WHAT DO LLM-BASED SOFTWARE ARCHITECTURE RECOVERY METHODS ACTUALLY DO? A DIAGNOSTIC STUDY

SYED AMEENUDDIN QUADRI

A THESIS SUBMITTED TO
THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING
AND COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

July 2026

© Syed Ameenuddin Quadri, 2026

Abstract

Software architecture recovery groups a system's files into components. It has traditionally been performed by deterministic clustering algorithms; Large Language Models have recently entered the field, reporting single runs with no account of what the method does when it succeeds. This thesis asks what these methods do, running agent configurations from free file reading to knowledge-graph-only exploration, with repeated runs, directory flattening, and identity stripping. Two findings dominate. Recovered groupings mirror the folder tree: flatten the directories and recovery degrades in proportion to the reference architecture's directory alignment. Component names are memorized: a name-only probe reproduces each system's published architecture, and stripping every project name, filename, comment and header leaves that naming intact, down to a floor of code identifiers. The two channels substitute, since removing the folder tree raises recitation. The contribution is a measurement discipline: report distributions and directory alignment, and check for contamination.

TABLE OF CONTENTS

Page

Abstract ii

Table of Contents iii

List of Tables vii

List of Figures viii

1. Introduction 1
   1.1 Motivation 1
   1.2 Problem Definition and Scope 3
   1.3 Research Questions 3
   1.4 Approach in Brief 4
   1.5 Contributions 5
   1.6 Thesis Outline 6

2. Literature Review 7
   2.1 Classical Clustering-Based Architecture Recovery 7
   2.2 LLM-Driven Architecture Recovery 8
   2.3 Knowledge-Graph- and Retrieval-Augmented LLM Code Agents 8
   2.4 Evaluation Methodology and LLM Non-Determinism 9
   2.5 Memorization, Recitation, and Structure Bias 9
   2.6 Gap and Positioning 10

3. Method 12
   3.1 Overview: Three Layers 12
   3.2 Knowledge-Graph Extraction 12
   3.3 The Configuration Ladder 14
      3.3.1 The Fixed Allocator and What "No-KG" Means 17
      3.3.2 The Agent Chassis 17
      3.3.3 Structure-Riding Probes (A-F-flat and A-KG-flat) 18
      3.3.4 The Flat-KG Construction 20
   3.4 Two-Tier Evaluation 20
   3.5 Metrics 21
      3.5.1 Behavioral Metric: Cypher Query Classification 22
      3.5.2 Behavioral Metric: Canonical-Overlap (Recitation and Structure-Riding) 26
   3.6 Baselines and External Comparators 26
   3.7 Subject Systems and Gold-Truth Handling 27

3.8 The File Universe . . . 28
3.9 Experimental and Statistical Design . . . 29

4. Experiments and Results . . . 31
4.1 Experimental Setup and Protocol . . . 31
4.2 Tier 1: Validity Probe (A-Solo) . . . 31
4.2.1 Corpus and Setup . . . 32
4.2.2 Coverage and Output Validity . . . 32
4.2.3 Granularity Spread . . . 34
4.2.4 Hallucinated and Out-of-Universe Allocations . . . 34
4.2.5 What the Validity Probe Establishes . . . 34
4.3 A-F: Component Proposal with Fixed Allocation . . . 35
4.3.1 Corpus and Setup . . . 35
4.3.2 Validity by Construction . . . 35
4.3.3 Cross-Codebase Quality . . . 35
4.3.4 Two Codebases Whose Score Is Structurally Degenerate . . . 36
4.3.5 Bash: Stable Concept Anchors over a Directory-Aligned Gold . . . 36
4.3.6 What A-F Establishes . . . 37
4.4 A-F+KG: Component Proposal with an Offered Knowledge Graph . . . 38
4.4.1 Corpus and Setup . . . 38
4.4.2 Cross-Codebase Quality, Paired Against A-F . . . 38
4.4.3 Behavioural KG Usage . . . 39
4.4.4 Cypher Query Content: What the Agent Asks the KG . . . 39
4.4.5 What A-F+KG Establishes . . . 40
4.5 A-KG: KG-Only Exploration (Shell Disabled) . . . 40
4.5.1 Corpus and Setup . . . 41
4.5.2 Cross-Codebase Quality, Paired Against A-F+KG and A-F . . . 41
4.5.3 Behavioural KG Usage . . . 43
4.5.4 What A-KG Establishes . . . 44
4.5.5 Probing the Structure-Riding Mechanism: the Flat-Variant Ablation . . . 45
4.6 Structure-Riding: the Flat-Variant Ablation . . . 45
4.6.1 The Two Pathways . . . 46
4.6.2 The Flat-KG Construction . . . 46
4.6.3 Why C/C++ Only . . . 47
4.6.4 Canonical vs Flat Quality . . . 47
4.6.5 The Quantitative Prediction . . . 48
4.7 Recitation: Component Naming Under Identity-Stripped Substrates . . . 49
4.7.1 The Question the Structure-Riding Result Leaves Open . . . 49
4.7.2 The Zero-Substrate Elicitation Probe . . . 50
4.7.3 Identity Stripping as a Substrate Transformation . . . 50

4.7.4 Design: Identity Stripping Crossed with Flattening . . . 52
4.7.5 Neither Transformation Removes Canonical Naming . . . 52
4.7.6 Did the Explorer Know What It Was Reading? . . . 53
4.7.7 Canonical-Overlap Across the Full Suite . . . 54
4.7.8 What This Section Establishes . . . 55
4.7.9 Threats Specific to This Section . . . 56
4.8 Directory Lens and Deterministic Reference Partitions . . . 57
4.8.1 The Immediate-Parent and dir@K Construction . . . 57
4.8.2 Louvain-on-Dependency-Graph and Louvain-on-KG Reference Partitions . . . 59
4.8.3 What the Lens Establishes . . . 61
4.9 External Comparators (Qualitative Positioning) . . . 61
4.10 Cost . . . 62
4.10.1 Phase-1, Phase-2, and Probe Cost . . . 63
4.10.2 Cost Versus What Is Learned . . . 64
4.11 Qualitative Analysis . . . 64
4.11.1 Largest gap below the directory baseline: A-F-flat ITK, run 345d5821, flattening collapses the partition . . . 65
4.11.2 Highest run-to-run IQR and largest two-run disagreement: A-F+KG ITK, depth-1 vs depth-2 . . . 65
4.11.4 What the Cases Show . . . 66
4.12 Summary of Results . . . 66

5. Threats to Validity . . . 68
Construct Validity . . . 68
Internal Validity . . . 69
External Validity . . . 70

6. Conclusion . . . 71
6.1 Summary of Contributions . . . 71
6.2 Limitations . . . 73
6.3 Future Work . . . 73
6.4 Concluding Remarks . . . 74

Bibliography . . . 75

Appendix A. Codebase and Gold-Truth Provenance and Change Log . . . 81
A.1 The Governing Principle: Select the File Universe by File Structure, Not by the Gold . . . 81
A.2 Per-Codebase Provenance and Universe-Selection Rule . . . 82
A.3 Gold-Truth Cleaning: Every Change, with Rationale . . . 83
A.3.1 bash: 356 files, 13 components . . . 83
A.3.2 hadoop: 560 files, 62 components . . . 83

A.3.3 ITK: 5,228 files, 11 components 83
A.3.4 archstudio4: 583 files, 57 components 84
A.3.5 libxml2 (supplementary): basename-keyed JSON gold (82 items) 84
A.3.6 jpetstore (supplementary): class-level CSV gold (23 classes to 3 services, from 26 CSV rows) 84
A.4 Independent Verification (Three-Way Audit) 84
A.5 Locked Directory Baseline (Cleaned Golds × Path-Scoped Universes) 85
A.6 Cross-Codebase Consistency and the Two-Tier Design 85
A.7 Cypher Query Corpus: Worked Examples 85
A.7.1 Matching trajectories to scored runs 86

Appendix B. Algorithm Listings 87
B.1 The Fixed Batch Allocator 87
B.2 Flat-Substrate Construction 87
B.3 Consolidated Scoring 88
B.4 The Canonical-Overlap (Recitation) Scorer 89
B.5 The Louvain Reference Partitions 89
B.6 Per-Run Evidence on the Identity-Stripped Substrate 90

LIST OF TABLES

Table 1: The six-configuration ladder . . . 14
Table 2: Structure-riding probes: pathway closed and the reading of each outcome . . . 19
Table 3: The four-stage flat-substrate construction . . . 20
Table 4: Metric families: role, computation, and per-suite caveat . . . 21
Table 5: The four cypher-query categories . . . 23
Table 6: Per-codebase gold-cleaning fixes and resulting counts . . . 28
Table 7: Immediate-parent directory baseline per codebase . . . 29
Table 8: A-Solo validity over 10 runs per codebase . . . 32
Table 9: A-Solo dominant failure mode per codebase . . . 32
Table 10: A-F quality per codebase . . . 35
Table 11: A-F+KG vs A-F paired comparison per codebase . . . 38
Table 12: Classification of the A-F+KG cypher_query calls . . . 39
Table 13: A-KG vs A-F+KG paired comparison per codebase . . . 41
Table 14: A-KG vs A-F paired comparison per codebase . . . 41
Table 15: Classification of the A-KG cypher_query calls . . . 43
Table 16: A-F vs A-F-flat per codebase . . . 47
Table 17: A-KG vs A-KG-flat per codebase . . . 47
Table 18: Per-codebase directory covariate ARI(dir@2, gold) . . . 48
Table 19: Identity channels closed by the stripping transform . . . 51
Table 20: Probe-containment under identity stripping and flattening . . . 52
Table 21: Project identification forked from the recovery runs . . . 54
Table 22: Canonical-overlap by channel per codebase . . . 55
Table 23: Deterministic directory lens per codebase . . . 58
Table 24: Deterministic Louvain reference partitions per codebase . . . 60
Table 25: Phase-1 scored grid cost by configuration . . . 63
Table 26: Per-codebase Phase-1 cost by configuration . . . 63
Table 27: Universe-selection rules and the gold-independence test . . . 81
Table 28: Per-codebase universe (folder rule) and gold-independence . . . 82
Table 29: Per-run evidence on the identity-stripped substrate . . . 90

LIST OF FIGURES

Figure 1: Software architecture recovery as a clustering task 1
Figure 2: The three-layer recovery pipeline 13
Figure 3: The per-codebase knowledge-graph schema 14
Figure 4: The configuration ladder 16
Figure 5: The graph-structure-agnostic fixed allocator 18
Figure 6: Classifying a cypher-query call 25
Figure 7: Per-configuration MoJoFM and ARI distributions 42
Figure 8: Median ARI across the A-F, A-F+KG, A-KG ladder 44
Figure 9: Flat-versus-canonical recovery drop by directory alignment 49
Figure 10: Mean cost per run by codebase and configuration 62

1. Introduction

1.1 Motivation

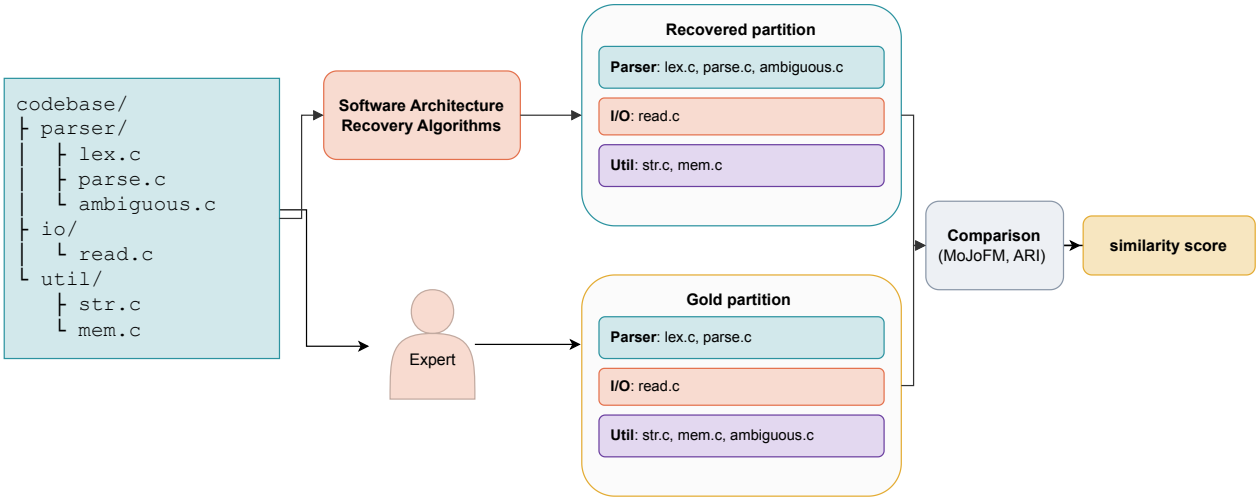


Figure 1: Software-architecture recovery as a clustering task: a codebase is a flat set of source files. A recovery method groups them into a partition of named components (Parser, Job Control, I/O, . . .). Quality is the partition's agreement with a developer-authored reference ("gold") grouping, measured by MoJoFM or ARI.

Software architecture recovery (SAR) is the task of reconstructing a system's component-level structure from its source code, so that a maintainer can understand, evolve, and reason about a codebase whose original design documentation is missing, outdated, or was not written in the first place. The dominant tradition treats recovery as a clustering problem: the system's source files (or finer code units) are sorted into groups, called **components**, such that every file belongs to exactly one group (a **partition**), and the groups correspond to the modules a developer would draw on a whiteboard, such as "Parser" or "Job Control" (Figure 1). To judge a recovered partition, the field compares it against a reference architecture prepared by developers or researchers who studied the system by hand. This reference is called the **ground truth** or, following common usage, the **gold**. Agreement between a recovered partition and the gold is summarized by similarity metrics, commonly on a zero-to-one scale, introduced in Chapter 3. This clustering-based line of work is established, with canonical algorithms such as Bunch, ACDC, LIMBO, and WCA, and a comparison protocol and metric suite consolidated by Garcia et al. [1] and extended by Lutellier et al. [2, 3].

An obstacle has shaped the field: constructing these gold architectures is expensive. The recovery-and-validation effort for a single benchmark system is reported to be on the order of a hundred person-hours, which is why the canonical benchmark set is small and reused across the literature. The same few systems (bash, hadoop, ITK, and a handful of others) appear in study after study. This expense is the kind of bottleneck that motivates automation, and the recent wave of Large Language Model (LLM) work has begun to apply LLMs to the task. LLMs enter in two main ways. One line keeps the classical clustering pipeline but uses the LLM to describe what each file does, giving the clustering algorithm richer information to group on. The most direct point of comparison in this line, SemArc [4], enriches a clustering pipeline with LLM-extracted summaries of each file's implementation and architectural role, and reports large gains over

classical baselines. Another line asks the LLM to classify code directly against an already-known reference architecture [5]. A third line, and the one this thesis instruments, is **agentic**: the LLM is run in a loop and given tools it can invoke, such as listing directories, reading files, or querying a database. The model decides which tool to call next, reads the result, and continues until it decides it is done. Such agents are often paired with a structured view of the repository, a **code knowledge graph (KG)**: a database of the codebase's files, functions, and classes and the call, include, and dependency relationships among them, which the agent queries instead of (or alongside) reading files. Explicit graph structure helps such agents navigate code (Section 2.3), so a recovery pipeline built over a KG inherits an expectation that the graph is doing real work. Whether it actually is, for recovery, has not been tested.

