Functionality-based clustering of source files and the distribution of fixed faults: a study of Buildbot (Python) and Apache Apex Malhar (Java)

Prepared by Devin (Cognition AI)

November 2026

Abstract

We describe a pipeline that (i) gives every source file of a project a natural-language description of the application functionality it implements, (ii) extracts a static file-level call graph, (iii) builds a four-level hierarchy of file clusters from the combination of what a file implements and what it uses, and (iv) tests whether files with similar numbers of fixed faults per line of code (FpLOC) fall into the same clusters. The pipeline was applied to a 2018 snapshot of the Buildbot continuous-integration framework (774 Python files, fault data for 743) and to Apache Apex Malhar (1,985 Java files). In both projects the clusters are strongly cohesive with respect to the call graph, and cluster membership explains significantly more of the variation in FpLOC than random groupings of the same sizes. The effect is weak at the broadest level (3% of FpLOC rank variance in Buildbot, 9% in Apex Malhar beyond chance) and grows as clusters become smaller (32% and 52% at the finest level). In Buildbot low-FpLOC files are more concentrated than high-FpLOC files; in Apex Malhar, where 90% of files have no recorded fault, faulty files are concentrated in a handful of functional areas and 18 of 35 mid-level clusters contain no faulty file. Clusters derived from the source directory or package structure explain as much or more of the variation as the functionality clusters. We discuss the factors that limit these conclusions, chiefly the absence of exposure (file age, change activity) and commit-level information in the fault data, and propose analyses that would give a more causal understanding.

1 Introduction

Fault fixes are not spread uniformly over the files of a software system. A recurring question is whether the files that need many fixes share some property beyond their size—in particular whether they implement related functionality or use one another. If faults cluster by functionality, a functional decomposition of the code is a useful unit for estimating where future faults will occur and for allocating review and testing effort.

This report documents an exploratory study with three stages, applied in the same way to two projects written in different languages:

1. **File description**: each file is given a description of the functionality it implements, and a list of the other project files containing functions or methods it calls (Section 3).
2. **Hierarchical clustering**: files are grouped into a hierarchy using both the functionality they implement and the functionality they use (Section 4).
3. **Fault analysis**: we measure the extent to which files with high, and files with low, FpLOC are clustered together (Sections 5–6).

Section 7 discusses the factors limiting accuracy and Section 8 suggests further work.

2 Data

Table 1: Projects and fault data.

	Buildbot	Apex Malhar
Language	Python	Java
Snapshot	2018 (`62ef53c`)	`0d98d05`
Source files in archive	774	1,985
Files with fault data	743	1,985
Lines of code (files with fault data)	175,658	302,364
Fixed faults ($\sum$ Freq)	6,168	230
Files with Freq > 0	680 (91.5%)	192 (9.7%)
Median FpLOC	0.0316	0
Pooled FpLOC ($\sum$Freq/ $\sum$LOC)	0.0351	0.00076
Spearman ρ(LOC, Freq)	0.66	0.02
Spearman ρ(LOC, FpLOC)	-0.15	0.00

For Buildbot the supplied CSV contained an `FpLOC` column; the 31 files without a row are almost all empty package initialisers. For Apex Malhar FpLOC was computed as `Freq/LOC`; the CSV's file set matched the archive's Java files exactly. Freq in Apex Malhar is 0 for 1,793 files, 1 for 162, 2 for 23, 3 for 6 and 4 for one file. The two distributions (Figure 1) are therefore very different: Buildbot FpLOC is continuous and right-skewed, Apex Malhar FpLOC is zero-inflated.

