Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ **Research Article**

High-Performance Git-Backed File Delivery with Node.js: From Bitbucket to Local Cache

Vivek Koodakkara Shanmughan
Independent Researcher, USA

ARTICLE INFO

Received: 02 Jan 2026

Revised: 08 Jan 2026

ABSTRACT

Modern companies need to ship configuration files and versioned assets to their web applications on demand while maintaining all the collaborative power of distributed version control systems such as Git. This article shows how to use Node.js and smart local caching to close the divide between a Git repository, such as Bitbucket, and actual delivery of content. Having Git as the canonical store, together with a detailed caching mechanism, allows the majority of requests to be served from the local disk to minimize latency. To maintain performance, multiple validation mechanisms can ensure that the content is not stale; these include checking the commit hash or ETag. Other features include ensuring files receive atomic writes, handling errors gracefully, managing memory, implementing cache eviction policies, securing the system via credential rotation, using JSON Web Tokens for authentication, encrypting data at rest, authenticating data for integrity checking with cryptographic hashes, and monitoring and logging cache performance, hits, and system state widely. Real-world performance has been shown to increase, with file retrieval times decreasing from hundreds of milliseconds to single-digit milliseconds, and a cache hit rate above eightyfive percent. The architecture isolates upstream Git infrastructure from being overloaded and throttled with requests. It also scales to tens of thousands of requests per hour on relatively low-power hardware, which makes it possible for enterprises to adopt a pattern of version-controlled asset management that has proven to work in production.

Keywords: Git-Backed Caching, Node.js File Delivery, Distributed Cache Architecture, Content Delivery Optimization, Version-Controlled Assets

1. Introduction

Modern software systems demand the distribution of configuration files, templates and customized material to web apps and libraries. Though Git repositories like Bitbucket have good version control, collaboration and change tracking, they were not built to support an effective high volume content distribution. The major point is that any HTTP request which is made by a browser will always be redirected to the browser cache to check whether it can have a valid entry in the browser cache which can be used to complete the request. The model of the cache is a key concept that can be used to construct scalable delivery systems [1]. In the case of failure to do so, direct hits on the API of Git hosting services are fatal to their performance because requests can reach to thousands per minute. This article describes a custom-built high-performance Git-backed file delivery architecture built on Node.js and smart local caching. This architecture allows using Bitbucket as the main storage system behind Node.js as a super-fast content distribution service to give organizations production-grade version-controlled file management capabilities for their projects. Empirical studies on distributed key-value stores show that the workload characteristics can have an important impact on cache performance. For example, in a deployment of Facebook's memcached, proper sizing and eviction policy decisions can considerably increase hit ratios while also reducing the load on backends [2]. If these principles are applied to a Git-based deployment, organizations can reduce the overhead caused by file fetching from hundreds of milliseconds to single-digit milliseconds, while still protecting the Git infrastructure from excessive traffic and rate-limit issues, which could lead to service downtime.

2. Core Architecture and Design Principles

The fundamental architecture of the product is to separate version control from the application. Bitbucket is the code versioning, review, and change management system of record and works as it would in normal Git workflows. The Node.js application layer's only purpose is to return the generated page content to the user at high speed and throughput. The advantage of this technique is its implementation of HTTP caching at a relatively low level in the hierarchy. When they are cached, this allows for the browser to avoid repeated round-trips to the network, therefore speeding up rendering and lowering the cost to the server [1]. When the git-backed files are retrieved over HTTP, this results in what would have been network requests instead being satisfied from a local cache, indistinguishable from reading from disk.

The application will check the local disk cache and compare locally stored metadata to the request in order to process file requests. When this is not a falsehood, files are served in the local disk cache. When the cache is empty or outdated, the application will perform calls on Bitbucket API that causes the performance bottleneck to shift not on accesses over the network, but on accesses in the disk. Analysis of key-value store workloads during bulk loads has demonstrated that the workloads of application are usually dominated by GET operations. Moreover, data sizes that are likely to be accessed in key-value stores usually possess some type of data distribution, with a large amount of retrieved requests to small objects, typically a couple of kilobytes or less [2]. These observations are captured in the design choices made on the characteristics of the caches, and the structure of the metadata organization to ensure that it is optimized to suit typical workloads on configuration files. The design assumes eventual consistency with strong guarantees, so that the files in the content cache may be stale with respect to the Bitbucket repository. Every file in the content cache contains a metadata set, which includes the commit hash, timestamps, or ETags that allow it to correctly validate and serve it. The overhead is small compared to fetching the files. But the cache architecture must also accommodate production concurrency patterns, where bursty traffic and correlated requests can overwhelm naive implementations. Experiments with distributed caching systems have demonstrated that requests are coalesced to prevent fetching the same file several times if multiple requests for the same file occur before the file is present in the cache; thus, avoiding excessive upstream traffic for burst requests.

