Welcome

Contents

Welcome to the Software Gardening Almanack, an open-source handbook of applied guidance and tools for sustainable software development and maintenance.

Inspiration

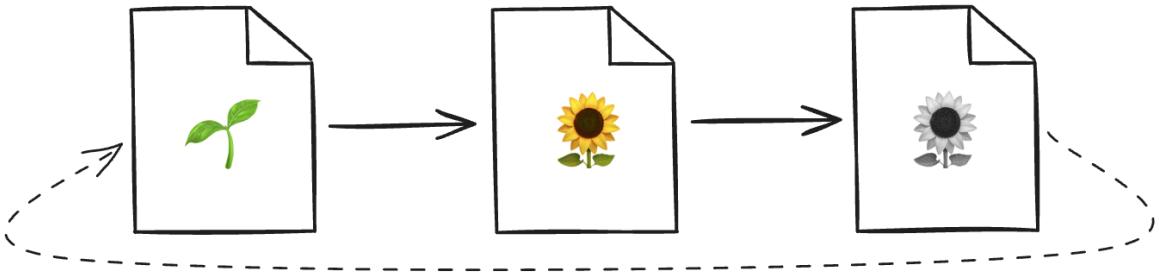


Fig. 1 Software is created, grows, and decays over time.

Software experiences development cycles, which accumulate errors over time. However, these cycles are not well understood nor are they explicitly cultivated with the impacts of time in mind. Why does software grow quickly only to decay just as fast? How do software bugs seem to appear in unlikely scenarios?

These cycles follow patterns from life: software is created, grows, decays, and so on (sometimes in surprising or seemingly unpredictable ways). Software is also connected within a complex system of relationships (similar to the complex ecology of a garden). The Software Gardening Almanack posits we can understand these software lifecycle patterns and complex relationships in order to build tools which sustain or maintain their development long-term.

> *"The 'planetary garden' is a means of considering ecology as the integration of humanity – the gardeners – into its smallest spaces. Its guiding philosophy is based on the principle of the 'garden in motion': do the most **for**, the minimum **against**."* - Gilles Clément

The content within the Software Gardening Almanack is inspired by ecological systems (for example, as in *planetary gardening*). We also are galvanized by the *scientific method*, and *almanacks* (or earlier *menologia rustica*) in documenting, expecting, and optimizing how we plan for time-based changes in agriculture (among other practices and traditions).

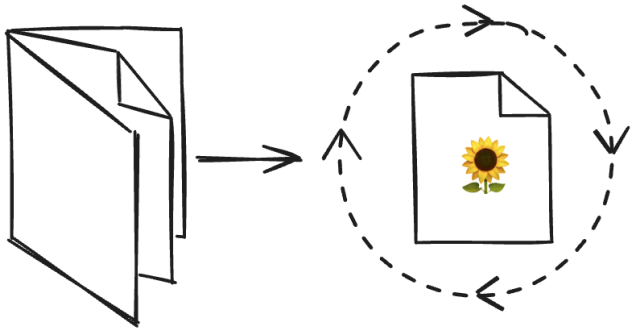


Fig. 2 Almanacks help us understand and influence the impacts of time on the things we grow.

The Software Gardening Almanack helps share knowledge on how to cultivate change in order to nurture software (and software practitioners) for long periods of time. We aspire to define, practice, and continually improve a craft of *software gardening* to nourish existing or yet-to-be software projects and embrace a more resilient future.

Motivation

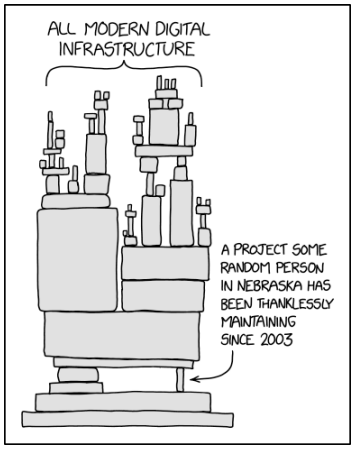


Fig. 3 We depend on a delicate network of software which changes over time.
(Image source: 'Dependency' comic by Randall Munroe, XKCD)

Our world is surrounded by systems of software which enable and enhance life. These systems are both invisible and sometimes brittle, breaking in surprising ways (for example, beneath uneven pressures as in the above XKCD comic). At the same time, loose coupling of these software systems also forms the basis of innovation through multidimensional, emergent growth patterns. We seek to embrace these software systems which are innately in motion, embracing diversity without destroying it to perpetuate discovery and enhance future life.

> *"Start where you are. Use what you have. Do what you can."* - Arthur Ashe

We suggest reading or using this content however it best makes sense to a reader. Just as with gardening, sometimes the best thing to do is jump in! We structure each section in a modular fashion, providing insights with cited prerequisites. We provide links and other reference materials for further reading where beneficial. Please let us know how we can improve by opening GitHub issues or creating new discussion board posts (see our `contributing.md` guide for more information)!

Who’s this for?

> *“We’ll share stories from the heart. All are welcome here.”* - Alexandra Penfold

The Software Gardening Almanack is designed for developers of any kind. When we say *developers* here we mean anyone engaging in pursuing their vision towards a goal using software (including scientists, engineers, project leads, managers, etc). The content will engage readers with pragmatic ideas, keeping the barrier to entry low.

Acknowledgements

This work was supported by the Better Scientific Software Fellowship Program, a collaborative effort of the U.S. Department of Energy (DOE), Office of Advanced Scientific Research via ANL under Contract DE-AC02-06CH11357 and the National Nuclear Security Administration Advanced Simulation and Computing Program via LLNL under Contract DE-AC52-07NA27344; and by the National Science Foundation (NSF) via SHI under Grant No. 2327079.

Garden Lattice

Fig. 4 The garden lattice facilitates growth on our software gardening journeys. (Image source: GL1).

> “To plant a garden is to believe in tomorrow.” - Audrey Hepburn

We enter the book through a garden lattice, both an acknowledgement and forward recognition of people who cultivate software. The garden lattice is a celebration of our individual vibrancy and connection to the world around us. “Here you will give your gifts and meet your responsibilities.” GL2 It also is a place to view our collective fates; as we grow an understanding emerges that we must commit ourselves toward a sustainable future through tending the garden. This sustainability is undeniably coupled to our natural relationships, persisted even in the software realm.

Early roots from Ada Lovelace’s work on an analytical engine algorithm in 1842 provide a glimpse of the natural world’s influence on software development: “… the Analytical Engine weaves algebraical patterns just as the Jacquard-loom weaves flowers and leaves.” GL3 Many years later, software development remains an effort of intertwining beauty and function to

enhance well-being through a growing system, or lattice, of interconnected parts built upon layers of personal and technical advances.

Lattices, reminiscent of living structures built with willow or bamboo, emerge as a weaved foundation, both flexible and systematically resilient, protecting and inspiring new growth. “The willow submits to the wind and prospers until one day it is many willows - a wall against the wind.” [GL4]. These lattices also signify our collective friendship and interdependence. They echo the sentiment of “Be like bamboo. The higher you grow, the deeper you bow,” (a Chinese proverb) embodying strength through flexibility and unity. This concept mirrors the organic growth seen in software gardens, where a system of interweaved latticework supports the development of intricate and resilient structures.

The lattice also guards against both direct and indirect challenges we face on our journey as software gardeners. These encumbrances can “overheat” our growth potential, leading us to early burnout or burn-down of gardens. The lattice helps us as a carbon sequester, mitigating the excess heat caused by an imbalance in emissions, similar to bamboo garden forests [GL5]. Our individual experiences and personal network consume these difficulties to further grow the lattice through acts of harmonization (such as collaboration or apprenticeship). We care for these experiences together on the lattice because “all flourishing is mutual.” [GL2].

[GL1] Benjamin Gimmel. Round window with lattice. 2005. URL: https://commons.wikimedia.org/wiki/File:Rundes_Fenster_mit_Gitter.jpg (visited on 2024-05-17).

[GL2](1,2) Robin Wall Kimmerer. *Braiding sweetgrass: indigenous wisdom, scientific knowledge and the teachings of plants*. Milkweed Editions, Minneapolis, Minn, first paperback edition edition, 2013. ISBN 978-1-57131-356-0.

