

Engineering at Anthropic

Redesigning source control at Anthropic

As our merges per day grew more than tenfold, source control became our biggest development bottleneck. Here, we share how we migrated to an in-house version control system in five months.

Published Oct 13, 2026

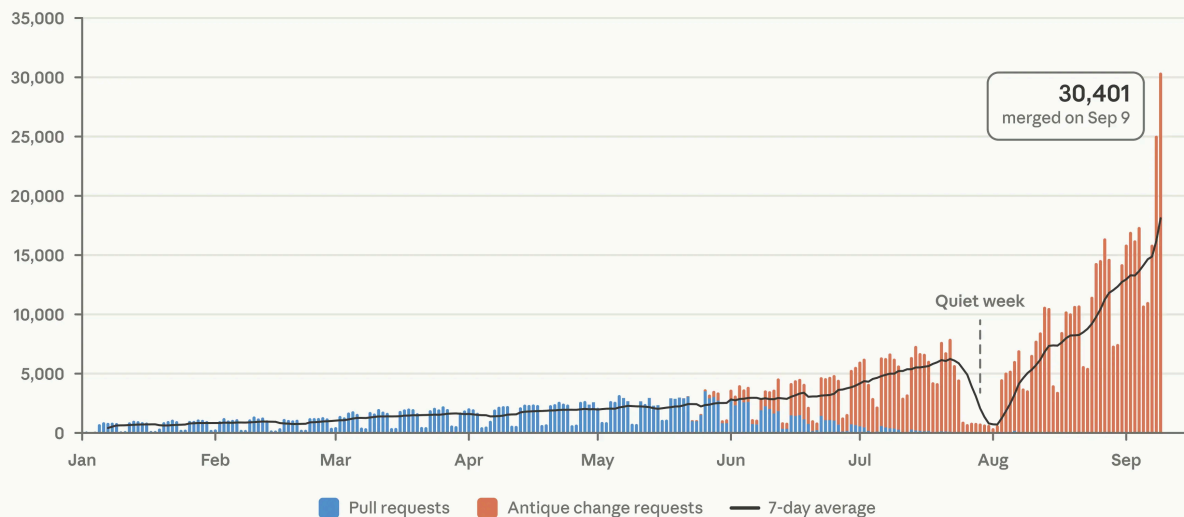
At Anthropic, coding is no longer the bottleneck to building systems or doing research. Claude is able to autonomously iterate on research loops, micro-optimize complex programs, and build new services from the ground up at a speed and scale we haven't seen before. **The bottleneck, now, is all of the developer infrastructure built for human scale: source control, builds, and CI.**

In May 2026, Anthropic averaged around **2,500** code changes merged per day; as of September, we merged over **30,000 per day**. We don't expect this rate of growth to slow down anytime soon; the latent demand for merging code is practically infinite, and every bottleneck we remove immediately leads to a further increase in the amount of code we produce.

← 3a 4 merge 12x per day

Changes merged per day

Anthropic monorepo, January–September 2026



Changes merged to our monorepo per day, from January to September 2026.

In order to support this growth, we built *Внезапная скорость миграции* Scion, an in-house version control system, and fully migrated our monorepo from git in five months. We also built and shipped a code review surface (called Antique), code search, a rich source control IAM system, novel ways of managing and landing large amounts of reviewed code, and more, all in that same time period. In this post, we'll focus on how we built Scion specifically. It makes for an interesting case study: Claude's ability to write code was both what put us in this situation in the first place *и* how we were able to quickly build a high-quality, scalable system to deal with it.

