# Using Reinforcement Learning to Sustain the Performance of Version Control Repositories

Shane McIntosh*, Luca Milanesio†, Antonio Barone†, Jacek Centkowski†, Marcin Czech†, Fabio Ponciroli†
* Software REBELs, University of Waterloo, Canada; shane.mcintosh@uwaterloo.ca
† GerritForge Inc., Sunnyvale, CA, USA; {luca,tony,jacek,marcin,ponch}@gerritforge.com

***Abstract*—Although decentralized Version Control Systems (VCSs) like Git support several organizational structures, a central copy of the repository is typically where development activity is coalesced and where official software releases are cut. Popular practices like trunk-based development and monolithic repositories (a.k.a., "monorepos") that span entire organizations strain central repositories. Remedial actions, such as performing garbage collection routines, can backfire because they are computationally expensive and if run at an inopportune moment, may degrade repository performance or cause the host to crash.**

**In this paper, we propose a reinforcement learning agent that can take remedial actions to sustain VCS performance. Since volumes of VCS activity are needed to train the agent, we first augment the VCS to enable a greater throughput, observing that the augmented VCS outperforms the stock VCS to a large, statistically significant degree. Then, we compare the performance that a VCS can sustain when the agent is applied against a schedule-based garbage collection policy and a no-action baseline, observing 64 to 82-fold improvements in the Area Under the Curve (AUC) that plots repository performance over time. This paper takes a promising first step towards automatically sustaining VCS performance under heavy workloads.**

## I. Introduction

In most development settings, the Version Control System (VCS)—the system that tracks changes to source code and accompanying artifacts—invisibly provides powerful features. Software organizations rely on the VCS to track the state of their software, with it serving as a hub for and an archive of development activity. Decentralized VCSs (e.g., Git) provide users with powerful branch and commit manipulation features.

Although DVCSs support several organizational structures, it is typical for teams to anoint a centralized copy of the VCS as the official repository from which releases are cut. This central repository and the infrastructure on which it is hosted needs to scale to meet the needs of software organizations. The pace at which the central repository can process read and write requests constrains the agility of a software oganization.

The central repository is strained when entire organizations host all software in a monolithic repository ("monorepo"). While a monorepo provides benefits [1, 4], it stresses development tools to new limits [2]. For example, each revision of a code change hosted by Gerrit amends a commit record in a central repository. These amendments continuously change paths within the VCS history graph, making the repository organization suboptimal and degrading protocol efficiency.

To cope with these production workloads, experts, such as build, release, or devops engineers [6], monitor repository performance and react to degradation by performing remedial actions. For example, performing a garbage collection routine will reduce the disk footprint of the repository and increase performance, by traversing the VCS history graph, removing unreachable records, and consolidating loose records into larger files that are less expensive to navigate and transfer over the network; however, the remedial actions further strain the system on which the repository is hosted. If remedial actions are poorly timed, they may degrade the performance of the repository or even crash a repository host, leading to downtime that likely coincides with periods of urgent development.

In this paper, we propose an autonomous agent that can automatically sustain the performance of VCS repositories under production workloads—a problem especially acute for (but not limited to) monorepos. More specifically, we address the following two Research Questions (RQs):

**(RQ1) To what degree can we accelerate the accrual of repository activity?**
Motivation: To effectively train our agent, we must expose it to a large amount of development activity. Hence, we set out to accelerate the rate at which the VCS can process development activity by altering its workflow for write operations.
Results: When under a simulated workload of 25,000 write operations—50 parallel clients each performing 500 write operations—we find that our improvements to the VCS yield large (Cliff's $\delta > 0.474$), statistically significant improvements in processing speed (Wilcoxon signed rank test, one-tailed, paired, $\alpha = 0.05$), which grow increasingly larger over time.

**(RQ2) How well does our agent sustain repository performance under a production workload?**
Motivation: Since contextual decision making is critical for our solution, we explore *reinforcement learning* for selecting when and which remedial actions to apply. As recommended by Shrikanth and Menzies [9], before rushing to adopt complex methods, we study whether a simple approach (i.e., the Q-learning approach [3], which learns to rank actions based on stochastic states and rewards) can provide value.
Results: The agent outperforms scheduled garbage collection and no-action baselines by factors of 64- and 82-fold on an Area Under the Curve (AUC) scale that plots repository performance over time.