[GL3] J. Fuegi and J. Francis. Lovelace & babbage and the creation of the 1843 'notes'. *IEEE Annals of the History of Computing*, 25(4):16–26, October 2003. URL: http://ieeexplore.ieee.org/document/1253887/ (visited on 2024-05-10), doi:10.1109/MAHC.2003.1253887.

[GL4] Frank Herbert and Brian Herbert. *Dune*. The Dune chronicles. Ace, New York, ace premium edition edition, 2010. ISBN 978-0-441-17271-9.

[GL5] Arun Jyoti Nath, Rattan Lal, and Ashesh Kumar Das. Managing woody bamboos for carbon farming and carbon trading. *Global Ecology and Conservation*, 3:654–663, January 2015. URL: https://linkinghub.elsevier.com/retrieve/pii/S2351989415000281 (visited on 2024-05-12), doi:10.1016/j.gecco.2015.03.002.

Understanding

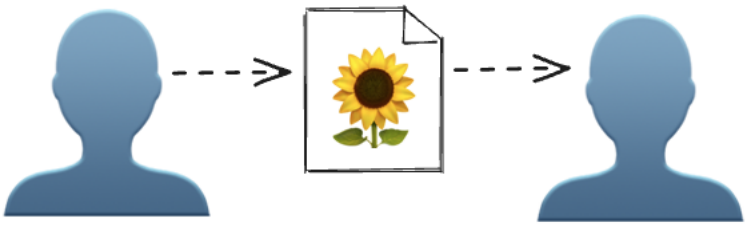


Fig. 5 We transfer understanding of software projects in the form of documentation artifacts.

Shared understanding forms the heartbeat of any successful software project. Peter Naur, in his work “Programming as Theory Building” (1985), emphasized that programming is fundamentally about building a shared theory of the project among all contributors [GLU1]. Understanding encompasses not only the technical aspects of software, but also the collaborative and ethical dimensions that ensure a software project’s growth and sustainability. Just as a gardener must understand the needs of each plant to cultivate a thriving garden, contributors to a project must grasp its goals, structure, and community standards to foster a productive and harmonious environment. Documentation, of which there are many different forms that we describe below, cultivates a shared understanding of a software project.

Common files for shared understanding

In software development, certain files are expected to ensure project scope, clarity, collaboration, and legal compliance. These files serve as the foundational elements that guide contributors and users, much like a well-tended garden benefits from clear paths and signs. When present, these files are also correlated with increased rates of project success [GLU2].

Historically, developers had written the following files in plain-text without formatting. More recently, developers prefer markdown to format the documents with rich styles and media. GitHub, GitLab, and other software source control hosting platforms often automatically render markdown materials in HTML.

README

The `README` file is the cornerstone of any repository, providing essential information about the project. Its presence signifies a well-documented and accessible project, inviting contributors and users alike to understand and engage with the work. A well-structured `README` will reference other pieces of important information that exist elsewhere, such as dependencies and other files we describe below. A repository thrives when its purpose and usage are communicated through a `README` file, much like a garden benefits from a clear plan and understanding of its layout.

> "A good rule of thumb is to assume that the information contained within the README will be the only documentation your users read. For this reason, your README should include how to install and configure your software, where to find its full documentation, under what license it's released, how to test it to ensure functionality, and acknowledgments." [GLU3]

For further reading on `README` file practices, see the following resources:

- The Turing Way: Landing Page - README File
- GitHub: About `README`s
- Makeareadme.com

CONTRIBUTING

A `CONTRIBUTING` file outlines guidelines for how to add to the project. Its presence fosters a welcoming environment for new contributors, ensuring that they have the necessary information to participate effectively. In the same way that a garden flourishes when there's good understanding of how to provide care to the garden and gardeners, a project grows stronger when contributors are guided by clear and inclusive instructions. The resulting nurturing environment fosters growth, resilience, and a sense of community, ensuring the project remains healthy and vibrant.

The `CONTRIBUTING` file should outline all development procedures that are specific to the project. For example, the file might contain testing guidelines, linting/formatting expectations, release cadence, and more. Consider writing this file from the perspective of both an outsider and a beginner: an outsider who might enhance your project by adding a new task or completing an existing task, and a beginner who seeks to understand your project from the ground up [GLU4].

For further reading on `CONTRIBUTING` file practices, see the following resources:

- GitHub: Setting guidelines for repository contributors
- Mozilla: Wrangling Web Contributions: How to Build a CONTRIBUTING.md

CODE_OF_CONDUCT

The `CODE_OF_CONDUCT` (CoC) file sets the standards for behavior within the project community. Its presence signals a commitment to maintaining a respectful and inclusive environment for all participants. It also may provide an outline for acceptable participation when it comes to conflicts of interest. A CoC is a foundational document that defines community values, guides governance and moderation, and signals inclusivity within the project, particularly for vulnerable contributors.

GLU5 A project community benefits from a shared understanding of respectful and supportive interactions, ensuring that all members can contribute positively, much like a garden requires a harmonious ecosystem to thrive.

For further reading and examples of `CODE_OF_CONDUCT` files, see the following:

- Contributor Covenant Code of Conduct
- Example: Python Software Foundation Code of Conduct
- Example: Rust Code of Conduct

LICENSE

A `LICENSE` file specifies the terms under which the project's code can be used, modified, and shared.

> "When you make a creative work (which includes code), the work is under exclusive copyright by default. Unless you include a license that specifies otherwise, nobody else can copy, distribute, or modify your work without being at risk of take-downs, shake-downs, or litigation. Once the work has other contributors (each a copyright holder), "nobody" starts including you." GLU6. The presence of a `LICENSE` file is crucial for legal clarity, encourages the responsible use or distribution of the project, and is considered an indicator of project maturity GLU7.

Just as a garden's health depends on understanding natural laws and respecting boundaries, a project's sustainability hinges on clear licensing that safeguards and empowers both creators and users, cultivating a culture of trust and collaboration.

For further reading and examples of `LICENSE` files, see the following:

- OSF: Licensing
- GitHub: Licensing a repository
- Choosealicense.com
- SPDX License List

Project documentation

Common documentation files like `README`'s, `CONTRIBUTING` guides, and `LICENSE` files are only a start towards more specific project information. Comprehensive project documentation is akin to a detailed gardener's notebook for a well-maintained project, illustrating how the project may be used and the guiding philosophy. This includes in-depth explanations of the project's architecture, practical usage examples, application programming interface (API) references, and development workflows. Oftentimes this type of documentation is provided through a "documentation website" or "docsite" to facilitate a user experience that includes a search bar, multimedia, and other HTML-enabled features.

Such documentation should strive to ensure that both novice and seasoned contributors can grasp the project's complexities and contribute effectively by delivering valuable information. Writing valuable content entails conveying information that the code alone cannot communicate to the user GLU8. In addition to increased value from understanding, software tends to be more extensively utilized when developers offer more comprehensive documentation GLU9. Just as a thriving garden benefits from meticulous care instructions and shared horticultural knowledge, a project flourishes when its documentation offers a clear and thorough guide to its inner workings, nurturing a collaborative and informed community.

Project documentation often exists within a dedicated `docs` directory where the materials may be created and versioned distinctly from other code. Oftentimes this material will leverage a specific documentation tooling technology in alignment with the programming language(s) being used (for example, Python projects often leverage Sphinx). These tools often increase the utility of the output by styling the material with pre-built HTML themes, automating API documentation generation, and an ecosystem of plugins to help extend the capabilities of your documentation without writing new code.

For further reading and examples of deep Project documentation, see the following:

- Berkeley Library: How to Write Good Documentation
- Write the Docs: Software documentation guide

- Overcoming Open Source Project Entry Barriers with a Portal for Newcomers

[GLU1] Peter Naur. Programming as theory building. *Microprocessing and Microprogramming*, 15(5):253–261, May 1985. URL: https://linkinghub.elsevier.com/retrieve/pii/0165607485900328 (visited on 2025-01-02), doi:10.1016/0165-6074(85)90032-8.

