Git celebrating its 20th anniversary this week

Ideas for building on Git's concepts and capabilities

This week, Git is celebrating its 20th anniversary. Over the past two decades, this version control system has become the de facto standard for software version management and development. Furthermore, Git forms the foundation of major platforms such as GitHub and GitLab, supporting tens of millions of users from all over the world. Currently 95 percent of all developers use Git as their primary version control system.

After taking a brief look at the origins of Git and what has been accomplished in software version management over the past 20 years, this article highlights several adjacent domains where Git's versioning concepts can be or already have been successfully applied. In addition we present various ideas for extending Git's capabilities and usefulness in significant ways.

"I really never wanted to do source control management at all and felt that it was just about the least interesting thing in the computing world"

Off to a quick start

Git was created by Linus Torvalds in April 2005 to support the development of the Linux kernel after BitKeeper was no longer (freely) available and no other existing free system offered similar functionality. At the time, Linus had only four design criteria: patching should take no more than three seconds; take CVS as an example of what *n*ot to do; support a distributed, BitKeeper-like workflow; and provide very strong safeguards against corruption, whether accidental or malicious.

The newly created system was self-hosting within a week, and the first kernel commits and merges took place within two weeks. By the end of July, the maintenance and further development of Git were transferred to major contributor Junio Hamano, who holds this role to this day.

Git

Git is a distributed version control system mostly used for – but not limited to – the development and documentation of software code, datasets and specifications. Its main feature is the support of massive non-linear development, allowing large numbers of collaborators to participate in the development and maintenance of a project in a semi-independent way.

Projects are brought under Git control locally by creating a set of metadata in a hidden directory. Others can get their own local copies of the very same project through a Git server and work on these copies at their own discretion. Being very precise about the identification of files and patches, and keeping close track of development history, Git allows for fast and easy synchronization of changes between repositories, at the same time providing a high level of granularity in doing so.

In practice, projects are hosted at a central location, from where copies are made by developers and where approved changes are fed back into the main branch. Being fast and highly scalable, Git forms the foundation of major hosting platforms such as GitHub and GitLab, supporting tens of millions of users all over the world.

World domination

"I really never wanted to do source control management at all and felt that it was just about the least interesting thing in the computing world." That's what Linus said in an interview on the occasion of Git's 10th anniversary. [1] According to him, the biggest problem with BitKeeper was that is wasn't open source, which made a lot of people involved in Linux kernel development not want to use it. After discussions came to a clash with the owner of BitKeeper, it took Linus only a few days to write the basics of Git, as he had already been thinking about its requirements and concepts for some time.

Since its inception 20 years ago, Git has taken over the world of software version management almost completely – much to the surprise of Linus himself. 95 percent of all developers currently use Git as their primary version control system,[1] making it the de facto standard. Git also forms the foundation of major hosting platforms such as GitHub and GitLab, supporting tens of millions of users worldwide in what Linus calls "social coding". He explains the immense success of Git through its "distributed" nature and that it's so easy to start a new project. While Linus used to joke about aiming for world domination with the Linux kernel [1, 2], it appears that it is Git that has now actually achieved this status.

FAIR

FAIR is a set of Guiding Principles to unlock research data and make them machine-actionable:

- **Findability**: machine-readable and searchable metadata allow for automatic discovery
- **Accessibility**: (meta)data is accessible through an open protocol that supports authentication and authorization
- **Interoperability**: unambiguous (meta)data definitions (e.g. ontology) allow data to be integrated with other data, and be used by applications and workflows
- **Reusability**: data should have clear descriptions, licences and provenance, and adhere to domain-specific community standards (relevance)

As such, FAIR is closely related to the Semantic Web, which aims to make internet data machine-actionable by adding ontological metadata to allow machine reasoning. FAIR data is also closely related to Open Data, Open Science and FOSS, although openness is not a requirement of FAIR itself.

FAIR was formally defined in 2016 by a consortium of scientific, industrial and other stakeholders [1]. Since then, its principles have been adopted by several research institutes and are actively promoted and researched by all major umbrella organizations in the research-data ecosystem.

