.
[doi:10.48550/arXiv.2406.01788](https://doi.org/10.48550/arXiv.2406.01788)
- [EVE25] EVERSE. Three-Tier Model of Research Software. *Research Software Quality Toolkit (RSQKit)*, Feb. 2025.
https://everse.software/RSQKit/three_tier_view
- [FGG⁺25] M. Felderer, M. Goedicke, L. Grunske, W. Hasselbring, A.-L. Lamprecht, B. Rumpe. Investigating Research Software Engineering: Toward RSE Research. *Communications of the ACM* 68(2):20–23, 2025.
[doi:10.1145/3685265](https://doi.org/10.1145/3685265)
- [HBB16] P.-S. Huang, S. E. Boyken, D. Baker. The coming of age of de novo protein design. *Nature* 537(7620):320–327, 2016.
[doi:10.1038/nature19946](https://doi.org/10.1038/nature19946)
- [HCH⁺20] W. Hasselbring, L. Carr, S. Hettrick, H. Packer, T. Tiropanis. Open Source Research Software. *Computer* 53(8):84–88, Aug. 2020.
[doi:10.1109/MC.2020.2998235](https://doi.org/10.1109/MC.2020.2998235)
- [HDB⁺25] W. Hasselbring, S. Druskat, J. Bernoth, P. Betker, M. Felderer, S. Ferenz, B. Hermann, A.-L. Lamprecht, J. Linxweiler, A. Prat, B. Rumpe, K. Schoening-Stierand, S. Yang. Multi-Dimensional Research Software Categorization. *Computing in Science & Engineering*, 2025.
[doi:10.1109/mcse.2025.3555023](https://doi.org/10.1109/mcse.2025.3555023)
- [Hin19] K. Hinsén. Dealing With Software Collapse. *Computing in Science Engineering* 21(3):104–108, May 2019.
[doi:10.1109/MCSE.2019.2900945](https://doi.org/10.1109/MCSE.2019.2900945)
- [HT21] T. Honeyman, A. Treloar. A National Agenda for Research Software (0.9). *Zenodo*, June 2021.
[doi:10.5281/zenodo.4940274](https://doi.org/10.5281/zenodo.4940274)
- [JLN⁺19] I. Jacobson, H. B. Lawson, P.-W. Ng, P. E. McMahon, M. Goedicke. *The Essentials of Modern Software Engineering: Free the Practices from the Method Prisons!* Association for Computing Machinery and Morgan & Claypool, 2019.
[doi:10.1145/3277669](https://doi.org/10.1145/3277669)
- [KJB25] D. S. Katz, E. A. Jensen, M. Barker. Understanding and advancing research software grant funding models. *Open Research Europe*, 2025. submitted.

- [Leh84] M. M. Lehman. Program evolution. *Information Processing & Management* 20(1-2):19–36, 1984.
[doi:10.1016/0306-4573\(84\)90037-2](https://doi.org/10.1016/0306-4573(84)90037-2)
- [LKK⁺25] W. Lu, E. Kasaadah, S. M. R. U. Karim, M. Germonprez, S. Goggins. Open Source Software Lifecycle Classification: Developing Wrangling Techniques for Complex Sociotechnical Systems. *arXiv*, 2025.
[doi:10.48550/arXiv.2504.16670](https://doi.org/10.48550/arXiv.2504.16670)
- [Moz13] Mozilla. jschannel: A JavaScript library which implements fancy IPC semantics on top of postMessage. *GitHub*, 2013.
<https://github.com/mozilla/jschannel>
- [Moz23] Mozilla. packages depending on jschannel. *npm*, 2023.
<https://www.npmjs.com/browse/depended/jschannel>
- [NK24] R. van Nieuwpoort, D. S. Katz. Defining the roles of research software (Version 2). *Upstream*, July 2024.
[doi:10.54900/xdh2x-kj281](https://doi.org/10.54900/xdh2x-kj281)
- [OMG24] OMG. About the Essence Specification Version 2.0 beta. *Essence*, 2024.
<https://www.omg.org/spec/Essence>
- [PPP⁺20] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, F. Massacci. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering* 48(5):1592–1609, 2020.
[doi:10.1109/TSE.2020.3025443](https://doi.org/10.1109/TSE.2020.3025443)
- [RB00] V. Rajlich, K. Bennett. A staged model for the software life cycle. *Computer* 33(7):66–71, July 2000.
[doi:10.1109/2.869374](https://doi.org/10.1109/2.869374)
- [RGH⁺19] R. Reussner, M. Goedicke, W. Hasselbring, B. Vogel-Heuser, J. Keim, L. Martin. *Managed Software Evolution*. Springer, Cham, 2019.
[doi:10.1007/978-3-030-13499-0](https://doi.org/10.1007/978-3-030-13499-0)
- [SMH18] T. Schlauch, M. Meinel, C. Haupt. DLR Software Engineering Guidelines. *Zenodo*, Aug. 2018.
[doi:10.5281/zenodo.1344612](https://doi.org/10.5281/zenodo.1344612)
- [TBPK20] D. A. Tamburri, K. Blincoe, F. Palomba, R. Kazman. “The Canary in the Coal Mine...” A cautionary tale from the decline of SourceForge. *Software: Practice and Experience* 50(10):1930–1951, 2020.
[doi:10.1002/spe.2874](https://doi.org/10.1002/spe.2874)
- [WC10] A. Wiggins, K. Crowston. Reclassifying success and tragedy in FLOSS projects. In *IFIP International Conference on Open Source Systems*. Pp. 294–307. 2010.
[doi:10.1007/978-3-642-13244-5_23](https://doi.org/10.1007/978-3-642-13244-5_23)

- [WJ04] D. E. Wynn Jr. Organizational structure of open source projects: A life cycle approach. *SAIS 2004 Proceedings*, 2004.
<https://aisel.aisnet.org/sais2004/47>
- [YCF⁺24] Y. Yehudi, M. Cashman, M. Felderer, M. Goedicke, W. Hasselbring, D. S. Katz, F. Löffler, S. Müller, B. Rumpe. Towards Defining Lifecycles and Categories of Research Software. *USRSE24 Conference Proceedings*, 2024.
[doi:10.5281/ZENODO.13974637](https://doi.org/10.5281/ZENODO.13974637)
- [YCF⁺25] Y. Yehudi, M. Cashman, M. Felderer, M. Goedicke, W. Hasselbring, D. S. Katz, F. Löffler, S. Müller, B. Rumpe. Towards Defining Lifecycles and Categories of Research Software. *deRSE25 Conference Proceedings*, 2025.
[doi:10.5281/ZENODO.15002660](https://doi.org/10.5281/ZENODO.15002660)
- [Yeh24] Y. Yehudi. *Human factors in research software engineering: an exploration of individuals, communities, and systemic challenges*. PhD thesis, The University of Manchester, Department of Computer Science, 2024.
<https://research.manchester.ac.uk/en/studentTheses/human-factors-in-research-software-engineering-an-exploration-of->