[GLU2] Jailton Coelho and Marco Tulio Valente. Why modern open source projects fail. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 186–196. Paderborn Germany, August 2017. ACM. URL: https://dl.acm.org/doi/10.1145/3106237.3106246 (visited on 2024-12-31), doi:10.1145/3106237.3106246.

[GLU3] Benjamin D. Lee. Ten simple rules for documenting scientific software. *PLOS Computational Biology*, 14(12):e1006561, December 2018. URL: https://dx.plos.org/10.1371/journal.pcbi.1006561 (visited on 2024-12-31), doi:10.1371/journal.pcbi.1006561.

[GLU4] Christoph Treude, Marco A. Gerosa, and Igor Steinmacher. Towards the First Code Contribution: Processes and Information Needs. 2024. Version Number: 1. URL: https://arxiv.org/abs/2404.18677 (visited on 2024-12-31), doi:10.48550/ARXIV.2404.18677.

[GLU5] Hana Frluckaj and James Howison. Codes of Conduct in Open Source. In Daniela Damian, Kelly Blincoe, Denae Ford, Alexander Serebrenik, and Zainab Masood, editors, *Equity, Diversity, and Inclusion in Software Engineering*, pages 295–308. Apress, Berkeley, CA, 2024. URL: https://link.springer.com/10.1007/978-1-4842-9651-6_17 (visited on 2025-01-02), doi:10.1007/978-1-4842-9651-6_17.

[GLU6] Inc. GitHub. No license. Accessed: 2025-01-02. URL: https://choosealicense.com/no-permission/.

[GLU7] Deekshitha, Rena Bakhshi, Jason Maassen, Carlos Martinez Ortiz, Rob van Nieuwpoort, and Slinger Jansen. RSMM: A Framework to Assess Maturity of Research Software Project. June 2024. arXiv:2406.01788 [cs]. URL: http://arxiv.org/abs/2406.01788 (visited on 2024-10-02).

[GLU8] **missing author in henney_97_2010**

[GLU9] Awan Afiaz, Andrey Ivanov, John Chamberlin, David Hanauer, Candace Savonen, Mary J. Goldman, Martin Morgan, Michael Reich, Alexander Getka, Aaron Holmes, Sarthak Pati, Dan Knight, Paul C. Boutros, Spyridon Bakas, J. Gregory Caporaso, Guilherme Del Fiol, Harry Hochheiser, Brian Haas, Patrick D. Schloss, James A. Eddy, Jake Albrecht, Andrey Fedorov, Levi Waldron, Ava M. Hoffman, Richard L. Bradshaw, Jeffrey T. Leek, and Carrie Wright. Evaluation of software impact designed for biomedical research: Are we measuring what's meaningful? June 2023. arXiv:2306.03255 [cs, q-bio]. URL: http://arxiv.org/abs/2306.03255 (visited on 2024-10-02).

Software Forest

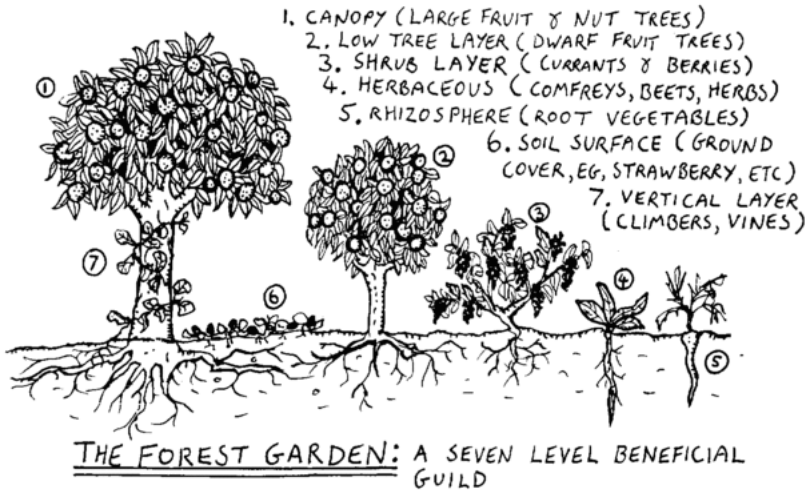


Fig. 6 The software forest is multi-layered like a forest garden. (Image source: SF1).

The software forest is a multidimensional garden, enriched by connections and minimal, iterative changes. Both the practice and artifacts of software development can seem daunting, like an unfamiliar forest full of mystery and surprises. However, through software gardening, we see these forests as opportunities for balanced cultivation.

> “Called Three-Dimensional Forestry or Forest Farming, … the three ‘dimensions’ of [Toyohiko Kagawa’s] 3-D system were the trees as conservers of the soil and suppliers of food and the livestock which benefited from them.” [SF2], [SF3]

Forest gardening offers innovations from an ecological cultivation perspective by caring for trees, understory vegetation, soil, and other environmental factors. Forest gardens take time to create because their slow-growing nature requires patience and careful, long-term nurturing. Software forests are also created gradually, “… starting with the existing, living, breathing system, and working outward, a step at a time, in such a way as to not undermine the system’s viability.” [SF4] Nourishing the system in this way helps us practice *primum non nocere* (“first, do no harm”) to the garden and components within, conserving energy for necessary adaptations.

Dense networks of relationships surround software forest gardens in the form of software environments. Some of these environments are portable, similar to Wardian cases that provide loosely coupled habitats for software organisms (for example, through environment managers). Others are exhibited through abstract relationships as *nurse logs*, *Hügelkultur*, or mycorrhizal-like associations which redistribute energy among the entirety of a software region.

As software gardeners we must consider how energy is transferred to various portions of the forest. Adventitious code, which “volunteers” by accident of growth, may create scenarios of energy waste in the form of cognitive or performance misalignment. All waste in the forest is food [SF5], therefore we must imagine and craft new homes for wasted energy through appropriate means. This chaos of this soft-scaping (curating the software garden towards a goal) is in constant flux, tugging on our capacities as student and teacher of the garden to encourage growth and avoid harm.

“What gardens demand of us is lifelong learning.” [SF6] We discover and are discovered within the software garden. As software forest gardens grow, they demonstrate emergent properties which are well suited for practices such as evolutionary architecture, object-orientated design, and coupling analysis. These assist software gardeners as a shed of multi-dimensional concepts that must be employed to truly achieve effective multi-layered software garden maintenance.

[SF1] Graham Burnett. Forest garden diagram to replace the one that currently exists on the article. 2006. URL: https://commons.wikimedia.org/wiki/File:Forgard2-003.gif (visited on 2024-05-17).

[SF2] Robert A. de J. Hart. *Forest gardening: cultivating an edible landscape*. Chelsea Green Pub. Co, White River Junction, Vt, 1996. ISBN 978-0-930031-84-8.

[SF3] Kagawa, T. and S. Fujisaki. Rittai nogyo no riron to jissen (theory and practice of three-dimensional structural farming). *Tokyo, Japan: Nihonhyoronsya*, 1935.

[SF4] Brian Foote and Joseph Yoder. Big ball of mud. *Pattern languages of program design*, 4:654–692, 1997.

[SF5] William McDonough. Waste equals food: our future and the making of things. *Awakening–The Upside of Y2k, The Printed Word, Spokane*, 1998.

[SF6] Viviane Stappmanns, Mateo Kries, Vitra Design Museum, and Wüstenrot Stiftung, editors. *Garden futures: designing with nature*. Vitra Design Museum, Weil am Rhein, 2023. ISBN 978-3-945852-53-8.

Verdant Sundial

Fig. 7 The verdant sundial casts shadows of past and future intention observed in the present. (Image source: VS1).

> “The present is the ever moving shadow that divides yesterday from tomorrow. In that lies hope.”
> - Frank Lloyd Wright

The software garden is intertwined in a story of time which is exhibited like a shadow on a moss-covered sundial. Time is observed indirectly through a kind of software archeology in past-tense material artifacts (files) and present-tense operation (procedures). These artifacts act as shadows of our prior creative intentions where procedures become the living present materialized promise of those intentions. “This brings an uncanny sense of living in a state where the future is a dimension of our present reality.” VS2 The future is elusive but present in software artifacts, similar to rings of a tree that depict the promise of continued stability or hope of change.