The GO FAIR Initiative is an international network aiming at helping implement the FAIR data principles, for example through its GO FAIR International Support and Coordination Offices (GFISCOs) and through the European Open Science Cloud (EOSC).

Free and Open-Source Software

Free and Open-Source Software (FOSS) is characterized by the use of specific copyright licensing. These licenses, however, are not primarily about the copyright of the code produced: in this case copyright legislation is merely used to facilitate collaborative development of public code within a community of (anonymous) contributors.

The current FOSS landscape is largely covered by two main licensing types. Both allow users to run, study, change and (re)distribute the (source) code.

- **copyleft**: Crucial in these licences is the requirement that the same rights *m*ust be preserved in derivative works. Most importantly this means that others cannot modify the code and redistribute it without also making the modified source code available.
- **permissive**: These licences *a*llow but do not require others to make modified source code of derivative works available, while at the same time forbidding any limitations on the usage and further development of the software. This way making sure the original software remains available and can be used with minimal restrictions – even in closed-source commercial software – makes permissive licences so attractive to businesses and industry.

FOSS licenses form the foundation of a highly dynamic ecosystem, combining strong competition and massive collaborative development and reuse in an evolutionary process. Over the past decades, enormous amounts of FOSS software have been created this way, including Firefox, LibreOffice, Linux, Python and WordPress. Its economic value is measured in tens of billions of euros annually [1].

[1] Stack Overflow Developer Survey 2022:
https://survey.stackoverflow.co/2022/#section-version-control-version-control-systems

To what extend Git implements the FAIR Principles

The FAIR Principles were not in themselves a new proposed standard, but a set of high-level computational behaviors and expectations to which many different possible standards could be applied. Below we give an indication of how Git Server implements the FAIR Principles.

F1. (meta)data are assigned a globally unique and eternally persistent identifier.

- Git Server uses multiple types of identifier systems to track many of the object types used by Git, such as repositories, users, commits and issues. These identifier systems largely fulfill the FAIR attributes of persistence, global uniqueness and resolvability.

F2. data are described with rich metadata.

- Although Git Server does not natively apply multiple, structured metadata schemas, Git-based platforms describe object types such as repositories, users, issues, pull requests and other objects.

F3. metadata clearly and explicitly include the identifier of the data it describes.

- Among modern versioning systems, Git Server excels at this important FAIR Principle; it allows for the separation of data and metadata through commit hashes and references, linking them with persistent identifiers. However, the formalized separation of data and metadata is more explicitly managed in Git platform implementations.

F4. (meta)data are registered or indexed in a searchable resource.

- The use of persistent identifiers and structured metadata in Git Server goes a long way to making Git platforms indexable registries of Git objects, which enables search.

A1. (meta)data are retrievable by their identifier using a standardised communications protocol. A1.1 the protocol is open, free, and universally implementable.

- While Git Servers use open and standard protocols, accessibility can still be constrained by platform-level control and policies.

A1.2 the protocol allows for an authentication and authorization procedure, where necessary.

- Git Server provides basic authentication (SSH, HTTPS), while advanced authorization mechanisms (e.g. role-based, branch protection and fine-grained API permissions) are implemented by Git platform providers.

A2. metadata are accessible, even when the data are no longer available.

- Git Server allows the preservation of metadata for archived or inactive repositories by keeping issues, pull requests and commit histories accessible. However, once a repository is deleted, its metadata is typically removed and no longer accessible.

I1. (meta)data use a formal, accessible, shared and broadly applicable language for knowledge representation.

- Git Server metadata is structured but does not use a formal semantic web language such as RDF or OWL.

I2. (meta)data use vocabularies that follow FAIR principles.

- Git Server clearly defines Git objects and actions, but does not use formal, FAIR vocabularies.

I3. (meta)data include qualified references to other (meta)data.

- Git Server supports linking to other data via, for example, repository dependencies, but does not support semantic linking between repositories, datasets or external resources.

R1. meta(data) have a plurality of accurate and relevant attributes. R1.1. (meta)data are released with a clear and accessible data usage license.