## II. Agent-Based Approach

To sustain the performance of VCSs, we train an autonomous agent to take remedial action. To train the agent, we apply Q-learning [3]. Below, we describe the set of agent actions (Section II-A) and the reward function (Section II-B).

### A. Agent Actions

When prompted, the agent assesses the state of the repository, e.g., recent miss rates for lookup optimizations and the quantity of loose repository records (i.e., those stored individually rather than in consolidated files) and then selects from a set of five actions.[1] First, the agent may determine that *no action* is needed. Second, the agent may *create a bitmap*, i.e., a map indicating whether each repository record can be reached from the other records stored within a subset of the repository. An up-to-date bitmap minimizes the lookup time for missing content when a read operation (e.g., `fetch`) is being performed. Third, the agent may *repack refs*, i.e., create an index where loose objects can be looked up. Fourth, the agent may *remove* `.keep` *files*, which prevent the premature removal of temporary files that are actively being used. Finally, the agent may perform a full *Garbage Collection* (GC) routine (the most expensive operation), which compresses file revisions, removes unreachable records, and (re)packs refs.

### B. Reward Calculation

After the agent selects an action, its database of rewards is updated based on the reward calculated for the most recent action. In a nutshell, the agent selects actions that have provided the maximum reward when the agent has observed the same repository state in its past. Agent behaviour is encouraged (discouraged) by providing large (small) rewards.

Agent performance depends on the definition of its reward function. Our reward comprises components that estimate the cost of the remedial action that was performed and the benefit that the action has provided. For the cost component, we use a relative scale, with a full GC routine generating no reward, no action producing the full reward, and other remedial actions providing a reward proportional to their cost with respect to a full GC routine. For the benefits component, we measure the impact of the action on the staleness of the repository bitmaps (i.e., the miss rate) and the time that is spent during the dominant (search for reuse) phase of read operations.

## III. Accelerated Processing of VCS Requests

To accelerate the accrual of repository activity, we make performance improvements to VCS tools. This is especially important for our Q-learning approach, which is known to converge slowly. We focus on Gerrit—a popular code review platform based on Git that is integral to the version control workflows of many software organizations—and JGit—a pure Java implementation of Git that is used by Gerrit. Our improvements accelerate the `receive-pack` and `upload-pack` commands, whose efficiency impacts the responsiveness and scalability of VCSs. The improvements have been uploaded to JGit,[2] Gerrit,[3] and the cached-refdb plugin of Gerrit.[4]

In JGit, we improve the routines for encoding in-memory buffers into a binary-stream (e.g., UTF-8 to binary), the I/O locking subsystem, and the concurrent usage of resources for processing packfiles and indices. Additionally, we fine-tune GC routines to enable greater use of concurrency. Similarly, in Gerrit, we enhance the performance of `receive-pack` in high-concurrency scenarios. Below, we describe the evaluation of these improvements, which we perform on a server with a Intel Xeon Gold 6438Y+ processor and 128 GB of RAM.

*(RQ1) To what degree can we accelerate the accrual of repository activity?*

RQ1: Approach: To address RQ1, we compare the performance of the improved and stock VCS when they are placed under a simulated load of 25,000 change operations that write to the repository. We distribute the load across 50 clients that each submit 500 change operations to a central VCS. The 50 clients perform change operations concurrently. For each change operation, we measure the duration of the operation and its exit code (i.e., whether the operation succeeded or failed). To account for uncertainty in performance measurements, we repeat the experiment three times. We perform the entire experiment in the two settings—once with the improvements applied and once with a stock VCS. These runs produce two sets of 75,000 rows ($25,000$ rows $\times$ 3 repetitions, each with an associated duration and exit code.

To investigate whether the differences are significant, we apply Wilcoxon signed rank tests (one-tailed, paired, $\alpha = 0.05$) to the median cross-run duration values across the improved and stock VCS samples. Moreover, we measure the effect size using Cliff's Delta ($\delta$), which is considered *negligible* when $\delta < 0.147$, *small* when $0.147 \leq \delta < 0.33$, *medium* when $0.33 \leq \delta < 0.474$, and *large* otherwise.

RQ1: Results: Figure 1 plots the median duration of the change operations across repetitions using line plots. The black line plots the improved VCS setting and the grey line plots the stock VCS setting. The lighter-shaded bands surrounding the trend lines indicate the standard deviation across the execution repetitions. The entire experiment is repeated with a sleep duration between client operations of one second (leftmost plot) and one minute (rightmost plot).