These software tree rings are vessels of temporal data which offer a unique insight into growth patterns. We can study these patterns like dendronchronology: “Peel away the hard, rough bark and there is a living document, history recorded in rings of wood cells.” VS3 This peeling helps us understand the chronological nature of software and how it evolves. It also shares patterns of temporal pulsing - not all times of the software garden are the same.

These software pulses give visibility into distinct growing or receding patterns. Some code exhibits annual behavior, requiring continual replanting efforts each season. Others are perennials, undergoing seasonal change perpetuated by periods of vibrancy and dormancy. Seasons of software promote differing patterns among these. Some systems may prepare for winter (for instance, during resource scarcity, or periods of high performance stress) through adaptable conservation only to fervently bloom during their spring.

Seasonal chronology in software benefits also from an awareness of kairos, the opportune timing of our actions amidst time. As we gain experience in software gardens we begin to anticipate seasonal change which helps us project preparatory change. Each software gardening moment amidst this cascade of season includes a series of options of varying kairological fitness. Assessing these moments of their fit within patterns helps us gain wisdom as we contemplate time in software gardens; like annual flowers, how do we anticipate and accept the limited lifecycle of some software? Like mighty trees, how do we build software to last 100 years or more?

[VS1] Elfward. Garden sundial in an epping garden. 2015. URL: https://commons.wikimedia.org/wiki/File:Sundial_2916_HDR.jpg (visited on 2024-05-18).

[VS2] Timothy Morton. *Hyperobjects: Philosophy and Ecology after the End of the World*. University of Minnesota Press, Minneapolis, 2013. ISBN 9780816689231.

[VS3] Carolyn Gramling. ‘tree story’ explores what tree rings can tell us about the past. *Science News*, June 2020. Accessed: 2024-05-18. URL: https://www.sciencenews.org/article/tree-story-book-explores-what-tree-rings-can-tell-us-about-past.

Seed Bank

The seed bank is a section of the Software Garden Almanack associated with applied demonstrations of concepts and technologies associated with other areas of content.

Visualizing PubMed article GitHub repository software information entropy

The content below seeks to better understand how software information entropy manifests in a dataset of ~10,000 PubMed article GitHub repositories.

PubMed GitHub repositories are extracted using the PubMed API to query for GitHub links within article abstracts. GitHub data about these repositories is gathered using the GitHub API. The code to perform data extractions may be found under the directory: gather-pubmed-repos .

Software entropy measurements are gathered using the notebook: software-information-entropy.ipynb .

We derive software information entropy using methods inspired from *Predicting faults using the complexity of code changes*[SB1]. Software information entropy within the context of this notebook is normalized for all files within a repository using the first and latest commits.

Project durations highlighted below is the number of days between the first and latest commit date.

Data Extraction

The following section extracts data which includes software entropy and GitHub-derived data. We merge the data to form a table which includes PubMed, GitHub, and Almanack software entropy information on the repositories.

```
import numpy as np
import pandas as pd
import plotly.express as px
import plotly.io as pio

# set plotly default theme
pio.templates.default = "plotly_white"

# read example data which includes pubmed github links detected from article abstracts
df = pd.merge(
    left=pd.read_parquet("repository_analysis_results"),
    right=pd.read_parquet(
        "gather-pubmed-repos/pubmed_github_links_with_github_data.parquet"
    ),
    left_on="Repository URL",
    right_on="github_link",
)
df
```

	Repository URL	Normalized Total Entropy	Date of First Commit	Date of Last Commit	Time of Existence (days)	PMID	artic
0	https://github.com/BUStools/BUSZ-format	0.000000	2022-11-08	2022-11-18	10.0	37129540	
1	https://github.com/pmelsted/BUSZ_paper	0.178897	2023-03-27	2023-03-28	1.0	37129540	
2	https://github.com/WormBase/scdefg	0.153966	2021-01-29	2022-02-02	368.0	35814290	202
3	https://github.com/ekg/guix-genomics	0.176469	2020-01-06	2024-01-22	1476.0	36448683	
4	https://github.com/ekg/seqwish	0.027122	2018-06-11	2023-12-09	2007.0	36448683	
...	...	...	...	...	...	...	
11153	https://github.com/maxplanck-ie/parkour	0.000449	2016-05-23	2022-08-24	2284.0	30239601	
11154	https://github.com/linsalrob/partie	0.000257	2016-09-19	2022-10-23	2224.0	28369246	
11155	https://github.com/louzounlab/CountingIsAlmost...	0.011558	2022-06-15	2022-11-04	141.0	36741395	202
11156	https://github.com/sysbio-polito/NWN_CElegans_...	0.002924	2021-02-12	2021-10-06	235.0	34765090	202

	Repository URL	Normalized Total Entropy	Date of First Commit	Date of Last Commit	Time of Existence (days)	PMID	artic
11157	https://github.com/lennylv/DGCddG	0.014386	2021-09-26	2024-06-12	990.0	37018301	202

11158 rows × 33 columns

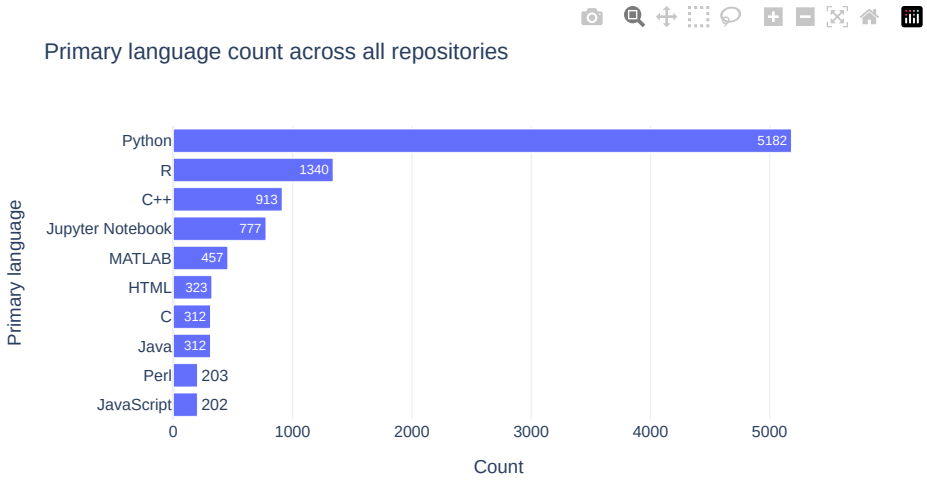
What languages are used within PubMed article repositories?

The following section observes the top 10 languages which are used in repositories from the dataset. Primary language is determined as the language which has the most lines of code within a repository.

```
language_grouped_data = (
    df.groupby(["Primary language"]).size().reset_index(name="Count")
)

# Create a horizontal bar chart
fig_languages = px.bar(
    language_grouped_data.sort_values(by="Count")[-10:],
    y="Primary language",
    x="Count",
    text="Count",
    orientation="h",
    width=700,
    height=400,
    title="Primary language count across all repositories",
)

fig_languages.show()
```

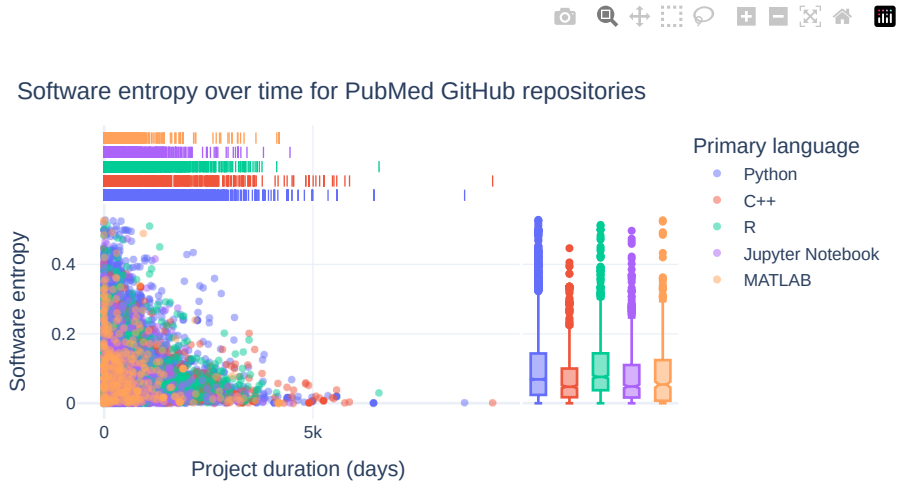


How is software entropy different across primary languages?

