Do Fault Fixes Cluster by Functionality? Evidence from buildbot and Apache Apex Malhar

Nov 8, 2026 · @Someone

Abstract

Files that implement related functionality share fault-proneness only modestly: cluster membership explains about 6–9% of the variation in faults fixed per line at the subsystem level, rising to about 31% for small groups of closely related files.

We studied two open-source systems: buildbot, a continuous-integration framework (774 Python files, 2018), and Apache Apex Malhar, a stream-processing operator library (1,985 Java files). Each file was given a natural-language description of the functionality it implements and a list of the other files whose functions it calls. These two signals were combined into a feature vector per file and clustered hierarchically into three levels, which were then named.

The clusters are coherent with the call structure: 55% (buildbot) and 71% (Apex Malhar) of call links stay within a top-level cluster, against 10–11% under random assignment. Faults fixed per line (FpLOC) then cluster significantly at every level in both systems (permutation $p < 0.01$), and the effect is not explained by file size or by test versus non-test files. But the subsystem a file belongs to says little about its fault-proneness; most variation is between individual files.

The main threats are short and sparse fault histories, fixes that touch several related files at once, file age, and the accuracy of the descriptions and call resolution. We outline studies, chiefly commit-level and time-aware analyses, that would separate these explanations.

1 Introduction

We ask whether files with many fixed faults sit together when files are grouped by the functionality they implement and use, and whether files with few fixed faults do the same.

Fault studies usually treat files as independent units, or group them by directory. Directories reflect packaging decisions, which only partly follow functionality. If fault-proneness is a property of a functional area rather than of individual files, then effort for

testing and review could be directed at areas. If it is mostly a property of individual files, area-level predictors will add little.

This paper makes four contributions:

1. A procedure for describing every file in a codebase by the functionality it implements (a short natural-language description) and the functionality it uses (the files it calls).
2. A hierarchical clustering of files built from those two signals, validated against the call structure.
3. Tests of whether fault-prone and fault-free files are concentrated in the same clusters, at three levels of granularity, in a Python and a Java system.
4. An account of the factors that limit these results, and of the studies needed to go further.

2 Projects and data

The two systems differ in language, size and, above all, in how much fault history they carry.

	buildbot	Apache Apex Malhar
Language	Python	Java
Snapshot	2018 (commit 62ef53c)	commit 0d98d05
Source files analysed	774	1,985
Files matched to fault data	743	1,985
Lines of code in matched files	175,658	302,364
Faults fixed in matched files	6,168	230
Files with no fault fixed	8%	90%
Faults fixed per 1,000 lines	35.1	0.76

Fault data were supplied as one row per file, giving its lines of code (LOC), the number of commits touching it (AllFreq) and the number of fault-fixing commits (Freq). The measure studied is faults fixed per line:

```
\mathrm{FpLOC} = \frac{\mathrm{Freq}}{\mathrm{LOC}}
```

In the first version of the buildbot data the FpLOC column held Freq divided by the project-wide total of fault fixes, a scaled fault count rather than a rate. This was detected because the column matched Freq/8,327 exactly, and corrected data were then supplied.

For Apex Malhar, FpLOC was computed from Freq and LOC.

The 31 buildbot files without fault data are almost all empty `__init__.py` package markers. In Apex Malhar most files have only two commits in total (median AllFreq = 2), and 162 of the 192 faulty files have exactly one fix. The Apex Malhar history is therefore short and sparse, which shapes both the tests used and the conclusions drawn.

3 Describing each file

Each file was described in two ways: a short text saying what functionality it implements, and the set of other project files containing functions it calls.

3.1 Functionality descriptions

Descriptions were written by a large language model (Claude) working from a structural summary of each file, with the source available for reference.

1. A script parsed every file and produced a summary: path, line count, the module or class documentation comment, and the declared types and functions. For Python this used the `ast` module (classes with their base classes and methods, top-level functions). For Java it used pattern matching on the source (class Javadoc, `class` / `interface` / `enum` declarations with `extends` / `implements`, method names).
2. The summaries were split into batches of 129 (buildbot) or 166 (Apex Malhar) files and given to parallel model instances. Each instance read the source whenever the summary was not enough to describe a file with confidence.
3. Each description was limited to one or two sentences (about 45 words), in the vocabulary of the application domain. Conventions covered special cases: test files say what they test, test helpers say what they fake, schema migrations say what they change, and empty package files are labelled as such.
4. Completeness was checked mechanically (one description per file) and quality by sampling against the source. One vague description out of the sample was rewritten.