The leftmost plot of Figure 1 shows that the stock VCS struggles to handle the load generated when the sleep duration is one second. The trend for the stock VCS shows a super-linearly increasing tendency as the experiment proceeds. The shaded band around the plot also tends to expand over time, indicating that the variance in performance tends to increase as well. The trend sharply decreases as we approach the end of the experiment due to a release of concurrency pressure as client task queues empty. By comparison, the improved

[1] Selected based on our professional experience in sustaining the performance of repositories under production workloads

[2] https://eclipse.gerrithub.io/c/eclipse-jgit/jgit/+/1195454

[3] https://gerrit-review.googlesource.com/c/gerrit/+/428537

[4] https://gerrit-review.googlesource.com/c/modules/cached-refdb/+/428378

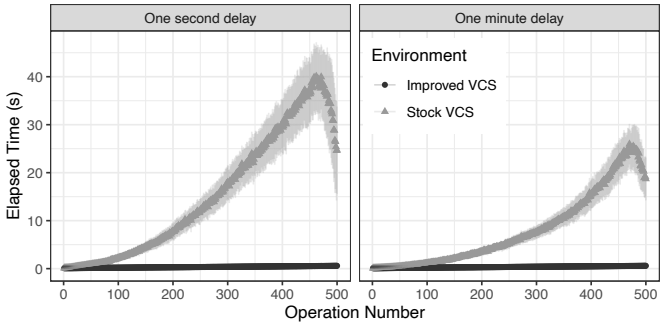


Fig. 1. The duration of change operations (in seconds) as our simulated load of 25,000 writes to the repository, i.e., 50 concurrently running clients each performing 500 serially queued write operations. The trends suggest that the improved VCS handles the simulated load substantially faster than the stock VCS, even when a delay of one minute is introduced between write operations.

VCS processes the workload efficiently, as indicated by the smooth linear trend line and tight shaded band. A Wilcoxon signed rank test comparing the median trends indicates that the differences are statistically significant ($p < 2.2 \times 10^{-16}$) and the effect size is large ($\delta = 0.9229$).

Since the confounding factor of concurrency pressure is at play, we investigate whether a longer sleep duration will impact the results by increasing the sleep duration to one minute. Comparing the stock VCS trends of the left and right plots of Figure 1, it is clear that the longer sleep duration eases the concurrency pressure. The magnitude of the stock VCS trend decreases and the breadth of the shaded band tightens when the sleep duration is set to one minute. The increased sleep duration also increases the overall duration of the experiment, which runs counter to our goal of accelerating the accrual of VCS activity. Moreover, even in this slower setting, the performance of the improved VCS remains significantly faster ($p = 2.2 \times 10^{-16}$) with a large effect size ($\delta = 0.8432$).

Figure 1 also shows that, in both sleep duration settings, the gap between the stock and improved VCS performance grows larger as time passes. Indeed, the stock VCS trends take a superlinear shape, whereas the the improved VCS trends remain consistently linear. It is possible that the improved VCS trend will curve superlinearly in a longer running experiment; however, the current results show great promise for accelerating the accrual of VCS activity for training our agent.

We do not observe any non-zero exit codes (i.e., processes that terminated erroneously) in any of the runs. This suggests that our accelerations do not introduce any obvious sources of new errors in the modified routines.

**Conclusion for RQ1**

Extending the sleep duration between operations avoids overburdening the test environment, but the VCS improvements still yield large, statistically significant improvements in processing speed, which grow larger at an accelerating pace as time passes.

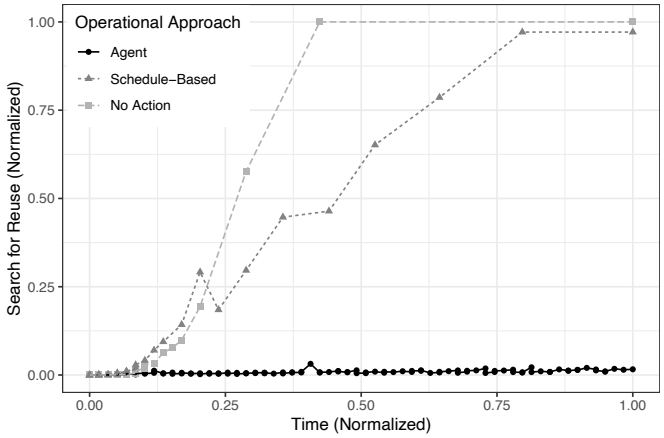


Fig. 2. A comparative overview of the performance (Y axis, lower search for reuse scores indicate better performance) of the proposed agent (black circles), schedule-based maintenance (dark grey triangles), and a baseline approach that takes no maintenance action during the experiment (light grey squares). As time progresses (X axis), the gap between the baseline and agent approaches grows, indicating that the benefits of the agent solution increases over time.