The following section explores how software entropy manifests differently across different primary languages for repositories.

```
fig = px.scatter(
    df[
        df["Primary language"].isin(
            language_grouped_data.sort_values(by="Count")[-5:]["Primary language"]
        )
    ],
    x="Time of Existence (days)",
    y="Normalized Total Entropy",
    hover_data=["Repository URL"],
    width=700,
    height=400,
    title="Software entropy over time for PubMed GitHub repositories",
    marginal_x="rug",
    marginal_y="box",
    opacity=0.5,
    color="Primary language",
)

fig.update_layout(
    font=dict(size=13),
    title={"yref": "container", "y": 0.8, "yanchor": "bottom"},
    xaxis_title="Project duration (days)",
    yaxis_title="Software entropy",
)
fig.write_image("images/software-information-entropy-top-5-langs.png")
fig.show()
```

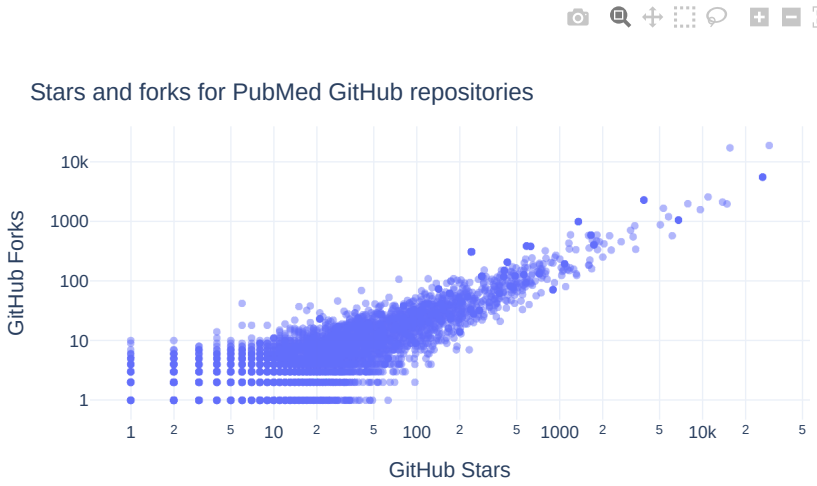


What is the relationship between GitHub Stars and Forks for repositories?

We next explore how GitHub Stars and Forks are related within the repositories.

```
fig = px.scatter(
    df,
    x="GitHub Stars",
    y="GitHub Forks",
    hover_data=["Repository URL"],
    width=700,
    height=400,
    title="Stars and forks for PubMed GitHub repositories",
    opacity=0.5,
    log_y=True,
    log_x=True,
)

fig.update_layout(
    font=dict(size=13),
    title={"yref": "container", "y": 0.8, "yanchor": "bottom"},
    xaxis_title="GitHub Stars",
    yaxis_title="GitHub Forks",
)
fig.write_image("images/pubmed-stars-and-forks.png")
fig.show()
```



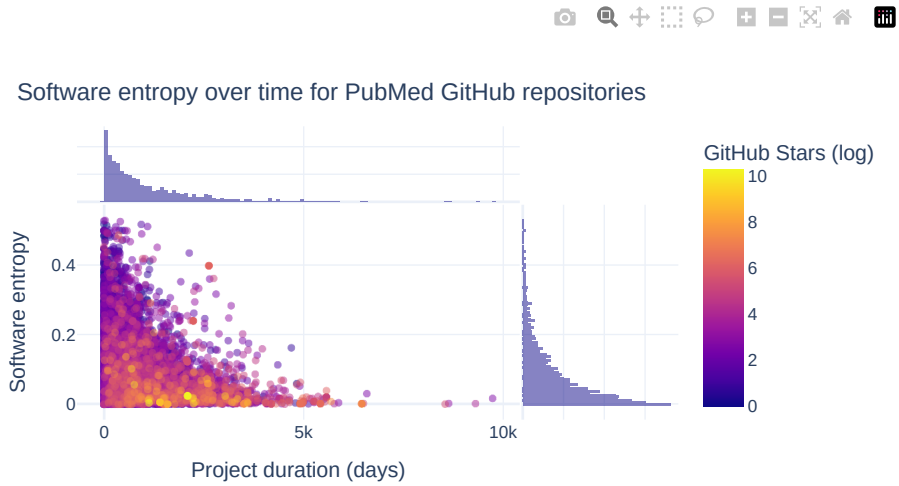
How do GitHub Stars, software entropy, and time relate?

The next section explores how GitHub Stars, software entropy, and time relate to one another.

```
df["GitHub Stars (log)"] = np.log(
    df["GitHub Stars"].apply(
        # move 0's to None to avoid divide by 0
        lambda x: x if x > 0 else None
    )
)

fig = px.scatter(
    df.dropna(subset="GitHub Stars (log)").sort_values(by="GitHub Stars (log)"),
    x="Time of Existence (days)",
    y="Normalized Total Entropy",
    hover_data=["Repository URL"],
    width=700,
    height=400,
    title="Software entropy over time for PubMed GitHub repositories",
    marginal_x="histogram",
    marginal_y="histogram",
    opacity=0.5,
    color="GitHub Stars (log)",
)

fig.update_layout(
    font=dict(size=13),
    title={"yref": "container", "y": 0.8, "yanchor": "bottom"},
    xaxis_title="Project duration (days)",
    yaxis_title="Software entropy",
)
fig.write_image("images/software-information-entropy-gh-stars.png")
fig.show()
```



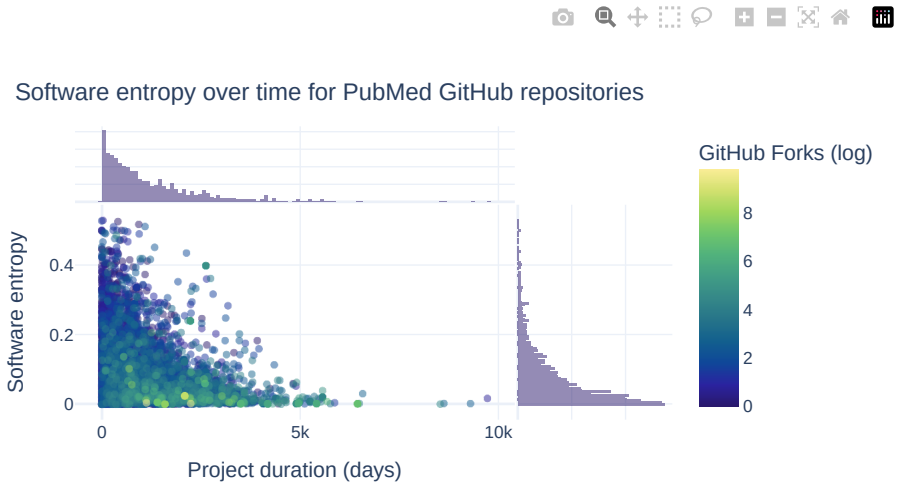
How do GitHub Forks, software entropy, and time relate?