Two methodological gaps run through this emerging LLM-SAR literature. The first concerns repetition. An LLM is **non-deterministic**: asked the same question twice, it can give different answers. A decoding setting called **temperature** controls how much randomness the model adds when it generates text, and temperature zero is the setting intended to make it as repeatable as possible. In practice, outputs still vary even there, a property the broader software-engineering measurement literature has flagged as a threat to the validity of conclusions [6, 7, 8]. Yet LLM-SAR results are reported as **single-run point estimates**: one score, from one attempt, at one pipeline configuration. Classical clustering algorithms are effectively deterministic (or were reported as the best of N attempts), so the discipline of running an experiment many times and reporting the spread of outcomes was not inherited by the LLM line of work.

The second gap concerns a simple baseline. A codebase's **directory structure** already groups files: every file lives in some folder, and grouping files by their folders is itself a partition, one that requires no analysis. Because developers tend to organize folders by module, this free partition is often close to the gold. Lutellier et al. [3] introduced this directory baseline together with the observation that it already matches several recovery techniques on some of the benchmark systems. We use the directory partition not as a competitor to outscore but as a **diagnostic lens**: for any recovered partition, it lets us ask how much of the apparent success was already explained by the folder layout alone, and therefore whether the method understood the code or merely mirrored its folders. That lens has not crossed into the LLM-driven recovery literature, and bringing it in, combined with multi-run reporting, is part of the methodological core of this thesis.

A third gap underlies the first two and is sharper. LLMs are trained on large collections of public text and code, their **pre-training corpus**, and the canonical SAR benchmarks are among the most-documented software systems: their architectures appear in textbooks, manuals, and online discussions, material almost certainly present in that corpus. A model can therefore produce a plausible decomposition of bash by *recalling* bash's published architecture from training rather than analysing the code in front of it. When a benchmark's answers leak into a model's training data, the phenomenon is called **benchmark contamination**, and it changes what a score means: a high score may measure memory, not capability. Contamination is established for code LLMs in adjacent fields [9, 10] but is neither measured nor disclosed in the LLM-SAR literature, even though it threatens the meaning of any reported score. Surfacing this threat, and measuring it, is the second half of the thesis's diagnostic core.

1.2 Problem Definition and Scope

This thesis asks one headline question, and it is a **diagnostic** one: **what do LLM, KG, and agentic methods for the clustering sub-task of software architecture recovery actually do, and do their components carry any weight?** This is deliberately *not* the question "do they win a benchmark." We answer it **internally**, by probing the behaviour of a ladder of our own pipeline configurations (does the KG change the output? is the output stable across runs? what does it reproduce when it appears to succeed?), rather than through a numeric head-to-head with any external paper's published numbers.

The study is scoped to the **clustering sub-task**: given an authoritative set of source files, how are they partitioned into components, and how good is that partition? The complementary problem of *which* files should enter the partition in the first place (filtering out build scripts, vendored third-party code, tests, and generated artifacts) is **out of scope** and disclosed as a limitation. This scoping is intentional and is reflected in our file-universe handling (Chapter 3): the set of files to be partitioned is fixed in advance by folder rules, without consulting the gold, so that no method is handed the answer key and all methods and baselines are scored over the same files.

1.3 Research Questions

The headline diagnostic question is decomposed into three research questions. The first two concern the methods' *internal behaviour*: does the knowledge graph matter, and is the output repeatable? The third, the headline, asks what the method is actually reproducing when it appears to succeed. Each question names the instruments that answer it. The instruments themselves are introduced in Section 1.4 and detailed in Chapter 3.

- **RQ1: Is the code knowledge graph load-bearing for LLM-based architecture recovery?** By load-bearing we mean carrying real weight: if the graph were taken away, or actually used, would the output change? We answer with two controlled comparisons in which exactly one ingredient changes at a time (such single-variable comparisons are called **ablations**): first the KG is *offered* to the agent alongside its ordinary file-reading tools, then the KG is *forced* by taking file reading away, so that querying the graph is the only way to see the codebase. Logs of every query the agent issues show both whether quality changes and how the graph was actually used.
- **RQ2: How stable is an LLM-driven recovery across identical repeated runs?** Every configuration is run repeatedly on identical input at temperature zero, the most repeatable setting available. We measure variation in quality, in coverage (on the one configuration where the agent decides for itself when to stop assigning files), and in the number of components proposed, and ask how much a single-run point estimate hides.
- **RQ3 (headline): When recovery appears to succeed, what does it actually reproduce?** We test two layers. The first is **structure-riding**: are the file groupings read off the existing folder layout rather than recovered from the code? We test this by hiding the folder structure (every file moved into a single flat directory) and watching whether recovery survives. The second is **recitation**: do the component *names* come from the model's memory of the system's published architecture rather

than from analysis? We test this by stripping the codebase's identity from the substrate, by asking the model for the architecture given nothing but the system's name, and by measuring how much of each run's naming vocabulary matches that memorized architecture. RQ1 and RQ2 are answered by internal, configuration-to-configuration comparison and reported as **distributions** over repeated runs. RQ3 uses the gold-independent directory partition as a diagnostic lens (Section 2.1). Cross-cutting all three is a **methodological contribution**: multi-run distributions, the directory lens, and a benchmark-contamination check.

1.4 Approach in Brief

We evaluate a **configuration ladder**: a series of pipeline configurations in which each rung differs from the previous one by exactly one ingredient, so that any change in outcome can be attributed to that ingredient. All rungs share the following: An LLM agent explores the codebase and proposes a list of components, each just a name and a one-line purpose of the component. A **fixed allocator**, a zero-shot LLM prompt, then assigns every file to exactly one of the proposed components. The allocator guarantees that every file is placed and no non-existent file is invented, so the configurations that use it always produce a valid partition, and it is held identical everywhere, so the only thing that varies is how the agent explores. The configuration names have been designed so that they may be read compositionally: **A** means an LLM agent drives exploration, **F** means it explores through the file system using a **shell** (a terminal in which it can run commands such as `ls`, `cat`, and `grep`), and **KG** means it explores through the knowledge graph, which it queries in Cypher, the query language of the Neo4j graph database that stores the KG. The motivating ladder is four rungs, each answering a question the previous rung raised:

- **A-Solo**: a single agent both explores the codebase and assigns files to components itself, with no fixed allocator. This rung exists as a *validity probe*: it asks whether an unconstrained agent can produce a well-formed partition. This failure becomes the reason for the Fixed Allocator in all the following configurations.
- **A-F**: the agent only proposes components, reading the codebase through the shell, and the fixed allocator assigns the files.
- **A-F+KG**: the same agent, additionally *offered* the KG query tool alongside the shell. It may use either.
- **A-KG**: the KG *forced* as the only exploration lens (the shell disabled), the direct test of whether the KG is load-bearing when actually used.

The ladder unfolds as its results arrive: A-Solo's self-allocation turns out to be *invalid* (it invents file paths and leaves large parts of the codebase unassigned), so allocation is taken off the agent (A-Solo to A-F). Because A-F only reads files, a KG is offered (A-F to A-F+KG). The agent then largely ignores it, which cannot distinguish a useless graph from an unused one, so the KG is forced (A-F+KG to A-KG). These four rungs are **gold-scored**, meaning their partitions are scored against the gold. Two **structure-riding probes**, **A-F-flat** and **A-KG-flat**, rerun A-F and A-KG on a copy of the codebase whose directory structure has been erased (every file renamed into one flat folder), removing the folder signal that RQ3 suspects the methods are riding. The same two configurations are also rerun on an **identity-stripped** copy of each C study system,

described in Section 3.3.4, so that the folder channel and the naming channel can be closed independently and together.

We run the design on six systems in total: four core canonical benchmark systems, **bash, hadoop, archstudio4, and ITK**, all used by Lutellier [3], plus two supplementary systems whose golds come from elsewhere. **libxml2**'s gold is the decomposition released with SARIF [11], labelled by that project's industrial collaborator rather than by libxml2's developers; **JPetStore**'s is the four-person expert microservice decomposition of Zaragoza et al. [12], digitized as a class-to-service table in the independent tool comparison of Wang et al. [13]. Both supplementary systems run through the full grid but are read as disclosed structure-riding exhibits and are not pooled with the core systems. Recovered partitions are scored with two agreement metrics defined in Chapter 3: the **Adjusted Rand Index (ARI)**, the primary metric, which scores agreement between two partitions and corrects for chance, and **MoJoFM**, the field's traditional metric, reported secondarily with a caveat because it is insensitive to how finely a partition is split on this suite. Because of the non-determinism of RQ2, every quality configuration is run ten times per codebase, and we report **distributions** (median, inter-quartile range, and worst-best gap over repeated runs), reading each configuration difference against that run-to-run spread.

1.5 Contributions

This thesis makes three contributions: *mechanism* (does the KG carry weight?), *reliability* (is the output stable?), and *what-it-does* (what does it reproduce?). The third is the load-bearing one and has two layers, a *structure-riding* layer (where the file groupings come from) and, beneath it, a *recitation* layer (where the component names come from). Two of the three do not involve the directory baseline:

1. **The KG is under-used, and even when forced it is not load-bearing here.** We present the first ablation of knowledge-graph augmentation in LLM-SAR, and we keep two claims distinct. First, **behavioural under-use**: when the KG query tool is *offered* alongside the shell (A-F versus A-F+KG, allocator held constant), the agent largely ignores it, so A-F+KG's output stays close to A-F's. The neglect persists even when the prompt spells out what the graph's relational edges are good for. Second, the **load-bearing test**: because under-use alone cannot tell whether the KG is useless or merely unused, we *force* the KG by disabling the shell (A-KG), the direct test of whether the substrate carries weight when actually used. The under-use is a *behavioural choice*: when the agent does issue relational graph queries, they succeed.

2. **Run-to-run variance.** We characterize the non-determinism of LLM-driven recovery at temperature zero: coverage (driven by the agent's own stop-or-continue decision) and the number of components proposed both vary across identical inputs. Such variance is new to the SAR setting, where classical methods are essentially deterministic, and it motivates reporting multi-run distributions.

3. **Structure-riding and recitation.** We identify, in two layers, what the method reproduces when it appears to succeed. The first layer is **structure-riding**: the method reproduces the existing directory groupings rather than recovering latent architecture, shown by the flattening experiment of RQ3,

whose damage per codebase is predicted by a directory-alignment score (how closely the gold follows the folders) that can be computed before any run. The second, deeper layer is **recitation**, the thesis's headline: the component *names* the method emits are a memorized canonical architecture. We establish this by anonymizing the codebase step by step (removing the system's name, then file and directory names, then comments and indicative text) and observing that the canonical names persist, reconstructed from the identifiers in the code itself. We further show that the model reproduces the published architecture when given nothing but the system's name, and that removing the directory tree does not suppress the recitation channel but amplifies it. Two corollaries follow for *measurement*, not for our method's standing: because the gold standards on this suite are largely directory-aligned, and because the benchmark systems are in the model's pre-training corpus, high scores can be earned by mirroring folders or by recall. **Benchmark contamination** is therefore a concrete threat to what these scores measure, and scores in this literature overstate semantic recovery.

Together, these contributions characterize what these methods do: on these famous, documentation-rich benchmarks an apparently successful recovery is substantially the sum of two priors, *directory-readable-for-free* and *training-data-recallable*, with a small residual attributable to analysis of the code itself. The full assessment appears in the conclusion (Section 6.4).

1.6 Thesis Outline

Chapter 2 reviews classical clustering-based SAR, the LLM-driven recovery literature, the knowledge-graph-augmented LLM code-agent lineage, the evaluation-methodology and LLM-non-determinism work, and the memorization and contamination literature behind the recitation and structure-riding findings, closing by positioning the contributions against the identified gap. Chapter 3 describes the method: the deterministic KG extraction, the configuration ladder and its ablations, the two-tier evaluation, the metrics and baselines, the subject systems and gold-truth handling, and the experimental and statistical design. Chapter 4 presents the results: the A-Solo validity probe, the ladder's per-configuration quality and behavioural analysis, the structure-riding flatten ablation (§4.6), and the recitation and identity-stripping study (§4.7) that establishes the headline finding. Chapter 5 collects the threats to validity. Chapter 6 concludes.

2. Literature Review

This chapter positions the thesis within five bodies of work: classical clustering-based software architecture recovery (SAR), the emerging LLM-driven recovery literature, the knowledge-graph- and retrieval-augmented LLM code-agent lineage, the evaluation-methodology and LLM-non-determinism work that motivates multi-run reporting, and the memorization and contamination literature that grounds the thesis's recitation and structure-riding findings. It closes by stating the gap and positioning the thesis's three contributions.

2.1 Classical Clustering-Based Architecture Recovery

The clustering tradition reconstructs a component view by partitioning a system's code units using the call, include, and use relationships among them. The canonical search-based clusterer is Bunch [14], which performs hill-climbing (and genetic) optimization of a modularisation-quality objective over the module dependency graph. The genetic-algorithm formulation of automatic clustering is due to Doval et al. [15]. ACDC [16] takes a comprehension-driven, pattern-based approach intended to produce decompositions that match how a maintainer would group the system. It is frequently used as a reference decomposition when human-labelled architectures are unavailable. Surveys of the clustering methodology [17] and of the evolution of recovery techniques [18] map the broader design space.

A recurring methodological question in this tradition is *which relationships* to feed the clusterer. Stavropoulou et al. [19] study which relations are most effective for clustering-based recovery. Lutellier et al. [2, 3] show that the *type* of dependency (include versus call versus symbol) materially changes recovery quality. The journal version of that work [3] is doubly important for this thesis. First, it introduced the **directory-structure baseline** (group each file by the package or directory in which it is declared) and reported that this baseline already matches several state-of-the-art recovery techniques on some of the benchmark systems, specifically archstudio4 and hadoop. Lutellier framed that observation competitively (a technique that cannot beat the directory structure offers little). We reuse the directory baseline but *not* that competitive reading. In this thesis the directory partition is a **diagnostic lens**. It is a reference for how much of any recovered output, or of any gold itself, is already explained by folder structure. Via the flatten ablation (Chapter 3), it is also a way to test whether what looks like recovery survives once the directory scaffolding is removed. Second, it is the source of the four core ground-truth architectures (bash, hadoop, archstudio4, ITK) reused throughout this thesis, so our work runs on that corpus. We adopt the directory baseline as **prior work** and apply it as a lens to LLM methods, rather than re-deriving the control or claiming it as a discovery. Our directory baseline is equivalent in *construction* to Lutellier's package baseline but is computed over a different file universe, so we do not place the two numbers in a numeric head-to-head (Section 3.8).