## IV. The Effectiveness of the Approach

*(RQ2) How well does our agent sustain repository performance under a production workload?*

RQ2: Approach: We evaluate the effectiveness of our proposed agent using *lift charts* (a.k.a., Alberg diagrams) [5]. First, we train our agent for two months while the target repository is under a simulated load of ten concurrent clients, which are each performing `receive-pack` (write) and `upload-pack` (read) operations. Next, we place our target repository under a similar load for the evaluation phase. For each read activity of the evaluation phase, we record: (a) the timestamp when the event occurred; and (b) a performance score. We plot each activity in the unit square space by normalizing the timestamp (X axis) by the earliest (0) and latest (1) observations, and by normalizing the performance score (Y axis) by its minimum (0) and maximum (1) values.

We plot curves for three operational scenarios. First, we plot a curve representing what happens if no maintenance is performed during the simulation (i.e., the "No Action" approach). The second curve plots what happens when we perform a garbage collection routine every five minutes (i.e., the "scheduled-based" approach). The final curve shows how our proposed learner performs (i.e., the "Agent" approach). The same (improved) VCS variant is used in all three scenarios.

To quantitatively compare the approaches, we compute the areas under their curves (AUCs). Since the plot is in the unit-square space, AUC values will range between zero and one. The smaller the AUC value, the better the performance.

RQ2: Results: Figure 2 shows the lift charts from our experiment. For the performance score, we use the time spent during the search for reuse phase of the read operation. We select this measure rather than more natural measures of performance, such as the total duration because the search for reuse phase is not influenced by external factors, such as network congestion.

Figure 2 shows that when no repository maintenance is performed or when performance is performed on a scheduled basis, read performance rapidly degrades. The no action baseline has a superlinearly growing search for reuse score until it reaches its maximum (worst) score just before the experiment has reached the halfway point. At this point, the repository host could not complete any further read operations within the experiment timeframe. Therefore, we extrapolate the no action trend to the end of the experiment.

The schedule-based solution fares substantially better than the no action approach. We observe a similar expontential growth as the no action baseline until the first garbage collection routine completes. This results in the drop in search for reuse that we observe just before the 0.25 mark of the experiment timeframe. Yet the performance again quickly begins to degrade until the next garbage collection routine completes. This time, the impact is smaller, simply slowing the growth just before the halfway mark of the experiment. The growth accelerates again before reaching its peak at around the 0.8 mark of the experiment timeframe. This marks the last read operation that could be completed within the timeframe, so we extrapolate the curve out to the end of the experiment.

Figure 2 also shows that the agent solution vastly outperforms the other approaches. First, the black trend is stable and without superlinear growth. Moreover, the normalized search for reuse score remains under 0.05. Second, a substantially larger number of read operations complete within the experiment timeframe, as can be observed by the more frequent and steady occurrence of points on the trend (circles) than the schedule-based (triangles) and no action (squares) trends.

To quantitatively compare the trends, we compute and compare the AUC values for each approach. We find that the AUC for the agent approach is 0.0088, whereas the AUC for the schedule-based and no action approaches are 0.5630 and 0.7251, respectively. On the relative scale, the agent approach is performing 64 to 82 times better on this AUC scale.

**Conclusion for RQ2**

The agent vastly outperforms the schedule-based maintenance and no action approaches, suggesting that it can sustain repository performance when repositories are put under heavy production loads.

## V. Related Work

Prior work has explored the feasibility of monolithic repository management for large software organizations. For example, Potvin and Levenberg [7] articulate the case for choosing a monolithic repository in the development setting at Google. As of 2016, the repository contained 86 TB of data spanning one billion files and 35 million commits, demonstrating that single-source model can scale to serve the needs of a large development organization. Keeping a VCS functioning under such a workload requires operators with deep expertise. Our agent-based approach aims to reduce that barrier to entry for monorepo management.