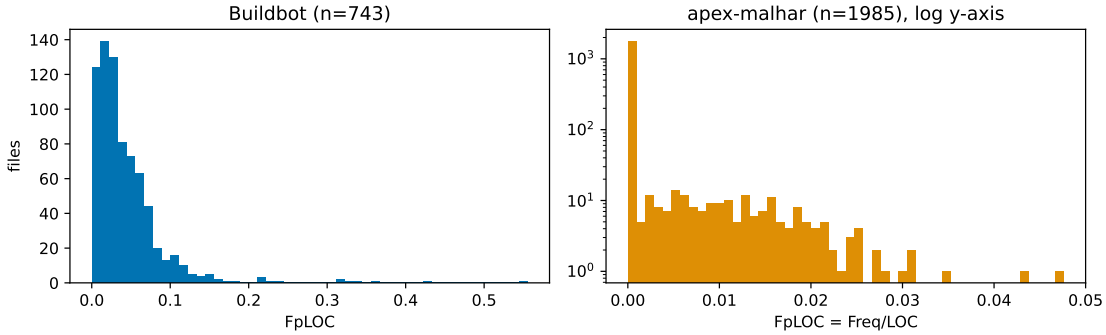


Figure 1: Distribution of FpLOC. Apex Malhar is shown on a logarithmic count axis because 90% of its files have FpLOC $= 0$.

3 Describing each file

3.1 Static call graph

For each file F we want the set of other project files that define at least one function, method or class (constructor) called from code in F. Both projects were analysed statically, i.e. from source text without executing it.

Python (Buildbot). Every call expression was located with the `parso` parser, and the callee was resolved with Jedi 0.18 (`goto` with `infer` as fallback), with the project's `master`, `worker`, `pkg`, `common` and web-plugin directories on the search path. A call contributes an edge $F \to G$ when its resolved definition lies in project file $G \neq F$. Of 52,044 call sites, 42,032 (81%) resolved to a definition; most of the remainder are calls into the standard library and third-party packages (Twisted, SQLAlchemy), which are correctly excluded. 587 files have at least one outgoing edge, and there are 2,039 distinct file-to-file edges. A heuristic that matched unresolved calls to project files by function name alone was tried and rejected because common names (`execute`, `select`, `Mock`) produced many false edges.

Java (Apex Malhar). A purpose-written analyser using JavaParser 3.25 and JavaSymbolSolver resolves `MethodCallExpr`, `ObjectCreationExpr`, explicit constructor invocations and method references. The type solver combines reflection (JDK), all project source roots, and the 153 dependency jars (Apex API/engine,

Hadoop, Guava, ...) that could be downloaded by Maven. Dependency jars matter: without them only 48% of the 75,606 call sites resolved, because most operators inherit from Apex base classes, so receiver types could not be inferred; with them 93% (70,375) resolved, 25,517 of them into project files. This yields 4,166 distinct file-to-file edges, with 1,315 files having at least one. Like Jedi, the solver resolves a virtual call to the declaring method of the static receiver type, not to the overriding implementations.

3.2 Functionality descriptions

Buildbot. A structural summary of each file was produced (module docstring, classes with their bases and docstrings, top-level functions, imports). From these summaries and the source, descriptions of production modules, test fakes and helpers, integration tests, packaging scripts and web plugins were written individually by the analyst (an AI assistant, not a Buildbot developer). The 272 unit-test files were mapped to the module under test by decomposing the test-file name (`test_process_buildstep.py` → `process/buildstep.py`), with manual overrides for irregular names; their descriptions name that module and repeat its description.

Apex Malhar. With almost 2,000 files and 1,385 class-level Javadoc comments available, descriptions were generated semi-automatically. For each file the first top-level type's Javadoc was cleaned (HTML and `{@link}` tags removed; text truncated at boilerplate such as "Properties:" or "Ports:"; first sentences up to ≈350 characters). If there was no usable Javadoc, a structural description was produced: kind, name split into words, superclass, interfaces and up to eight non-accessor methods. A hand-written description of each of the 205 packages was appended. Test classes were mapped to the class under test by name (`XTest`, `XIT`, `TestX`, ..., preferring the same package, then module); 420 of 695 test files were mapped and described as "Tests for *path* (*description of target*)". 78 files declare no type (e.g. `package-info.java`).

Validation. For both projects we checked that every source file appears exactly once, every listed callee file exists, no file lists itself, and CSV quoting is correct.

4 Hierarchical clustering

4.1 Features

Two feature blocks are built for every file.