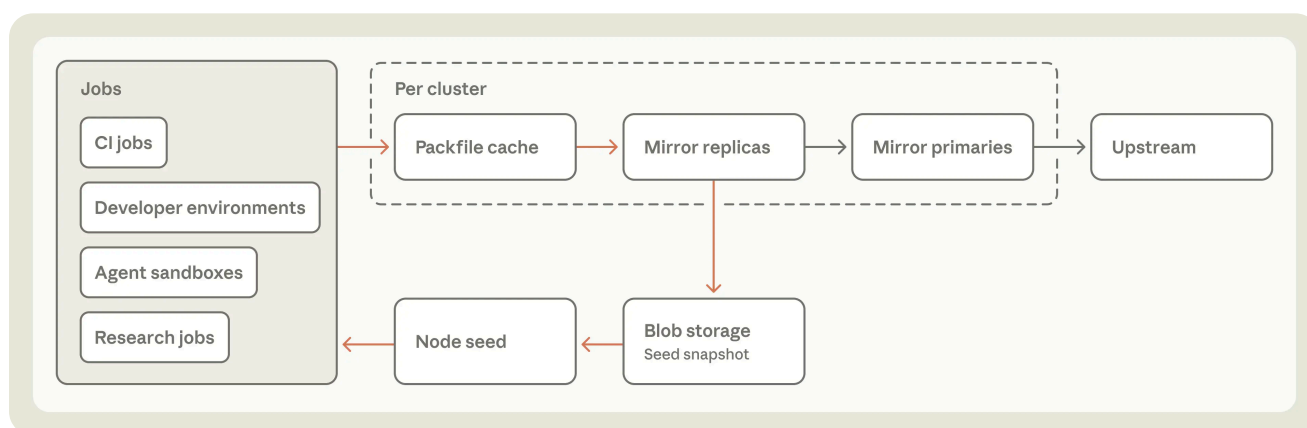
Growing our monorepo

Before we explore what we built, we should acknowledge how unusual our workload is. Unsurprisingly, Anthropic uses Claude to write software far more than almost any other company, so we're often the first to see where tools built for human scale start to strain. Source control is just one example of this. While we don't expect many organizations to have these problems in the very near term, we suspect it won't be too long before others who use AI coding tools experience these same pressure points, which is why we wanted to share what we built and what we learned.

Git is a decentralized system: every clone is a complete, self-contained repository that can compute diffs, read history and blame, and check out any revision while fully offline. For the vast majority of repositories, this is the right design, and it's a large part of why git became the standard.

As our monorepo grows, it increasingly doesn't fit that design space. The history of our main branch alone is almost 100 GB, and a *compressed* working copy is over 25 GB. We perform on the order of 25 million code checkouts per day for CI, before counting developer environments, agent sandboxes, and research jobs. Fetched naively, that would be over 600 PB of git traffic per day.

Git offers tools for working with a slice of a repository, such as shallow clones, sparse checkouts, and promisor remotes. Beyond these, there are standard workarounds for large monorepos, like internal git mirrors and repository snapshots (seeds) kept in blob storage. We've used all of them—at our peak, we operated several hundred internal git mirrors to deal with the load. This bought us time, but they became harder to operate as our merge rate increased. Seeds fell further from the tip of main between refreshes, and it took more and more care to keep operations that depend on history, like diff and blame, working on partial clones.



The typical setup for scaling git checkouts.

Taking a step back, we noticed that, like most companies, we were using git as a centralized system with one authoritative copy, but nearly all of our clients were agents and jobs that are online by definition. The properties that make git excellent for distributed, offline work were actively undermining our ability to scale as the repo grew. So we asked ourselves: if we designed source control with the fact that agents are always online in mind from the start, what would it look like?

Principles for agent-scale source control

With agentic coding, many common assumptions about how software is developed are evolving. We embraced these changes when building Scion, which allowed us to take a fundamentally different approach than git on several core principles.

Source control for huge monorepos has prior art, notably Google's Piper and Meta's

Sapling, so we were able to build Scion with their hard-won lessons in mind. Our principles are:

Clients are always online. Git assumes that clients work offline and need a complete copy of the repository. **Since nearly all of our clients are agents and jobs**, we designed Scion's client and protocols around a server that is always reachable, and have clients fetch data when they need it.

Client-server asymmetry. In git, every client is capable of being a full-fledged server. In Scion, the **server answers questions about file history and blame**, creates commits, performs merges and rebases, and so on. The **client maintains a bounded local object cache**, lays out the working copy files on disk, monitors for filesystem changes, and gives users an ergonomic way to work with the server. It can compute very little on its own but, in exchange, it requires much less data locally.

A comprehensive source control service can power many novel applications.

With git, a CI job or tool that needs to ask a question about the repository, such as who last changed a line or how two commits are related, has to maintain a local copy of the repo or hope that vendor-specific APIs provide the answer. **With Scion**, **all of the source control queries the client uses are also exposed via API**, so in many cases jobs don't need a local repository at all. Many internal tools, such as our automated bisect and revert pipelines, wouldn't be practical at scale without this cheap, programmatic access to source control.

Everything must be horizontally scalable. At our peak, we ran several hundred git mirrors, which significantly strained capacity and no longer scaled once the repo outgrew the mirrors' disks or memory limits. In Scion, the repository's data is distributed, and every component is independently scalable; **there is no fully materialized copy of the repo anywhere**, even on the server. There is no equivalent of packfiles (git's compressed bundles of objects), so no repository maintenance is needed.

