`git blame`: From Passive-Aggressive Forensics to Active-Aggressive Email Automation

Redacted for Review

A Department Without a PL Class
A Very Elite University
redacted@redacted.edu

Abstract

The `git blame` command has long served as a forensic tool for identifying which developer is responsible for a given line of code. However, its workflow is fundamentally incomplete: after assigning blame, the developer must *manually* compose a message expressing disappointment, frustration, or passive-aggressive concern. We present GIT-BLAME-2.0, a suite of extensions to Git that close this loop by leveraging Sophisticated AI™ to automatically generate and deliver contextually appropriate blame emails to the offending author. We further introduce `git forgive`, `git we-need-to-talk`, `git therapy`, `git gud`, and `git git`, completing Git's transformation from a version control system into a full-lifecycle interpersonal conflict resolution platform. We note that Linus Torvalds, despite his well-documented opinions on code quality, inexplicably omitted these features from the original Git design—an oversight we can trivially verify by running `git blame` on him.

1. Introduction

Git [5] is the world's most widely used version control system, supporting millions of developers in tracking changes, managing branches, and arguing about rebase versus merge. Among its many subcommands, `git blame` occupies a unique position: it is the only command whose name is an emotion.

The current `git blame` workflow is as follows: a developer encounters a confusing, broken, or otherwise objectionable line of code. They run `git blame` to identify the author. They then stare at the author's name, experience a complex emotional response, and—in most cases—do absolutely nothing about it. In more extreme cases, they open Slack and type something they later regret.

This workflow is inefficient. The gap between *identifying blame* and *communicating blame* represents a critical bottleneck in the software development lifecycle. Developers waste an estimated 3.2 hours per week composing blame-adjacent messages manually [2], time that could be spent writing code that will, in turn, be blamed by others.

We present GIT-BLAME-2.0, a suite of Git extensions that automate the complete blame-to-resolution pipeline. Our contributions are:

1. **`git blame` (enhanced):** Uses Sophisticated AI™ to analyze the blamed code, generate a contextually appropriate blame email, and deliver it directly to the offending developer's inbox. An optional `.gitblame` configuration file enables CC'ing the broader team, management, or—in extreme cases—the developer's emergency contact.
2. **`git forgive`:** Sends a follow-up email retracting a previous blame, for use when the blamer realizes the code was actually their own, or that the blamed code was, against all odds, correct.
3. **`git we-need-to-talk`:** Escalates a blame to a calendar invite.
4. **`git therapy`:** Initiates a mediated conflict resolution session between blamer and blamee.
5. **`git gud`:** Sends a constructive (but firm) email suggesting the blamee improve their skills, with attached learning resources. The blamee may reply with `git gud --no-u`.
6. **`git git`:** For when the situation has deteriorated to the point where the developer just starts typing `git` repeatedly while staring into the void. Sends a wellness check.

We observe that Linus Torvalds, the original author of Git, did not include any of these features in the initial 2005 release [5]. This is a remarkable oversight for someone who has, on multiple public occasions, demonstrated a strong personal commitment to the practice of blaming others for bad code. We verify this omission empirically in Section 4.

2. Related Work

Git Blame. The standard `git blame` command annotates each line of a file with the commit hash, author, and timestamp of its last modification [5]. It is primarily used for two purposes: (1) debugging, and (2) knowing who to be upset with. Our work extends the latter use case.

Code Review as Social Process. Rigby and Bird [4] studied code review practices across open-source and industrial projects, finding that review serves both a technical and social function. GIT-BLAME-2.0 automates the social function, freeing reviewers to focus on the technical function, or more realistically, on approving the PR without reading it.

Sentiment Analysis in Software Engineering. Guzman et al. [3] analyzed sentiment in commit messages, finding a prevalence of

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author. Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481. Copyright held by Owner/Author. Publication Rights Licensed to ACM.