Implemented functionality (text). A document formed from the description (weight 2), the path split into words (weight 2) and the declared identifiers (class, superclass, interface and method names, camelCase split). For files without content (empty `__init__.py`, `package-info.java`) the descriptions of sibling files in the same directory are used. Documents are converted to sublinear TF-IDF vectors (terms in ≥ 2 files and $\leq 50\%$ of files; generic words such as *test*, *operator*, *apache* removed).

Used functionality (graph). Let A be the file call-graph adjacency matrix. Each file is represented by its row of $G = \mathbf{1}[A + A^{\mathsf{T}} + I > 0]$, i.e. the files it calls, the files that call it, and itself. A TF-IDF transform is applied so that hub files used by almost everything (utility modules, base classes) carry little weight.

Both blocks are L_2-normalised, concatenated with equal weight, reduced with truncated SVD (200 components for Buildbot, 250 for Apex Malhar) and re-normalised. Two files are therefore similar if they talk about the same concepts and/or have overlapping call neighbourhoods.

4.2 Algorithm

Ward agglomerative clustering (Euclidean distance on the normalised SVD vectors) produces a full binary dendrogram. Cutting a dendrogram at a fixed height produces one giant cluster plus many tiny ones, so levels were obtained by top-down splitting: starting from the root, the largest node is repeatedly replaced by its two children until no node exceeds a size limit. Level 1 is obtained from the root with limit s_1; each level-1 node is split with limit s_2 to give level 2, and so on. The limits were $(70, 22, 7)$ for Buildbot and $(150, 45, 12)$ for Apex Malhar. Level-1 and level-2 clusters were then named by inspecting their dominant terms (the terms whose mean TF-IDF weight in the cluster most exceeds the global mean) and directories; level-3 clusters carry automatic labels. Finally level-1 clusters were grouped by hand into seven broad areas (level 0).

Table 2: Hierarchy sizes and call-graph cohesion: percentage of file-to-file call edges whose two ends are in the same cluster, and the expected percentage for a random assignment of files to clusters of the same sizes.

	Buildbot			Apex Malhar		
Level	clusters	intra-edge %	random %	clusters	intra-edge %	random %
0 (manual)	7	58.6	15.7	7	73.5	17.0
1	24	45.4	5.7	35	64.8	4.4
2	60	31.4	1.9	82	53.3	1.4
3	170	–	–	250	38.5	0.4

The Buildbot level-0 areas are: build execution pipeline; workers and master–worker communication; change ingestion and external integration; persistence, Data API and messaging; web UI and status; packaging, configuration, security and operations; shared utilities and integration testing. The Apex Malhar areas are: example/benchmark applications; database and key-value connectors; messaging, streaming-system and network connectors; file-system and block I/O; core operator library (algorithms, math, parsing, utilities); state management, windowing and high-level stream/SQL APIs; App Data and dimensional aggregation. Because test files' descriptions name the module they test and they call it, unit tests are placed next to the code they test.

5 Methods for relating faults to clusters

Several complementary statistics were computed for each partition (each hierarchy level, and for comparison test vs. non-test, source directory and, for Apex Malhar, module and package).

Between-cluster variance. FpLOC is converted to ranks r (ties averaged) to remove the influence of extreme values; $\eta^2 = SS_{\text{between}}/SS_{\text{total}}$ is the share of rank variance explained by cluster membership. Because η^2 grows with the number of clusters even for random groupings, its null distribution was obtained by permuting FpLOC over files (5,000 permutations for Buildbot, 2,000 for Apex Malhar), and we report the variance explained *beyond chance*, $(\eta^2 - \bar{\eta}_0^2)/(1 - \bar{\eta}_0^2)$, together with a permutation p-value.

High/low concentration (join counts). Files are labelled high or low and we count pairs of high (low) files that share a cluster, compared with the mean count under permutation. A ratio of 1 means no clustering. For Buildbot high/low are the upper/lower FpLOC quartiles (≥ 0.0569, ≤ 0.0168). For Apex Malhar quartiles are all zero, so we use *faulty* (Freq > 0, 192 files), *high* (FpLOC ≥ 0.0107, the median of faulty files, 96 files) and *zero* (1,793 files).