The resulting texts are specific, for example: *Kudu input operator that repeatedly scans a table with a SQL template, stepping a bound column from a seed by a configured increment each iteration.*

3.2 Call links in Python (buildbot)

Python is dynamically typed, so a call's target was resolved by a purpose-written static analyser built on the `ast` module. Only calls made inside function or method bodies were counted, as the research question concerns functions.

A call was resolved to a **definite** target through:

- imports, including relative imports, `from ... import *`, and names re-exported by package `__init__` files;
- attribute access on imported modules and classes;
- the class hierarchy, for `self.m()`, `cls.m()`, `super().m()` and `ClassName.m()`, following base classes across files;
- constructors, counted as calls to the file defining the `__init__` method found along the class hierarchy;
- simple type inference: a local variable or `self` attribute assigned from a constructor call takes that class as its type.

Calls on objects of unknown type were resolved, as **probable** targets, by two heuristics. The first used buildbot conventions (`master.db.<component>` maps to `db/<component>.py`; `master.data.updates`, `master.data` and `master.mq` map to their connector modules). The second accepted a method name defined in exactly one project file, excluding names of built-in, standard-library, Twisted and SQLAlchemy methods; production files were never linked to test code this way.

This gave 1,883 definite and 380 probable cross-file links; 577 of the 774 files have at least one definite link.

3.3 Call links in Java (Apex Malhar)

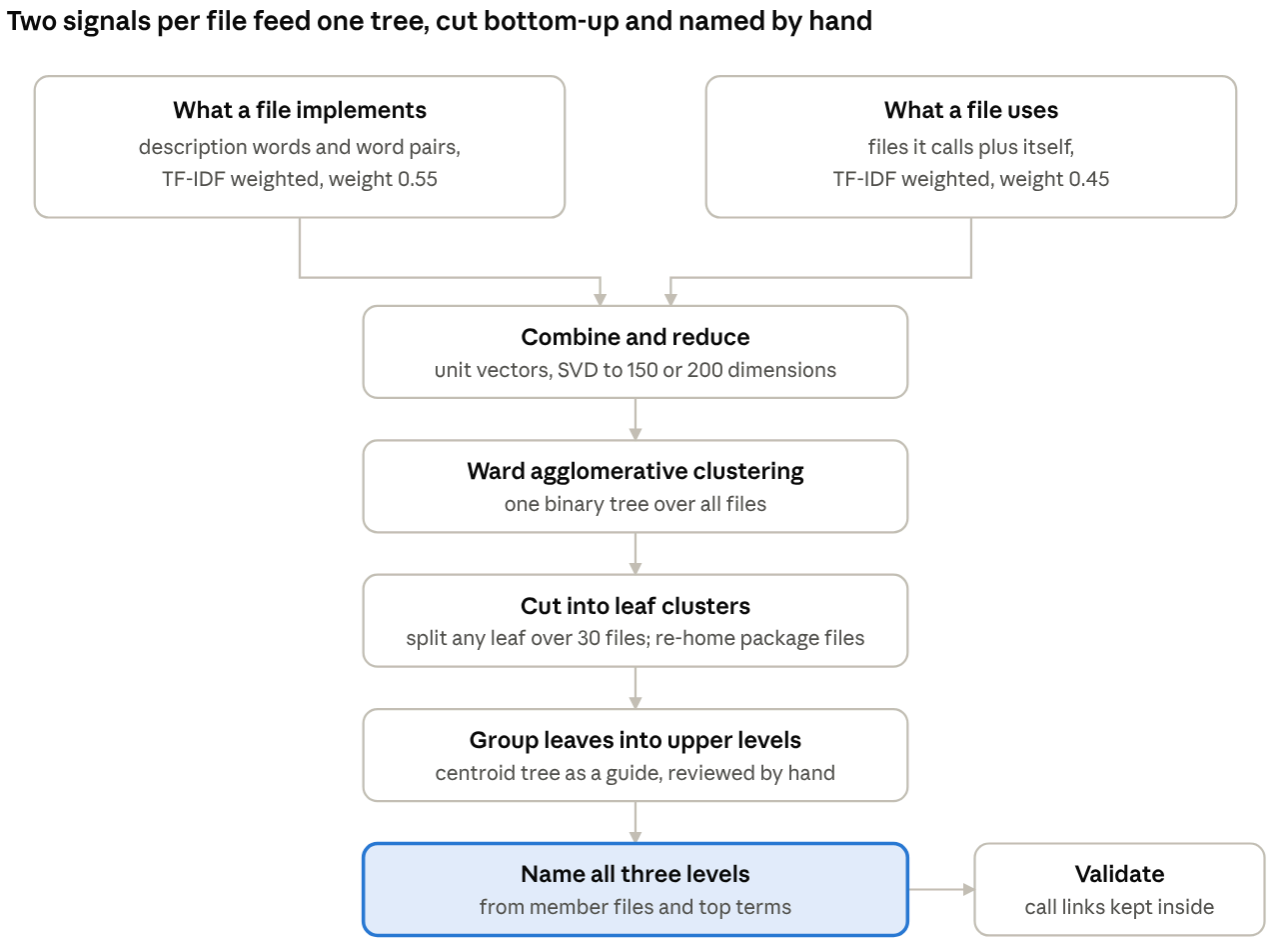
For Java the compiler itself resolved calls. A small tool used the `javac` compiler API (JDK 21) to parse and attribute each of the 45 Maven modules in turn. The other modules' sources were placed on the source path, so that calls across modules resolved to the right files.