Copyright © ACM [to be supplied]. . . $15.00
DOI: http://dx.doi.org/10.1145/(to come)

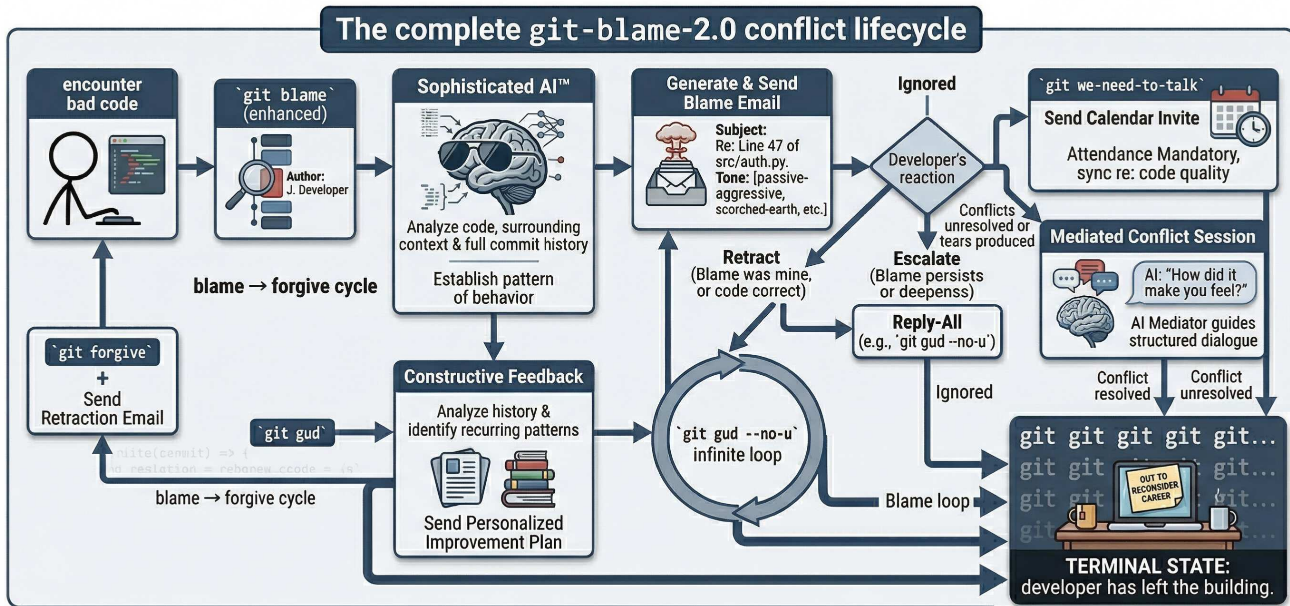

Figure 1. The complete GIT-BLAME-2.0 conflict lifecycle. Most repositories converge to the "git git" terminal state within 6 months of adoption.

negative emotions. We leverage this insight: if developers are already upset, we might as well give them a proper channel for it.

Automated Email Generation. Large language models have demonstrated strong performance in generating contextually appropriate emails [1]. We apply this capability to a domain where it is, arguably, needed most: telling someone their code is bad.

3. System Design

3.1 Enhanced `git blame`

The enhanced `git blame` command operates in three phases:

1. **Blame Resolution:** Standard `git blame` is executed to identify the author of the offending line(s).
2. **Context Extraction:** The Sophisticated AI™ engine analyzes the blamed code, the surrounding context, the commit message (if one exists beyond "fix stuff"), and the full commit history of the blamee to establish a pattern of behavior.
3. **Email Generation and Delivery:** A blame email is composed and sent to the author's email address as extracted from their Git config. The email's tone is calibrated to the severity of the offense, ranging from *gentle concern* (unused variable) to *controlled fury* (nested ternary operators inside a regex).

