A visual guide to

Line Impact

And how fusing commits together reduces tech debt & code review time
GitClear provides an engineering insights tool built **to measurably increase the output of a developer team**. We do it by identifying which directories harbor tech debt, which commits should be reviewed, and who are the experts?

We seek to help developers enjoy reading code -- like it’s a good book. By making code easier to read, we help them reduce (or even prevent) the tech debt that saps developer enthusiasm on so many large projects.

GitClear went live to customers in 2018 and has been used by hundreds of teams to improve their code quality & velocity.
A development team might have 100+ commits made across their git repos every day. Nobody can keep track of everything. The average repo in 2020 is a black box that periodically kicks out PRs when it’s working right.

At best, a couple developers know how the big picture connects.
Critical Moments Lost in the Jumble

A new bug is being introduced
A developer is refactoring legacy code that will prevent tremendous headaches down the road
A junior developer is creating tech debt by duplicating an existing utility function

When you’re lucky, you find out about these issues before your customers do. But just as often, critical moments happen and nobody knows about it until weeks or months later, if at all.
Where Do We Look for Critical Commits?

Certainly not in the commit list
You’re not going to catch critical moments in a flat list of hundreds of commits, separated by repo. Nobody in their right mind will use this to browse commit activity.
Where Do We Look for Critical Commits?

Not in PRs

PRs are *supposed* to clean up the mess that happens in development, but you’ve seen how that goes...

- **The change is too overwhelming.** Unless your team always keeps its PRs below 20 commits, they become overwhelming and are glossed over.
- **Bugs and tech debt abound by the time PR stage reached.** Especially for new team members or Junior Developers, when weeks may pass between PRs.
- **The top people don’t have time to read them.** Senior Architects are too busy to read PRs. If it’s small, expect a quick glance. If it’s more than 20 commits, you don’t want to know...
- **On small teams, much work happens outside of the PR process.** Such work usually gets reviewed by no one.

![Example PR](image-url)
GitClear Opening the “Repo Black Box”

Let’s zoom into a single commit to see how GitClear’s rich commit parsing engine uses Line Impact to simplify even the most intricate commit activity
When current git tools interpret a commit, they relate a list of **green** and **red** changes with an occasional dash of **gray**.

By pretending every change is binary, other git tools put the onus on code reviewers to piece together the real story. Developers must scour a PR to confirm whether an apparent block of "added code" is truly new, or just some legacy code moved in from another file?

Since 30% of all line changes are moved code, the work of reviewing binary changes adds up quick.
GitClear: Commit.rich_parse()

Meanwhile, GitClear recognizes...

**Commit ABCD**

File 1

- Update
- Update
- Delete
- Update
- NO OP
- Keyword
- + Add
- ▶️ Update

File 2

+ Add
+ Add
- ▶️ Update
NO OP
- Move
- ▶️ Update
- Copy/Paste
- Delete

RICH PARSING ENGINE PUTS CHANGE IN CONTEXT, LIKE A DEVELOPER

The commit parsing engine behind Line Impact understands code changes in the terms a Senior Developer would use. This helps us understand and condense code changes.
Here are seven types of operations we recognize in commits. Each operation is accompanied by a screenshot of how the operation looks in the GitClear diff viewer.

1. **Addition**

   ![Addition Example]

   Each added line of code counts for up to 10 points.

2. **Deletion**

   ![Deletion Example]

   Each deleted line of code can count for up to 25 points. By default, Line Impact prizes code deletion most of all, for its role in reducing long-term tech debt.

3. **Move**

   ![Move Example]

   Moved code (about 30% of all changed lines) is assigned no Line Impact.

4. **No-ops**

   ![No-op Example]

   One of the most common types of code change is the "no-op." This encompasses all changes to white space, blank lines added, and lines whose only change was their line number.
Operations Recognized by Line Impact

5. Update

When a line changes in part, we consider this an "update." Updates can count for up to 10 points.

6. Find & replace

When a developer applies the same change to several lines en masse, this is detected as "Find & replace." Such lines are worth up to 3 points.

7. Copy/Paste

When a developer repeatedly adds ("pastes") the same line or block in multiple locations, across one or more commits. Copy/pasted code is assigned no Line Impact for its role in creating tech debt.
Technical Details

Language Idioms Recognized by Line Impact

Assuming you’re using one of the 40 programming languages (including every modern language), Line Impact will know a few more tricks out of the box. Below, a subset of language-specific idioms we recognize.

1. Keywords
   Added keywords have no value
   ![Image](image1)

   About 10% of lines changes are language keywords: these are transparent to Line Impact.

2. Comments
   New commenting worth about a point
   ![Image](image2)

   Comments are assigned negligible Line Impact.

3. Multiline statements
   Multi-line declaration, value assigned to first line
   ![Image](image3)

   Multi-line statements have their value assigned to the first line.

4. Include statements
   Include statements have negligible value
   ![Image](image4)

   “Include statements” can comprise up to 5% of changed lines in React repos. Line Impact treats them as trivial changes.
Once our commit parse is complete, we assign a provisional score based on the estimated cognitive energy needed to produce the changes in the commit [1].

Scores range from 0-30 per line, with the highest scores reserved for updates and deletions to legacy code, where tech debt tends to accumulate and only experts dare tread.

Changes like no-ops, moved code, and copy/paste get no Line Impact. Most non-trivial changes get 5-10 per changed line. New code earns little value since it is prone to churn and begets maintainability costs.

[1] https://www.gitclear.com/line_impact_factors
GitClear rearranges the flat list of commits you’re used to into a colorful, 2d browsable map of commits. Commits are grouped by issue and condensed to hide irrelevant changes. Learn more about Commit Activity Browser.
Commit Groups Expose Critical Moments

What Happens by Fusing Commits **ABCD** + **EFGH**?