Size control. LOC is the denominator of FpLOC and small files give noisy ratios. We repeated the analysis with ranks adjusted by regressing r on rank(LOC), and on files with LOC ≥ 50 only.

Individual clusters. Each level-1 cluster was compared with the rest (Mann–Whitney for Buildbot FpLOC; Fisher's exact test of the proportion of faulty files for Apex Malhar), with Benjamini–Hochberg correction for 24 or 35 tests.

Call-graph assortativity. Spearman correlation of FpLOC ranks between the two ends of each call edge, with a permutation null; for Apex Malhar also the number of edges joining two faulty files.

6 Results

Table 3: FpLOC rank variance explained beyond chance (%) and concentration ratios. All η^2 values for hierarchy levels have permutation $p < 0.001$. For pair ratios, $p \geq 0.05$ is shown in brackets.

Buildbot (743 files) Partition	clusters	beyond chance	size-adjusted	LOC≥50	high pairs	low pairs
Level 0	7	2.7	4.2	5.2	1.07	(1.01)
Level 1	24	9.6	12.0	12.9	(1.08)	1.28
Level 2	60	16.7	18.0	19.7	1.21	1.28
Level 3	170	32.4	33.4	34.5	1.40	1.86
Test vs non-test	2	5.5	4.5	6.8	1.13	1.03
Source directory	79	26.2	28.0	23.8	(0.34)	(1.19)
Apex Malhar (1,985 files) Partition	clusters	beyond chance	size-adjusted	LOC≥50	faulty pairs	high pairs
Level 0	7	8.6	8.6	8.8	1.47	1.83
Level 1	35	20.9	20.9	21.7	4.10	6.27
Level 2	82	39.3	39.3	40.3	4.48	7.29
Level 3	250	51.9	51.9	53.7	5.87	9.89
Test vs non-test	2	0.0	0.0	0.0	(0.98)	(1.00)
Module	45	32.0	32.0	30.4	(0.94)	(0.78)
Package	205	57.5	57.4	58.9	3.88	5.38
Source directory	354	58.9	58.8	60.4	4.16	4.80

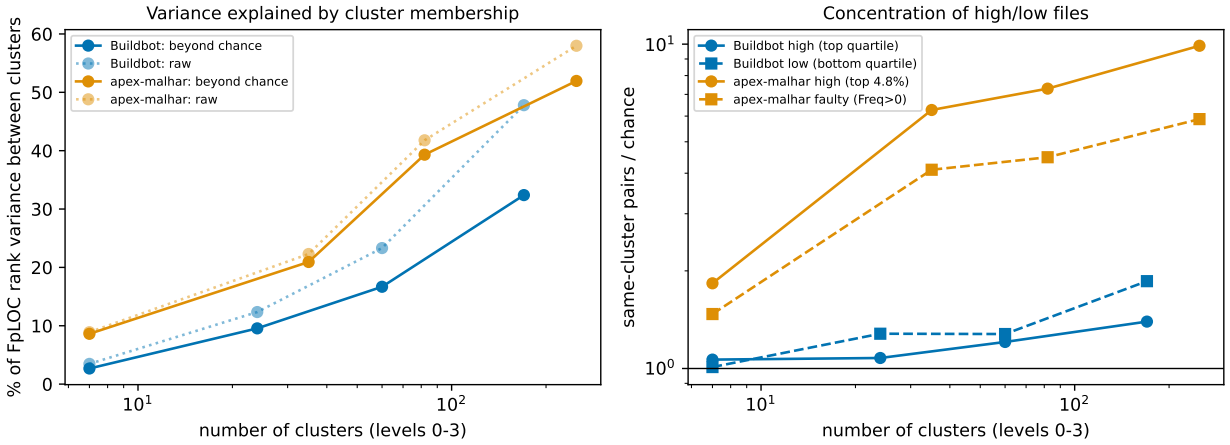


Figure 2: Left: share of FpLOC rank variance between clusters, raw and beyond chance, against the number of clusters. Right: number of same-cluster pairs of high (low, faulty) files relative to chance.