The next section explores how GitHub Forks, software entropy, and time relate to one another.

```
df["GitHub Forks (log)"] = np.log(
    df["GitHub Forks"].apply(
        # move 0's to None to avoid divide by 0
        lambda x: x if x > 0 else None
    )
)

fig = px.scatter(
    df.dropna(subset="GitHub Forks (log)").sort_values(by="GitHub Forks (log)"),
    x="Time of Existence (days)",
    y="Normalized Total Entropy",
    hover_data=["Repository URL"],
    width=700,
    height=400,
    title="Software entropy over time for PubMed GitHub repositories",
    marginal_x="histogram",
    marginal_y="histogram",
    opacity=0.5,
    color="GitHub Forks (log)",
    color_continuous_scale=px.colors.sequential.haline,
)

fig.update_layout(
    font=dict(size=13),
    title={"yref": "container", "y": 0.8, "yanchor": "bottom"},
    xaxis_title="Project duration (days)",
    yaxis_title="Software entropy",
)
fig.write_image("images/software-information-entropy-forks.png")
fig.show()
```

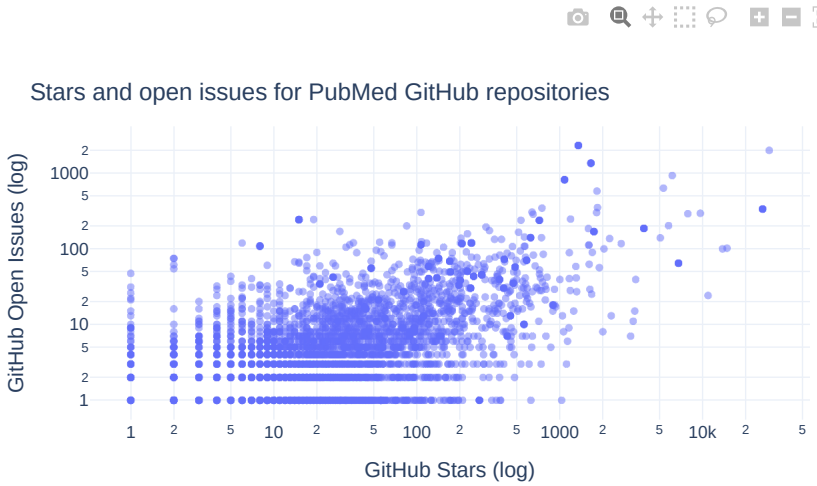


What is the relationship between GitHub Stars and Open Issues for the repositories?

Below we explore how GitHub Stars and Open Issues are related for the repositories.

```
df["GitHub Open Issues (log)"] = np.log(
    df["GitHub Open Issues"].apply(
        # move 0's to None to avoid divide by 0
        lambda x: x if x > 0 else None
    )
)
fig = px.scatter(
    df,
    x="GitHub Stars",
    y="GitHub Open Issues",
    hover_data=["Repository URL"],
    width=700,
    height=400,
    title="Stars and open issues for PubMed GitHub repositories",
    opacity=0.5,
    log_y=True,
    log_x=True,
)

fig.update_layout(
    font=dict(size=13),
    title={"yref": "container", "y": 0.8, "yanchor": "bottom"},
    xaxis_title="GitHub Stars (log)",
    yaxis_title="GitHub Open Issues (log)",
)
fig.write_image("images/pubmed-stars-and-open-issues.png")
fig.show()
```



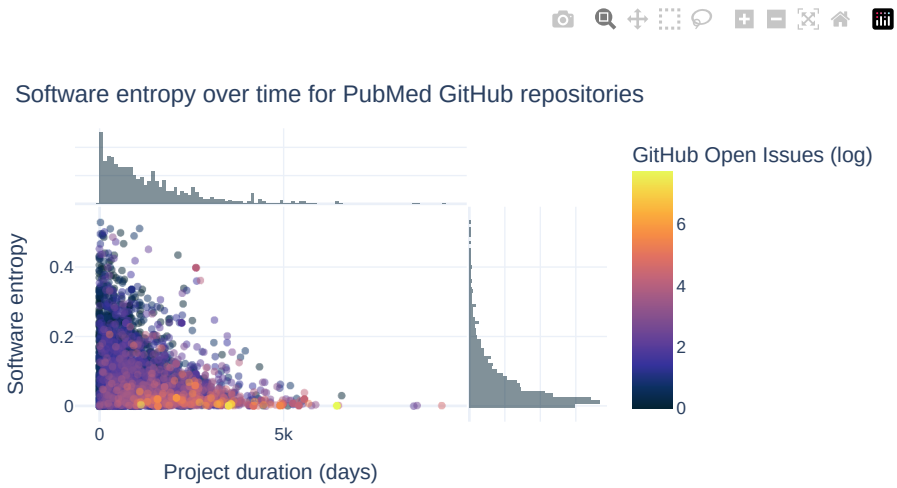
How do software entropy, time, and GitHub issues relate to one another?

The next section visualizes how software entropy, time, and GitHub issues relate to one another.

```
df["GitHub Open Issues (log)"] = np.log(
    df["GitHub Open Issues"].apply(
        # move 0's to None to avoid divide by 0
        lambda x: x if x > 0 else None
    )
)

fig = px.scatter(
    df.dropna(subset="GitHub Open Issues (log)").sort_values(
        by="GitHub Open Issues (log)"
    ),
    x="Time of Existence (days)",
    y="Normalized Total Entropy",
    hover_data=["Repository URL"],
    width=700,
    height=400,
    title="Software entropy over time for PubMed GitHub repositories",
    marginal_x="histogram",
    marginal_y="histogram",
    opacity=0.5,
    color="GitHub Open Issues (log)",
    color_continuous_scale=px.colors.sequential.thermal,
)

fig.update_layout(
    font=dict(size=13),
    title={"yref": "container", "y": 0.8, "yanchor": "bottom"},
    xaxis_title="Project duration (days)",
    yaxis_title="Software entropy",
)
fig.write_image("images/software-information-entropy-open-issues.png")
fig.show()
```



[SB1] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *2009 IEEE 31st International Conference on Software Engineering*, 78–88. Vancouver, BC, Canada, 2009. IEEE. URL: http://ieeexplore.ieee.org/document/5070510 (visited on 2024-07-15), doi:10.1109/ICSE.2009.5070510.

Garden Circle

The garden circle is a section of the Software Garden Almanack associated with contributions, planning, and internal maintenance of the project.

Contributing

First of all, thank you for contributing to the Software Gardening Almanack! 🎉 💯 🪴 We're so grateful for the kindness of your effort applied to grow this project into the best that it can be.

This document contains guidelines on how to most effectively contribute to this project.

If you would like clarifications, please feel free to ask any questions or for help. We suggest asking for help from GitHub where you may need it (for example, in a GitHub issue or a pull request) by "at (@) mentioning" a Software Gardening Almanack maintainer (for example, `@username`).

Code of conduct

Our Code of Conduct (CoC) policy (located here) governs this project. By participating in this project, we expect you to uphold this code. Please report unacceptable behavior by following the procedures listed under the CoC Enforcement section.

Security

Please see our Security policy (located here) for more information on security practices and recommendations associated with this project.

Quick links

- Documentation: software-gardening/almanack
- Issue tracker: software-gardening/almanack#issues

Process

Reporting bugs or suggesting enhancements

We're deeply committed to a smooth and intuitive user experience which helps people benefit from the content found within this project. This commitment requires a good relationship and open communication with our users.

We encourage you to file a GitHub issue to report bugs or propose enhancements to improve the Software Gardening Almanack.

First, figure out if your idea is already implemented by reading existing issues or pull requests! Check the issues (software-gardening/almanack#issues) and pull requests (software-gardening/almanack) to see if someone else has already documented or began implementation of your idea. If you do find your idea in an existing issue, please comment on the existing issue noting that you are also interested in the functionality. If you do not find your idea, please open a new issue and document why it would be helpful for your particular use case.

Please also provide the following specifics:

- The version of the Software Gardening Almanack you're referencing.
- Specific error messages or how the issue exhibits itself.
- Operating system (e.g. MacOS, Windows, etc.)
- Device type (e.g. laptop, phone, etc.)
- Any examples of similar capabilities

Your first code contribution

Contributing code for the first time can be a daunting task. However, in our community, we strive to be as welcoming as possible to newcomers, while ensuring sustainable software development practices.

The first thing to figure out is specifically what you're going to contribute! We describe all future work as individual GitHub issues. For first time contributors we have specifically labeled as good first issue.