In the real world, commits don't happen in isolation. Line Impact is built to get more accurate as more commits are added.

By fusing together related commits into Commit Groups, we cancel out noise from churn and get the ground truth about what precisely changed over the course of an issue’s implementation.

Fusing commits into commit groups is key to spotting bugs before they reach production.
Technical Details

Canceling the Noise

Commit Group: ABCD + EFGH

Commit ABCD
File 1
- Add
- Update
- Delete

File 2
- Add
- Update
- Move
- Delete

Commit EFGH
File 1
- Delete
- Update

File 2
- Delete
- Update
- Move
- Delete

WORK RENDERED OBSOLETE by Commit EFGH

100 - 50 (obsolete) Line Impact = 50 New Impact

WORK RENDERED OBSOLETE by Commit ABCD

30 - 10 (obsolete) Line Impact = 20 New Impact

+ 50 Line Impact

+ 20 Line Impact
Technical Details
Making a Commit Group

Commits **ABCD** and **EFGH**

- **File 1**
  - Delete
  - Update
  - Delete
  - Update
  - Update
  - Update
  - Delete
  - Update
  - Update
  - Delete

- **File 2**
  - Update
  - Move
  - Update
  - Delete

**+130 Line Impact**

Commit Group **ABCD + EFGH**

- **File 1**
  - Delete
  - Delete
  - Update
  - Delete
  - Update
  - Delete
  - Update

- **File 2**
  - Update
  - Move
  - Update
  - Delete

**+70 Line Impact**

**40% Less Code to Review**

**Commits to Coalesce**

**Reduces Value of CHURN LINES**
The Difference is (Git)Clear

Without GitClear:
- Bugs are created and nobody knows about it
- Developers interrupted to get updates
- Junior Developers repeat same mistakes

With GitClear:
★ Code and Jira together from first commit
★ Real-time developer status, no interruption
★ Team crafts more durable code together
Hundreds of teams trust GitClear to make sense of the activity in their repo. Our customers are data-oriented, nimble teams, who use GitClear to outpace their larger competitors.
## Leading Developer Measurement & Code Review Tools

<table>
<thead>
<tr>
<th>Product summary</th>
<th>FLOW</th>
<th>GitClear</th>
<th>Pinpoint</th>
<th>CODE CLIMATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product summary</td>
<td>“Ship faster because you know more. Not because your team's rushing.”</td>
<td>“Next-level software developer metrics powered by the best code review tool.”</td>
<td>“Streamline how you build software. Focus on the things that matter. Less tabs, more work.”</td>
<td>“Actionable metrics for engineering leaders. Turn data into insights you can lead with.”</td>
</tr>
<tr>
<td>3 developers monthly cost</td>
<td>$126</td>
<td>$27</td>
<td>$50</td>
<td>$112</td>
</tr>
<tr>
<td>10 developers monthly cost</td>
<td>$416</td>
<td>$90</td>
<td>$100</td>
<td>$374</td>
</tr>
<tr>
<td>50 developers monthly cost</td>
<td>$2,079</td>
<td>$450</td>
<td>$500</td>
<td>$1,871</td>
</tr>
<tr>
<td>100 developers monthly cost</td>
<td>$4,158</td>
<td>$900</td>
<td>$1,000</td>
<td>$3,742</td>
</tr>
<tr>
<td>Price per dev/month (baseline)</td>
<td>$42</td>
<td>$9</td>
<td>$10</td>
<td>$37</td>
</tr>
<tr>
<td>Price per dev/month (enterprise)</td>
<td>$50</td>
<td>$29</td>
<td>$25</td>
<td>(Too much to publish)</td>
</tr>
</tbody>
</table>

Last updated September 2020
Bonus Content

Encore reports for overachievers
Historic Stats: Long-Term Line Impact

Historic stats show how temporal factors impact developer output
Issue stats make clear when bugs increase, or when the team is working on undocumented work.
### Committer Performance by Domain

<table>
<thead>
<tr>
<th>Domain</th>
<th>Most Prolific</th>
<th>Velocity</th>
<th>Per Month</th>
<th>Similar Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android</td>
<td>JinsukKim</td>
<td>9</td>
<td>1050</td>
<td>redartwichik (997)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ted Choc</td>
</tr>
<tr>
<td>Angular</td>
<td>clydin</td>
<td>20</td>
<td>1439</td>
<td>Alan Aghius (1137)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Sachin Grover (487)</td>
</tr>
<tr>
<td>Autogenerated</td>
<td>claudiahdz</td>
<td>65</td>
<td>2</td>
<td>Kasper Timm Hansen (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Go Xiang Tan (0)</td>
</tr>
<tr>
<td>C</td>
<td>Gleb Smiروف</td>
<td>58</td>
<td>2726</td>
<td>Tom Lane (2491)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Arthur Heymann (3139)</td>
</tr>
<tr>
<td>C#</td>
<td>Michael Sawczyn</td>
<td>33</td>
<td>2752</td>
<td>soraryu (413)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>hoch (187)</td>
</tr>
<tr>
<td>C++</td>
<td>Ivan</td>
<td>47</td>
<td>3683</td>
<td>Eladhaj (2294)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Alexey Milevidov (2282)</td>
</tr>
<tr>
<td>Configuration</td>
<td>Roberto Carrillo</td>
<td>22</td>
<td>1521</td>
<td>anshulbehel (1211)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Sumit Jaiswal (829)</td>
</tr>
</tbody>
</table>

A customer favorite, our Domain Experts report uses git data to identify who in the company has the most experience writing every type of code across your projects.

These people can form the interview team if you need to hire more experts. They’re also prime candidates if you need to mentor aspiring Junior Developers.

Actual developer data, aggregated across 20 of the largest open source repos over past 12 months.