6.1 Buildbot

Cluster membership explains a statistically significant but modest part of the variation in FpLOC: 2.7% at the seven broad areas, 9.6% at level 1, 16.7% at level 2 and 32% for the 170 small level-3 clusters (3–7 files each). Most of the variation is within clusters. Low-FpLOC files cluster more strongly than high-FpLOC files (1.86× vs 1.40× chance at level 3; at level 1 the high-FpLOC concentration is not significant). Controlling for size does not weaken the result (it slightly strengthens it). Test files have lower FpLOC than non-test files (median 0.026 vs 0.043). FpLOC is weakly assortative over call edges (Spearman 0.14 over 2,038 edges, $p < 0.001$).

At level 1, three clusters have significantly higher FpLOC than the rest after correction—Schedulers (median 0.063, 53% of files in the top quartile), database connector and component tables (0.045) and change sources / master core (0.049)—and three have significantly lower FpLOC: reporters and web change hooks (0.014, 53% in the bottom quartile), secrets management (0.013) and master/worker integration tests (0.012). The remaining 18 clusters lie close to the overall median (Figure 3). Grouping by source directory explains more variation (26% with 79 groups) than the functionality clusters with a comparable number of groups (17% with 60).

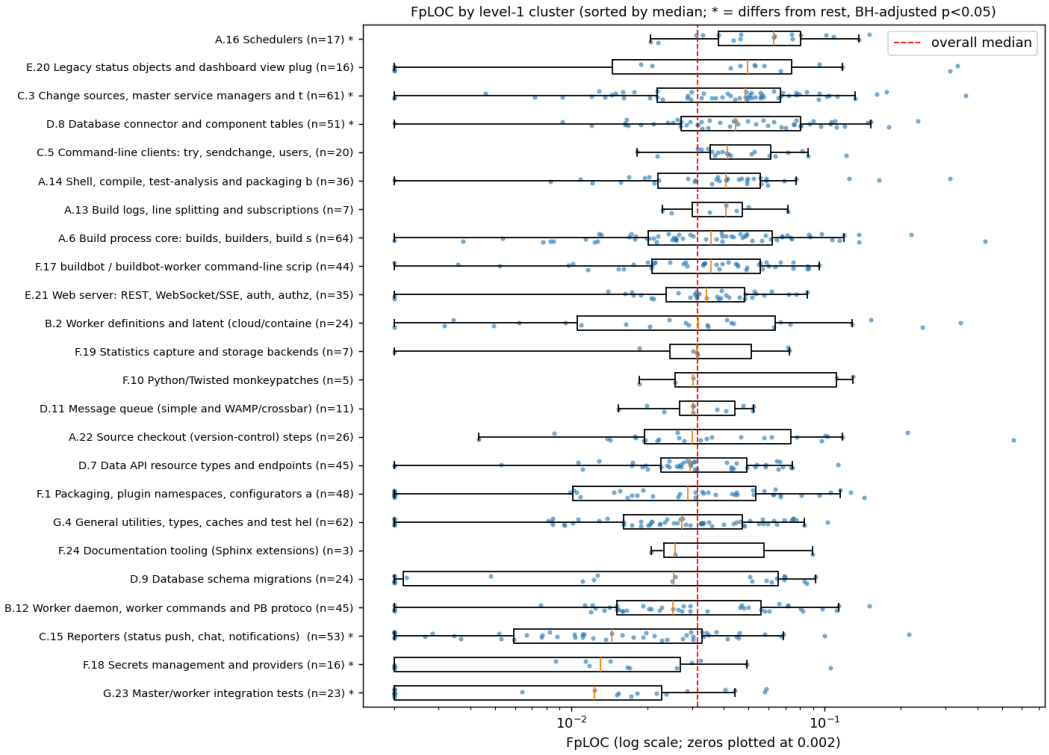


Figure 3: Buildbot: FpLOC by level-1 cluster.