- Although the Git Server specification itself does not natively support licensing features, Git platforms typically allow and encourage users to specify licenses for their content.

R1.2. (meta)data are associated with their provenance.

- Although Git Server does not explicitly align with formal provenance standards like W3C PROV or DataCite, Git provides a detailed versioning system (commit histories) recording changes, authors, timestamps and commit messages that constitute precise provenance metadata to track project evolution.

R1.3. (meta)data meet domain-relevant community standards.

- Although Git Server does not impose domain-specific metadata structures, it does allow users to include domain-specific standards in metadata for repositories using README files and JSON/YAML metadata files.

Git has played an important role in developing the data culture we have today. As of 2025, GitHub alone hosts over a billion open data files.

Git and AI

Git has profoundly impacted the development of AI, including the current wave of foundation models and Generative AI (GenAI). This is for multiple reasons.

1. Git and hosting platforms such as GitHub have played a key role in sharing the open-source implementations that drive modern AI in particular and data science more generally. Today, most academic outlets – e.g. journals and conference proceedings – require or at least encourage that papers be published together with relevant code and data.
2. Git has evolved from a version control system focused primarily on source code into the foundation of a vast ecosystem for sharing data, documentation, specifications and more. Publicly available repositories often serve as training data for increasingly data-hungry, specialized, large language models (LLMs) such as Copilot and Codex. More broadly, Git has played an important role in developing the data culture we have today. As of 2025, GitHub alone hosts over a billion open data files [1].
3. The Git ecosystem has fostered a culture of international, decentralized collaboration in which scientists and practitioners around the world can access software and data, work on them locally, and then either contribute their changes upstream or maintain their own branches. This has implications beyond just technical benefits like increased robustness and the ability to take over abandoned open-source projects. It has also democratized who and how millions of people can contribute to projects ranging from small to large, and how they can learn about modern AI technologies.
4. Git has promoted reproducibility, replicability and transparency in science and AI development. This is especially true for major models published not only as open source, but also accompanied by essential metadata required to build on them, such as data about hyperparameters and a license clarifying conditions for reuse. Before the rise of Git, the idea that researchers would routinely share their data and code online for anyone to view, copy and comment on, would have seemed almost unthinkable.
5. Often overlooked, Git's temporal dimension provides a rich source of historical data that future AI models can utilize. Each revision of code, data or documentation captures an implicit narrative of evolving project goals, community feedback, feature integration, issue resolution and conflicts. While platforms like Stack Overflow explicitly provide such interaction as their primary offering – e.g. by suggesting best practice patterns for common coding problems – Git captures this implicitly, but on a much larger scale, across diverse domains and with richer metadata about the process. Interestingly, GitHub's new Copilot provides AI tooling to help improve hosted code, effectively closing the loop.

All that said, a lot of work remains to be done, as many of the datasets available on Git-based platforms are still not actively used for AI model training. A consequent push to align these datasets with the FAIR Principles has the potential to change this.

Before the rise of Git, the idea that researchers would routinely share their data and code online for anyone to view, copy and comment on, would have seemed almost unthinkable

Putting FAIR into Git

Between 2005 and 2010, Git designers discovered and implemented many aspects of FAIR, which would be explicitly formulated only later in 2016. In Git, datasets (mostly software code) are stored in a well-structured environment with rich-provenance metadata, which serves as the primary asset for making data reusable. Git's metadata by design is highly controlled and structured. So much so that versioning is even computable.

Git's metadata on stored files – whether they are software code, ontologies or FAIR vocabularies, or even a monolithic blob – reduces ambiguity and increases context, which are core objectives of FAIR, thereby enhancing machine-actionability. Git designers had discovered the development of FAIR repositories even before FAIR was invented.

Git and FAIR

The similarities between Git and FAIR extend further to their higher-level objectives. The 2016 paper that outlined the modern form of the FAIR Principles was itself focused on "scientific data management and stewardship". While systemic approaches toward effective data stewardship are still in their infancy, the FAIR stewardship of software provided by Git has been robust, trusted and cost-effective for two decades now. By 2012, GitHub had inspired generalist data repositories like the Open Science Framework to be a "GitHub for science" [1].