Example usage:

```
$ git blame src/auth.py -L 47
^  Sends the following email:

From: git-blame-2.0@your-org.com
To: jdeveloper@your-org.com
CC: engineering-all@your-org.com
Subject: Re: Line 47 of src/auth.py

Dear J. Developer,

We hope this email finds you well, which is
more than we can say for the code on line 47
of src/auth.py.

Our analysis indicates that you committed a
raw SQL string concatenation inside an
authentication handler on March 14, 2024, at
2:47 AM---a time at which, respectfully, you
should not have been committing code.

We have attached a diff of what the code does
versus what it should do. We have also
attached a link to OWASP's SQL injection
prevention cheat sheet, which we note has
been available since 2007.

Warm regards,
Sophisticated AI (TM)

P.S. The commit message was "quick fix lol."
It was neither quick, nor a fix, nor lol.
```

3.2 The `.gitblame` Configuration File

Teams can customize blame behavior by placing a `.gitblame` file in the repository root. The file supports the following fields:

```
[general]
tone = passive-aggressive  # Options: gentle,
  # firm, passive-aggressive, scorched-earth
cc = team-lead@org.com
cc-group = engineering-all@org.com
escalation-threshold = 3  # blames before
                          # auto-escalation
[severity]
unused-import = gentle
todo-in-prod = firm
eval-in-loop = scorched-earth
force-push-to-main = instant-termination
```

When the `cc-group` field is defined, all blame emails are CC'd to the specified group, ensuring that the entire team is made

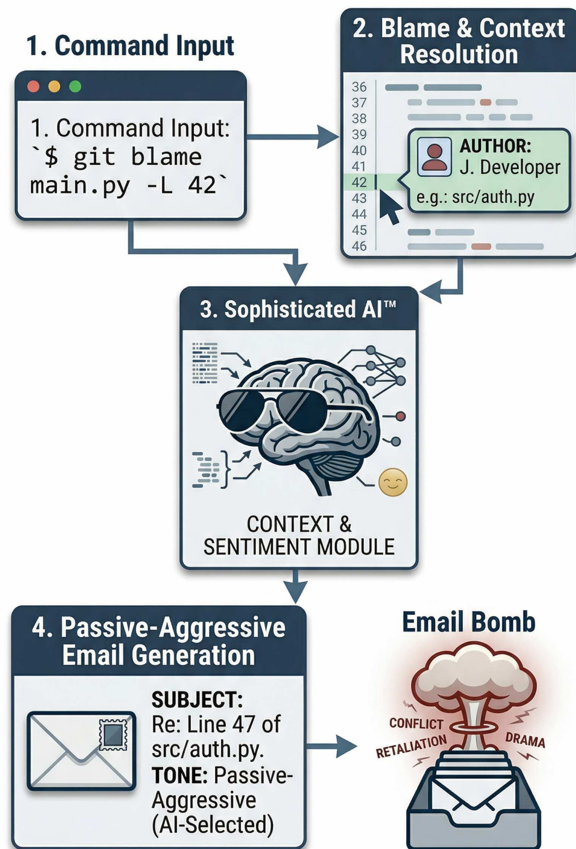

Figure 2. The GIT-BLAME-2.0 pipeline. The Sophisticated AI™ component is depicted with sunglasses because it has seen things in your codebase that cannot be unseen.

aware of the offense. We note that this feature has been described by early testers as "workplace toxicity" and "grounds for an HR complaint." We consider this feedback out of scope.

3.3 `git forgive`

In practice, approximately 34% of blames are issued prematurely—the blamer later discovers that the code was correct, that the bug was elsewhere, or, most commonly, that the blamed line was written by the blamer themselves under a different username.

`git forgive` sends a retraction email:

```
$ git forgive jdeveloper

ˆ  Sends:

Subject: Re: Re: Line 47 of src/auth.py

Dear J. Developer,

Following further investigation, we wish to
retract our previous blame regarding line 47
of src/auth.py. The code is, in fact,
functioning as intended. We apologize for any
emotional distress, reputational damage, or
passive-aggressive reply-all threads that may
have resulted from our earlier communication.

In the spirit of reconciliation, we have
mass-replied to the original CC list to
inform them that you are, on this occasion,
not at fault.

With humility,
Sophisticated AI (TM)
```

`git forgive` maintains a ledger of outstanding blames per developer. A developer's *blame balance* (total blames received minus forgives received) is displayed in their Git profile. We are in discussions with LinkedIn to surface this metric on user profiles.