Separating the storage and transport of Git data allows for further optimization of each layer for its specific requirements, and complex VCS-like requirements (versioning, branches, merge conflicts, access control policies, etc.) become decoupled from the requirements of the caching layer, which deals with high-throughput and low-latency data transport as its primary requirements. This design allows developers to continue using their Git workflows and does not degrade performance because the caching layer shields the back-end Git service from direct request load. Furthermore, at least at first, cached candidates for suggestions are returned regardless of the size of the repository or commit rate.

Metric	Direct Git API Access	Cached Delivery	Improvement
Average Response Time	Network-dependent (high latency)	Disk I/O bound (minimal latency)	Response time reduction to single-digit milliseconds
Cache Hit Rate	Not applicable	Typical range after warm-up	The majority of requests are served from local storage
Concurrent	Limited by API rate limits	Node.js non-blocking	Supports thousands of

Connections		I/O capacity	simultaneous connections
Backend Load	Every request hits the Git infrastructure	Only cache misses reach upstream	Dramatic reduction in upstream API calls

Table 1: Core Architecture Performance Characteristics [3, 4]

3. Intelligent Cache Management and Invalidation

Advanced cache invalidation techniques are required for optimal caching. Some strategies overinvalidate the cache, deleting items too early, while others are under-invoking and will cache items forever. HTTP cache validation defines mechanisms for determining whether a particular response is still considered fresh under constraints defined by a cache. The HTTP cache architecture defines some rules. These rules describe how browsers and intermediate caches should read response-level cachecontrol headers. This reading tells whether a response can be cached and reused. The same principles also apply to application-level caches. In this case, the use of validation mechanisms is necessary in order to avoid sending stale data, but designing these mechanisms to be light enough not to offset the gain in cache efficiency is a challenge.

In its simplest form, the application makes a lightweight HEAD request to the Bitbucket API which returns the hash or ETag of the file without downloading it and compares it with a local metadata file to determine whether a refresh of the resource is necessary. This check has very low overhead and is virtually an HTTP conditional request, yet it is well-guaranteed to be a very fresh content. This resembles the use of ETags and Last-Modified headers by HTTP caching. When the objects have a high turnover, e.g. customer configuration bundles, proxy auto-configuration files, then the cache can be re-populated without eliciting a client request, by using low-load background refreshes. The workload tracing in very large caching systems reveal that a workload has temporal localities (bursty access to objects) and the object sizes is distributed so that most of the objects accessed in the caching system are small, which preemptive fetching is possible both in the network and in the storage front [2].

For cache invalidation, event-driven approaches can also be used. Upstream changes can be detected using Bitbucket webhooks. Webhooks can also be configured to be triggered when a particular file or directory changes. This is faster and typically uses fewer resources than polling the server for changes. A webhook-based solution to cache invalidation turns the invalidation from constant polling for the presence of a change to an event-based reaction to the existence of a change, minimizing the time taken for the change to be propagated dramatically. This prescriptive behavior is consistent with the design style of distributed systems; event-based systems are typically more loosely coupled and more scalable than request-response architectures.

Cache eviction policies must consider the access patterns to allow a cache that works well with a small amount of data to still work with a larger data set. An example of such a policy is Least Recently Used (LRU). The distribution of the sizes of the files can also be important for eviction algorithms. In production caching workloads, the sizes of cached objects usually follow some distribution, which can be exploited to improve cache performance. In addition, cache designs must consider that upstream Git repositories might become briefly unreachable. This is accounted for by a graceful degradation strategy, which favors availability in case of the unavailability of upstream Git repositories over having an up-to-date cache.

Strategy	Mechanism	Latency Impact	Use Case

Table 2: Cache Management and Invalidation Strategies [5, 6]

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ **Research Article**

4. Implementation Considerations and Best Practices

In practice, a production implementation of a system has numerous concerns beyond the core caching logic, including file organization, atomicity, error handling, and memory management. For example, the organization of files can affect performance, ease of maintenance, and mode of use: cache directories may mirror the repository directory structure, use commit hashes or branch names as directory names, or support version co-existence. Modern storage systems have very different performance characteristics, and understanding the hierarchy from memory down to hard disk makes optimized file placement/access patterns possible [1]. Future caches should also use filesystem properties to optimize performance. These properties include organizing directory lists to improve lookup times and avoiding the performance degradation caused by using a directory with too many files.