- Every method call, constructor call (`new`) and method reference inside a method, constructor or lambda body was resolved to the declaring element.
- The callee file is the source file of the top-level class enclosing that element. Implicit default constructors were skipped.
- The static target was recorded: a call through an interface counts as a call to the interface's file, not to its implementations.
- Third-party libraries (Apex core, Hadoop and others) were not in the archive. The compiler was configured to continue attribution despite the resulting errors; calls whose receiver type depends on a missing library cannot be resolved.

In the largest module, 70% of call sites resolved. Calls to the JDK and to other libraries were discarded, leaving 3,575 cross-file links; 1,252 of the 1,985 files call at least one other project file.

4 Hierarchical clustering

Files were clustered by agglomerative (Ward) clustering on a combined text-and-call feature vector, cut bottom-up into leaves, grouped into two upper levels and named.



clustering pipeline · 2 inputs, 6 steps

4.1 Features

- **Implements.** Each description became a TF-IDF vector over words and adjacent word pairs that occur in at least two descriptions, with sublinear term frequency. English stop words and words carrying no functional meaning (test, class, file, helper, the project name) were removed.
- **Uses.** Each file became a bag of tokens: the files it calls, plus its own path counted twice. Including the file itself gives a file and its callers a shared feature, so a test file lands beside the code it tests. TF-IDF weighting down-weights callees that almost every file uses, such as utility modules. In buildbot, probable links were counted once against two for the file itself.

Both vectors were scaled to unit length and concatenated with weights, so that the cosine similarity of two files is a weighted sum of their text and call similarities:

```
\mathrm{sim}(a,b) = 0.55\,\cos(t_a,t_b) + 0.45\,\cos(u_a,u_b)
```

The combined vectors were reduced by truncated singular value decomposition to 150 (buildbot) or 200 (Apex Malhar) dimensions and normalised again.

4.2 Building the tree

Ward's method merges, at each step, the two clusters whose union least increases the within-cluster sum of squares. On unit vectors the Euclidean distance it uses is a monotone function of cosine distance, so it groups files by the similarity above.

Cutting the tree from the top produced unbalanced groups: in buildbot, two catch-all clusters of 295 and 162 files. The hierarchy was therefore built bottom-up:

1. The tree was cut into 70 (buildbot) or 110 (Apex Malhar) leaf clusters. Any leaf with more than 30 files was split by re-clustering its members, giving 74 and 132 leaves.
2. Empty package files (`__init__.py`, `package-info.java`) clustered together on their similar descriptions. Each was moved to the leaf holding most of its sibling files (78 files in Apex Malhar).
3. Leaf centroids were clustered by average linkage on cosine distance, and the result cut at about 10 and 25 groups as a guide.
4. The guide was reviewed against each leaf's member files. Leaves the centroid tree had misplaced, such as generic utilities grouped with reporters, were moved by hand.
5. Every cluster at every level was named from its member files and its highest-weighted terms.

The final hierarchies have 10, 25 and 70 clusters (buildbot) and 12, 33 and 126 clusters (Apex Malhar) at levels 1 to 3.

4.3 Validation against the call structure

A hierarchy that reflects functionality should keep calls inside clusters. For each level we computed the share of call links whose two files share a cluster. We compared it with the mean share over 200 random relabellings that keep cluster sizes fixed.