It's important to note, however, that Git is not formally semantic. There are no ontologies for Git, and there is no RDF capability in Git platforms to embed domain-specific semantics into Git metadata records. Although many parallels between Git and FAIR can be drawn, the *I*nteroperability principles appear to be lacking.

Also note that the FAIR Principles were not in themselves a new proposed standard, but a set of high-level computational behaviors and expectations to which many different possible standards could be applied. In any given instance, the implementation of FAIR needs actual services (such as identifier systems, indexing and registries, controlled vocabularies), software (for example, to manage authentication and authorization) and data representations that follow reporting frameworks agreed upon by domain communities.

Making Git data interoperable

Given the specialized use case for Git – primarily distributed software code versioning – and given its wide adoption, the world has already reached broad consensus on the "meaning" of Git statements. The question now is: can Git be made more semantic, particularly at the domain content level? In other words, can we FAIRify Git with respect to *I*nteroperability?

A number of options to accomplish this present themselves: One way to capture domain-specific, machine-actionable FAIR metadata is to use the CEDAR Embeddable Editor [1]. Another idea would be to use nanopublication-based FAIR Digital Objects[a] to create machine-readable metadata statements on the Internet to point to. These would then describe critical resources in GitHub, GitLab and Gitea, such as stored files, README files, Issues, Patches and Pull Requests.

Using either technology and a minimal set of reusable metadata templates, a rich semantic metadata (proxy) layer could be created for any Git repository. It would support the findability and discoverability of resources, describe (or even enforce) access controls, augment the licensing, and add a provenance metadata layer native to Git. Adding the I for *I*nteroperability in this way would make Git fully adhere to the FAIR Principles.

[a] Magagna, B., Bonino da Silva Santos, L. O., Kuhn, T., Ferreira Pires, L., & Schultes, E. (2025). Nanopublications as FAIR Digital Object Implementations. Open Conference Proceedings, 5. https://doi.org/10.52825/ocp.v5i.1417

It turns out that if you say "patch" instead of "amendment" and "code freeze" instead of "plenary vote", many members of the software community suddenly understand what you are talking about![2]

[2] EPFSUG: The hacker perspective on lawmaking

Law as code

Just as software code is a formalization of an algorithm or intent, legislative texts aim to formalize legal code. And just like the development of software, the legislative process requires precise version management and collaboration among many participants. People involved in legislative processes have recognized these similarities and have put to work 'software version control'-like systems – or even Git itself – for their own purposes.

European Legislation

One example is LEOS, short for Legislation Editing Open Software. [1, 2] This tool facilitates the drafting of legislative texts and generates legislation in an XML format, that way supporting interoperability between European institutions. LEOS is the best-known FOSS tool created by the European Commission and freely available under the EUPL license. The software is also used by several Member States and various other public administrations.

Parltrack [1] is unrelated but can be considered complementary to LEOS. It combines information on dossiers, representatives, vote results and committee agendas of the European Parliament into a single database and allows the tracking of dossiers using e-mail and RSS. The platform improves the transparency of legislative processes. For example, it allows you to see who are the most influential Members of the European Parliament related to a specific dossier. Most of the data presented on the website is also available in JSON format for further processing, just as a dump of the full database. The Parltrack software itself is available under a free software license.

"GitLaw"

Both LEOS and Parltrack attach great importance to the history of the development process, one of the strongpoints of Git. Washington D.C. has even published an authoritative copy of its laws on GitHub [1]. It allowed one of its citizens to fix a typo using a pull request [1].

Others have been philosophizing about a public "GitLaw" system specifically for legislative texts. Such a system would allow both drafters and citizens (through crowdsourcing) to propose bills and amendements using pull requests. Problems could be addressed through a mechanism of issues and fixes (patches). For every change it would be fully transparent who had proposed it (tracking). Legislative texts and snippets could easily be reused. And notarial deeds such as wills could be digitally signed and stored in a protected section of the GitLaw system. [1, 2]

It turns out that tech enthusiasts welcome the publication of any body of law or structured information on the legislative process, and are eager to explore how Git can be used to exploit this data. [1]

Top-down?