Commits must be cheap, incremental, and lock-free. In git, you commit locally and then push to share your work. **In Scion, every commit, rebase, or merge happens on the server**, so there's nothing to push, but each of those operations needs to be fast. **The cost of creating a commit and updating its downstream data structures has to scale with the size of the change, not the overall size of the repo**, or the server can't keep up. The development lifecycle in Scion discourages refs (pointers to commits, such as branches and tags), except for the main ref or release branches, because **global state that every server has to agree on is a looming scaling problem.**

⏏ Skip to footer ↗

The working copy is optional, and sparse by design. Because our monorepo is so large, and because we check out code tens of millions of times a day for CI, we wanted to avoid checkouts whenever we could. With Scion, a CI job that needs the diff between a commit and its merge base, for example, can ask the server for it. The same goes for writes: many write workloads can also skip the checkout and function entirely via API. **Scion's protocol is centered around sparse checkouts, which only contain the directories the client asks for, and virtualized checkouts, which materialize files on demand, so when a job does need files on disk, the client only pays for what it actually reads.**

Чел-то напоминает базу данных и клиентов.
 ↳ Данные хранятся в базе данных
 ↳ Клиент может добывать данные через API и ему редко нужно тянуть данные к себе на диск.

How Scion works

It's helpful to understand how these principles came together in practice when building Scion. These insights can apply to other source control systems or, more broadly, to other stages of the software development lifecycle.

Data structures

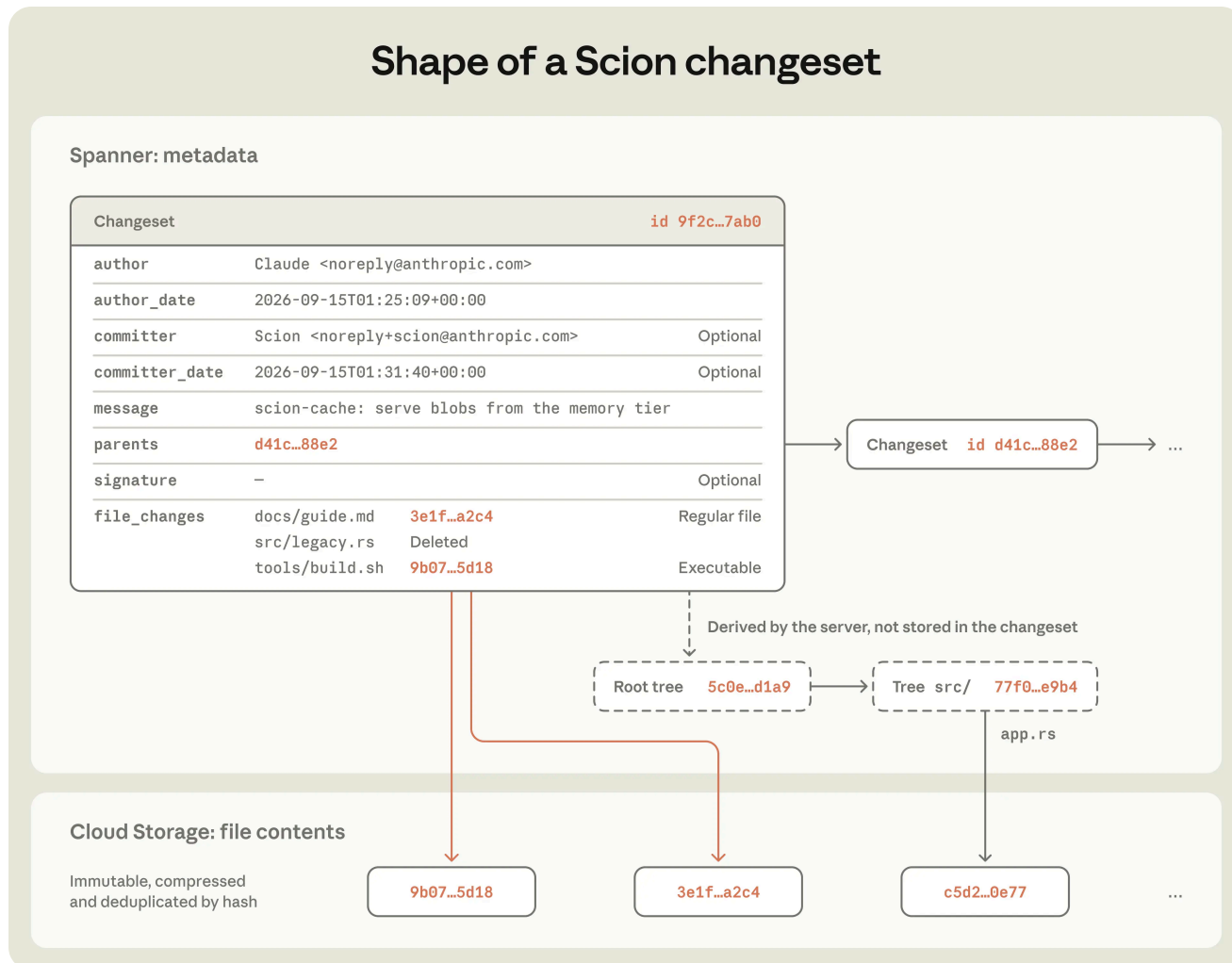
In Scion, the unit of work is a "changeset," an entry in a write-ahead log. A changeset stores most of what you'd expect from a commit in git. But where a git commit stores a pointer to the tree, a changeset stores a list of the files that changed relative to its first parent,¹ along with content-addressed pointers to the compressed file contents in blob storage. **Changesets are created server-side and are available everywhere, instantly.**