5 Results

Fault-prone and fault-free files are concentrated in the same clusters far more than chance allows, but the effect is weak at the subsystem level and grows as clusters get smaller.

5.1 The clusters follow the call structure

The top levels are recognisable subsystems. In buildbot they are build execution, scheduling and change detection, the database, the Data API and messaging, reporting, workers, the web UI, configuration and tools, core infrastructure, and integration tests. In Apex Malhar they include windowing and the stream API, state management, computational operators, database connectors, messaging connectors, file-system I/O, app data, and example applications.

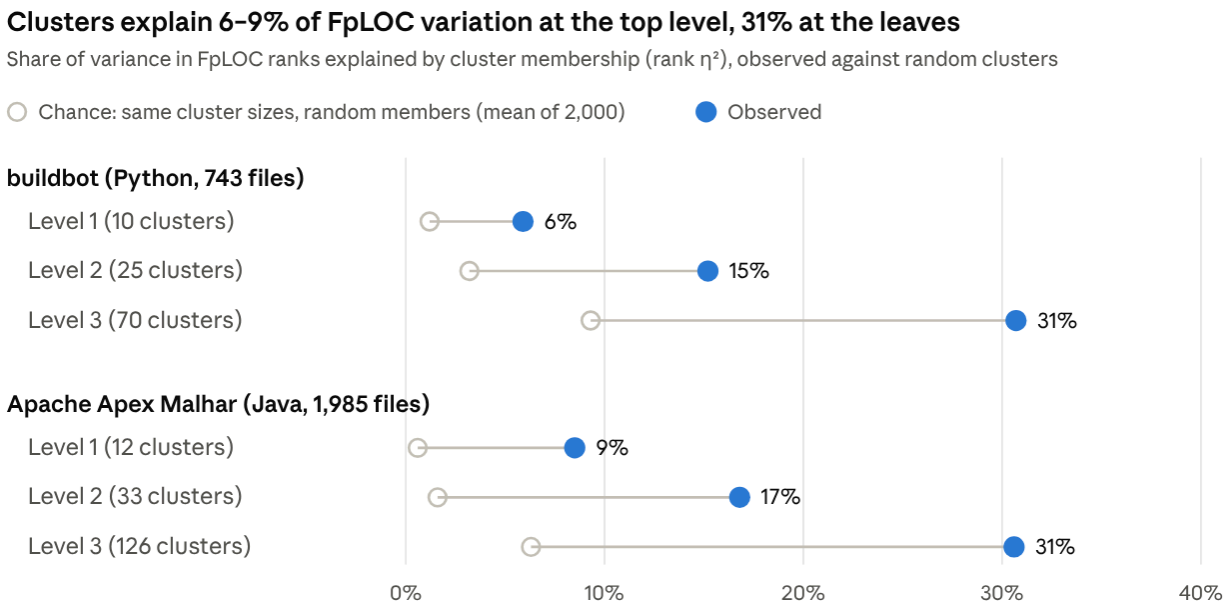
Level	buildbot: call links inside a cluster	Chance	Apex Malhar: call links inside a cluster	Chance
1	55%	11%	71%	10%
2	42%	5%	60%	5%
3	30%	2%	46%	1%

Most calls that cross top-level clusters go into shared infrastructure, or from integration tests into the code they exercise.

5.2 Does FpLOC cluster?

Three statistics were computed at each level, each against 2,000 random relabellings of cluster membership that keep cluster sizes fixed:

- **Variance explained (rank η^2).** The between-cluster share of the variance of FpLOC ranks. Ranks were used because FpLOC is highly skewed.
- **Controlled η^2.** The same, after regressing the ranks on log LOC and a test-file indicator, so that clustering by size or by test status is removed.
- **Same-cluster pairs.** In buildbot: among pairs of files in the same cluster that are both in the top or both in the bottom quarter of FpLOC, the share that are alike. In Apex Malhar 90% of files have FpLOC = 0, so quarters cannot be formed; instead we counted same-cluster pairs of files that both have a fault fixed, against the count expected by chance.



permutation tests on the corrected buildbot data and the Apex Malhar data · all files

Project	Level	Controlled η²	Same-cluster pair measure	Chance value
buildbot	1	6%	52% alike	50%
buildbot	2	16%	58% alike	50%
buildbot	3	32%	65% alike	50%
Apex Malhar	1	9%	1.19 × chance	1.00
Apex Malhar	2	17%	1.80 × chance	1.00
Apex Malhar	3	31%	3.87 × chance	1.00

Every statistic exceeds chance with permutation $p \leqslant 0.006$. Controlling for size and test status changes nothing, because FpLOC is barely related to file size (rank correlation −0.15 in buildbot, 0.00 in Apex Malhar).