The comparison protocol and the evaluation metrics consolidated by Garcia et al. [1], together with the Rigi Standard Format ground-truth representation and the use of MoJoFM and related cluster-similarity measures, form the methodological backbone the present study inherits. The strongest recent non-LLM system, SARIF [11], fuses typed/weighted dependencies, LDA-topic code text, and quality-filtered folder

structure with adaptive per-source weights. It is the classical ceiling against which an LLM method should be judged. We treat it as a qualitative specimen (Section 3.6). Reflexion-model analysis [20] and graph-matching formulations of recovery [21, 22] round out the classical context.

2.2 LLM-Driven Architecture Recovery

A recent strand substitutes LLMs into the recovery pipeline. The closest published method, SemArc [4], uses an LLM to extract implementation-level semantics (what each file does) and architecture-level semantics (which canonical pattern a file manifests), and anchors a clustering algorithm on pattern-derived component seeds. It reports large average gains over classical baselines on a fifteen-system corpus that **overlaps our benchmark set** (bash, hadoop, archstudio4) and uses the same Lutellier-derived ground truth. For this thesis, SemArc is a one-shot enrichment of a clustering pipeline that reports single-run point estimates at default (stochastic) decoding, and uses no directory baseline. These are the two methodological gaps this study targets. We therefore treat SemArc as a measurement-methodology **specimen** (to expose the single-run, model-confounded reporting style) and a legitimacy anchor.

Rukmono et al. [5] propose a deductive, top-down recovery that uses chain-of-thought prompting to classify code units against a *known* reference architecture. Their method requires that reference architecture as a precondition, whereas this study assumes none. Soliman and Keim [23] probe whether LLMs contain software-architectural knowledge in an exploratory case study with GPT. Dhar et al. [24] and Díaz-Pace et al. [25] study LLMs for generating and assisting architectural *design decisions*, which are adjacent forward-design tasks rather than recovery against ground truth. Johansson et al. [26] map source code to architecture with LLMs. Broader surveys of generative AI for software architecture [27] and of the software-architecture-meets-LLMs literature [28] situate the subfield. MDRE-LLM [29] is representative of LLM tooling for reverse engineering more generally. Across this strand, multi-run stability of an LLM recovery and a directory-baseline control are absent.

2.3 Knowledge-Graph- and Retrieval-Augmented LLM Code Agents

The mechanism this thesis ablates, an LLM reasoning over a persistent code knowledge graph, has an established lineage outside SAR. Graph-guided agents for code localization (LocAgent [30]) report that an explicit code graph improves the agent's ability to find the right files for a task. CodexGraph [31] stands a code repository up as a graph database and lets the LLM author graph queries (the closest published precedent for the mechanism A-F+KG adds: an LLM issuing Cypher-style queries over a code knowledge graph). RepoGraph [32] packages a repository-level code graph as a plug-in module that improves software-engineering agents on generation and repair. The GraphRAG foundations [33] and the graph-reasoning paradigms (Think-on-Graph [34], Reasoning-on-Graphs [35]) supply the general "LLM-over-graph" retrieval-and-reasoning pattern. These results establish the *expectation* that a KG substrate should be load-bearing. This thesis tests that premise in the recovery setting and finds it unsupported (Section 2.6 and Chapter 4). The distinction is task: those settings reward *navigation* to a small set of relevant entities, whereas recovery requires a *global partition* of the whole system. On directory-aligned benchmarks the file path already encodes most of the partition, so the relational graph that helps with navigation has little to add. We observe it going unused.

2.4 Evaluation Methodology and LLM Non-Determinism

Two metric families anchor the evaluation. The Adjusted Rand Index [36] measures chance-corrected agreement between a recovered and a reference partition, and MoJoFM [37] measures the move-and-join transformation cost normalized to a similarity score. Both are the field-standard cluster-vs-ground-truth measures used by Garcia, SARIF, and SemArc. Two properties of these metrics matter for our reporting and are discussed in Chapter 3: MoJoFM is direction- and granularity-sensitive and is heavily compressed on this benchmark suite, so ARI is treated as primary.

The case for **multi-run reporting** comes from the LLM-non-determinism literature. Ouyang et al. [6] argue that the non-determinism of LLMs is an obstacle to conclusion validity and that temperature zero does not guarantee determinism. Atil et al. [7] quantify substantial accuracy variation and large best-worst gaps across repeated runs under "deterministic" settings. Song et al. [8] argue that single-output evaluation is misleading. These works justify the design choice to report distributions (median, inter-quartile range, and worst-best gap) over repeated runs. No paper we have seen this far in the SAR literature measures the multi-run variance of an LLM-based recovery against developer ground truth using clustering metrics. This makes the variance characterization in this thesis novel for the recovery setting.

2.5 Memorization, Recitation, and Structure Bias

The thesis's headline findings, recitation and structure-riding, each have an established lineage that the LLM-SAR literature has not yet connected to.

The recitation finding rests on the memorization literature. Large language models reciting their training data was established by Carlini et al., who extracted verbatim training sequences from GPT-2 by adversarial prompting [38], and later quantified memorization with three scaling laws: it grows with model capacity, with a fragment's duplication count in the training set, and with the amount of prompting context [39]. All three axes predict maximal memorization for the canonical SAR benchmarks, which are among the most-mirrored, most-quoted codebases on the web. Zhang et al. distinguish rare-fact memorization from *common-knowledge* memorization of high-frequency material [40]. The published architectures of bash or hadoop are common knowledge in this sense, so a model does not need to have seen a study's gold file to reproduce its contents. In code models specifically, memorization is directly measurable [41, 42], multifaceted (spanning verbatim recitation, reconstruction, and recollection [43]), and can be *functional*: the logic is reproduced even when the surface form is not [44]. Large-scale audits find the resulting **benchmark contamination** inflates software-engineering evaluation scores [45], and recall is strongest for *popular* entities [46]. The tendency to favour memorized knowledge over the artifact in hand was first formalized as the model over-relying on memorized information rather than reading the provided context [47]. From another lens, Soliman and Keim's finding that GPT already contains these systems' architectures [23] is reframed as a capability. No LLM-SAR study measures or discloses contamination, which this thesis treats as its headline construct threat (Chapters 4 and 5).

The structure-riding finding has independent corroboration from outside the LLM setting. A non-LLM graph-representation recovery finds that folding in folder structure raises gold alignment by drifting from dependency cohesion [48]. The Garcia group (custodians of the canonical ground truth used in this thesis)

report that the classical state of the art, SARIF, must have its absolute-file-path input removed before a valid comparison, because that path encodes the ground-truth module boundaries [49]. On the LLM side, code models have been shown to ride high-level structural familiarity over the specific implementation, an effect that survives identifier randomization [50]. Together these results say the two shortcuts this thesis measures are not artifacts of one pipeline: folder structure leaks the answer for classical methods too, and structural familiarity rides through even aggressive surface changes.

Obfuscating real code to defeat a memorization advantage is itself established practice in the code-generation benchmark literature, and this thesis inherits the technique rather than inventing it. OBFUSEVAL rebuilds five real C projects with renamed symbols and semantics-preserving rewrites, then re-runs code generation on the result [51]; CODEMORPH applies dependency-preserving transformations across files for the same purpose [52]; and a parallel line isolates the naming channel specifically, stripping identifiers to ask what comprehension survives [53]. The transform used here is **orthogonal** to theirs rather than stronger, and the difference matters for what can be concluded. All three rewrite code identifiers, which this thesis deliberately retains because a method that cannot read function names is not performing architecture recovery; conversely, none of them alters the project name, file names, directory structure, comments and copyright headers, or identity string literals, which is what this thesis removes. Neither transform contains the other. The second difference is the dependent variable: all three measure whether the model can still *complete a task* on an isolated excerpt, none asks whether it can still recognize the project or recall its published design, and none verifies that identity was removed rather than inferring it from a performance drop.

Whether identity survives the removal of its surface is, in a broader sense, an old question with two established answers. Code stylometry attributes an anonymous program to its *author* from lexical, layout and syntactic features, at around 95% accuracy over 250 candidate programmers, and the attribution is largely unaffected by obfuscators that strip comments and rename identifiers, though an obfuscator that rewrites program structure cuts it to 67% [54]. Obfuscation-resilient library detection identifies a *third-party library, and its version*, inside Android bytecode whose identifiers have been renamed, by matching a structural fingerprint against a database profiled in advance from the original artifacts [55]. Both establish that identity survives surface transformation; neither is a language model, and in both the identity recovered is a label selected from a set assembled beforehand, never a name the method produces on its own.

Closest to the present setting, Fang et al. report in passing that GPT-4 named the Heron project, and Twitter as its origin, from source that had been de-commented and contained no occurrence of “Twitter”, and that it recognized 22 of 100 obfuscated contest entries as belonging to that contest [56]. Their study is an evaluation of code *comprehension* under obfuscation, and the recognition is an incidental observation within it: one channel removed, one keyword checked, at file granularity, with no protocol and no reported accuracy for the project case. They read the behaviour as potentially useful. The present study treats the same behaviour as a threat to the validity of architecture-recovery evaluation, and measures it.

2.6 Gap and Positioning

The literature has converged on clustering-based recovery (Garcia 2013 [1], Lutellier 2018 [3], SARIF 2023), and recent work substitutes LLMs into that pipeline (SemArc) or uses them for one-shot classification against

a fixed reference architecture [5]. What is missing is a *diagnostic* study that, in the recovery setting, asks what these methods actually do and whether their components carry weight. Such a study (a) tests whether the **KG substrate is load-bearing** for an LLM rather than assuming it. It (b) treats **run-to-run non-determinism** as a first-class object of measurement. It (c) explains *how* an LLM recovery arrives at what it produces by identifying a **structure-riding** mechanism in the file groupings and a **recitation** mechanism in the component names, using the gold-independent directory partition as a diagnostic lens (Section 2.1), and grounded in the memorization and structure-bias evidence of Section 2.5. The thesis fills this gap with the three contributions stated in Section 1.5.

3. Method

This chapter describes the recovery pipeline and the experimental design. Section 3.1 gives the three-layer overview. Section 3.2 gives the deterministic knowledge-graph (KG) extraction. Section 3.3 gives the configuration ladder (the A-Solo, A-F, A-F+KG, A-KG motivating chain plus the A-F-flat and A-KG-flat structure-riding probes) and its ablations. Section 3.4 gives the two-tier evaluation. Section 3.5 gives the metrics. Section 3.6 gives the baselines and external comparators. Section 3.7 gives the subject systems and gold-truth handling. Section 3.8 gives the file universe. Section 3.9 gives the experimental and statistical design.

3.1 Overview: Three Layers

The system is organized into three logical layers (Figure 2), each layer's output serving as the next layer's input.

1. **Ingest**: a codebase on disk is extracted into a knowledge graph in Neo4j. This runs once per codebase, is deterministic, and completes in minutes.
2. **Decompose**: the KG is turned into a hierarchical component tree (files grouped and nested). All recovery methods and the deterministic baselines live here. The LLM-based methods are stochastic and are therefore run repeatedly.
3. **Evaluate**: the component tree is scored against a human-authored ground truth. This is post-processing: deterministic, and a matter of seconds.

The key architectural property is that **the KG extraction is deterministic**, so *all* non-determinism reported in this thesis lives downstream, in the Decompose layer. The handoff from Decompose to Evaluate is a flat file-to-component partition plus a parent-child component tree, dumped from Neo4j as a snapshot file. Because every method emits the same output shape, all methods are scored identically.

3.2 Knowledge-Graph Extraction

Extraction walks every in-scope source file and uses tree-sitter [57] to pull entities and relationships into Neo4j, with one extractor per language. All three extractors share a common skeleton. `Directory` and `File` nodes mirror the on-disk tree. `CONTAINS` edges record containment (directory-to-file, file-to-entity, class-to-method). `CALLS` edges run between callable entities, each carrying a confidence score and the disambiguation rule that resolved the call. `USES` edges run from callables to the types they reference. A derived file-to-file `DEPENDS_ON_FILE` edge aggregates each language's cross-file relationships into one weighted dependency edge. On top of this skeleton each language contributes its own entity vocabulary and relational edges. The C extractor (bash, libxml2) emits `Function`, `Struct`, `Enum`, and `Union` nodes, plus file-to-file `INCLUDES` edges from `#include` directives resolved to in-repo files. The C++ extractor (ITK) emits `Namespace`, `Class` (C++ structs are folded into `Class` via a `kind` property), `Function`, `Method`, and `Enum` nodes, plus `INCLUDES`, class-to-class `INHERITS`, and an entity-level `DEPENDS_ON` aggregate. The Java extractor (hadoop, archstudio4, jpetstore) emits `Class`, `Interface`, `Enum`, and `Method` nodes, plus `IMPORTS` (from

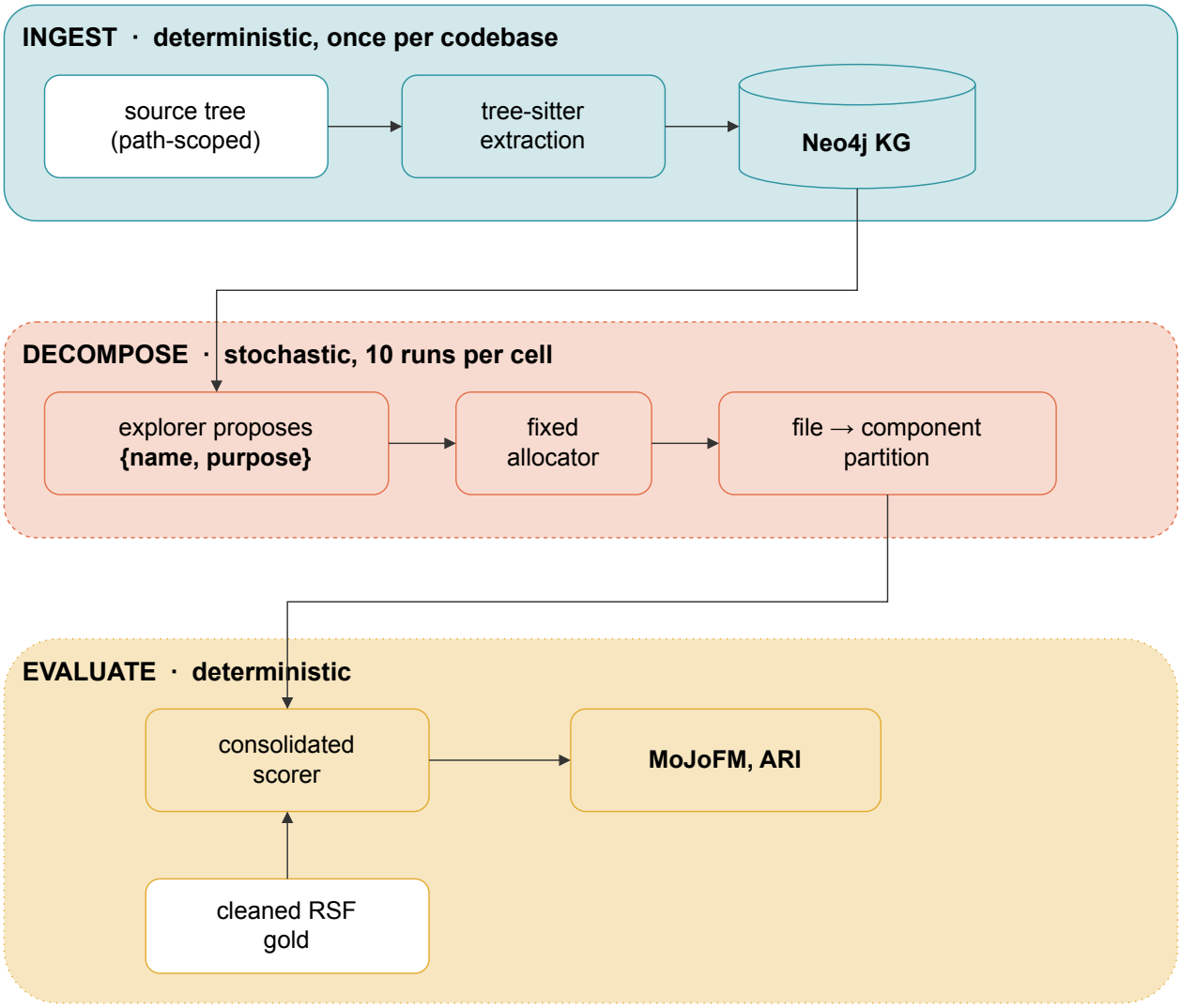


Figure 2: The three-layer recovery pipeline: deterministic ingest (tree-sitter to a Neo4j knowledge graph), stochastic decomposition (ten runs per cell. The explorer proposes components, the fixed allocator fills them), and deterministic evaluation against the cleaned gold. The fixed allocator uses path, name, and contents only. No relational edges enter allocation.