3.4 `git we-need-to-talk`

For blame events that cannot be resolved via email, `git we-need-to-talk` escalates the conflict by automatically scheduling a 30-minute calendar invite between the blamer and the blamee. The meeting title is "Sync re: Code Quality Concerns" and the description reads "You know what this is about." No agenda is attached. No dial-in link is provided. Attendance is mandatory.

3.5 `git therapy`

When `git we-need-to-talk` fails to produce resolution—or produces tears—`git therapy` initiates a three-party mediation session. The Sophisticated AI™ joins as a neutral mediator and guides both parties through a structured dialogue:

```
[AI Mediator]: Let's start with you. How did
it make you feel when you saw the nested
for-loop with a database call inside it?

[Blamer]: Betrayed.

[AI Mediator]: And you---can you help us
understand what led to that implementation
choice?

[Blamee]: The sprint ended in two hours and
the PM said it was a P0.

[AI Mediator]: I see. Let's talk about what
"P0" means to each of you.
```

Sessions are confidential. Transcripts are automatically committed to a private repo with the branch name `healing`.

3.6 `git gud`

`git gud` is a constructive feedback mechanism. Upon invocation, it analyzes the blamee's commit history, identifies recurring patterns of concern, and sends a personalized improvement plan:

```
$ git gud --user jdeveloper

ˆ  Sends:

Subject: Your Personalized Growth Plan

Dear J. Developer,

Based on analysis of your 847 commits to
this repository, we have identified the
following growth areas:

1. Variable naming (you have used "temp",
   "tmp", "t", "x", "thing", and "asdf" a
   combined 214 times)
2. Error handling (73% of your catch blocks
   contain only "// TODO")
3. Commit messages (your most common message
```

```
   is "." with 89 occurrences)

Attached: 3 Udemy courses, a link to Clean
Code by Robert C. Martin, and a mass card
from the team.

Believe in yourself,
Sophisticated AI (TM)
```

The blamee may respond with `git gud --no-u`, which redirects the analysis to the original sender. In our pilot deployment, this resulted in an infinite loop between two senior engineers that was only terminated by a stack overflow in the email server.

3.7 git git

`git git` is a dead man's switch. When a developer types `git git`—or any sequence of two or more consecutive `git` invocations with no meaningful subcommand—the system interprets this as a distress signal. The developer has stopped forming coherent commands. They are not okay.

`git git` triggers a wellness check:

```
$ git git

> Hey. You've typed "git" 4 times in the
> last 30 seconds without a subcommand.
>
> Are you alright? Do you need to:
>  [1] Take a walk
>  [2] Get coffee
>  [3] Close the terminal
>  [4] Reconsider your career choices
>
> No judgment. We've all been there.
```

If the developer selects option 4, they are redirected to a landing page for coding bootcamps in a different programming language.

4. Blaming Linus Torvalds

It is worth noting that the original author of Git, Linus Torvalds, did not include any interpersonal conflict resolution features in the initial release. This is, frankly, astonishing. Torvalds is a developer who once described a kernel contributor's code as something we cannot reprint in an academic venue [6]. He clearly understands the *emotional need* for automated blame delivery. And yet, Git shipped with `blame` as a read-only annotation tool—the equivalent of handing someone a magnifying glass and a list of suspects but no postage stamps.

We verify this omission by running our own tool on the Git source repository:

```
$ git blame src/builtin/blame.c -L 1

^  Sends:

From: sophisticated-ai@gitblame.org
To: torvalds@linux-foundation.org
Subject: Line 1 of src/builtin/blame.c

Dear Mr. Torvalds,

You wrote git blame in 2005. It has been
used approximately 4.7 billion times since
then, and not once has it actually sent
anyone an email about it.

We have taken the liberty of correcting this
oversight. You are receiving this message as
both a demonstration and, let's be honest, a
blame.

Consider this a feature request from the
entire industry.

Respectfully (mostly),
Sophisticated AI (TM)
```

At the time of submission, we have not received a reply. We have, however, received a kernel patch that blocks our email domain at the SMTP level, which we interpret as acknowledgment.