WAL как из баз данных :)

На самом деле instantly в распределенных базах данных не бывает. Возможно, здесь имеется в виду гарантия консистентности в виде инкрементальности.

Тут мешает ограничение физики в виде скорости света.

Shape of a Scion changeset



The shape of a changeset in Scion.

This structure lets us accept changesets with very high throughput because there are no packfiles to accept, no locking or cross-server coordination, and no refs to move around. Scion does technically support refs, but since neither creating nor fetching commits involves them, moving one is a compare-and-swap on a single

Spanner row that happens separately from creating changesets.

Spanner тут это связано с Google Spanner base данных? Нормализует транзакции и использует CQRS и оптимизирует write и read path (которые раскидывают по разным серверам)

A changeset records only what's changed, so on its own, it can't quickly answer questions like who last edited a line or which commits affected a file. To answer those, Scion builds additional data structures from the changesets, including dedicated structures for file blame and path history and skip lists for the commit graph. We call everything that isn't directly part of the changeset "derived data," and we compute it all through one generic mechanism. Each type of derived data is defined recursively, so its value for a changeset is computed from its values for that changeset's parents.

① Возможность расширяться

This mechanism has three benefits. First, we can easily add a new type of derived data when a new data structure would make some operation faster. Second, for each type, we can choose whether to compute it eagerly or defer the work, so we avoid materializing data we don't need. And third, because the client never sees the

② Интерпретируем

③ Интерпретируем

internals of these data structures, we can optimize them without changing the client. We've added new types of derived data many times as the system has grown, such as new indices for performing copy detection or path-based file search.

Scaling Scion

A small fleet of Scion servers provides source control APIs for any operation a client needs, such as getting changesets, computing diffs, finding merge bases, and fetching blobs. All source control data is content-addressed, so a piece of data never changes once it's stored, and any server can cache it without checking with the others. As a result, the servers are horizontally scalable and require little coordination. They are essentially large caches that can also compute derived data on the fly and write the results back to Spanner. ← Это больше свойство. Оно действительно сильно помогает с оптимизацией типа кеширования