resolved `import` statements), `EXTENDS`, `IMPLEMENTS`, and entity-level `DEPENDS_ON`. Java has no `INCLUDES` and no free-function label. The `DEPENDS_ON_FILE` derivation likewise differs by language (in C it aggregates cross-file calls, type uses, and includes. In C++, it aggregates entity dependencies and includes. In Java, it aggregates class-level dependencies and imports), so the uniform file-level dependency edge is comparable across languages even though its ingredients are not identical. Figure 3 gives the per-codebase schema, with edges coloured by structural-versus-relational family. Extraction is deterministic (the same codebase produces a bit-identical graph), cheap (minutes per codebase, with ITK the largest), and cacheable (the same populated database is re-read across repeated decomposition runs).

The KG is the *substrate* every LLM-based method reads from, and it is the variable under test in the central

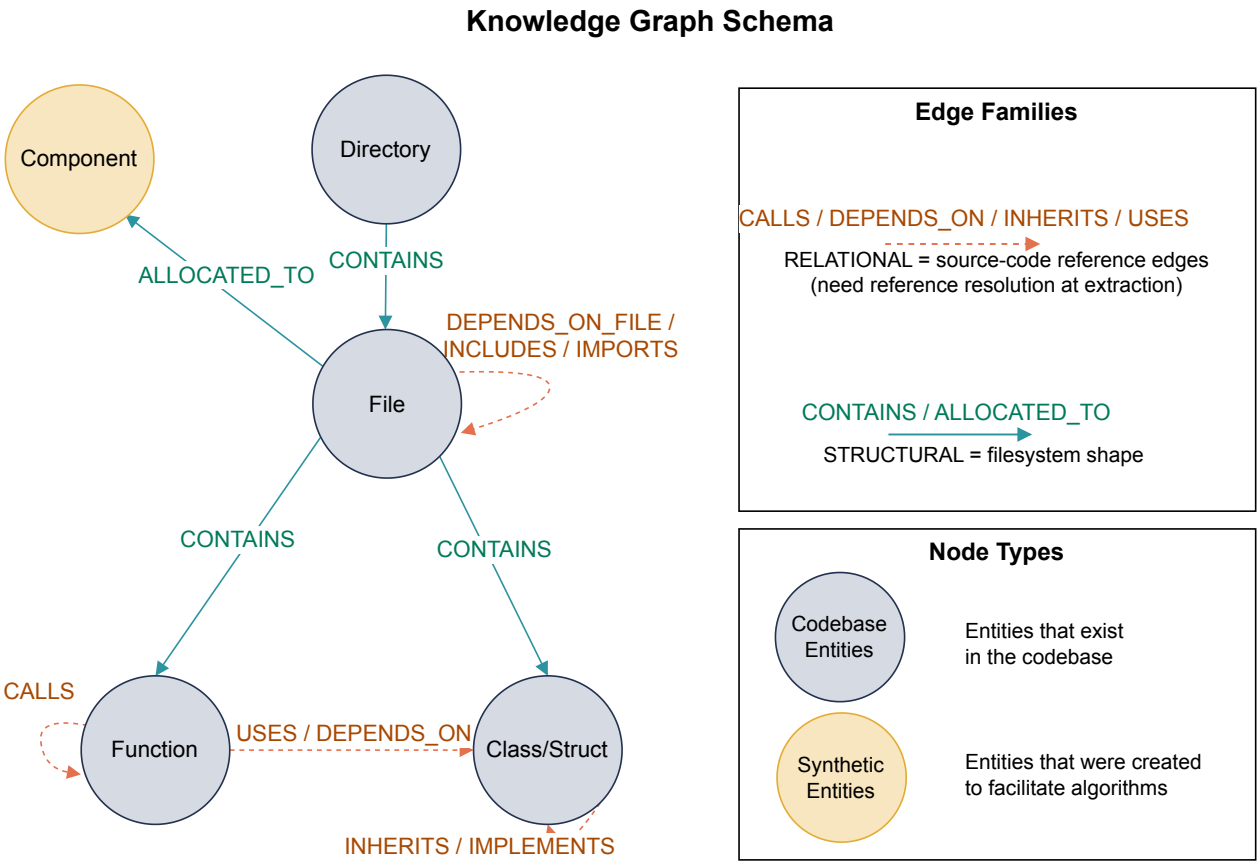


Figure 3: The per-codebase knowledge-graph schema, drawn as the union across the three language extractors (each language emits the subset described in the text), with edges coloured by family: structural (CONTAINS, Directory-to-File, ALLOCATED_TO) versus relational (CALLS, DEPENDS_ON, INCLUDES, . . .). The ALLOCATED_TO edge to the bootstrap Component{System} node is file enumeration, not the agent inspecting its own work.

ablation (Section 3.3). We distinguish the KG used as **storage and enumeration** from the KG used as a **structural-reasoning signal**: only the latter is the object of the KG-load-bearing contribution. This distinction is made precise when the fixed allocator is described below, and it is revisited as a threat to validity.

3.3 The Configuration Ladder

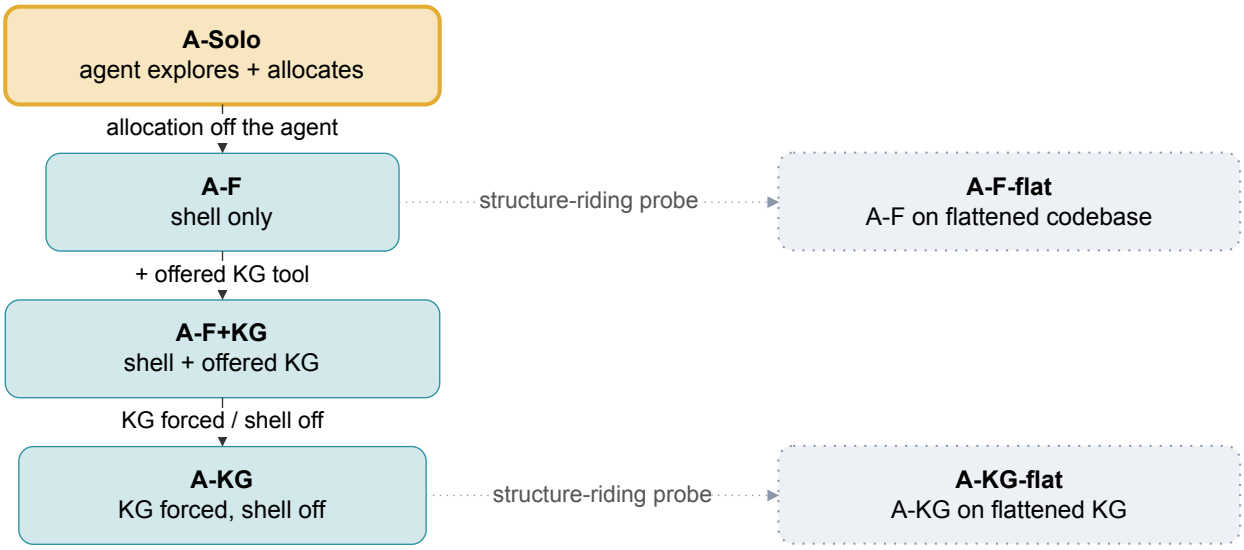
The study evaluates a configuration ladder (Figure 4) in which certain variables are changed on each rung. The gold-scored core is a **four-rung motivating chain** (A-Solo, A-F, A-F+KG, A-KG). Two **structure-riding probes** (A-F-flat and A-KG-flat) are attached to test whether the recovery survives the removal of the directory scaffolding. Every configuration operates at **file granularity**, at **temperature 0.0**, with the primary model `gpt-5.4-mini-2026-03-17`. Every rung except A-Solo shares the *same* fixed `BatchFileAllocator` (batch size 30), held constant across all quality configurations. Only the *explorer* (its tools) varies, together with the input the explorer sees for the two probes.

TABLE 1

The six-configuration ladder. The model, caps, and fixed allocator are held constant across all rungs.

Config	Explorer / scaffold	Tools in exploration	Allocator	Input	Tier / role
A-Solo	agent explores and allocates, flat	shell only	the agent itself	path-scoped tree	validity probe (Tier 1)
A-F	agent proposes components {name, purpose}	shell only	fixed allocator	path-scoped tree	quality (Tier 2)
A-F+KG	A-F plus an offered read-only KG query tool	shell **+ KG**	fixed allocator	path-scoped tree	quality (Tier 2)
A-KG	A-F+KG with the shell disabled, the KG the only lens	**KG-only** (shell forced off)	fixed allocator	path-scoped tree	quality (Tier 2)
A-F-flat	A-F, run on a directory-flattened codebase	shell only	fixed allocator	**flattened** tree	structure-riding probe
A-KG-flat	A-KG, run against the directory-flattened KG	KG-only	fixed allocator	**flattened** tree	structure-riding probe

Naming legend: the configuration codes are to be read compositionally. **A** = an LLM agent drives exploration. **F** = file/shell access. **KG** = knowledge-graph access. **-flat** = run on a directory-flattened codebase.



Reading a configuration name (e.g. **A-F+KG-flat**)

segment	what it sets	meaning
A / W	who explores	**A** = the SWE-agent explores · **W** = a scripted workflow explores
F / KG	exploration substrate	**F** = file system (shell) · **KG** = knowledge graph · **F+KG** = both
-flat	codebase	run on a **flattened** codebase (directory structure removed)
A-Solo	(exception)	the agent explores **and** allocates itself, with no fixed allocator

node role (colour *and* border pattern, so it reads in black & white):

Tier-1 validity (thick solid) Tier-2 quality (thin solid)

structure-riding probe (dotted)

Figure 4: The configuration ladder: each arrow changes one factor (allocation off the agent, offered KG, forced KG). A-F-flat and A-KG-flat are structure-riding probes run on a directory-flattened substrate. Model, allocator, granularity, and temperature are held constant.

The motivating chain yields three gold-scored ablations, each a single-variable comparison, and each answering a question the previous rung raised:

- **A-Solo to A-F**: asks whether moving allocation off the agent onto a fixed step closes A-Solo's expected validity failures (hallucinated paths and severe under-coverage, measured in Section 4.2), converting a validity question into a quality one. Everything but the allocation step is held constant.
- **A-F to A-F+KG**: asks how an *offered* read-only KG `cypher_query` tool is taken up alongside the shell, a behavioural question and **not yet** a claim about whether the KG is helpful (see Section 4.4 and the no-KG note of Section 3.3.1). Model, allocator, input, and shell access are held constant.
- **A-F+KG to A-KG**: *the central ablation*, which, by **forcing** the KG, asks whether an under-used KG

is *useless* or merely *unused*: the shell is disabled so the KG is the explorer's only lens, everything else held constant. This forced-KG rung is the test of whether the KG is **load-bearing** when actually used, the basis of Contribution 1.

The structure-riding probes (A-F-flat and A-KG-flat) are detailed in Section 3.3.3.

3.3.1 The Fixed Allocator and What "No-KG" Means

The fixed allocator (the `BatchFileAllocator`) consumes the authoritative file set, reads each file's path, name, and contents and the proposed components' names and purposes, and assigns every file to exactly one component (pseudocode in Appendix B.1). Coverage is enforced by bounded retries, and every reported run reached 100 percent. The allocation decision uses only file path/name/contents and component name/purpose: **no `CALLS`, `DEPENDS_ON`, or other inter-entity edges enter allocation**. The allocator touches Neo4j only as storage and enumeration. It is therefore *graph-structure-agnostic* when it comes to slight differences in edges and nodes due to the properties of various programming languages. The allocator is held identical across all quality configurations.

Consequently, "no-KG" in A-F denotes the **absence of KG in the agent's exploration**, not the absence of KG infrastructure: the same storage-only allocator runs underneath every configuration. This is the only sense in which "no-KG" can be meant when a fixed allocator is shared, and it is the right framing for the KG ablations. The only place graph *structure* can be exploited for reasoning is exploration (Figure 5), the variable the A-F, A-F+KG, and A-KG rungs isolate (A-F shell-only, A-F+KG shell plus an *offered* KG tool, A-KG the KG *forced* as the sole lens).

3.3.2 The Agent Chassis

The agentic configurations wrap a lightweight code agent (a mini-SWE-agent-style chassis) that reads files, runs shell commands, and, from A-F+KG onward, issues KG queries through a `cypher_query` tool. A-Solo additionally produces its own file allocation by appending {`component, file`} decisions to a log and deciding when it is complete. For the shell-bearing configurations this iterative design avoids a single-completion output-token cap, so incompleteness reflects the agent's judgement rather than a harness limit. The forced-KG rungs are the exception: with no shell they cannot append to a persistent file, so they emit their whole component set in one final submission. That is entailed by removing the shell rather than an independent choice, but it means the prompt and the output mechanism change alongside the tool surface on the A-F+KG to A-KG step. All other configurations aside from A-Solo emit only component proposals ({`name, purpose`}) and delegate allocation to the fixed allocator. A-F+KG *offers* the KG tool alongside the shell. **A-KG forces it** by disabling the shell, so the KG is the only lens.