Three further checks bear on robustness:

- Dropping the 17 buildbot files under 20 lines leaves the results unchanged.
- In Apex Malhar, restricting the analysis to non-test files removes the top-level effect (pairs 1.06 × chance, $p = 0.21$) but leaves levels 2 and 3 strong (1.89 and 3.66 ×).
- Among the 192 faulty Apex Malhar files, the size of FpLOC also clusters: 19%, 27% and 50% of rank variance explained, against 6%, 12% and 25% by chance.

5.3 High-fault and low-fault areas

The concentration is visible in the level-2 clusters with the highest and lowest rates (faults fixed per 1,000 lines).

Project	Highest-rate clusters	Lowest-rate clusters
buildbot (overall 35.1)	Compatibility and deprecation 70.9; legacy status and Perspective Broker 63.9; database engine and schema migrations 56.9; schedulers 52.0	Reporters 15.6; web change hooks 16.7; worker daemon 18.2; integration tests 19.2
Apex Malhar (overall 0.76)	Managed state 3.48; local-mode application tests 3.21; windowed operators 2.85; partitioning and codecs 2.81	No fix at all in NoSQL and search stores (186 files), message-queue connectors (97), app-data queries and schemas (83), Kudu (36)

Test files behave differently in the two systems. In buildbot they have about half the fault rate of other code (25.6 against 49.1 per 1,000 lines). In Apex Malhar the rates are similar (0.85 against 0.72).

6 Factors limiting accuracy

The largest uncertainties come from the fault data, not from the clustering: fix counts are sparse, not independent across files, and not adjusted for how long each file has existed.

6.1 Fault data

- **Short, sparse history.** Apex Malhar has 230 fixes over 1,985 files, and most files have two commits in total. Its FpLOC is zero for 90% of files and takes only a few distinct values for the rest, which limits statistical power and makes rank statistics tie-heavy.
- **Fixes that touch several files.** One fault-fixing commit that changes several related files marks them all as faulty. Related files tend to share a cluster, so this alone produces clustering. In Apex Malhar, 17 files of the JDBC example share 31 fixes. File-level counts are therefore not independent, and the permutation tests, which treat files as exchangeable, overstate significance.
- **Fixes are not faults.** One fault can take several commits to fix, and one commit can fix several faults. How commits were classified as fault fixes (keywords, issue links or manual labels) is not known here, and misclassification adds noise.

- **Exposure and age.** A file that has existed for ten years has had more chances to have faults fixed than one added last year. In both systems the low-fault clusters are largely recent subsystems (secrets, the newer reporters and web hooks in buildbot), while the high-fault clusters are long-lived legacy code.
- **History versus snapshot.** Counts cover a file's history, but LOC and functionality come from one snapshot. Files that were renamed or moved, as in Apex Malhar's package renaming, may have lost earlier history.
- **Data errors.** The first buildbot file computed FpLOC as a share of all fixes rather than per line. It was caught only because the column matched another calculation exactly, so other undetected errors are possible.

6.2 File descriptions

- Descriptions were written by a language model from summaries and source, then spot-checked rather than fully reviewed. Errors are most likely in large files that serve several purposes.
- Twelve or six independent model instances wrote different batches, so the same idea may be worded differently across batches. TF-IDF treats different words as unrelated, which can split or merge clusters for reasons of wording.
- Wording can dominate structure. Empty package files clustered together purely because their descriptions were similar, and had to be reassigned by rule.

6.3 Call links

- **buildbot.** Python's dynamic dispatch leaves many calls unresolved, especially calls through attributes such as `self.master.db`. The name-matching heuristic for probable links can be wrong. Calls made through callbacks, plugins, decorators or names held in strings are missed.
- **Apex Malhar.** About 30% of call sites could not be resolved because third-party libraries were not in the archive. Only the static target is recorded, so a call through an interface is credited to the interface, not to the classes that implement it. Reflection and dependency injection are invisible.
- Calls are only one kind of use. Inheritance, shared data, configuration and messages between components are not represented.

6.4 Clustering choices

- Several settings were chosen by judgement and not tuned or varied: the 0.55/0.45 weighting, the number of SVD dimensions, Ward linkage, the leaf cut and the 30-file maximum leaf size.