If you want to contribute code that we haven't already outlined, please start a discussion in a new issue before writing any code. A discussion will clarify the new code and reduce merge time. Plus, it's possible your contribution belongs in a different code base, and we do not want to waste your time (or ours)!

Pull requests

After you've decided to contribute code and have written it up, please file a pull request. We specifically follow a forked pull request model. Please create a fork of Software Gardening Almanack repository, clone the fork, and then create a new, feature-specific branch. Once you make the necessary changes on this branch, you should file a pull request to incorporate your changes into the main The Software Gardening Almanack repository.

The content and description of your pull request are directly related to the speed at which we are able to review, approve, and merge your contribution into The Software Gardening Almanack. To ensure an efficient review process please perform the following steps:

1. Follow all instructions in the pull request template
2. Triple check that your pull request is adding *one* specific feature. Small, bite-sized pull requests move so much faster than large pull requests.
3. After submitting your pull request, ensure that your contribution passes all status checks (e.g. passes all tests)

We require pull request review and approval by at least one project maintainer in order to merge. We will do our best to review the code additions in as soon as we can. Ensuring that you follow all steps above will increase our speed and ability to review.

We will check for accuracy, style, code coverage, and scope.

Git commit messages

For all commit messages, please use a short phrase that describes the specific change. For example, “Add feature to check normalization method string” is much preferred to “change code”. When appropriate, reference issues (via `#` plus number) .

Development

Overview

We write and develop the Software Gardening Almanack in Python through Jupyter Book with related environments managed by Python Poetry. We use Node and NPM dependencies to assist development activities (such as performing linting checks or other capabilities). We use GitHub actions for CI/CD procedures such as automated tests.

Getting started

To enable local development, perform the following steps.

1. Install Python (suggested: use `pyenv` for managing Python versions)
2. Install Poetry
3. Install Poetry environment: `poetry install`
4. Install Node (suggested: use `nvm` for managing Node versions)
5. Install Node environment: `npm install`
6. Install Vale dependencies: `poetry run vale sync`

Development Tasks

We use Poe the Poet to define common development tasks, which simplifies repeated commands. We include Poe the Poet as a Python Poetry `dev` group dependency, which users access through the Poetry environment. Please see the `pyproject.toml` file’s `[tool.poe.tasks]` table for a list of available tasks.

For example:

```
# example of how Poe the Poet commands may be used.
poetry run poe <task_name>

# example of a task which runs jupyter-book build for the project
poetry run poe build-book
```

Linting

pre-commit automatically checks all work added to this repository. We implement these checks using GitHub Actions. Pre-commit can work alongside your local git with git-hooks After installing pre-commit within your development environment, the following command performs the same checks:

```
% pre-commit run --all-files
```

We strive to provide comments about each pre-commit check within the `.pre-commit-config.yaml` file. Comments are provided within the file to make sure we single-source documentation where possible, avoiding possible drift between the documentation and the code which runs the checks.

Automated CI/CD with GitHub Actions

We use GitHub Actions to help perform automated CI/CD as part of this project. GitHub Actions involves defining workflows through YAML files. These workflows include one or more jobs which are collections of individual processes (or steps) which run as part of a job. We define GitHub Actions work under the `.github` directory. We suggest the use of `act` to help test GitHub Actions work during development.

Releases

We utilize semantic versioning ("semver") to distinguish between major, minor, and patch releases. We publish source code by using GitHub Releases available here. Contents of the book are distributed as both a website and PDF. We distribute a Python package through the Python Packaging Index (PyPI) available here which both includes and provides tooling for applying the book's content.

Release Publishing Process

Publishing Software Gardening Almanack releases involves several manual and automated steps. See below for an overview of how this works.

Version specifications

We follow semantic version (semver) specifications with this project through the following technologies.

- `poetry-dynamic-versioning` leveraging `dunamai` creates version data based on git tags and commits.
- GitHub Releases automatically create git tags and source code collections available from the GitHub repository.
- `release-drafter` infers and describes changes since last release within automatically drafted GitHub Releases after pull requests are merged (draft releases are published as decided by maintainers).

Version specification process

1. Open a pull request and use a repository label for `release-<semver release type>` to label the pull request for visibility with `release-drafter` (for example, see almanack#43 as a reference of a semver patch update).
2. On merging the pull request for the release, a GitHub Actions workflow defined in `draft-release.yml` leveraging `release-drafter` will draft a release for maintainers.
3. The draft GitHub release will include a version tag based on the GitHub PR label applied and `release-drafter`.
4. Make modifications as necessary to the draft GitHub release, then publish the release (the draft release does not normally need additional modifications).
5. On publishing the release, another GitHub Actions workflow defined in `publish-pypi.yml` will automatically build and deploy the Python package to PyPI (utilizing the earlier modified `pyproject.toml` semantic version reference for labeling the release).

Attribution

We sourced portions of this contribution guide from `pyctyominer`. Big thanks go to the developers and contributors of that repository.

Garden Map

Our garden map is where you can learn about what features we're working on, what stage they're in, and when we expect to bring them to you. The content here follows that of a technical roadmap (providing strategic visibility on implementation details and timelines).

Ethos

We intend for the garden map to provide useful tools for us to efficiently create and maintain the Software Gardening Almanack. We practice keeping the garden map as “petal-light” as possible through built-in tools, automation, and an avoidance of analysis paralysis where possible.

We’d love your input

We welcome your input on the garden map and elements found within it! Please see the contributing guide for more information on how to participate in this process.

Active garden map(s)

The following are active garden maps for the Software Gardening Almanack. Please see below for a further description of these projects.

- **2024 BSSw Fellowship Milestones**: this project communicates activities associated with the 2024 BSSw Fellowship associated with the Software Gardening Almanack.

Sections

You can explore the garden map with different views:

- **“Canopy-view” development visibility**: We track new, existing, and previous work at a high-level using organization-wide GitHub Projects.
- **“Ground-level” development visibility**: We track repository-specific work using GitHub issues and a GitHub Project field called “status”, which we associate with each issue (indicated as “Todo”, “In progress”, “Paused”, or “Done”).
- **Major milestones**: We track major milestones using common GitHub issues labels across all repositories through their associated issues.

Visualizations

It can be helpful to visualize the garden map using data plots or other images. We use GitHub Projects insights to assist with automated creation and updates to garden map activity. Please see this project insights page for an example of these (using the insights button within any GitHub Projects for viewing others).

Acknowledgments

We drew inspiration for this content from the GitHub public roadmap. Big thanks to the original works found there.

`almanack` Package API

Metrics

Data

This module computes data for GitHub Repositories

`almanack.metrics.data._get_almanack_version() → str`

Seeks the current version of almanack using either pkg_resources or dunamai to determine the current version being used.

Returns:

str

A string representing the version of almanack currently being used.

almanack.metrics.data.compute_almanack_score(*almanack_table: List[Dict[str, int | float | bool]]*) → Dict[str, int | float]

Computes an Almanack score by counting boolean Almanack table metrics to provide a quick summary of software sustainability.

Parameters:

almanack_table (*List[Dict[str, Union[int, float, bool]]]*) – A list of dictionaries containing metrics. Each dictionary must have a “result” key with a value that is an int, float, or bool. A “sustainability_correlation” key is included for values to specify the relationship to sustainability: - 1 (positive correlation) - 0 (no correlation) - -1 (negative correlation)

Returns:

Dictionary of length three, including the following: 1) number of Almanack boolean metrics that passed (numerator), 2) number of total Almanack boolean metrics considered (denominator), and 3) a score that represents how likely the repository will be maintained over time based (numerator / denominator).

Return type:

Dict[str, Union[int, float]]

almanack.metrics.data.compute_pr_data(*repo_path: str*, *pr_branch: str*, *main_branch: str*) → Dict[str, Any]

Computes entropy data for a PR compared to the main branch.

Parameters:

- **repo_path** (*str*) – The local path to the Git repository.
- **pr_branch** (*str*) – The branch name for the PR.
- **main_branch** (*str*) – The branch name for the main branch.