Per-exploration budget and the not-resource-limited property. Each agent exploration runs under a step-and-cost budget, which exists for cost control and as a guaranteed-termination safety net. The budget is designed *not* to be the binding constraint on agent behaviour. The empirical evidence that agents self-terminate with ample headroom (which matters for interpreting the negative findings as behaviour rather than under-resourcing) is reported with the grid results. Context-compaction (history-summarization / scratchpad)

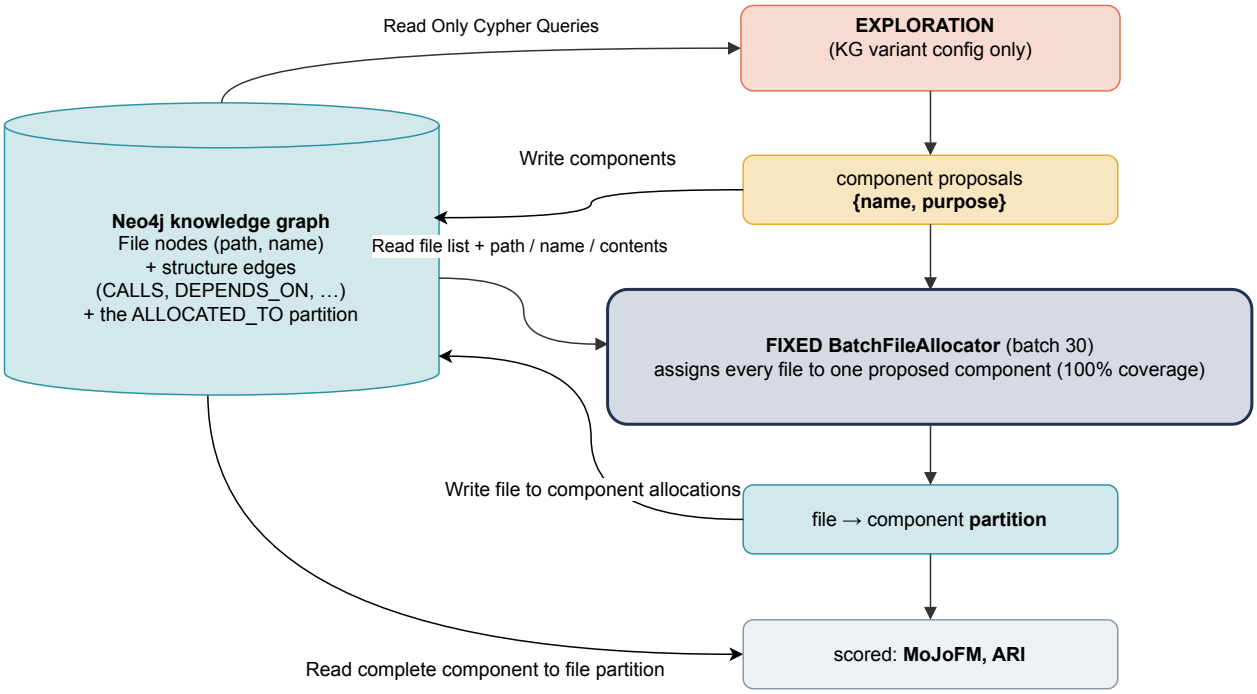


Figure 5: The fixed allocator is graph-structure-agnostic. The explorer (whose tools vary per rung) emits only component proposals {name, purpose}. The allocator assigns every file from its path, name, and contents, and no relational KG edges enter allocation. Graph structure can therefore be exploited only in exploration, the variable the A-F, A-F+KG, and A-KG rungs isolate.

was *not* added, both because context exhaustion is not the bottleneck and because a summarizer would inject summary-quality as a new confound into the controlled comparison.

The two structure-riding probes reuse this same chassis and allocator: A-F-flat is the A-F agent run against the flattened tree, and A-KG-flat is the A-KG agent run against the flattened KG (Section 3.3.3).

3.3.3 Structure-Riding Probes (A-F-flat and A-KG-flat)

Two configurations sit off the gold-scored chain and exist to ask the following question: *is the apparent recovery semantic, or is it riding the directory structure?* Both are diagnostic probes whose output is read through the directory lens (Section 3.6, threat T1). They are paired flat-variant probes that close the structure-riding leak at its two sources: the shell pathway and the Cypher pathway.

The flat-variant ablation operates on a constructed alternate substrate. It has a **flat codebase** in which every file sits in a single directory under a flat, collision-free name, and a **flat knowledge graph** extracted from that codebase in which `File.path` *is* the flat name (no directory segments beyond one constant source-root prefix, identical for every file and therefore carrying no grouping signal) and `:Directory` collapses to a single trivial root. The construction is detailed in Section 3.3.4. With that substrate in hand, two configurations test the two pathways through which the canonical configurations can ride directory structure (Table 2): **A-F-flat** closes the **shell pathway** and **A-KG-flat** closes the **Cypher pathway**, each leaving everything else identical to its parent so a collapse localizes the leak to that one interface.

TABLE 2

The two structure-riding probes: which directory-riding pathway each closes, the substrate change that closes it, and the diagnostic reading if the probe collapses relative to its parent versus survives.

Probe	Parent	Pathway closed	Substrate change	If it collapses vs parent	If it survives
A-F-flat	A-F	shell	shell retained, but file listings return flat names with no directory segments to group by	the A-F recovery was largely shell-driven directory parroting	the explorer does something beyond mirroring folders
A-KG-flat	A-KG	Cypher	KG queries retained, but path-string patterns (Section 3.5.1) return nothing useful and directory containment collapses to one trivial root	the A-KG quality was largely Cypher-driven directory parroting	the KG's relational edges, not path strings, carry the load

Two natural-sounding configurations are omitted from this pair. There is no **A-F+KG-flat**, because it would conflate the shell and Cypher pathways within a single cell and dilute attribution (the design intent is to *separate* the two leaks), and no **A-Solo-flat**, because A-Solo is a Tier-1 validity probe (Section 3.4), not a quality configuration.

Codebase scope for the flat-variant probes is **C and C++ only** (bash, libxml2, ITK). Java codebases (arch-studio4, hadoop, jpetstore) are excluded by language-resolution semantics. Directory, package, namespace, and dependency-resolution key coincide in Java, so a mechanically flat Java view either rewrites every binding (requiring a JVM-side symbol solver that is out of scope for this thesis) or produces a flat KG whose dependency edges are silently wrong (per threat T9). The Java codebases are still subjected to structure-riding analysis through the directory-lens covariate `ARI(directory_at_depth_K, gold)`, which needs no flattening (Section 3.6, threat T1). The experiments chapter handles it on that lens in Section 4.6.7.

3.3.4 The Flat-KG Construction

The flat-variant probes of Section 3.3.3 require a substrate in which both structure-riding pathways are closed at their source, and only a flat substrate (source tree and KG together) achieves this: hiding directories from the shell alone still leaks the structure through the path strings in the canonical KG, and rewriting paths in the agent's terminal alone leaves them intact for the allocator and scorer downstream.

The flat substrate is built deterministically in four post-verifiable stages (Table 3; pseudocode in Appendix B.2). Each stage carries its own verification gate, so a rewrite failure that would unfairly cripple A-KG-flat is caught before the substrate is used. The Stage-1 rename gate halts the build inline; the Stage-3 edge-parity gate is run as a separate step against the extracted graph, and is not invoked from the run path.

TABLE 3

The four deterministic stages that build the flat substrate, each with the verification gate that must pass before the next stage runs.

Stage	Action	Verification gate
1	assign every file an order-stable, collision-free flat name (a rename map)	rename map is bijective, complete, and count-consistent
2	rewrite every C/C++ `#include` directive (both the `"..."` and $<...>$ forms) to its flat target, without silent guessing	ambiguous/unresolved includes (including system headers) left intact and recorded as residuals, disclosed (threat T7)
3	re-point the canonical extractor (Section 3.2) at the flat tree	each tracked dependency-edge type present in the canonical KG (`:INCLUDES`, `:DEPENDS_ON_FILE`, `:DEPENDS_ON`, `:CALLS`) within tolerance of the canonical counts
4	at scoring time, translate the recovered flat-keyed partition back to canonical paths via the same rename map	flat and non-flat runs scored against identical gold

The Stage-2 residuals are disclosed with the results: the per-codebase totals (rewritten / ambiguous / unresolved / system) are bash 1742 / 70 / 15 / 1132 of 2959 (88% non-residual), libxml2 477 / 0 / 12 / 290 of 779 (96%), and ITK 23816 / 842 / 757 / 3136 of 28551 (97%).

The flat substrate is built once per codebase and reused across all flat-variant runs. Section 4.6 reports the structure-riding ablation: whether the flat configurations recover the gold less well, the same, or better than their canonical counterparts, and whether any drop is quantitatively predicted by how directory-aligned each codebase's gold itself is.

3.4 Two-Tier Evaluation

Evaluation is split into two tiers matched to what each configuration can be asked.

- **Tier 1 (validity / feasibility)** applies to **A-Solo only** and is gold-truth-free. It measures completion, coverage / missed-file rate, hallucinated-file rate (split into *nonexistent* paths and *out-of-universe* real files), premature-completion incidence, granularity (component-count) spread, revision churn, and process cost.
- **Tier 2 (quality)** applies to the gold-scored quality rungs **A-F, A-F+KG, and A-KG** (ten runs per configuration-by-codebase cell) and, diagnostically, to the structure-riding probes **A-F-flat** and **A-KG-flat** (run counts in Section 3.9, probe results read through the directory lens of Section 3.6). These configurations are valid in every reported run (the fixed allocator's validation filter admits no hallucinated paths, and its retry loop reached 100 percent coverage throughout, Appendix B.1) and skip Tier 1.

3.5 Metrics

Three metric families are reported, summarized in Table 4: a primary gold-scored similarity (ARI), a secondary gold-scored similarity (MoJoFM), and, recorded as a reproducibility disclosure rather than a study axis, run cost. Each carries a caveat specific to this benchmark suite.

Two further instruments, defined in Sections 3.5.1 and 3.5.2, measure the behaviour that produced a partition rather than the partition itself: which edges the agent queries, and where its component names come from. Both report classifications and token-overlap counts, not quality scores.

TABLE 4

The three metric families, their role in the study, how each is computed, and the key caveat each carries on this benchmark suite.

Metric	Role	How computed	Key caveat on this suite
ARI [36]	primary quality	`sklearn`. Chance-corrected, recovery treated as set partitioning	discriminates far better than MoJoFM here (observed range roughly 0.10 to 0.82)
MoJoFM [37]	secondary quality	canonical Wen-Tzerpos optimal via the standard `mojo.jar` (not a greedy approximation, which diverged materially). Direction fixed as MoJoFM(recovered, gold)	joins are free and the scale is compressed near the top. Caveats developed in Chapter 4 and Threats

Metric	Role	How computed	Key caveat on this suite
Cost	reproducibility disclosure	raw per-call tokens, dollars, wall-time + a single documented pricing table, applied uniformly	reader can recompute at their own rates

3.5.1 Behavioral Metric: Cypher Query Classification

The quality metrics above ask what the recovered partition looks like, not what the *agent did* to produce it. For the KG-enabled configurations (A-F+KG, A-KG, and later the Cypher query classification measures the behavioural question: when the agent can, or must, query the KG, does it exercise the **relational** source-code edges that only reference resolution at extraction time can provide, or does it issue queries whose returns approximate `find` or `grep`?

A note on the KG's bootstrap state One property of the KG at agent time must be recorded before the classifier is defined. Every per-codebase KG ships with a single bootstrap node, `(:Component {name:'System'})`, to which the entire file universe is pre-allocated via `:ALLOCATED_TO` edges *before the agent or the fixed allocator runs*. The agent's own component proposals are written to a JSONL file during exploration and become `:Component` nodes only at the end of the exploration phase. Consequently, queries of the form

```
MATCH (f:File)-[:ALLOCATED_TO]->(c:Component {name:'System'})
RETURN f.path ORDER BY f.path LIMIT 50
```

are **enumeration of the file set through the bootstrap System component**, semantically equivalent to `find /work -type f | sort | head -50`. They are classified as STRUCTURAL, not as introspection of agent state.

Four categories Each `cypher_query` invocation across the corpus is extracted from the agent's trajectory and classified into one of four categories, defined in Table 5 by the edges their MATCH pattern touches and the information the query returns. (Table 5 gives the category *contents*. The order in which the tests are applied is the decision rule of Figure 6.)

TABLE 5

The four cypher-query categories: the edges each category's MATCH pattern contains, what the query is equivalent to, and where examples are located.

Category	MATCH-pattern edges	Equivalent to / what it returns	Examples
STRUCTURAL	only structural edges (`:CONTAINS`, or `:ALLOCATED_TO` against the bootstrap System component) or no edge. Path-string WHERE ops (`STARTS WITH`, `ENDS WITH`, `CONTAINS`, `split(path,'/')[n]`) count as structural	what a filesystem walk plus a parse pass could equivalently produce	Appendix A.7
RELATIONAL	one or more source-code relational edges (`:CALLS`, `:DEPENDS_ON_FILE`, `:DEPENDS_ON`, `:INCLUDES`, `:IMPORTS`, `:INHERITS`, `:IMPLEMENTS`, `:USES`)	information only extraction-time reference resolution can provide	Appendix A.7
HYBRID	a relational edge **and** a structural edge co-traversed, canonical shape `(Container)-[:CONTAINS]->(Unit)-[:CALLS]->(Unit)`	relational structure scoped by container	verbatim example below
SCHEMA	introspection (`CALL db.labels()`, `CALL db.schema.visualization()`, generic counts)	the shape of the KG itself, not the codebase	Appendix A.7

Decision rule Classification proceeds by tokenising the query on Cypher keywords (MATCH, WHERE, RETURN, WITH, ORDER BY, LIMIT, SKIP, UNION, CALL) and identifying which segments are MATCH segments versus WHERE segments. The rule is then (Figure 6):

1. If any introspection token is present (`db.labels`, `db.schema`, `db.relationshipTypes`), classify as

SCHEMA.

2. Else, scan MATCH segments for relational and structural edges.
    - If MATCH segments contain *both* a relational edge AND a structural edge, classify as HYBRID.
    - Else if MATCH segments contain a relational edge, classify as RELATIONAL. (Structural patterns in the query's WHERE segments do not change this classification.)
    - Else, classify as STRUCTURAL.

The decision rule is permissive about structural filters in WHERE clauses. A query of the form `MATCH (f:File)-[:DEPENDS_ON_FILE]->(g:File) WHERE f.path STARTS WITH 'hadoop/mapred/' RETURN ...` is RELATIONAL because the substantive operation (the dependency traversal) is relational. The WHERE filter restricts the scope but does not change what kind of structure the query is computing. HYBRID is reserved for coordinated co-traversals, not for scope restrictions of this kind.