6.2 Apex Malhar

Faulty files are much more strongly clustered than in Buildbot. Cluster membership explains 8.6% of FpLOC rank variance beyond chance at level 0, 21% at level 1, 39% at level 2 and 52% at level 3. Pairs of faulty files share a level-1 cluster 4.1 times as often as chance, and pairs of high-FpLOC files 6.3 times as often. File size plays no role: LOC is uncorrelated with both Freq and FpLOC, and the size-adjusted and LOC ≥ 50 results are unchanged. Test and non-test files do not differ (10.2% vs 9.4% faulty). FpLOC is more assortative over the call graph than in Buildbot (Spearman 0.30 over 4,166 edges); 168 call edges join two faulty files, against 39 expected by chance.

The concentration is visible at every level (Figure 4). At level 0, the proportion of faulty files ranges from 22.4% (state management, windowing and APIs) and 21.2% (examples) through 16.6% (file I/O) down to 1.4% (messaging connectors) and 0.3% (database connectors). At level 1, five clusters are significantly richer in faulty files—example applications (45%), windowed operators and the high-level stream API (33%), dimension and metric aggregators (28%), block readers / file input / S3 / DTFile (21%) and managed and spillable state (20%)—and 18 of the 35 clusters (760 files) contain no faulty file at all, five of them significantly so (NoSQL/search connectors, algorithmic and utility operators, App Data, benchmarks, and database stores with a single faulty file).

An important caveat is that in several example modules *every* file has Freq > 0 (all 17 files of `examples/jdbc`, and all files of `enricher`, `throttle`, `csvformatter`, `partition`), which suggests single commits touching whole modules rather than independent faults. Excluding the `examples` tree (267 files) reduces but does not remove the effect: 10.7%, 17.7%, 31.3% and 41.6% beyond chance at levels 0–3. As in Buildbot, the package (57.5%) and directory (58.9%) groupings explain somewhat more than the functionality clusters at comparable granularity, and the module grouping is dominated by the 930-file `library` module.