Returns:

A dictionary containing the following key-value pairs:

- ”pr_branch”: The PR branch being analyzed.
- ”main_branch”: The main branch being compared.
- ”total_entropy_introduced”: The total entropy introduced by the PR.
- ”number_of_files_changed”: The number of files changed in the PR.
- ”entropy_per_file”: A dictionary of entropy values for each changed file.
- ”commits”: A tuple containing the most recent commits on the PR and main branches.

Return type:

dict

almanack.metrics.data.compute_repo_data(*repo_path: str*) → None

Computes comprehensive data for a GitHub repository.

Parameters:

repo_path (*str*) – The local path to the Git repository.

Returns:

A dictionary containing data key-pairs.

Return type:

dict

almanack.metrics.data.days_of_development(*repo: Repository*) → float

Parameters:

repo (*pygit2.Repository*) – Path to the git repository.

Returns:

The average number of commits per day over the period of time.

Return type:

float

almanack.metrics.data.gather_failed_almanack_metric_checks(*repo_path: str*) → List[Dict[str, Any]]

Gather checks on the repository metrics and returns a list of failed checks for use in helping others understand the failed checks and rectify them.

Parameters:

repo_path (*str*) – The file path to the repository which will have metrics calculated and includes boolean checks.

Returns:

A list of dictionaries containing the metrics and their associated results. Each dictionary includes the name, id, and

Return type:

List[Dict[str, Any]]

guidance on how to fix each failed check. The dictionary also

includes data about the almanack score for use in summarizing the results.

almanack.metrics.data.get_github_build_metrics(*repo_url: str*, *branch: str = 'main'*, *max_runs: int = 100*, *github_api_endpoint: str = 'https://api.github.com/repos'*) → dict

Fetches the success ratio of the latest GitHub Actions build runs for a specified branch.

Parameters:

- **repo_url** (*str*) – The full URL of the repository (e.g., ‘ software-gardening/almanack’).
- **branch** (*str*) – The branch to filter for the workflow runs (default: “main”).
- **max_runs** (*int*) – The maximum number of latest workflow runs to analyze.
- **github_api_endpoint** (*str*) – Base API endpoint for GitHub repositories.

Returns:

The success ratio and details of the analyzed workflow runs.

Return type:

dict

almanack.metrics.data.get_table(*repo_path: str*) → List[Dict[str, Any]]

Gather metrics on a repository and return the results in a structured format.

This function reads a metrics table from a predefined YAML file, computes relevant data from the specified repository, and associates the computed results with the metrics defined in the metrics table. If an error occurs during data computation, an exception is raised.

Parameters:

- **repo_path** (*str*) – The file path to the repository for which metrics are
- **performed.** (*to be*)

Returns:

A list of dictionaries containing the metrics and their associated results. Each dictionary includes the original metrics data along with the computed result under the key “result”.

Return type:

List[Dict[str, Any]]

Raises:

- **ReferenceError** – If there is an error encountered while processing the
- **data, providing context in the error message.** –

almanack.metrics.data.measure_coverage(*repo: Repository, primary_language: str | None*) → Dict[str, Any] | None

Measures code coverage for a given repository.

Parameters:

- **repo** (*pygit2.Repository*) – The pygit2 repository object to analyze.
- **primary_language** (*Optional[str]*) – The primary programming language of the repository.

Returns:

Code coverage data or an empty dictionary if unable to find code coverage data.

Return type:

Optional[dict[str,Any]]

almanack.metrics.data.parse_python_coverage_data(*repo: Repository*) → Dict[str, Any] | None

Parses coverage.py data from recognized formats such as JSON, XML, or LCOV. See here for more information: https://coverage.readthedocs.io/en/latest/cmd.html#cmd-report

Parameters:

repo (*pygit2.Repository*) – The pygit2 repository object containing code.

Returns:

A dictionary with standardized code coverage data or an empty dict if no data is found.

Return type:

Optional[Dict[str, Any]]

almanack.metrics.data.process_repo_for_analysis(*repo_url: str*) → Tuple[float | None, str | None, str | None, int | None]

Processes GitHub repository URL’s to calculate entropy and other metadata. This is used to prepare data for analysis, particularly for the seedbank notebook that process PUBMED repositories.

Parameters:

repo_url (*str*) – The URL of the GitHub repository.

Returns:

A tuple containing the normalized total entropy, the date of the first commit,

the date of the most recent commit, and the total time of existence in days.

Return type:

tuple

Garden Lattice

Understanding

This module focuses on the Almanack's Garden Lattice materials which encompass aspects of human understanding.

almanack.metrics.garden_lattice.understanding.includes_common_docs(*repo: Repository*) → bool

Check whether the repo includes common documentation files and directories associated with building docsites.

Parameters:

repo (*pygit2.Repository*) – The repository object.

Returns:

True if any common documentation files are found, False otherwise.

Return type:

bool

Connectedness

Module for Almanack metrics covering human connection and engagement in software projects such as contributor activity, collaboration frequency.

almanack.metrics.garden_lattice.connectedness.count_unique_contributors(*repo: Repository*, *since: datetime | None = None*) → int

Counts the number of unique contributors to a repository.

If a *since* datetime is provided, counts contributors who made commits after the specified datetime. Otherwise, counts all contributors.

Parameters:

- **repo** (*pygit2.Repository*) – The repository to analyze.
- **since** (*Optional[datetime]*) – The cutoff datetime. Only contributions after this datetime are counted. If None, all contributions are considered.

Returns:

The number of unique contributors.

Return type:

int

almanack.metrics.garden_lattice.connectedness.default_branch_is_not_master(*repo: Repository*) → bool

Checks if the default branch of the specified repository is "master".

Parameters:

repo (*Repository*) – A pygit2.Repository object representing the Git repository.

Returns:

True if the default branch is "master", False otherwise.

Return type:

bool

almanack.metrics.garden_lattice.connectedness.detect_social_media_links(*content: str*) → Dict[str, List[str]]

Analyzes README.md content to identify social media links.

Parameters:

readme_content (*str*) – The content of the README.md file as a string.

Returns:

A dictionary containing social media details discovered from readme.md content.

Return type:

Dict[str, List[str]]

almanack.metrics.garden_lattice.connectedness.find_doi_citation_data(*repo: Repository*) → Dict[str, Any]

Find and validate DOI information from a CITATION.cff file in a repository.

This function searches for a *CITATION.cff* file in the provided repository, extracts the DOI (if available), validates its format, checks its resolvability via HTTP, and performs an exact DOI lookup on the OpenAlex API.

Parameters:

repo (*pygit2.Repository*) – The repository object to search for the CITATION.cff file.

Returns:

A dictionary containing DOI-related information and metadata.

Return type:

Dict[str, Any]

almanack.metrics.garden_lattice.connectedness.is_citable(*repo: Repository*) → bool

Check if the given repository is citable.

A repository is considered citable if it contains a CITATION.cff or CITATION.bib file, or if the README.md file contains a citation section indicated by “## Citation” or “## Citing”.

Parameters:

repo (*pygit2.Repository*) – The repository to check for citation files.

Returns:

True if the repository is citable, False otherwise.

Return type:

bool

Practicality

This module focuses on the Almanack’s Garden Lattice materials which involve how people can apply software in practice.

almanack.metrics.garden_lattice.practicality.count_repo_tags(*repo: Repository*, *since: datetime | None* = *None*) → int

Counts the number of tags in a pygit2 repository.

If a *since* datetime is provided, counts only tags associated with commits made after the specified datetime. Otherwise, counts all tags in the repository.

Parameters:

- **repo** (*pygit2.Repository*) – The repository to analyze.

- **since** (*Optional[datetime]*) – The cutoff datetime. Only tags for commits after this datetime are counted. If None, all tags are counted.

Returns:

The number of tags in the repository that meet the criteria.

Return type:

int

`almanack.metrics.garden_lattice.practicality.get_ecosystems_package_metrics(repo_url: str) → Dict[str, Any]`

Fetches package data from the ecosyste.ms API and calculates metrics about the number of unique ecosystems, total version counts, and the list of ecosystem names.

Parameters:

repo_url (*str*) – The repository URL of the package to query.

Returns:

A dictionary containing information about packages related to the repository.

Return type:

Dict[str, Any]