Worked examples Two verbatim examples from the corpus illustrate the poles of the classification. Further examples (with the naming conventions for locating the trajectories) are collected in Appendix A.7, and the full query log is preserved with the per-run artifacts. Bracketed identifiers give a short-id to the codebase and run. The short-id can be used to identify the particular run in the experiments artifact.
STRUCTURAL: bootstrap-component enumeration, semantically a paged file listing:

```
[hadoop/54fe64e8]  MATCH (f:File)-[:ALLOCATED_TO]->(c:Component {name:'System'})
                   RETURN f.path AS path ORDER BY path SKIP 100 LIMIT 100
```

This is the shape that looks like introspection of the agent's own allocation but, per the bootstrap note above, is file enumeration: a paged `find`.
HYBRID: coordinated container-and-call co-traversal:

```
[bash/4961f9ed]    MATCH (f:File)-[:CONTAINS]->(fn:Function)-[:CALLS]->(callee:Function)
                   WITH f, count(DISTINCT callee) AS calls
                   RETURN f.path, calls ORDER BY calls DESC LIMIT 20
```

The MATCH pattern joins the structural `:CONTAINS` edge to the relational `:CALLS` edge in a single traversal. This is the canonical HYBRID shape in our corpus.
Trajectories carry no run identifier, so they are matched to scored runs by the set of component names the explorer proposed, which both the trajectory transcript and the run's allocation log record independently. The match is exclusive, one trajectory per run. Timestamps cannot serve as the key here: grid runs launch in parallel, so several runs share a start second, and a trajectory is written before its run ends. Every scored run in the A-F, A-F+KG and A-KG corpora matches, so the behavioural and quality corpora coincide (matching mechanics in Appendix A.7).

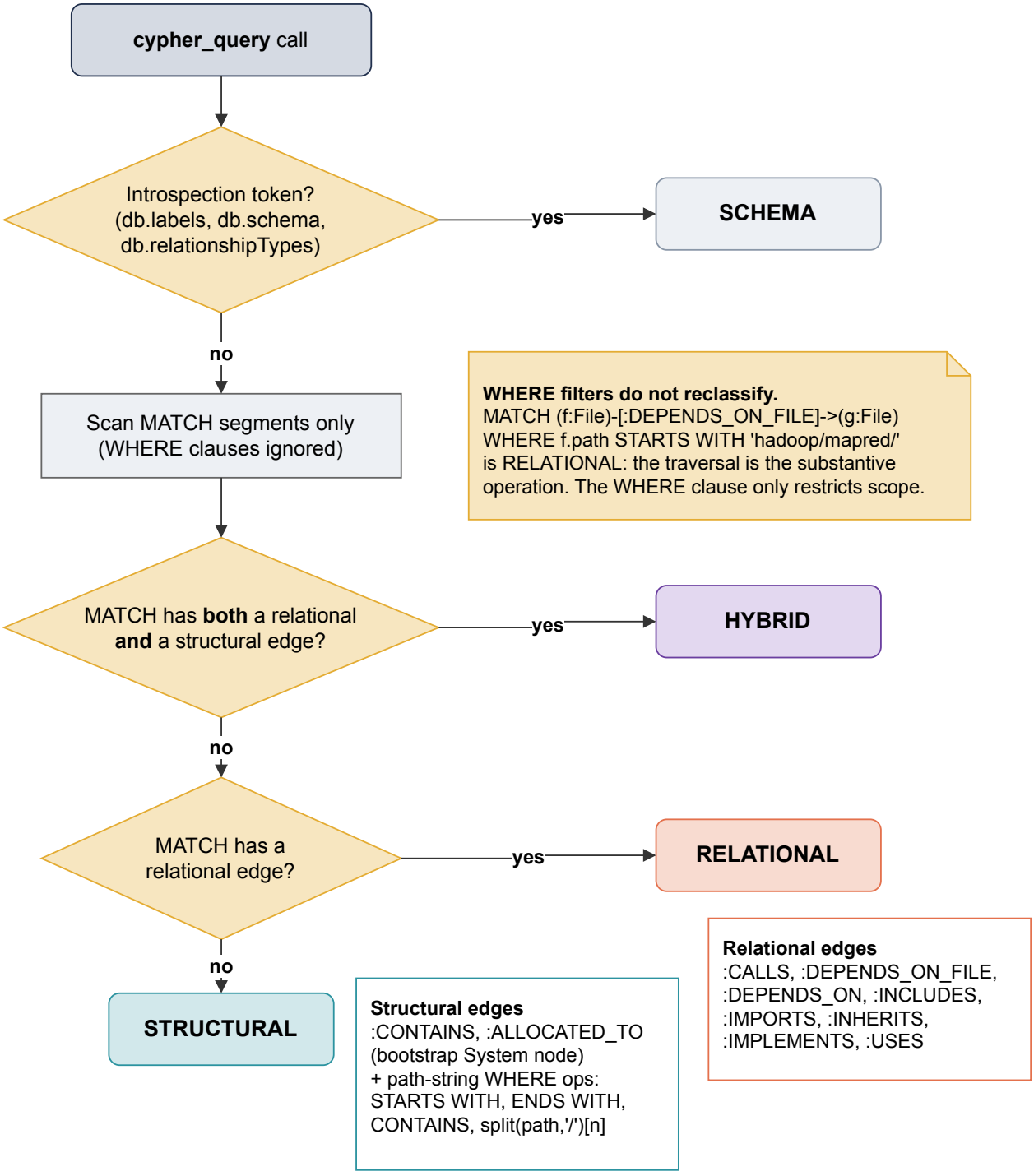


Figure 6: Classification of a `cypher_query` call. The query is tokenized and its MATCH segments are scanned (WHERE clauses ignored). The first matching rule wins, tested in the order SCHEMA, HYBRID, RELATIONAL, STRUCTURAL. Structural edges and path-string filters count as structural. The relational edges are the KG's source-code references.

Reporting conventions For each configuration and codebase we report (i) the matched-run count, (ii) the total cypher_query call count, (iii) the per-category breakdown, and (iv) the relational+hybrid fraction of total cypher calls, the single summary number for "how much is the agent using the KG for what it was built for" (RELATIONAL and HYBRID together capture every query that exercises a source-code relational edge).
The classifier is mechanical and does not infer intent: it asks what the surface form of the query is, not what the agent meant by it. The harness does not persist the agent's reasoning, so a classifier cannot read intent without projecting it, and this one does not try (threat T6). The behavioural claims the paper makes from these counts are framed accordingly: "the agent's queries are predominantly structural" rather than "the agent does not understand the KG."

3.5.2 Behavioral Metric: Canonical-Overlap (Recitation and Structure-Riding)

The Cypher classification (Section 3.5.1) measures how the agent *explores*. The canonical-overlap metric measures what its *output names* are made of. It is the quantitative backbone of the RQ3 recitation finding (Chapter 4). For each run we take the set of component *names* the method proposes and compare that name-set, by token-set overlap, against two references:

- the **zero-shot elicitation probe**: the component names the same model emits for the same system when given no codebase (only the system's name). High overlap here indicates **recitation** (the names are recalled canonical architecture, not read off the artifact).
- the **directory vocabulary**: the set of path segments in the real source tree. High overlap here indicates **structure-riding** (the names are lifted from folder names).

The headline quantities are token-level **containment** $|M \cap R| \,/\, |M|$ (what fraction of the method's name tokens appear in the reference, the recitation / structure-riding signal) and **coverage** $|M \cap R| \,/\, |R|$ (what fraction of the reference the method reproduces). We report both, plus a name-level match table for intuition. Normalization is **light**: lowercasing, splitting on non-alphanumerics, a short generic stopword list (`system`, `core`, `util`, `manager`, and the like), and plural-s folding, with **no hand-tuned synonym or alias table**. We confirm the cross-codebase ranking is stable when the stopword list is removed (a `--stopwords none` sensitivity pass).
The metric is reported three ways in Chapter 4: a main-grid containment table (probe versus directory vocabulary, per configuration and codebase), a crossed comparison of identity stripping against directory flattening on the two C study systems, and a cue-presence count that separates names that rode an in-run cue from those recited cold. The scorer is deterministic given the run snapshots and the probe (pseudocode in Appendix B.4), and every reported number is regenerable from it.

3.6 Baselines and External Comparators

Directory baseline as a diagnostic lens (gold-independent). The directory baseline groups files by the directory that immediately contains them: one component per directory of declaration. It is used not as an opponent to beat but as a **diagnostic lens** for how much of any recovered output, or of any gold, is

already explained by folder structure alone, and, together with the structure-riding probes of Section 3.3.3, for whether the recovery survives when the folder scaffolding is removed. It is equivalent in construction to the package/directory ("PKG") baseline of Lutellier et al. [3], while remaining gold-independent. Because the baseline depends on the path-prefix depth used to group files, the **immediate-parent-directory value is the locked comparator, fixed in advance**, and the full depth sweep is disclosed as a descriptive sensitivity figure (threat T1).

Louvain baselines. Two deterministic non-LLM baselines are reported at one run each: Louvain community detection on the basic file-level dependency graph (the `DEPENDS_ON_FILE` edges), and Louvain on the enriched KG projected to file level (every file-file relational edge type plus the entity-level relational edges lifted to their containing files, weights summed per file pair). The latter addresses the question "is the KG doing the work, or the LLM?". Both use a seeded, deterministic Louvain runner (seed fixed at 42, so the run is reproducible, pseudocode in Appendix B.5) with the pipeline's three-tier small-community policy (isolated files form an orphan bucket. Communities below 5 percent of the file universe are kept if disconnected, otherwise merged into their strongest neighbour), and are scored through the same consolidated scorer as every other partition (Section 3.5).

External comparators (qualitative specimens, not rivals). We treat SARIF [11], the strongest classical, non-LLM recovery, and SemArc [4], the closest LLM-augmented recovery, as **qualitative specimens** of the surrounding literature. This study performs **no quantitative head-to-head**. Neither a rescore of their shipped outputs under our metrics nor a matched-model rerun is carried out. Their metric conventions, gold versions, and file universes differ from ours, and bringing their shipped outputs onto this study's apparatus is future work (Section 6.3). The positioning against both is developed qualitatively in Section 4.9.

3.7 Subject Systems and Gold-Truth Handling

The four **core** subject systems are the canonical benchmark systems **bash 4.2 (C), hadoop 0.19 (Java), archstudio4 (Java), and ITK 4.5.2 (C++)**. These are four of the systems used by Lutellier, overlapping the SARIF and SemArc evaluation suites, each with a Rigi Standard Format (RSF) developer-authored ground truth. Two **supplementary** systems of different provenance are included as disclosed structure-riding exhibits: **libxml2** (a flat repository whose gold is not directory-aligned) and **JPetStore** (a microservice-decomposition gold whose business services cut across the layer directories). Neither supplementary gold is developer-authored. libxml2's is the labelled decomposition released with SARIF [11], produced by that project's industrial collaborator; JPetStore's is the expert microservice decomposition of Zaragoza et al. [12], digitized as a class-to-service table in the independent tool comparison of Wang et al. [13].

The gold-truth handling follows the corrections locked for this study. The canonical Rigi-Standard-Format golds inherited from the Lutellier-derived corpus are not all valid single-membership partitions as shipped (multi-membership files, typographic component-name splits, and generated or phantom entries), and both ARI and MoJoFM are undefined for overlapping clusters, so each gold was cleaned, *per codebase*, into a valid single-membership partition, with one general rule applied uniformly: **drop any gold entry that has no on-disk source file**. Every count below was verified against the source RSF files. The original golds were left untouched and the cleaned golds written to a separate location, so the cleaning is reproducible and

auditable. Appendix A gives the complete line-by-line change log, distractor tails, and singleton-component counts. The headline fixes per codebase are collected in Table 6.

TABLE 6
The headline gold-cleaning fixes per core codebase and the resulting valid single-membership partition. The full change log is in the cited appendix section.

Codebase	Headline fixes	Files	Components	Log
bash	seven multi-membership `lib/readline` files dropped. Phantom entry `builtins.c` remapped to its on-disk path	356	13	A.3.1
hadoop	shipped *cover* resolved to a partition (numbered architectural label preferred, 31 truly multi-architectural files excluded). 24 phantom/generated entries dropped	560	62	A.3.2
ITK	`group_numberics`/`group_numerics` typo merged. Vendored `group_thirdParty` set (2,428 files) excluded. One non-source entry dropped	5,228	11	A.3.3
archstudio4	already a valid single-membership partition, unchanged	583	57	A.3.4

ITK's gold *is* its developer module structure (an easy / special case, disclosed in Threats), so conclusions are weighted toward bash, hadoop, and archstudio4.
The two supplementary systems were also checked: JPetStore's lone duplicate (`AbstractActionBean`, a shared base class) is disclosed and excluded, and libxml2's group/item hierarchy is clean, so neither needed RSF-style cleaning.

3.8 The File Universe

How the file universe is selected. The file set each method is scored over is chosen by per-codebase path rules about where the source lives (for example, hadoop is the `src/core`, `hdfs`, and `mapred` subtrees), applied without consulting the gold. The rules are tabulated in Appendix A.2, and ITK's universe is the full source under `Modules/` (8,547 files). Non-architectural distractor files (tests, build helpers, utilities, vendored code) are retained. The pipeline must itself avoid mis-assigning them. Metrics are computed over the intersection of the recovered partition with the gold, which is standard practice (the rationale is given in Appendix A.1).
Locked directory-baseline characterization numbers. Over the cleaned golds (Section 3.7) and these universes, the immediate-parent-directory baseline scores the following canonical MoJoFM and ARI values against the gold. These are **baseline-characterization numbers, not grid results**: they characterize how directory-aligned each gold is.

TABLE 7
Immediate-parent directory baseline per codebase (canonical MoJoFM and ARI against the cleaned gold): a characterization of how directory-aligned each gold is, not a grid result.

Codebase	Gold files	Directory-baseline MoJoFM	Directory-baseline ARI
bash	356	0.939	0.817
hadoop	560	0.691	0.392
archstudio4	583	0.832	0.252
ITK	5,228	0.945	0.151

These numbers carry two points into Threats: the gold-based metrics on this suite largely measure directory recovery (the golds are substantially directory-aligned), and **MoJoFM is heavily compressed** (the directory baseline alone scores about 0.94 on bash and ITK), so ARI discriminates far better. The grid and all scoring use the cleaned golds and these universes.

No numeric comparison with Lutellier's published baseline. Our directory baseline is reported as a self-computed comparator only, not against Lutellier's published "directory structure" MoJoFM: a forensic reproduction showed the gap (our 0.94 versus their 0.57 on bash) to be a file-universe construction difference, their full extracted tree carrying a non-architectural file tail that our path-scoped trees do not.

3.9 Experimental and Statistical Design

The Tier-2 grid is **ten runs per (configuration x codebase) cell** for the gold-scored quality rungs A-F, A-F+KG, and A-KG across all six subject codebases (the four core systems plus the two supplementary substrates, libxml2 and JPetStore), at **temperature 0.0**, with the primary model `gpt-5.4-mini-2026-03-17`. The two structure-riding probes are run alongside the chain: **A-F-flat** on the directory-flattened codebases and **A-KG-flat** on the flattened KG, each on the three C/C++ codebases only (§3.3.3); their per-cell run counts are given with the results in §4.6. The design is **single-model throughout**: every executed configuration uses the primary model above, conclusions are scoped to it, and testing whether the diagnostic findings generalize to other model families is left to future work (Section 6.3). The external-validity threat is recorded in Chapter 5. A-Solo is a Tier-1 validity probe, budgeted at roughly three to five runs per codebase. The probe proved cheap enough that the executed corpus matches the grid at ten runs per codebase (Section 4.2). Deterministic baselines (Directory and Louvain) are run once each.