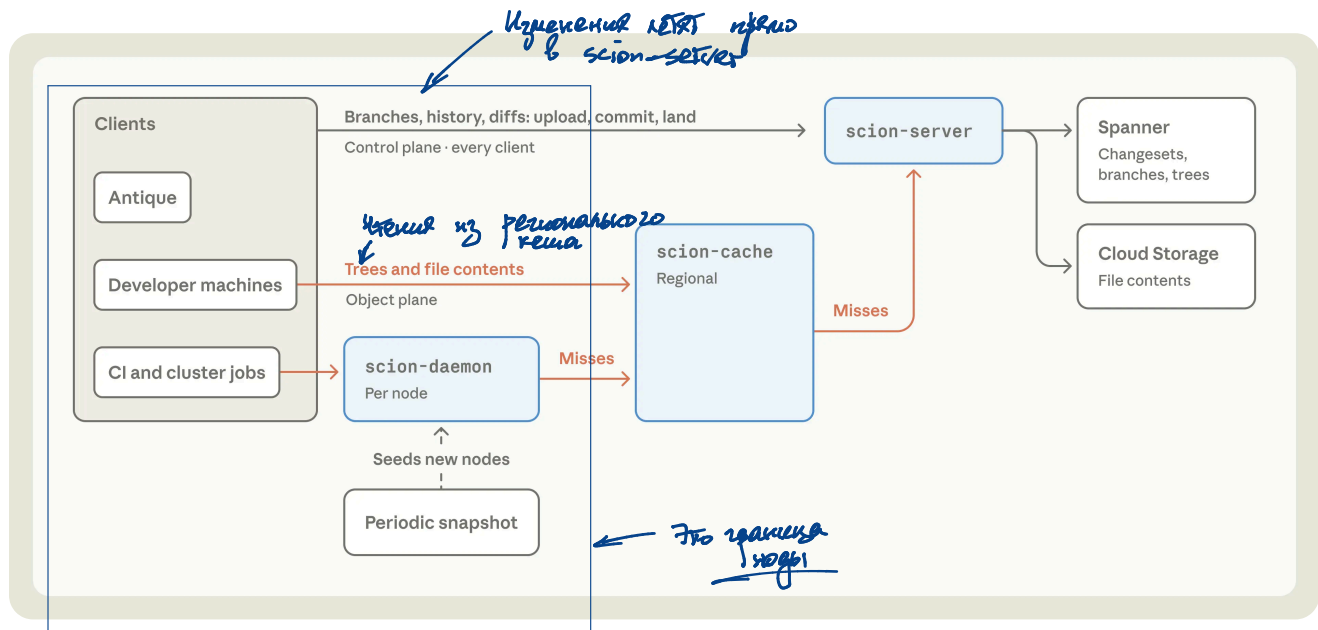
Scion's many consumers no longer need to maintain local repositories of their own. Fetching and maintaining the repository used to be a significant maintenance burden for CI jobs or tools that needed to ask source control questions, particularly about commit ancestry and blame. Early in the migration, before we had our own client, repo format, or protocols, Scion only answered these "metadata" queries. Fetching file contents and materializing working copies are far more data-intensive operations, and they needed a design of their own. ← Хорошо разделили типы API и реализовали их выкачивали схемы и их имплементацию отдельно

Scion's scn client deviates from git's fetch protocols in several important ways and, as a result, lends itself to a highly scalable architecture that requires significantly less CPU than git servers. Scion's negotiation doesn't tell the server what the client already has; instead, it operates exclusively on trees and blobs the client wants. Starting from a root tree, the client checks its local store for missing trees and requests them recursively, then repeats the process for blobs. Every request is for a specific object that never changes, so any cache can answer it, and the architecture closely resembles traditional tiered caches. The process is also efficient and resumable, because there is no single multi-gigabyte packfile response. Sparse and even virtualized checkouts don't need special support, because the client simply doesn't ask for trees it doesn't want. ← Инициализируется в начале - это важно

In the fleet, the client talks over a socket to a daemon, scion-daemon, that stores objects for the whole node. Most of our nodes need access to the monorepo, and jobs on the same node reuse most of the same objects. Each daemon starts from a periodically refreshed snapshot, then talks only to the tree and blob cache servers in its region, called scion-cache. These cache servers stream objects directly from memory, in the disk format that the daemon will consume, so they don't need to spend time computing the client's missing objects or performing delta ← Идея на уровне node, это все обрабатывает все объекты из для разных запросов, а как как у нас данные инициализируются то к ним можно сразу получить доступ

compression. When a cache is missing an object, it asks the authoritative scion-servers, which read from Spanner or cloud storage as needed.

The scion-daemon, scion-cache, and scion-server all respond to identical RPCs for this protocol, so the client doesn't need to know the cache topology. Any tier could theoretically talk to any other, but we strictly enforce the tiering to isolate failure domains. We want to avoid a situation where a failure in a regional scion-cache overwhelms the upstream scion-servers and brings down the entire system.



Scion's checkout architecture.

Each tier of this system maintains a cache hit rate of over 99% and does not require long-lived state, so it's easy to scale up if needed. Our Scion infrastructure footprint is now 200x smaller than our previous git mirroring infrastructure in terms of CPU and memory requirements. CI and research jobs check out code two orders of magnitude faster (in seconds rather than minutes), which frees up a sizable amount of capacity across the fleet; CI alone realized a 25% CPU capacity win. Neither clients nor caches need to perform repository maintenance, which simplifies the lifecycle management of our jobs.

What we learned building Scion

We started building Scion around the same time as Claude Mythos Preview and Claude Tag became available internally. Both of these releases simultaneously amplified the need for Scion and meaningfully accelerated how quickly we could build it.

