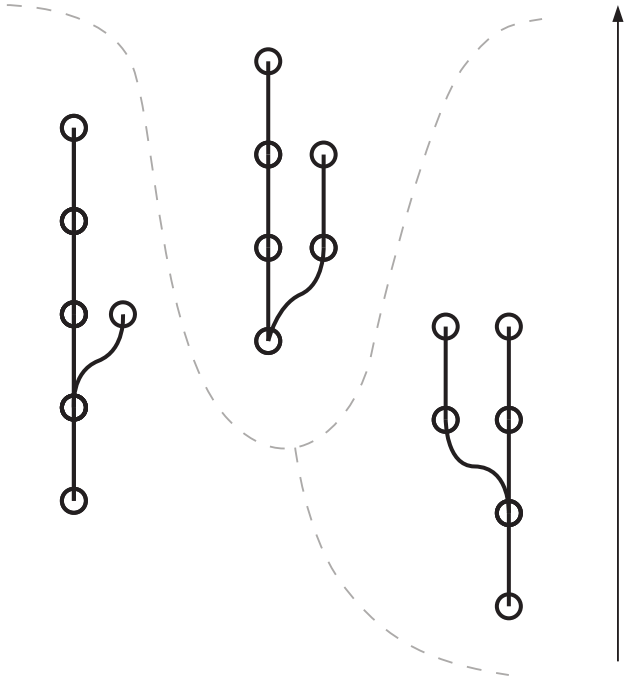
The scale of change

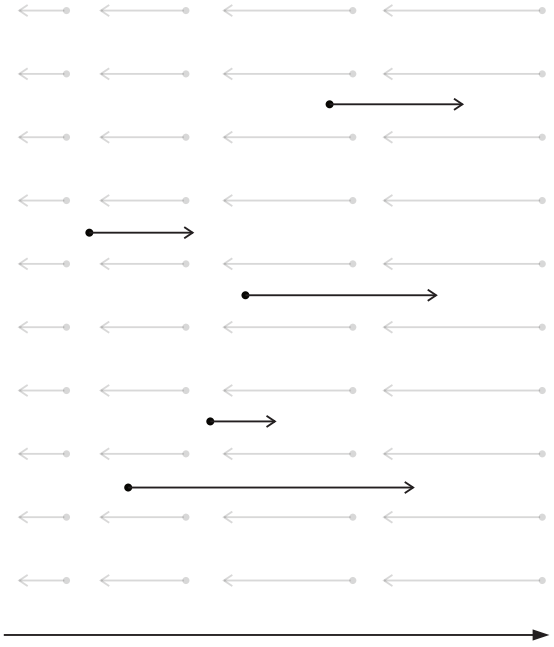
You are starting a software company. Whether it's an online service, a mobile app, or a deep tech startup, your long term goal is probably not to write the software itself, but to achieve certain business targets using the software you're developing as a tool.

But what brings value to your software? An inert piece of software will only be able to deal with the conditions considered when it was created. But the world isn't static, and neither can your software be. As time passes, everything outside your software changes: the data you work with, the way your software is used, the platforms it runs on, etc.

Therefore, in order to stay relevant and make your software actually valuable, you have no choice but to keep changing your software. In fact, the longer your software exists, the more work you need to put towards keeping up with the times. Worst case scenario, at some point you either lose the velocity and stop making progress, or even fall behind rapidly. So how can you catch up with the time?



Even though different projects are developed at the same time, it's difficult to establish a connection between them as there's no link in the data model behind the collection of different histories



Most companies use version control systems to manage their source code. Version control systems provide a way to keep track of changes. Git is probably the most popular version control system today. On top of version control systems like Git, platforms like GitHub provide tools for collaboration such as code review, issue tracking and so on.

At the core of any version control system there's a concept of change. In Git, a single atomic change is referred to as "commit". Developers contribute commits one by one, often preceded by automated validity checks and code review. Every commit represents its own unique state of code.

A chain of changes comprises a history. In terms of Git, every repository usually contains one history, often scoped to a specific project. The rise of collaboration platforms like GitHub has contributed to the popularity of development models where every project gets a separate repository. This becomes especially relevant over time: repositories are often created to enable code sharing and collaboration.

This, in turn, means that different projects have different histories that aren't connected, even if they are developed on the same time axis. This might be acceptable when the projects are completely independent, however, in practice that is rarely the case.