Atomic file operations, to avoid cache pollution from partial or corrupted downloads. In this pattern, files are first written to a temporary location, then checked against a checksum. When a valid file is downloaded, it is then moved to its final destination. This way, the contents of the cache are always valid, notwithstanding network outages or server/API errors when fetching content from the Internet; this ensures the invariant that anything in the cache can be served. This atomic operation is especially useful in a highly concurrent context where multiple instances of an application are trying to cache the same file, to avoid duplication of work and to only expose files that are complete and valid.

Error handling is important. Knowing that upstream Git services can go down, and that cascading failure patterns of distributed systems suggest that systems sometimes fail, software systems should limit and not increase the failures they experience as they become larger. If Bitbucket goes offline or a rate limit is reached, the cache should serve stale content without re-validation. Circuit breakers can propagate back pressure to the caching layer by refusing requests when there is a sustained high failure rate and when it is detected that a request is likely to fail. This prevents caching from creating a load on the busted upstream services, while also providing the end user with a response that their requested resource is still available.

Memory management needs to be considered as well. Disk caching is often complemented with an inmemory index of the cached files and their metadata (such as file paths, commit hashes, timestamps, and content lengths) to speed cache lookups. It allows the program to avoid disk I/O and infer cache status in the common case of files being present and current. This index may need additional constraints applied to it, so that the memory consumed does not grow indefinitely if the cache size grows beyond reasonable limits, and also eviction policies and size limits may be needed. Memory handling strategies may also take into account the access patterns of production workloads, such as when a few files have the majority of the requests. In this situation, tiered cache designs are often applied, and hot data is kept in memory, while cold data is stored on disk [2].

Component	Technique	Benefit
File Storage Organization	Hierarchical structure with commit hashes	Version coexistence and simplified rollback
Atomic Operations	Write to the temporary location, then rename	Prevention of cache corruption
Error Handling	Circuit breaker and exponential backoff	Graceful degradation during failures
Memory Management	Bounded in-memory metadata index	Fast lookups without memory exhaustion
Concurrency Control	Request coalescing for duplicate fetches	Reduced upstream load during bursts

Table 3: Implementation Best Practices [5, 6]

5. Security, Monitoring, and Operational Excellence

Security considerations comprise everything in a production Git-based file deployment workflow. If privileged-level Git-based repository access is used via Bitbucket API tokens, it is critical to securely store these tokens as environment variables, in a secret management service (such as HashiCorp Vault), in a service using the provider's secret service, and to ensure that they are not stored in code repositories, logs, or error messages. As part of least privilege, the permissions of the generated API tokens can also be scoped to allow reading files from the repository only, and nothing else.

Since authentication and authorization must occur outside of Bitbucket, the Node.js application must impose its own restrictions so that users only have access to files they are permitted to access. JSON Web Tokens (JWT) are a compact, URL-safe means of representing claims to be transferred between two parties. The token contains all the passive information about the user and permissions the application needs to make authorization decisions without having to access a database [8]. JWTbased authentication is stateless because the application can validate the tokens locally using the public keys of trusted identity providers. It is also a good approach for large-scale applications, because no network calls to authentication services are needed; cryptographic operations are sufficient for token validation. Applications should validate each token, including signature verification, expiration checking, and issuer verification (if using public key signature). Incorrect validation can lead to security vulnerabilities.

In very sensitive situations, cache contents can also be encrypted when at rest to prevent exposure of the underlying storage. Encryption overhead is generally small upon modern hardware with hardware accelerators or optimized encryption algorithms, especially for workloads that are mostly reads from the storage devices and are able to cache data on the filesystem-level to avoid decrypting the same data repeatedly. The Secure Hash Algorithm 1 (SHA-1) is used to compute the message digest. Modern implementations should use SHA-256 or higher, since SHA-1 is broken per [9]. Apart from encryption, the integrity of the filesystem can be further improved by HMAC (Hash-based Message Authentication Code) mechanisms that can detect unexpected modifications to files. The Python HMAC module has a reference implementation of a keyed digest function for authenticating messages (see [10]), which can be combined with other cryptographic primitives to provide both encryption and an authenticated hashing as a defense-in-depth strategy.

The service can be traced and monitored through the detailed logging and observation. The logs of all the cache operations must be properly structured e.g. cache hit rate, mean time of accessing the cache, count of calls on Bitbucket APIs, count of times errors are sent back, etc. This information is all employed to refine the service, solve and address the faults such that they do not impact on end users. Widely-spread logging can however be a bottleneck and what to log coupled with how close is a tradeoff. This allows recording operations in their entirety but summarizing high-frequency operations with descriptive statistics. This can optionally warn about anomalies so that problems can be corrected before affecting users. A sudden drop in cache hit rate may indicate a configuration mistake or file churn in the repository. In the Bitbucket API, an increase in error rate may indicate authentication failure or failure of an upstream service. These can lead operations to respond before a large user-facing outage occurs.