The benefits and challenges of adopting monorepos has also been studied. Through triangulation of developer perceptions with quantitative data, Jaspan *et al.* [4] observe that codebase transparency is a key benefit because it helps Google staff to discover APIs and patterns of their usage, and to propagate API changes directly to client code. Toolchain standardization was perceived as both a benefit and a challenge, since teams whose desired tools are not supported are forced to use an alternative. Brito *et al.* [1] found that research on monorepos was sparse, but through a survey of the grey literature, arrive at a list of benefits, such as improvements to work culture, and challenges, such as the required investments in tooling.

The choice of repository organization style is not a simple one. Brosse [2] shows that the tradeoffs of repository organization styles are not one that can be universally optimized. He argues that context should be considered to select well for each organization, but also argues that the primary benefit of a monorepo is likely cultural. Klein argues the case against the monorepo, explaining that for a monorepo to "pay off", the organization must combat the tendency to tightly couple components and must account for the additional "herculean" effort required to scale a VCS.[??] In this paper, we strive to reduce that VCS scalability burden by training an agent to intervene automatically when remedial actions are needed.

## VI. Threats to Validity

*Construct validity* is threatened by the measures we select for repository performance. Although incomplete, our measures dominate (read) operation duration (e.g., search for reuse time), and would provide performance uplift if improved.

*Internal validity* is threatened by the fluctuating compute load on our experimental machines. To counteract this threat, we repeat our RQ1 analysis three times (with minimal cross-run variation). Moreover, the performance gap between the agent-based approach and others in RQ2 is so large that such fluctuations are unlikely to explain the differences.

The *external validity* of our study is threatened by the simulated workload that we generate. Our results may not generalize to real-world workloads if they differ substantially. We are actively performing a more comprehensive study to evaluate the generalizability of the approach.

## VII. Future Plans

In this paper, we propose an agent-based approach to sustain VCS performance autonomously (Section II). To do so, we make improvements to VCS internals to accelerate the accrual of repository operations (Section III). We train and evaluate our agent-based solution, demonstrating its superiority with respect to naïve and state-of-the-art baselines (Section IV).

In future work, we are expanding the set of remedial actions that the agent can take (e.g., geometric repacking[5]), conducting a broader evaluation of the agent under real-world workloads, and training the agent using deep reinforcement learning [8]. Replication: Our data and scripts are publicly available.[6]

[5]https://github.blog/open-source/git/highlights-from-git-2-33/

[6]https://doi.org/10.5281/zenodo.13917389

## References


[1] G. Brito, R. Terra, and M. T. Valente, "Monorepos: A Multivocal Literature Review," in *Proc. of the Brazilian Workshop on Software Visualization, Evolution and Maintenance*, 2018, pp. 43–50.

[2] N. Brousse, "The Issue of Monorepo and Polyrepo In Large Enterprises," in *Proc. of the Int'l Conf. on the Art, Science, and Engineering of Programming*, 2019, pp. 2:1–2:4.

[3] J. Clifton and E. Laber, "Q-learning: Theory and Applications," *Annual Review of Statistics and Its Application*, vol. 7, no. 1, pp. 279–301, 2020.

[4] C. Jaspan, M. Jorde, A. Knight, C. Sadowski, E. W. Smith, C. Winter, and E. Murphy-Hill, "Advantages and Disadvantages of a Monolithic Repository: A Case Study at Google," in *Proc. of the Software Engineering in Practice track of the Int'l Conf. on Software Engineering*, 2018, pp. 225–234.

[5] N. Ohlsson and H. Alberg, "Predicting Fault-Prone Software Modules in Telephone Switches," *Transactions on Software Engineering*, vol. 22, no. 12, pp. 886–894, 1996.

[6] S. Phillips, T. Zimmermann, and C. Bird, "Understanding and Improving Software Build Teams," in *Proc. of the Int'l Conf. on Software Engineering*, 2014, pp. 735–744.

[7] R. Potvin and J. Levenberg, "Why Google Stores Billions of Lines of Code in a Single Repository," *Communications of the ACM*, vol. 59, no. 7, pp. 78–87, 2016.

[8] M. Sewak, *Deep Reinforcement Learning*. Springer, 2019.

[9] N. C. Shrikanth and T. Menzies, "Assessing the Early Bird Heuristic (for Predicting Project Quality)," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 116:1–116:39, 2023.