Handwritten notes in Russian: 'Пресловутый самоактивирующийся цикл со сложными формами капитализации' (The so-called self-activating cycle with complex forms of capitalization) and 'условно, технический прогресс, где улучшение не складывается и не переносится' (conditionally, technical progress, where improvement does not accumulate and is not transferred).

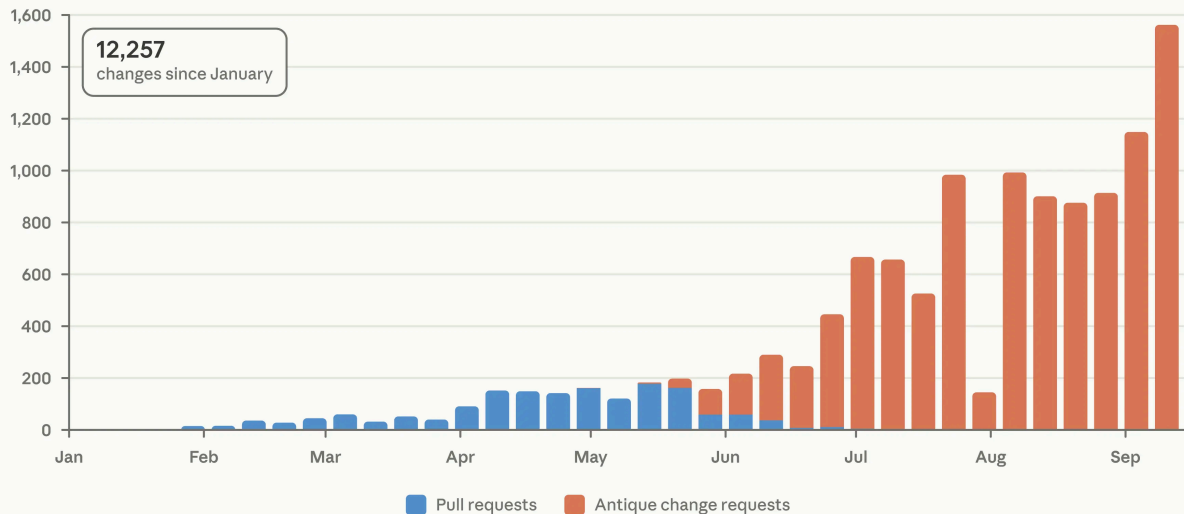
Here's a particularly striking example. Earlier in the year, we had largely handwritten the first version of the git-to-Scion ingestion pipeline as a prototype.

We thought it was good—until we got our hands on an early internal snapshot of **Claude Mythos Preview**. We asked it to rewrite the whole thing, and with no further instructions, it produced a version that consumed 20x less memory and was 4x faster, while preserving all of the implicit invariants around ingestion ordering and data structures.

Такие результаты от AI получаются & перескальются сферичности

Changes to the Scion and Antique codebases per week

scm/ and antique/ directories of the Anthropic monorepo, January–September 2026



Merged changes to the Scion and Antique codebases per week, from January to September 2026.

What we learned building and operating Scion offers some interesting lessons for what building agent-scale software could look like.

Rapid development with a small team will become more common. A team of seven people built both Scion and Antique (our code review tool, which we could easily write a whole other blog post about), not to mention half a dozen related systems, and fully migrated to them in five months. We did all this while the volume of code merged per day increased by 10x. In the past, developing and deploying a source control system like this would have taken a company years of iterative migration, with many more engineers. We think this sort of fast, nimble migration will become more common in the future.

Engineers are no less valuable, but their work is changing. All of the engineers who worked on this system would now rate Claude as better at writing code than them. But our engineers still supplied the invaluable understanding of the high-level inter-system components and the design-level tradeoffs, making informed choices around the problems we wanted to solve and the scale we knew we needed to achieve. Very often, we found that the role of the engineers was to tell Claude