Results are reported as **distributions**: per cell, the median, inter-quartile range, and minimum/maximum over the ten runs, plus the worst-best gap. Named configuration comparisons are reported as **paired per-codebase median deltas** read against the within-cell run-to-run spread (a delta inside the spread is called near-noise, not a difference). We do not report **omnibus or pairwise significance tests**: with six codebases, two of which are structurally degenerate for quality scoring (Section 4.3.4), the paired-sample size is too small for such tests to carry evidential weight, and reporting them would lend false precision. Every method is reported **alongside the per-codebase directory reference** (the diagnostic lens of Section 3.6) in addition to the gold (threat T1). Every run is logged with a fixed per-call and per-run schema so that all figures can

be regenerated offline from cached artifacts.

Scoring conventions. Every recovered partition is scored through one consolidated scorer (pseudocode in Appendix B.3): deterministic leaf-component selection from each run's hierarchical snapshot, key alignment chosen mechanically among three fixed namespace transforms (as-is, drop-first-segment, basename) by key-set overlap with the gold's file keys (the chooser aligns namespaces and does not see the scores), and coverage reported alongside both metrics, because both are computed on the recovered-gold intersection (threat T5).

Cost budgeting. Per-run cost scales steeply with codebase size, and ITK dominates the grid: pilot runs cost about $0.87 per run on bash but about $7.97 per run on ITK (file counts as staged for those pilots. The locked universes of Section 3.8 differ slightly), putting the full design on the order of several hundred dollars. We keep ten runs per cell on all six codebases rather than trade statistical power for cost. The single-pricing-table convention of Section 3.5 keeps the disclosed costs directly comparable across configurations and recomputable at a reader's own rates.

External comparators are not re-run. No matched-model comparison or output rescore of SemArc or SARIF is part of the executed design, and a quantitative rescore is left as future work (Section 6.3).

4. Experiments and Results

This chapter reports the executed study: the Phase-1 configuration grid (the A-Solo validity probe, §4.2; the quality rungs A-F / A-F+KG / A-KG, §§4.3 to 4.5; and the flat-variant ablation, §4.6), the Phase-2 recitation study on bash and libxml2 (§4.7), then the directory-lens attribution (§4.8), comparator positioning (§4.9), cost (§4.10), mechanically-selected qualitative cases (§4.11), and the synthesis (§4.12). Every quantitative value traces to the scored-run ledger or the recitation runs, recomputed through the single consolidated scorer of §4.1.

4.1 Experimental Setup and Protocol

The study executes the locked single-variable design of Method §3.3: a four-rung gold-scored chain (A-Solo, A-F, A-F+KG, A-KG) in which each rung changes one factor from its predecessor (where file allocation happens, whether a knowledge graph is offered, whether it is forced by disabling the shell), plus the two structure-riding probes A-F-flat and A-KG-flat. The subject systems are the four core canonical benchmark systems **bash 4.2, hadoop 0.19, archstudio4, and ITK 4.5.2** plus the two supplementary systems **libxml2 and JPetStore** (six in total) for the Phase-1 analyses, and **bash and libxml2** (both C) for the Phase-2 recitation study. Per-codebase gold, universe, and distractor counts and the directory baselines are fixed in Method §3.8.

All configurations use **gpt-5.4-mini-2026-03-17 at temperature 0**, File granularity, and the same fixed `BatchFileAllocator` (batch size 30), except A-Solo, which self-allocates. The Tier-2 quality rungs (A-F, A-F+KG, A-KG) are scored at **n = 10** per (config, codebase) cell. A-Solo is a Tier-1 validity probe reported diagnostically (§4.2). The recitation study is **n = 10** per cell on the real and deeply identity-stripped substrates, and **n = 4 to 5** on the intermediate stripped substrate (§4.7). The directory multi-depth partition is the deterministic reference baseline (§4.8). No run approached the $1.00 cost cap on any agent exploration (the maximum instance cost is $0.06) and no Tier-2 result is resource-limited; the two exceptions are A-Solo runs that terminated on agent-side resource limits rather than submitting (§4.2.2).

Every metric is recomputed through a **single consolidated scorer**: canonical `mojo.jar` MoJoFM, scikit-learn ARI over the intersection of file keys, coverage beside both, one fixed path transform, and deterministic leaf-component selection from each run's KG snapshot (Method §3.5, under the scoring conventions of §3.9). Only pipeline-`exit_status=success` runs are scored. Per-cell logging captures tokens, USD, wall-time, and the dumped KG snapshot.

4.2 Tier 1: Validity Probe (A-Solo)

A-Solo is the only rung where the agent both explores the codebase and allocates each file to a component itself. As established in Section 3.4, it is a *validity* probe, scored gold-truth-free against the authoritative file universe rather than the gold's component labels. The question Tier 1 asks is narrow: when an unconstrained LLM agent is left to self-allocate, does it produce a *valid* partition at all (covering the codebase, drawing only on files that exist in the authoritative universe, with stable enough output that a reader could even compare runs)?

4.2.1 Corpus and Setup

Each of the six subject systems (bash, hadoop, archstudio4, ITK, libxml2, jpetstore) was run under the canonical A-Solo configuration of Section 3.3.2 (minimal JSONL self-allocation prompt, caps matched to the rest of the ladder), across 10 independent runs per codebase, for 60 runs in total. The authoritative file universe is the cleaned gold RSF per Section 3.7. Run artifacts are persisted per run and re-scored offline, so the validity metrics reported here are reproducible without re-running the agent. Fifty-eight of the 60 runs completed with the agent submitting at a fraction of the per-run cap; the remaining two terminated on agent-side resource limits (one `LimitsExceeded`, one `ContextWindowExceededError`).

4.2.2 Coverage and Output Validity

The principal Tier-1 outcomes are *coverage* (the fraction of the authoritative file universe the agent assigns to a component) and *output validity* (whether the agent produced a parseable allocation log at all). Coverage of zero is reported as a *collapse* (the run produced no usable partition). Coverage strictly between zero and one is a *partial* run (the agent stopped before allocating the full universe). Coverage of one is a *full* run. Table 8 summarizes the 60-run corpus per codebase.

TABLE 8

A-Solo validity over 10 runs per codebase, scored against the cleaned gold universe (Section 3.7). *Coverage* is the fraction of the universe allocated. *Collapse* means a run produced no usable partition (coverage 0). Component-count and coverage statistics aggregate the full 10 runs per row, including any collapsed runs.

codebase	universe	coverage mean	coverage range	n_components range	collapsed
bash	356	0.813	[0.49, 1.00]	[7, 21]	0 / 10
libxml2	82	0.683	[0.52, 0.99]	[7, 12]	0 / 10
archstudio4	583	0.638	[0.00, 1.00]	[0, 24]	1 / 10
hadoop	560	0.531	[0.00, 1.00]	[0, 25]	2 / 10
jpetstore	23	0.700	[0.00, 1.00]	[0, 8]	3 / 10
ITK	5228	0.317	[0.00, 1.00]	[0, 140]	3 / 10

Three observations follow from the table.

A-Solo's output is non-deterministic at temperature 0. Across the 60 runs, **9 are collapses** (15%, no usable partition), **19 reach full coverage**, and **32 are partials**. Only about one run in three allocates the whole codebase; the agent makes a different stop-or-continue decision on each draw despite identical input and decoding temperature.

Codebase size does not predict where the agent fails. The lowest-collapse codebases (bash and libxml2, 0 / 10 each) differ by more than a factor of four in file count. The highest-collapse (jpetstore and ITK, 3 / 10 each) differ by two orders of magnitude. What organizes the data is instead the *kind* of failure each codebase elicits (Table 9).

TABLE 9

The A-Solo failure modes per codebase over the 10 runs, and their signatures. Failure tracks codebase character; file count does not predict it.

Codebase	Failure mode	Signature / mechanism
ITK	window overrun and prefix drift	of three collapses: one raises `ContextWindowExceededError`; one emits 256 allocations under a `Modules/`-stripped prefix, none of which land in the universe; one writes a 21-line log carrying no allocation records
jpetstore	empty component names and prefix drift	of three collapses: one submits nothing; one submits 139 well-formed records whose component name is empty, so no component can be formed; one emits 117 allocations under a `src/main/java/`-stripped prefix, none of which land in the universe
hadoop	empty component names	two runs write allocation logs (314 and 11 lines) that yield no usable partition: 299 of the longer log's records carry an empty component name, and the shorter contains no allocation records
archstudio4	malformed record syntax	one run emits all 609 of its allocation records single-quoted rather than as JSON, so none parse (coverage 0)
bash	occasional partials	no collapse; no dominant mode
libxml2	all-partial coverage ceiling	`find -maxdepth 2` blind to depth-3 `include/libxml/` (detailed below)

libxml2 is a pattern in itself: all ten runs land in the partial band (coverage 0.52 to 0.99). The agent's

standard opening command, `find /work -maxdepth 2 -type f`, is structurally blind to depth 3, where 35 of libxml2's 82 gold files (43%) sit inside `include/libxml/`. Runs that issue no deeper listing hit a coverage ceiling near the **47 / 82 ≈ 0.57** predicted by allocating everything outside `include/libxml/`, and only runs that diversify exploration clear it. In no run does the agent notice that its standard exploration left a stable region of the codebase unexamined.

The failure surface is the self-allocation step. Across all 41 non-full runs the agent reads the codebase before failing: what fails is the *contract* by which it emits its allocation (empty component names, file lists that stop mid-universe, a context-window overflow, a submission with no allocation lines). None of these implicates the LLM's ability to recognize source files. They implicate its ability to do its own bookkeeping for several hundred file decisions in a single session without dropping part of the job.

4.2.3 Granularity Spread

Even on runs that succeed, the agent's *component count* is unstable: excluding collapses, it varies by a factor of between roughly two and three on three of the four non-extreme codebases (libxml2 is tighter, at 1.7) on identical input at temperature 0, and at the extremes by eight-fold (jpetstore) and by an order of magnitude (two full-coverage ITK runs propose roughly 12 and 140 components). Partition shape is also unstable (one ITK run puts 89% of its own allocations, and 83% of the gold's files, into a single bucket). A single A-Solo run is therefore not a representative point estimate, and the convention of reporting LLM-driven SAR as a single run hides this variance.

4.2.4 Hallucinated and Out-of-Universe Allocations

Per Section 3.4, hallucinated allocations split into *nonexistent* and *out-of-universe* paths. Confabulated paths are rare: 52 of the 60 runs record zero, and libxml2's maximum of 32 consists of directory names allocated as files (`result/HTML`, `result/SVG`). Out-of-universe allocation, however, is neither rare nor benign. Two runs, one on ITK (256 allocations) and one on jpetstore (117), emitted well-formed, cleanly parsing allocation logs in which *every* path had lost its universe prefix (`Modules/` and `src/main/java/` respectively), so both score coverage 0 despite having allocated most of the codebase. They account for two of the nine collapses.

A-Solo's instability is therefore two distinct failures. The dominant one is *under*-coverage of real, in-universe files: the agent drops parts of the codebase it has already seen. The rarer one is a complete answer emitted against the wrong path root, which is indistinguishable from total failure under any file-key-matching metric.

4.2.5 What the Validity Probe Establishes

Across ten independent draws per codebase, an unconstrained LLM agent left to allocate the codebase by itself produces *unreliable* output at temperature 0: in coverage, granularity, partition shape, and whether a parseable submission appears at all. Instability in coverage, granularity and partition shape appears across all six codebases, both language families, and the full size range; outright failure to produce a usable submission is confined to four of the six (hadoop, jpetstore, ITK, and one archstudio4 run). The unreliability is structural to the self-allocation step specifically. Tier 1 does *not* speak to whether the decompositions are architecturally

good. Those questions are deferred to Tier 2, which is only meaningful once a configuration is *valid by design*. The localization of A-Solo's failures suggests the next step: isolate the LLM's job to *proposing components* and move file allocation to a deterministic procedure that consumes the authoritative file set directly. This is **A-F** (Section 3.3, Table 1), which has full coverage through the allocator's bounded-retry enforcement (Appendix B.1), cannot hallucinate paths, and cannot produce a malformed allocation log, so it skips Tier 1.

4.3 A-F: Component Proposal with Fixed Allocation

A-F is the first Tier-2 configuration: the agent reads the codebase via shell and emits a list of components (each as {`name, purpose`}), and the fixed `BatchFileAllocator` (Section 3.3.2) assigns every file in the authoritative universe to one of those components in 30-file batches. The prompt is A-Solo's, *verbatim*, minus the file-allocation clause. The model, caps, allocator, and universe are held constant across the rest of the ladder. The single variable changing between A-Solo and A-F is therefore *where file allocation happens*: inside the agent or in a deterministic procedure outside it.

4.3.1 Corpus and Setup

Each codebase was run ten times under the canonical A-F configuration, for 60 runs in total. Run artifacts are persisted per run, including a per-run KG snapshot with the full partition and hierarchy. Scoring is performed offline from the snapshots, under the canonical leaf-extraction rule of Sections 3.5 and 3.9, restricted to pipeline-`exit_status=success` runs.

4.3.2 Validity by Construction

A-F delivers on its design promise: **all 60 runs are pipeline-`exit_status=success`, all 60 agents `Submitted` without hitting the step or cost cap, coverage is 1.00 in every run (the allocator's retry loop left no file unallocated), and no run produces a hallucinated path or a malformed allocation log.** Every failure mode catalogued in the A-Solo validity probe (Section 4.2) is removed by moving file-allocation off the agent.

4.3.3 Cross-Codebase Quality

Table 10 reports per-codebase quality across the corpus.

TABLE 10

A-F quality per codebase. ARI (sklearn) is the primary similarity-to-gold metric. Canonical MoJoFM (`mojo.jar`, Wen-Tzerpos optimal) is the secondary. Component count is the run's leaf-component count (Section 3.5). Numbers are computed from the scored-run ledger over the canonical success-gated corpus.

codebase	n	MoJoFM median [range]	ARI median [range]	n_components range
bash	10	0.567 [0.516, 0.595]	0.258 [0.234, 0.295]	[8, 17]

codebase	n	MoJoFM median [range]	ARI median [range]	n_components range
hadoop	10	0.551 [0.411, 0.591]	0.437 [0.283, 0.553]	[8, 20]
archstudio4	10	0.583 [0.408, 0.700]	0.537 [0.206, 0.723]	[4, 21]
libxml2	10	0.397 [0.247, 0.480]	0.274 [0.150, 0.352]	[5, 11]
jpetstore	10	0.350 [0.350, 0.350]	−0.050 [−0.050, −0.050]	[6, 8]
ITK	10	0.917 [0.814, 0.983]	0.756 [0.418, 0.968]	[10, 50]

Quality varies substantially across codebases (ARI medians span $-0.050 \rightarrow 0.756$), and the per-cell ranges are wide relative to the cross-codebase median deltas. The cross-codebase pattern therefore needs disaggregation, which the next three subsections provide.