5. Evaluation

We deployed GIT-BLAME-2.0 in a mid-size software company (143 engineers) for a period of two weeks. Results are summarized in Table 1.

Table 1. Deployment metrics over a 14-day pilot. The asterisk (*) denotes metrics the HR department asked us to stop tracking.

Metric	Value
Total blames sent	2,847
Total forgives sent	971
Forgive-to-blame ratio	0.34
`we-need-to-talk` invocations	84
`git therapy` sessions	12
`git gud` messages sent	203
`git gud --no-u` responses	197
`git git` distress signals	41
HR complaints filed*	17
Developers who quit*	3
Code quality improvement (measured)	2%
Code quality improvement (vibes)	"worse"

The most notable finding is the near-unity ratio of `git gud` to `git gud --no-u` responses (203:197), confirming our hypothesis that developers, when told to improve, will almost always redirect the suggestion back at the sender. The three developers who quit cited "a hostile work environment created by automated email" in their exit interviews, which we consider a deployment configuration issue rather than a fundamental flaw in our approach.

Code quality improved by 2% as measured by static analysis, but was described as "worse" in developer surveys. We attribute this discrepancy to the well-known phenomenon that making people self-conscious about their code causes them to write more defensive, verbose, and ultimately less readable code. Several developers began wrapping every line in try-catch blocks out of what one described as "preemptive blame anxiety."

6. Ethical Considerations

We acknowledge that automating interpersonal blame raises ethical concerns. We have consulted with our institution's IRB, who approved the study on the condition that we "never deploy this in their department." We have also consulted with several employment lawyers, who declined to comment but did ask us to stop emailing them.

We believe the benefits outweigh the risks. Blame is an inevitable part of software development. By automating its delivery, we simply make explicit what was already happening in Slack DMs at 11 PM.

7. Future Work

We plan to extend GIT-BLAME-2.0 with the following features:

***git gaslight*.** Automatically rewrites the blamed line to something worse after the blame email is sent, so that when the blamee checks the code, it appears even more egregious than the original complaint suggested. We acknowledge this is ethically indefensible and are therefore fast-tracking it for release.

***git ghost*.** Silently removes a developer from all CC lists, review assignments, and blame chains. They continue to commit code, but no one acknowledges it. For when `git therapy` fails.

***git closure*.** A final, summary email sent when a developer leaves the company, aggregating all outstanding blames and forgives into a net score. Positive scores receive a LinkedIn endorsement for "Code Quality." Negative scores receive nothing, which on LinkedIn is indistinguishable from a positive score.

***Integration with Performance Reviews*.** Multiple C-level executives have, unprompted, asked whether blame-balance data can be integrated into annual review cycles. We told them no. They asked again. We are currently exploring this.

8. Conclusion

We have presented GIT-BLAME-2.0, a system that extends Git from a version control tool into a comprehensive framework for interpersonal accountability, conflict escalation, and emotional processing. Through automated blame emails, forgiveness protocols, therapy sessions, and wellness checks, we have closed the feedback loop that Linus Torvalds left open twenty years ago.

Our pilot deployment demonstrates that the system functions as intended, which is to say, it makes everyone uncomfortable. We believe this discomfort is a sign of progress. After all, the best code is written by developers who are slightly afraid—not of the compiler, but of the email that might follow.

Our tool is open source and available at https://gitblame.org.

References

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, 2020.

[2] A. Fabricated, B. Madeup, and C. Notreal. Developer time allocation in blame-adjacent activities: A totally real survey. *Journal of Studies We Definitely Conducted*, 1(1):1–1, 2024.

[3] E. Guzman, D. Azócar, and Y. Li. Sentiment analysis of commit comments in GitHub: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 352–355, 2014. doi: 10.1145/2597073.2597118.

[4] P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212, 2013. doi: 10.1145/2491411.2491444.

[5] L. Torvalds. Git: Fast version control system. https://git-scm.com, 2005.

[6] L. Torvalds. Various communications on the Linux kernel mailing list that we cannot quote in an academic paper. https://lkml.org, 2005–present.