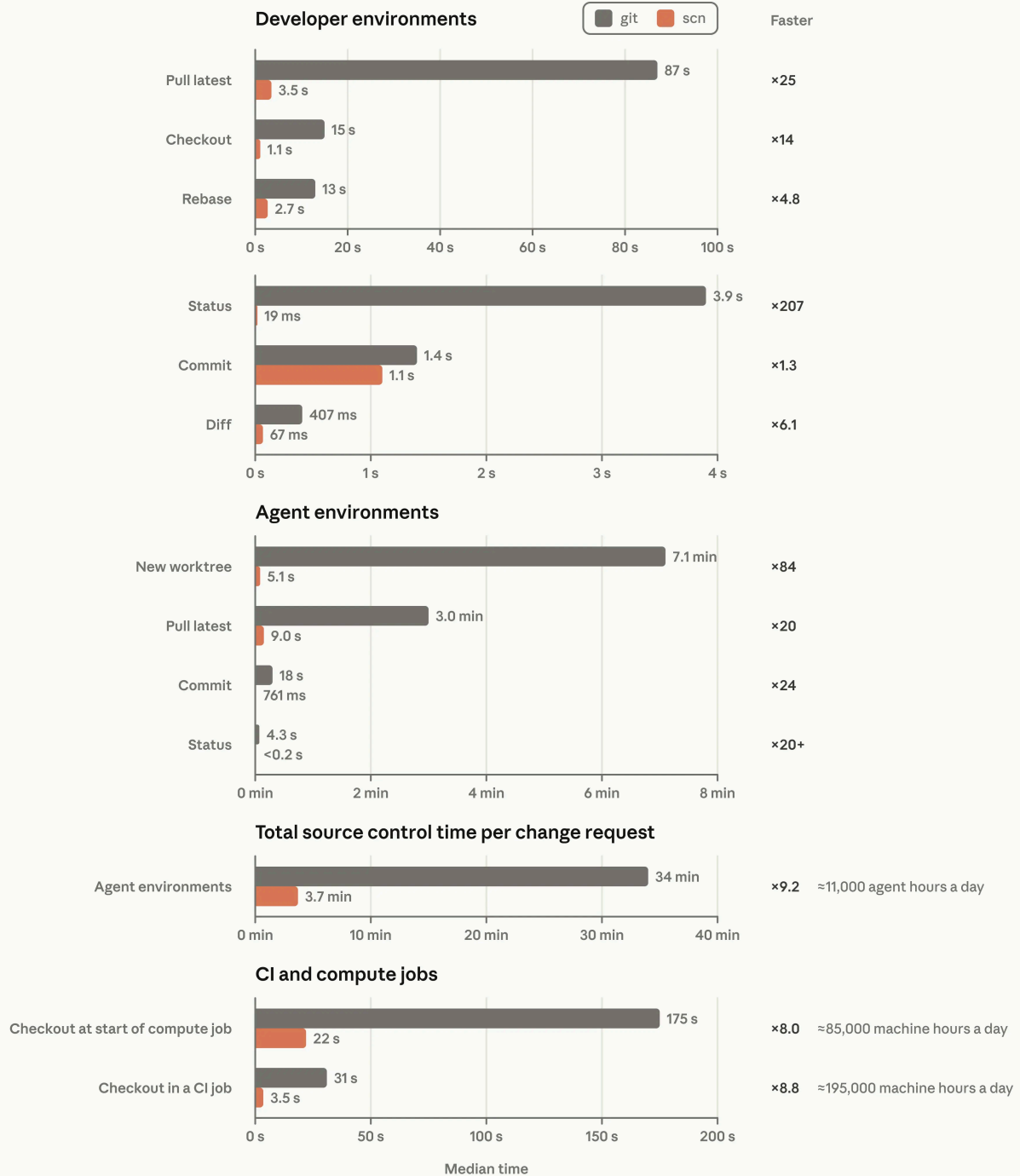
Бот делает increment от изменений

“no.” Agents will sometimes violate core system constraints to satisfy a pattern that should never have been allowed in the first place. This was particularly common in our migration. For example, there were multiple systems that tried to brute force looking up when a change merged to main by linearly reading the commit log instead of asking our code review tool. Anthropic also makes extensive use of adversarial review, and, while this is incredibly effective in many ways, it can also be myopic, so skilled engineers needed to steer Claude away from over-engineering parts of the system in response to hypothetical concerns. For example, one such review was overly defensive about trusting another internal service, which resulted in Claude adding excessive validation and complexity. Our team members still review every change request for the system, but they now examine the intent of the change as opposed to reading every line of code. ← A bot is probably interested.

We were able to substantially improve optimization and quality. By virtue of Claude’s ability to write immense amounts of code much faster than we could ourselves, Scion is among the most well-tested and well-defended codebases we have ever worked on. Even on our tight timeline, we were able to build hermetic simulations of every non-trivial infrastructure and source control interaction we’ve ever encountered, as well as fuzzing and sophisticated rate limiting. Claude also casually found significant optimizations—for example, an algorithmic optimization to ancestry queries that missed one of our indexes—that would have taken an expert days or weeks of analysis and tuning... and then did it all again the next day.