The bottom-up approach of a Git-like or Git-based system could provide a solid foundation to build more advanced functionality on.

The Greek 3GM project, however, shows that a top-down approach can be used too. This GSoC 2018 project parsed, analyzed and compared laws and amendments from the Greek Government Gazette using Natural Language Processing techniques. That allowed them to have amendments automatically merged into the law in the correct order, providing a fully codified, current version of each law at any given moment. The tool also clustered the laws according to their content, and ranked them based on incoming references. [1, 2]

AT4AM (Automatic Tool for AMendments) does something similar for the European Parliament: it was developed to help creating, editing and managing amendments ("diffs"). [1, 2, 3, 4, 5] Just like LEOS, it is based on the Akoma Ntoso XML schema, part of the OASIS LegalXML initiative [1]. AT4LEX (Authoring Tool for Legal Texts) was later developed for the creation of intial report drafts. Both tools are part of the e-Parliament Programme, aiming to establish a fully digital legislative text production chain. [1, 2]

Clear and clean

From the above, it appears that Git after 20 years has not only fully matured but has also become the main versioning tool in modern software development and the foundation of a global collaborative ecosystem. In addition to Git's "distributed" nature, the initial code and concepts being clear and clean have undoubtedly contributed to its tremendous success. "I remember that I was very impressed by the simplicity of the design and the clarity of the code," Junio recalled in an interview on Git's 15th anniversary [1].

Yet the main reason for Junio to start contributing to Git – rather than any of the other open-source version control systems available at the time – was that he wanted Linus to return to his work on the Linux kernel as soon as possible. He recalls that quite a few developers from the kernel community were implementing Git features in a rather chaotic competition. Working harder and faster on well designed and well implemented features, and presenting these better than others, is what Junio believes was decisive in Linus picking Git's new lead developer.

The best is yet to come

Despite everything that has already been accomplished, Junio said he believes the best features are yet to come – and we tend to agree. Git's impact on AI alone shows how relevant this versioning system is even at the forefront of today's most advanced technological developments. A notable example is Hugging Face, which is currently building a new ecosystem for machine learning, rooted in Git and FOSS principles. Above you could read about the idea of "GitLaw" and how Git's versioning concepts could be applied to legislative texts. Here below you can read how Git could be extended in various ways using domain-specific languages. Still, we are convinced that with the ideas presented in this article we have only just began to scratch the surface: the best is yet to come!

Extending Git using Domain-Specific Languages

A highly valuable extension would be to make the Git versioning system suitable for all types of files, including non-textual ones that currently can only be handled (stored) as monoliths. Making them "diff-able" in a meaningful way would unlock these files to the full capabilities of Git.

Bringing non-textual files under the Git versioning regime could best be achieved by inserting little codec layers into the Git system. Using Domain-Specific Languages (DSLs) for this would allow Git's functional domain to be extended in various important ways:

- Introducing a textual DSL to represent application-specific, non-textual data would allow for easy translation back and forth between the two formats. Instead of storing the opaque, non-textual version, Git would store the declarative DSL equivalent and be able to manage the data based on meaningful, line-by-line differences. In this setup, the original non-textual format would only be used for import/export-like functionality.
  Using a textual DSL for data storage to enable high-granularity versioning is a proven approach. [1]
- A DSL could also be used to provide an interface that protects sensitive data within a private repository by filtering out confidential information before it can be accessed by the user.
  In this case, a DSL would be used to specify precisely what policies to be applied. Nescio is an example of such a filtering DSL (for network traffic). [1]
- A similar approach could be used to make privacy-sensitive data from a private repository available to others. For example, access to anonymised patient records could be granted to researchers without violating GDPR/privacy rules.
  Again, the privacy policies specified in the DSL would be part of the private repository and enable responsible access.

In all of these cases, DSL code would specify desired policies that can be attached to Git hooks [1] on the server.

- GO FAIR Foundation: https://www.gofair.foundation/
- Fairscholar Foundation