- The upper levels and all names were set by hand with the centroid tree as a guide. Another analyst could group some leaves differently, and a few leaves are mixed (for example, IRC bot code with SSL and Windows-service utilities).
- Call links are both an input to the clustering and the measure used to validate it, so the cohesion figures partly confirm the method's own assumption.
- No stability analysis was done. Resampling files or perturbing features might move a noticeable share of files between clusters.

6.5 Statistical analysis

- The chance level of η^2 rises with the number of clusters (from 1% to 9%), so raw η^2 values are not comparable across levels without it.
- Many tests were run on the same data without correction for multiple comparisons, although most p-values are far below any plausible threshold.
- The analysis is correlational. Unmeasured file properties such as complexity, churn, ownership and number of authors may explain both functional grouping and fault-proneness.

7 Towards a better understanding

The most valuable next step is to analyse fix commits rather than per-file counts, because that separates genuinely fault-prone areas from fixes that simply touch several related files.

7.1 Work at the level of commits

- Treat each fault-fixing commit as one event touching a set of files. Count distinct fix commits per cluster, so that a fix spanning ten files counts once.
- Test whether the files changed by one fix fall in the same cluster more often than chance. This measures fix locality directly, and its complement, fixes spanning several clusters, may point to cross-cutting design problems.
- Use SZZ-style blame analysis to find the commits that introduced each fixed fault, and the lines a fix changed. This moves the analysis from files to the code actually involved.

7.2 Account for time and exposure

- Measure fault density per file-year of existence, or per line-year, instead of per line at one snapshot.

- Split the history into periods. Test whether a cluster's fault rate in one period predicts the rate of its files in the next. Out-of-sample prediction is the practical test of whether clusters matter.
- Follow files through renames and moves, so that history is not lost.

7.3 Model confounders explicitly

Replace the separate permutation tests with a multilevel count model. It estimates how much variation sits at each cluster level after other factors are accounted for:

```
\mathrm{Freq}_i \sim \mathrm{NegBin}(\mu_i), \quad \log \mu_i = \log
\mathrm{LOC}_i + \beta^{\top} x_i + u_{L1(i)} + u_{L2(i)} + u_{L3(i)}
```

Here x holds file age, non-fix churn (AllFreq minus Freq), complexity, number of authors and a test-file indicator. The variances of the random effects u at each level give the share of fault-proneness attributable to the subsystem, the area and the small group.

7.4 Improve the inputs

- **Fault data.** Link fixes to issue trackers (JIRA keys for Apex Malhar, GitHub and Trac issues for buildbot). Manually validate a random sample of fix classifications to estimate the error rate.
- **Descriptions.** Generate them in one pass with a fixed vocabulary or taxonomy, or produce two independent descriptions per file and measure agreement. Have a person review a random sample. Replace TF-IDF with sentence embeddings, so that synonyms count as similar.
- **Call links.** Build Apex Malhar with its Maven dependencies so that calls into libraries resolve. Add class-hierarchy analysis to credit interface calls to their implementations. For Python, use a type checker such as pyright, or record calls dynamically while running the test suite.

7.5 Test the robustness and value of the clusters

- Vary the weights, the dimensionality, the linkage method and the cut levels, and measure agreement between the resulting hierarchies (for example, the adjusted Rand index). Bootstrap files to estimate how stable each cluster is.
- Compare the functional clusters with simpler groupings: directories and packages, co-change clusters, and random groupings of the same sizes. If directories explain fault-proneness as well as functional clusters do, the extra effort buys little.
- Repeat the study on more systems, including ones with long, dense fault histories.

7.6 Look for mechanisms

The high-fault clusters have plausible explanations to check: compatibility shims and schema migrations change whenever their dependencies change, and managed state and windowing are algorithmically intricate. Relating cluster fault rates to dependency volatility, code complexity and developer turnover would turn a correlation into an explanation.

8 Conclusion

Files that implement and use related functionality do share fault-proneness, but mostly within small groups of closely related files rather than across whole subsystems.

In both buildbot and Apache Apex Malhar, a hierarchy built from file descriptions and call links reproduced the call structure well and separated fault-prone from fault-free files far better than chance. At the subsystem level, however, cluster membership explained only 6–9% of the variation in faults fixed per line. At the level of groups of a dozen or so files it explained about 31%.

These figures are upper bounds on what functionality itself explains. Fixes that span related files, short histories and file age all push in the same direction. Commit-level, time-aware analysis with an explicit model of confounders is needed before clusters can be recommended as a unit for directing testing and review effort.