Security Layer	Implementation	Purpose
Credential Management	Environment variables or secret management systems	Protection of API tokens
Authentication	JWT-based validation	Stateless authorization decisions
Encryption at Rest	Hardware-accelerated algorithms	Content confidentiality
Integrity Verification	HMAC-based cryptographic hashing	Tamper detection

Logging	Structured log entries with sampling	Operational visibility and troubleshooting
Metrics Collection	Cache hit rates, latency, error rates	Performance monitoring and optimization
Alerting	Anomaly detection on key indicators	Proactive problem resolution

Table 4: Security and Monitoring Framework [9, 10]

Conclusion

A Git-backed file system with clever local caching implemented as a Node.js server is well-suited to organizations that require a versioned digital asset management system with the performance and capabilities of a production-grade web app. With a Git-based architecture, versioning and delivery are decoupled. Organizations can combine the benefits of a collaborative Git-based workflow (branching strategies, code reviews, and full audit trails) with low-latency response times. Because of the twolayer architecture of this solution, it can specialize the software in each layer. Git deals with controlling complex versioning and dependencies, Bitbucket is a centralized repository for managed repositories, and the Node.js caching layer can specialize in serving up the highest throughput files with as low a latency as possible. Overall, this has resulted in substantial performance improvements. File fetching has gone from network-bound times in the hundreds of milliseconds to disk-bound times that are single-digit milliseconds. The cache layer also protects the service's underlying Git infrastructure from consuming or being affected by traffic that reaches rate limit thresholds and degrades performance. This is important during peak traffic periods or when doing large batch publishing. This leads to a more complex set of concerns in caching that go beyond a naive caching implementation. Examples include atomicity handling, freshness/validity logic, error handling to gracefully service requests in the event of an upstream service failure, and memory management handling for caching thousands of items. Security features include credential management to secure access to the repository, stateless authorization decisions with JWT authentication, at-rest encryption of cache files, and integrity checks of cache files served to clients. Operational excellence features include structured logging that logs events happening during cache operations, exposing metrics for monitoring the performance of the software, and configuring alerting that triggers when a desired condition is met. The architecture's modularity and composable nature enable more complex forms, such as multi-tiered caching hierarchies, distributed cache synchronization, and caching across multiple global locations, blue-green deployment for no system downtime on updates, and gradual rollout, among others. The architectural pattern has been proven for organizations needing to manage configuration files, customer templates, or similar resources on an on-demand basis at the scale and frequency required to serve a production-facing shared service, in particular, maximizing the collaborative benefits shared by Git-based workflows, and the required enterprise scale and performance.

References

[1] Ilya Grigorik and Jeff Posnick, "Prevent unnecessary network requests with the HTTP Cache," Web.dev, 2018. [Online]. Available: https://web.dev/articles/http-cache
[2] Berk Atikoglu et al., "Workload analysis of a large-scale key-value store," ACM SIGMETRICS Performance Evaluation Review, 2012. [Online]. Available: https://dl.acm.org/doi/10.1145/2318857.2254766
[3] R. Govindan and A. Reddy, "An analysis of Internet inter-domain topology and route stability," Proceedings of INFOCOM '97 [Online]. Available: https://ieeexplore.ieee.org/document/644557
[4] Grzegorz Malewicz et al., "Pregel: A System for Large-Scale Graph Processing," SIGMOD'10, 2010. [Online]. Available: https://15799.courses.cs.cmu.edu/fall2013/static/papers/p135-malewicz.pdf

[5] Mahadev Satyanarayanan et al., "Pervasive Personal Computing in an Internet Suspend/Resume System," IEEE Internet Computing, 2007. [Online]. Available: https://www.computer.org/csdl/magazine/ic/2007/02/w2016/13rRUxNW21m

[6] Brian F. Cooper et al., "PNUTS: Yahoo!'s hosted data serving platform," Proceedings of the VLDB Endowment, 2008. [Online]. Available: https://dl.acm.org/doi/10.14778/1454159.1454167

[7] Stephen Checkoway et al., "Comprehensive Experimental Analyses of Automotive Attack Surfaces". [Online]. Available: https://www.autosec.org/pubs/cars-usenixsec2011.pdf

[8] JWT.io, "Introduction to JSON Web Tokens" [Online]. Available: https://www.jwt.io/introduction

[9] Donald E. Eastlake and Paul Jones, "US Secure Hash Algorithm 1 (SHA1)," 2001. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3174

[10] Python Software Foundation, "hmac — Keyed-Hashing for Message Authentication," 2025. [Online]. Available: https://docs.python.org/3/library/hmac.html