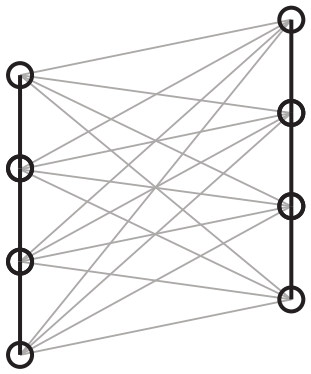
So what happens when your projects aren't actually independent? There are different ways to connect projects that developers came up with over time since the multi-repository model became popular. However, one common denominator is the need to decide between either specifying the exact points in history which are connected, or leaving one end loose and just always pointing to the latest. Both of these approaches present their own challenges that can't be solved easily.

When using Git, a common approach is to just pick up the latest version of the dependency from a repository and a branch. While this seems simple enough, it doesn't scale neither in time nor in terms of the size of the codebase: every time an upstream dependency changes, its tests only take into account its own code without downstream consumers. Therefore, changing a software entity that other entities depend upon is like making changes blindly without knowing whether or not they introduce problems.

Instead, it is common to pin dependency versions by specifying commit SHA1 of a submodule or using the SHA1 in some other way, like with language-specific package versions, it would be possible to have reproducible builds and tests. However, every time the dependency is updated, the update wouldn't propagate automatically to the software entity consuming it. There are different solutions for automatically bumping dependency versions, but something most of them have in common is that the bump happens after the change is already merged in the dependency.

Moreover, pinning means that from the whole exponentially growing set of different version combinations, only some versions are meant to work together. Building a working set of different versions therefore involves both guesswork and reliance on human-created rules such as semantic versioning. Attempts to automate verification of different versions combinations are doomed to fail as they would require handling an exponentially growing number of combinations.

This means that in a context of a multi-repository development model with projects connected by a dependency relationship, change that appears to exist within a context of a single repository might actually be a change within a context of multiple ones. However, this is not reflected in the development model or acknowledged by the developers. As a result, every time a change is made, its impact is only seen much later, requiring extra runs of this particularly long feedback loop.



When projects with disconnected histories are in a dependency relationship, the potential number of version combinations grows exponentially — even when using semantic versioning

This contradiction eventually resulted in an attempt to move away from the multi-repository paradigm. This contributed to the rise of popularity of monorepos.

In a monorepo, multiple software projects reside together in one repository. Some of these projects may be connected by dependency relationships. This arrangement enables developers to make atomic changes across several different components. Because components are tested together, there is no possibility of confusion about what versions of what components work together: any given commit represents a valid state. Moreover, when making changes to software components that other projects in the monorepo depend upon, the effect of the changes is visible immediately, dramatically shortening the feedback cycle.

However, monorepos come at a cost. First of all, most software development platforms like GitHub were not made with monorepos in mind. This is mainly reflected in UI and UX choices: developers are used to the model of one repository per project, with issues, pull request discussions and other metadata scoped per-project as well. This creates a psychological effect for developers where putting multiple projects in a single repo is seen as removing a degree of separation and mixing code that shouldn't be mixed — even though the repository layout has no effect on those qualities.

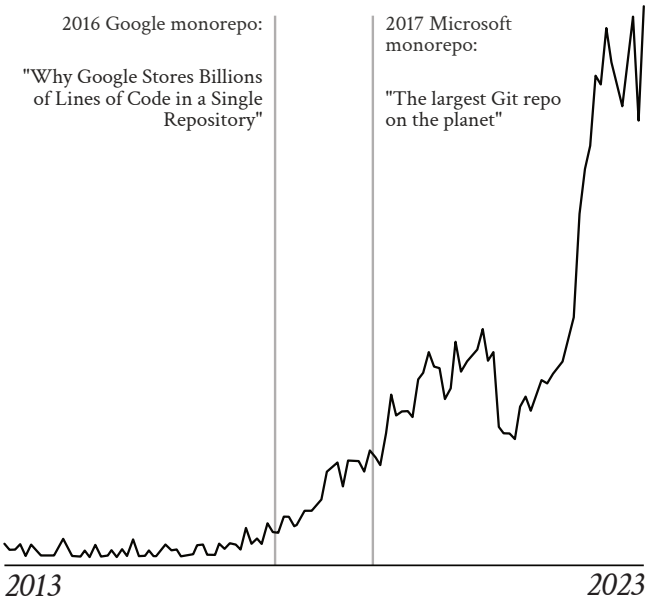
Additionally, at a certain scale, the lack of scoping of metadata such as issues and pull requests creates information noise for teams that only are interested in a subset of projects.