↑
Dreki bryxno bce onucano :)

Everyday source control commands on the monorepo



Observed performance improvements on native Scion repos.

We laid the groundwork for new forms of monorepo security. Managing safety, security, and compliance in a massive monorepo is as important as managing the size and scale. We built Scion with these in mind from the start. **It supports rich access control on repositories, as well as the ability to apply permissions to individual directories in order to protect the most sensitive code without losing the benefits of a monorepo.** Furthermore, the code review tooling we built can express **complex semantic constraints on approvals**, which lets us differentiate simple

things like comment-only changes from sensitive logical changes, so that human reviewers can focus on the most consequential changes.

A new level of operational excellence is possible. Our team uses Claude Tag heavily, which has helped us monitor each component of the system. For example, we created Slack channels where we tasked Claude with being obsessively focused on the health of a single component. We call these Claudes “superfans.” We have a scion-daemon superfan, a scion-cache superfan, an Antique merge queue superfan, and so on. They have access to our **telemetry and logs**, and act like an **experienced engineer** who has nothing better to do than to refresh charts and look for anything strange. We also have traditional alerts for all of our systems, but our superfans can detect subtle and novel issues that make these alerts seem one-dimensional by comparison.

A bot u эффективный SRE инженер.

We’re completely rethinking our priors. This project prompted us to take a very hard look at other workflows designed for human scale. A proper merge queue, for example, doesn’t make much sense at agent scale. What does CI even mean when hundreds or even thousands of changes can merge while tests are running? We haven’t been able to effectively replicate the industry standard of a global repository lock on the merge queue, so instead, we’ve invested in post-merge and pre-deploy automated testing. This does mean that concurrent changes can conflict, but Claude can typically fix these conflicts faster than humans can even identify them.

A bot u re-thinking u re-design процессов разработки

Contributing to the wider ecosystem

We see Scion as just one part of the answer to the scaling concerns that increasingly capable AI agents have raised. We’re sharing these details as an example of how teams can tackle similar challenges, though we don’t expect our solution, or our considerations, to suit every case.

Scion is by no means a suitable replacement for git for general use. The concepts we’ve outlined here really only make sense for truly enormous monorepos, with either tens of thousands of engineers or swarms of agents prolifically writing code. Plus, Anthropic is unique in that our ability to spend tokens for internal use far exceeds what any other business would realistically want to spend. We do this both to address our own software development needs, and to test Claude’s capabilities.

таким
dog fooding

Practically speaking, it’s difficult to offer Scion in a form that would be broadly useful to other organizations. It diverges significantly from git, meaning companies would need to migrate every piece of the development lifecycle, and it

requires its own code review, code search, and CI. Our internal versions of these components are highly bespoke, so they would be difficult for other organizations to adopt, and we worry that attempting to build a public version that tries to please everyone wouldn't actually be very useful. ← *Стандартная история с кодом, рожденным внутри систем.*

However, we anticipate that this class of problems will become more commonplace over the next six to 12 months, and it would be an inefficient use of engineering resources for every company using git to face similar challenges alone. Given that, we think the most impactful thing we can do is to share our thinking, so that other developers can apply some of the most successful parts of Scion to git without departing from the overall model. A git server doesn't have to work the same way as a git client does; novel server architectures exist that provide significant scaling improvements. Git platforms can provide comprehensive server-side RPCs like Scion, perhaps even in a standardized way across forges that the git client can learn to talk to. The negotiation protocol doesn't have to look exactly like it does today, either, and the various sparse and shallow features in git can be made to compose well.

← *Многузи, що це нагадує. Це лише OpenAI з математикою*

Lastly, we want to acknowledge that the git community has spent nearly two decades scaling the platform for millions of developers. Our contribution is far narrower: sharing our early view into how agentic software development interacts with these systems, so that we can partner with GitHub, GitLab, and the OSS community to support the changing needs of developers. We're all in it together as the software development lifecycle intersects with the AI exponential.

Acknowledgements

Written by Nick G. and the source control team: S.G., R.K., Dan C., Garrett H., Ben L., Colin M., Joe P. We're grateful to Vilas J. and Sergiu D. for supporting the source control team on this ambitious journey and Amir O. for early contributions to Antique.

Footnotes

1. Storing changed files relative to the first parent of a merge is a stylistic choice; in practice, we've found that people only care about history or blame for commits on the main line in trunk-based development in a monorepo. ↩

Get the developer newsletter