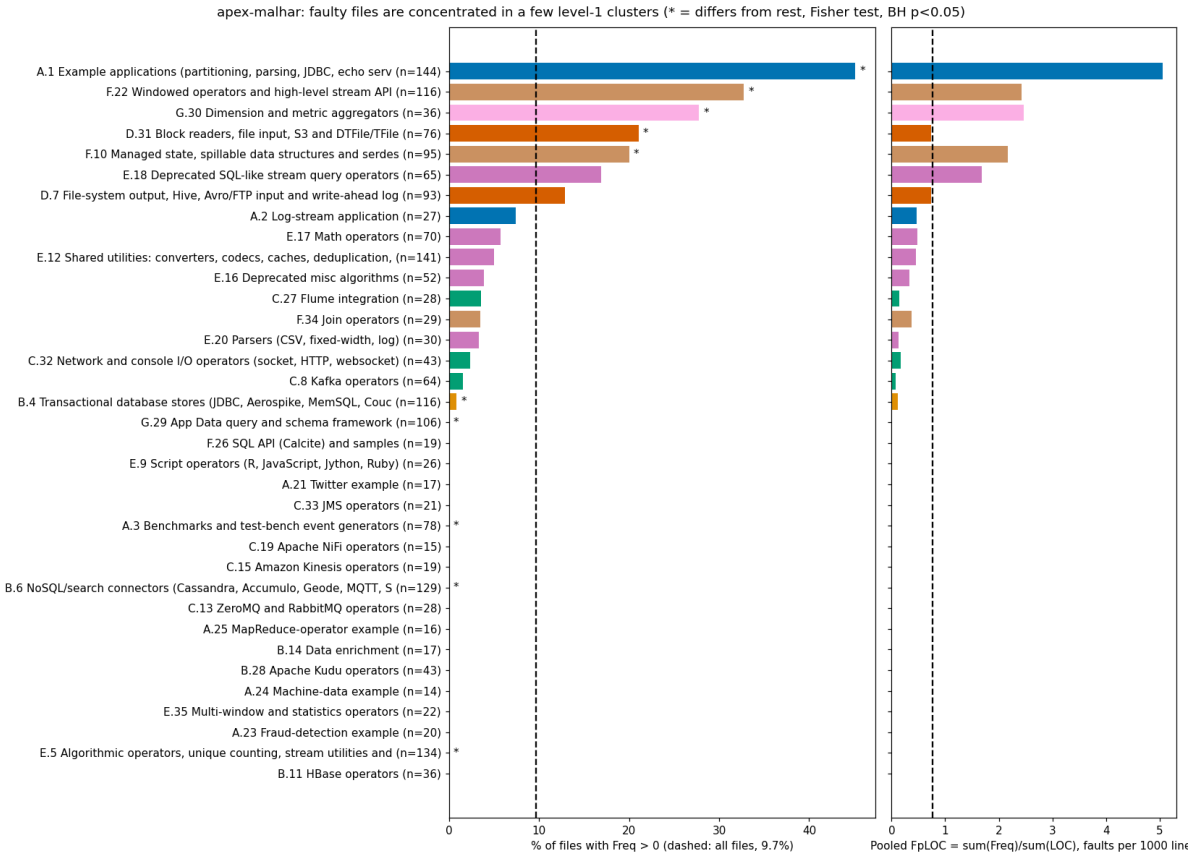


Figure 4: Apex Malhar: share of faulty files and pooled FpLOC by level-1 cluster.

6.3 Summary of findings

1. In both projects FpLOC is significantly associated with functional cluster membership; the association is weak at the broadest level and strengthens monotonically as clusters become smaller.
2. The pattern of concentration differs: Buildbot, in which almost all files have been fixed, shows stronger clustering of *low*-FpLOC files; Apex Malhar, in which few files have been fixed, shows fixed faults confined to a few functional areas.
3. The association is not explained by file size.
4. Files connected by calls have positively correlated FpLOC.
5. Directory/package structure is at least as good a grouping as the functionality hierarchy, which is unsurprising since paths are among the clustering features and developers organise code by function.

7 Factors limiting accuracy

Fault data.

- *Attribution.* How Freq was derived is not documented here; if it counts fix commits identified from commit messages, then tangled commits, commits that touch many files (renames, licence or API migrations), and misclassified commits all add noise. The 100%-faulty example modules in Apex Malhar are a probable instance of bulk attribution. Because one commit typically fixes several related files, co-fixed files are not independent observations, and that alone produces apparent clustering.
- *Exposure.* Faults are fixed only in code that exists and is used. Files differ in age, number of revisions and usage; a recently added file has had less opportunity to accumulate fixes. FpLOC normalises by current size only. Clusters of new functionality may appear low-fault for this reason alone.

- *Snapshot mismatch.* Fault counts accumulate over a history, while clustering uses a single snapshot. Files that were moved, renamed, split or merged may have lost history, and the LOC denominator may not represent the file's size when faults were fixed.
- *Zero inflation and small counts.* In Apex Malhar 90% of files have Freq = 0 and most others have 1; FpLOC is then mostly a binary indicator divided by LOC, and rank methods with massive ties have limited resolution. Conversely, Buildbot quartile thresholds lie in a dense part of the distribution, so high/low labels are sensitive to small count changes.

File descriptions and call graph.

- Descriptions were written or generated by an AI assistant from docstrings, Javadoc, identifiers and code structure, and have not been validated by project developers. Apex Malhar descriptions depend on Javadoc quality; missing or stale Javadoc gives structural descriptions only. Test-file descriptions largely copy those of the tested class.
- Static call resolution is incomplete and biased: dynamic dispatch is resolved to the declared type rather than to the runtime implementations; reflection, plugin registries (e.g. Buildbot's `buildbot.plugins` namespaces), configuration-driven wiring, callbacks and Twisted deferreds are largely invisible; and in Apex Malhar only part of the Maven dependency set could be downloaded, so some receiver types could not be resolved. 19% (Buildbot) and 7% (Apex Malhar) of call sites were unresolved.

Clustering.