Secondly, as the amount of code in monorepo increases, it becomes increasingly more difficult to deal with the size of code. One of the most common complaints about monorepos is the process of working with them being slow. Developers often have negative experiences working with really big Git or Subversion repositories that were not using any additional tooling to improve the monorepo experience. To start working with the monorepo often means running a full clone for hours or even days.

Finally, lack of granular access control means that any given developer can access either all of the code or none of it. It becomes even more complicated when organizations want to share a part of their codebase with other organizations or as an open-source project.

Therefore, while monorepos became more popular, only the companies that could afford spending on custom tooling to make the workflow more bearable. Examples include tools that enable Git clients to do partial checkout, as well as automatic repository synchronization tools.

This presents a problem of scale that over time becomes more and more relevant in a world increasingly defined by software.



Relative search interest in monorepos

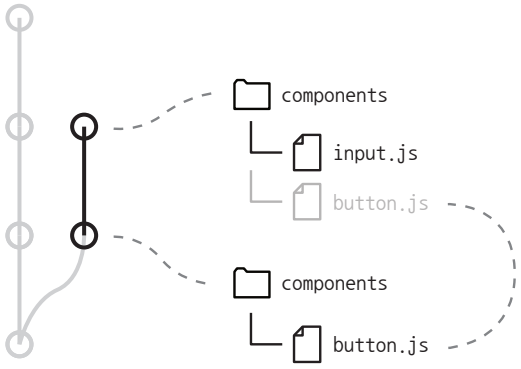
Data: Google Trends

While the benefits of using a monorepo are well defined and very attractive for a software organization, the downsides hinder adoption. So what can be done about the problem of scale?

One assumption that might arise from the available literature on the topic on monorepos is that there exists a hard choice between a single and multiple repository model. Let's examine this assumption in depth in context of Git as a version control system.

In Git, repository history consists of commits. Commits are nodes in a directed acyclic graph. Every commit refers to a tree of objects. Trees represent structures similar to how files and folders are laid out in a filesystem; every intermediate tree node represents a path component, and leaves of the tree link to objects that store the contents of the files.

Every Git commit can be identified by the combined hash of the object tree it points to and other data such as commit parents. This means that any given Git history can be transformed in a variety of ways by going through the history and recombining parts of the trees to get a projection of the repository. This projection would only contain relevant parts of the whole history.



The data model of Git makes efficient history transformations possible

Because of how the Git data model is designed, this history transformation can be performed incrementally, with parts of transformed histories cached for reuse. Therefore, this transformation can be extremely cheap, even on really large repositories.

This presents a new realm for Git repositories: any repository could be viewed through a number of different projections, each only

focusing on relevant components. This new Git history would be just like any other repository history: it would be possible to clone it and make new commits. In practice, it would give developers a way to work on a projection of a monorepo, only working with commits and code relevant to them.

Moreover, it would be possible to perform the opposite transformation: when a developer pushes a commit to such a projection, because the initial transformation is known, and can be restricted to be made fully reversible. This means that every change pushed to a projection, even though made from a limited view, is still a part of the single history, and in turn, the benefits of monorepos are preserved even though developers do not work with the whole big repository.

There have been attempts at writing tools for editing Git histories. For example, "git-filter-branch" command, included in Git, offers a command line interface for one-time operations, like removing unwanted files from a history. Like most other tools of that nature, it is too slow for practical use in this scenario. The reverse transformation is also not supported.

However, this is still a theoretical description. To make it work in the real world, we can wrap this history filtering mechanism into a service. There are different ways this service could work; we will look at one of the possible scenarios.

The core of this technology was implemented in an open source project "Josh" — from "Just One Single History". Josh provides a bare-bone history transformation solution for companies to build on top of, and integrate in their own tooling.

Metahead is a continuation of this idea that aims to provide a complete software engineering and version control platform, democratizing this technology by making it available for companies of any size. Metahead erases the boundary between monorepos and multirepos without requiring organizations to develop their own complex tooling. It also scales with your organization, enabling you to avoid the wall of exponential complexity and keep making changes.

Metahead connects to your existing repos and combines them in one single history in a meta-repository which serves as the source of truth. It enables you to easily rearrange them into views as required by your business needs. Each of these views is a fully-fledged git repository. Pushes to them are reflected in every affected view, where CI checks can ensure the validity of the main history. Individual views can also have different access permissions, enabling granular access control.

Your software is only as valuable as the changes you make — and with Metahead, you bring value to your organization through your software.