- Many choices are arbitrary: the equal weighting of text and graph features, the TF-IDF and SVD settings, Ward linkage, the size limits defining levels, and the manual grouping at level 0. Their effect on the fault results was not systematically tested, nor was cluster stability (e.g. under bootstrap resampling).
- Path words are part of the text features, so the clusters partly reproduce the directory structure, making the comparison with directory-based grouping not fully independent.
- Placing unit tests with the code they test is a design choice; it mixes two populations with different fault behaviour in Buildbot.

Statistics.

- η^2 and join counts are sensitive to the number and size distribution of clusters; the "beyond chance" normalisation corrects for this only on average. Join counts weight large clusters heavily (e.g. the module grouping in Apex Malhar).
- Permutation tests assume exchangeability of files; dependence between co-changed files violates this and makes p-values optimistic.
- The analysis is associational: cluster membership correlates with FpLOC, but the data cannot distinguish intrinsic difficulty of functionality from differences in age, churn, usage, developers or testing.
- Only two projects were studied, with differently defined fault data (91% vs 10% of files fixed), so cross-project comparisons are qualitative.

8 Towards a better understanding

1. **Use the commit history.** Obtain the fix commits behind Freq, identify bulk commits (many files, mechanical changes) and either exclude them or weight each file's share by 1/(files in commit). Apply SZZ-style analysis to locate the lines changed by each fix, and record when each fault was introduced and fixed.
2. **Model exposure explicitly.** Fit count models (negative binomial or zero-inflated/hurdle Poisson) for Freq with offsets log(LOC) and covariates for file age, number of commits (churn), number of developers, and test coverage; include clusters as random effects at each hierarchy level. The intraclass correlation then measures how much fault-proneness is shared within clusters after controlling for exposure.

3. **Compare alternative decompositions.** Build clusterings from different signals—call graph only, text only, co-change (logical coupling) from the version history, directory structure—and compare their predictive power for held-out files or for faults in a later time window. Co-change clustering is particularly relevant because it captures the files that are fixed together.

4. **Improve the call graph.** Use class-hierarchy or points-to analysis to resolve virtual calls to all implementations; resolve all Maven dependencies; complement static analysis with dynamic call graphs collected while running the test suites.

5. **Validate descriptions and clusters.** Have project developers review a sample of descriptions and cluster assignments; test cluster stability with bootstrap resampling and adjusted Rand index; examine sensitivity to the feature weights and level limits.

6. **Graph-based fault statistics.** Treat FpLOC as a signal on the call graph and use spatial-autocorrelation measures (Moran's I with permutation tests), and test whether fan-in, fan-out or centrality predict FpLOC.

7. **Temporal prediction.** Split history into periods and test whether cluster-level fault density in one period predicts faults in files of the same cluster in the next, which directly tests the practical value of functional clusters.

8. **More projects.** Repeat the analysis on further projects with consistently defined fault data to establish whether the patterns observed here generalise.

9 Conclusion

A combination of written or generated functional descriptions and static call graphs yields file hierarchies that are strongly cohesive with respect to calls. In both Buildbot and Apex Malhar, files with similar fault density tend to be in the same clusters, beyond what random grouping produces and independently of file size. The effect is small for broad functional areas and substantial for small clusters of closely related files; it is stronger in Apex Malhar, where fixes are rare and confined to a few areas. Directory structure captures the same information at least as well. Establishing whether functionality itself drives fault-proneness, rather than age, change activity or bulk fix commits, requires commit-level fault data and exposure-adjusted models.

A Artefacts

Buildbot	Apex Malhar
`buildbot_files.csv`	`apex_malhar_files.csv`
`buildbot_clusters.csv`	`apex_malhar_clusters.csv`
`buildbot_cluster_hierarchy.md`	`apex_malhar_cluster_hierarchy.md`
`fploc_partition_stats.csv`	`apex_malhar_fploc_partition_stats.csv`
`fploc_level1_clusters.csv`	`apex_malhar_fploc_level1_clusters.csv`
`fploc_merged.csv`	`apex_malhar_fploc_merged.csv`