4.3.4 Two Codebases Whose Score Is Structurally Degenerate

Two of the six codebases produce A-F scores that are structurally pinned or structurally inflated and that should not be read as quality signals.

jpetstore is pinned constant. All 10 runs scored *exactly* MoJoFM 0.350 / ARI −0.050 despite proposing six to eight different component counts. The constancy is a property of the gold. jpetstore's gold assigns 23 of the project's classes to three business domains (account, catalog, order), and because both metrics are computed only over the intersection of the run's partition with that 23-file gold universe, any run-to-run disagreement about the remaining files is scored away. What survives the intersection is the same partition every time: the agent consistently recovered the four technical layers (domain model, persistence mappers, service layer, web action beans), and restricted to the 23 gold files all 10 runs induced the *identical* four-block layer sub-partition. The score is therefore pinned by construction. Layers cut across business domains, so the resulting ARI sits marginally below chance. We report jpetstore's numbers throughout the chapter as a disclosed exhibit, but do not pool them into cross-codebase aggregates.

ITK MoJoFM is granularity-insensitive. Across the 10 ITK runs MoJoFM holds steady at 0.81 to 0.98 while the proposed component count ranges over 10 to 50 on identical input, and ARI varies meaningfully with that swing (0.418 to 0.968). The optimal Wen-Tzerpos MoJo charges only cheap Join operations to merge a finer-but-pure partition into a coarser gold, so a 50-component run of one large bucket (a "ThirdParty"-family anchor in 8 of 10 runs) plus 49 small splinters scores the same as the 10-or-11-component runs. On ITK, MoJoFM represents how much of the gold's dominant directory-aligned structure is preserved regardless of granularity (Section 3.5). ARI is the metric that reflects the run-to-run variation.

4.3.5 Bash: Stable Concept Anchors over a Directory-Aligned Gold

Bash is the third case requiring its own treatment. It has the **tightest ARI band** in the corpus (a spread of 0.061 across 10 runs, where archstudio4 spans 0.517 and ITK 0.550). The tightness is not uniform skill but a property of how bash's input shape interacts with A-F's pipeline. A trajectory and snapshot audit of the 10 bash A-F runs identifies the mechanism with three observations.

Bash's gold is essentially the directory tree. Ten of the thirteen bash gold clusters are 100% directory-pure, drawing every file from a single directory. Only "general", libbuiltins, and the small job_control cluster require content-level reasoning. The directory partition (grouping files by their immediate parent directory, as a diagnostic lens per Section 3.6) scores **ARI 0.82 against bash gold** on its own, so the bash recovery task is structurally close to a directory-clustering task.

The agent's bash exploration is shallow and near-identical across runs. Every one of the 10 bash A-F trajectories opens with the *same* command, `find /work -maxdepth 2 -type f | sed 's#^/work/##' | sort | head -200`, then reads the top of 8 to 24 canonical bash files across one to four batched source-reading commands, proposes 8 to 17 components, and submits (median trajectory length 5 LLM calls, the shortest in the ladder). Almost every read is a head-of-file slice (`sed -n '1,80p'`, `head -60`), and no run dumps a file wholesale, though eight slices request more lines than the file holds, and the 10 runs together open 39 distinct files, 9% of the 416-file source tree (the gold labels 356 of those). In all 10 runs the agent **opens no file in `bash/lib/sh/`, `bash/lib/intl/`, `bash/lib/malloc/`, or `bash/lib/termcap/`**: 120 files, 29% of that tree, with no content-level reasoning. The proposals nonetheless stabilize in *concept space*, and the allocator routes files to those anchors by directory-path cues, reproducibly across runs.

One measurement completes the picture. For each of the 10 bash A-F partitions, the agreement with the directory partition and the agreement with the gold are *the same* **to within 0.02 ARI** (e.g. one run scores 0.283 vs the directory baseline and 0.285 vs the gold. Another scores 0.301 vs directory and 0.295 vs gold). The directory partition itself scores ARI 0.82 against the gold. So **bash A-F partitions do not reproduce the directory tree** (which would score ˜0.82). They are a *fixed coarsening* of it: the stable concept anchors hold libreadline / libbuiltins / libglob+libtilde / signals as dedicated buckets and reliably collapse `lib/sh + lib/intl + lib/malloc + lib/termcap` into a single mega-bucket, identical across runs because the exploration is identical. The bash result is the first place in this chapter where the directory partition (Section 3.6) becomes load-bearing for interpretation: for this codebase the A-F partition is close to a deterministic function of the filesystem layout. Whether the dynamic is bash-specific or generalizes across the suite is a question we return to in Section 4.6.

4.3.6 What A-F Establishes

A-F's results read along three axes. First, **A-F removes A-Solo's validity gap by construction**, with all 60 runs succeeding. Second, **A-F's quality is not interpretable as a single number**: two codebases produce structurally degenerate scores (jpetstore pinned, ITK MoJoFM granularity-insensitive), and on bash the ARI tightness is itself the product of the agent doing little to reach a stable coarse partition. Third, **the directory partition has surfaced as an interpretive lens** on bash. Its relevance varies across the suite, and the suite-wide treatment is deferred to Section 4.6. A-F's exploration on every codebase is essentially a *file-system query* (list files, read a handful, propose buckets), so the follow-up, taken up next, is whether an explicit relational view of the codebase changes the proposals.

4.4 A-F+KG: Component Proposal with an Offered Knowledge Graph

A-F+KG is the minimal change from A-F: the same prompt, shell, and fixed allocator, plus a `cypher_query` tool issuing read-only Cypher against the extracted Neo4j knowledge graph (Section 3.2), whose `:CALLS`, `:DEPENDS_ON_FILE`, `:INCLUDES`, `:INHERITS`, `:IMPLEMENTS`, `:USES`, and `:CONTAINS` edges span `:File` / `:Function` / `:Class` / `:Method` / `:Directory` nodes. The agent is free to ignore the new tool: this rung *offers* the KG, it does not force it. That distinction is load-bearing. A-F+KG answers a behavioural question (which tool does the agent use, and how?) and a quality question (does offering the KG change the partition, given the choice the agent actually makes?). What it *cannot* answer is whether the KG is *useful when used*: if the partition does not change, we cannot distinguish "the KG carries no architectural information" from "the agent did not draw on it." That distinction is what the next rung A-KG resolves by *forcing* KG-only exploration (Section 4.5). We keep the two claims distinct throughout.

4.4.1 Corpus and Setup

Each codebase was run ten times under the canonical A-F+KG configuration, for 60 runs in total, all succeeding with coverage 1.00 (§4.3.2). The corpus has been verified for cohort uniformity (byte-identical canonical system prompt, uniform output path, logged model and temperature), with pre-realignment artifacts archived and de-registered before this analysis (details in the provenance appendix). The agent's interface to the KG is a single tool call, `cypher_query "<Cypher string>"`, issued through the shell against a sidecar Neo4j container. Cypher activity is therefore counted at the shell-command level (every `cypher_query` call is also a shell call, but not conversely).

4.4.2 Cross-Codebase Quality, Paired Against A-F

The test of A-F+KG's quality is the *paired* comparison: same codebase, same allocator, same prompt, with vs without the offered KG. Table 11 gives the per-codebase delta between A-F and A-F+KG at the n=10 corpus on both sides.

TABLE 11

A-F+KG vs A-F paired comparison per codebase. Δ columns are A-F+KG minus A-F medians. Both configurations are at n=10 on every codebase.

codebase	A-F MoJoFM	A-F+KG MoJoFM	Δ MoJoFM	A-F ARI	A-F+KG ARI	Δ ARI
bash	0.567	0.587	+0.020	0.258	0.288	+0.030
hadoop	0.551	0.474	−0.077	0.437	0.373	−0.064
archstudio4	0.583	0.538	−0.045	0.537	0.300	**−0.237**
libxml2	0.397	0.432	+0.035	0.274	0.259	−0.015
jpetstore	0.350	0.350	0	−0.050	−0.050	0
ITK	0.917	0.906	−0.011	0.756	0.766	+0.010

Offering the KG is **not uniformly worse than withholding it, and not uniformly better either**. The ARI deltas are inconsistent in sign (two up, three down, one pinned), and none of them is large relative to the variation already present within each cell. Every delta in Table 11, on both metrics, is no larger than the ten-run range of each of the two cells it is computed from: the biggest, archstudio4 ARI -0.237, is set against A-F and A-F+KG ranges of 0.517 and 0.561 on the same codebase, and bash's +0.030 against ranges of 0.061 and 0.316. jpetstore is pinned constant, the layer-vs-domain case of §4.3.4, mathematically unaffected by any change to the agent's exploration. We therefore draw no per-codebase conclusion from Table 11 and report no significance test, per the conventions of §3.9. What the table supports is the weaker claim that **offering the KG did not change the partition in any consistent direction**, neither helping nor hurting. Explaining any of these effects requires looking at what the agent actually *does* with the KG when it has the choice, and the next two subsections show that in a specific and verifiable sense, its behaviour does not test the question the configuration was designed to ask.

4.4.3 Behavioural KG Usage

Across the canonical A-F+KG corpus, the agent rarely invoked the KG. All 60 scored runs are 1:1 matched to a trajectory file by the proposed-component-set procedure of Section 3.5.1, so the behavioural corpus and the quality corpus are the same 60 runs. The 60 runs issued **135 cypher_query calls in total**: a median of two calls per run, a maximum of seven, and two runs that issued none at all. The KG is invoked on every codebase, but shell calls substantially exceed cypher calls everywhere, consistent with the agent treating the KG as a minor supplement rather than a primary reasoning surface. The substantive question is *what the agent asks the KG to do*, which the next subsection takes up.

4.4.4 Cypher Query Content: What the Agent Asks the KG

Each `cypher_query` invocation across the corpus is extracted from the agent's trajectory and classified under the four-category scheme of Section 3.5.1 (STRUCTURAL, RELATIONAL, HYBRID, SCHEMA; decision rule, bootstrap-System note, and worked examples there). The classifier is mechanical and does not infer intent.

TABLE 12
Classification of all 135 cypher_query calls in the 60 scored A-F+KG runs.

codebase	runs	queries	structural	relational	hybrid	schema
ITK	10	28	28	0	0	0
archstudio4	10	25	25	0	0	0
bash	10	29	25	3	1	0
hadoop	10	27	26	1	0	0
jpetstore	10	12	12	0	0	0
libxml2	10	14	14	0	0	0

codebase	runs	queries	structural	relational	hybrid	schema
TOTAL	60	135	130	4	1	0

Two findings follow.

Finding 1: 96.3% of all cypher_query calls are structural. One hundred and thirty of 135 invocations return information equivalent to what a `find` plus possibly a `grep`/`wc` would produce (paging the file universe via the bootstrap System component, directory-tree statistics, path-prefix counts, function-count probes; worked examples in Section 3.5.1). None draws on the KG's relational source-code edges: the agent uses the KG as a file-and-code enumeration tool that extends `find` only slightly.

Finding 2: Only 5 of 135 queries (3.7%) exercise the KG's relational source-code edges, and they are concentrated in bash. Three pure-relational queries probe `:INCLUDES` and `:DEPENDS_ON_FILE` traversals and one hybrid query coordinates `:CONTAINS` and `:CALLS` to compute a per-file call-graph density, all four on bash; a fifth, on hadoop, counts `:IMPORTS` edges per file within `mapred/`. The remaining four codebases issue *zero* relational queries across 10 independent runs each. On a five-call sample spread over two codebases there is no basis for reading bash as a systematic exception, and we do not treat it as one.

4.4.5 What A-F+KG Establishes

A-F+KG was designed to ask whether offering an explicit relational view changes the partition. The corpus answers a different question: the quality picture is muted (six small deltas, inconsistent in sign, none of them large against the run-to-run variation already present in the cells they are computed from), the agent uses the KG almost exclusively as a structural enumeration tool (a fancier `find`), and the relational signal is barely tested, with five relational calls in the whole corpus. The agent's behavioural choice about *how* to use the KG therefore dominates the measurement: the "no consistent quality lift" reading measures an agent that did not engage the load-bearing structure, leaving the KG's capacity untested. To answer "is the KG useful for SAR?", the agent's choice has to be removed from the measurement, which the next section does by disabling the shell so that `cypher_query` becomes the agent's *only* exploration interface.

4.5 A-KG: KG-Only Exploration (Shell Disabled)

A-KG is the minimal single-variable change from A-F+KG: the shell tool is disabled, and with the same prompt, model, and fixed `BatchFileAllocator`, component proposals must be made entirely from what Cypher returns. This rung resolves the question A-F+KG could not judge: A-KG removes the agent's choice and forces it onto Cypher for everything. If the partition *improves* relative to A-F+KG, the offered-KG result was a measurement of the agent's choice, leaving the KG's capacity untested. If A-KG also matches or exceeds A-F, the KG's relational signal carries weight beyond what filesystem-driven exploration extracts.

4.5.1 Corpus and Setup

Each codebase was run under the canonical A-KG configuration (same caps, same per-cell n=10 target). The scored corpus is **10 runs on each of the six codebases, 60 runs in total**. All 60 scored runs succeed with coverage 1.00.

4.5.2 Cross-Codebase Quality, Paired Against A-F+KG and A-F

Table 13 reports A-KG quality per codebase at the current corpus and the paired delta against A-F+KG (the same-tool-set comparator, offered vs forced).

TABLE 13

A-KG vs A-F+KG paired comparison per codebase. Both configurations are at n=10 on every codebase.

codebase	A-F+KG MoJo / ARI	A-KG MoJo / ARI	Δ MoJoFM	Δ ARI
bash	0.587 / 0.288	0.609 / 0.327	+0.022	+0.039
hadoop	0.474 / 0.373	0.532 / 0.491	+0.058	**+0.118**
archstudio4	0.538 / 0.300	0.628 / 0.457	+0.090	**+0.157**
libxml2	0.432 / 0.259	0.541 / 0.325	**+0.109**	+0.066
jpetstore	0.350 / −0.050	0.350 / −0.050	0	0
ITK	0.906 / 0.766	0.822 / 0.466	−0.084	**−0.300**

Forcing the KG produces ARI lift on four of the six codebases (bash, hadoop, archstudio4, libxml2). Jpetstore is pinned at or near ARI −0.05 (§4.3.4), and ITK is the one informative codebase where forcing the KG *underperforms* the offered-KG configuration, with a wide, bimodal ARI distribution taken up in §4.5.4 (Figure 7, where run-to-run spread dominates the configuration effect). This is direct evidence that A-F+KG's muted quality picture was at least partly a measurement of the agent's behaviour, with the KG's capacity left untested there: available-but-mostly-unused on A-F+KG (§4.4.5), the KG is used under A-KG, and quality lifts. Table 14 compares the same corpus against the shell-only A-F baseline (Section 4.3).

TABLE 14

A-KG vs A-F paired comparison per codebase. Both configurations are at n=10 on every codebase.

codebase	A-F MoJo / ARI	A-KG MoJo / ARI	Δ MoJoFM	Δ ARI
bash	0.567 / 0.258	0.609 / 0.327	+0.042	+0.069
hadoop	0.551 / 0.437	0.532 / 0.491	−0.019	+0.054
archstudio4	0.583 / 0.537	0.628 / 0.457	+0.045	**−0.080**
libxml2	0.397 / 0.274	0.541 / 0.325	**+0.144**	+0.051
jpetstore	0.350 / −0.050	0.350 / −0.050	0	0
ITK	0.917 / 0.756	0.822 / 0.466	−0.095	**−0.290**

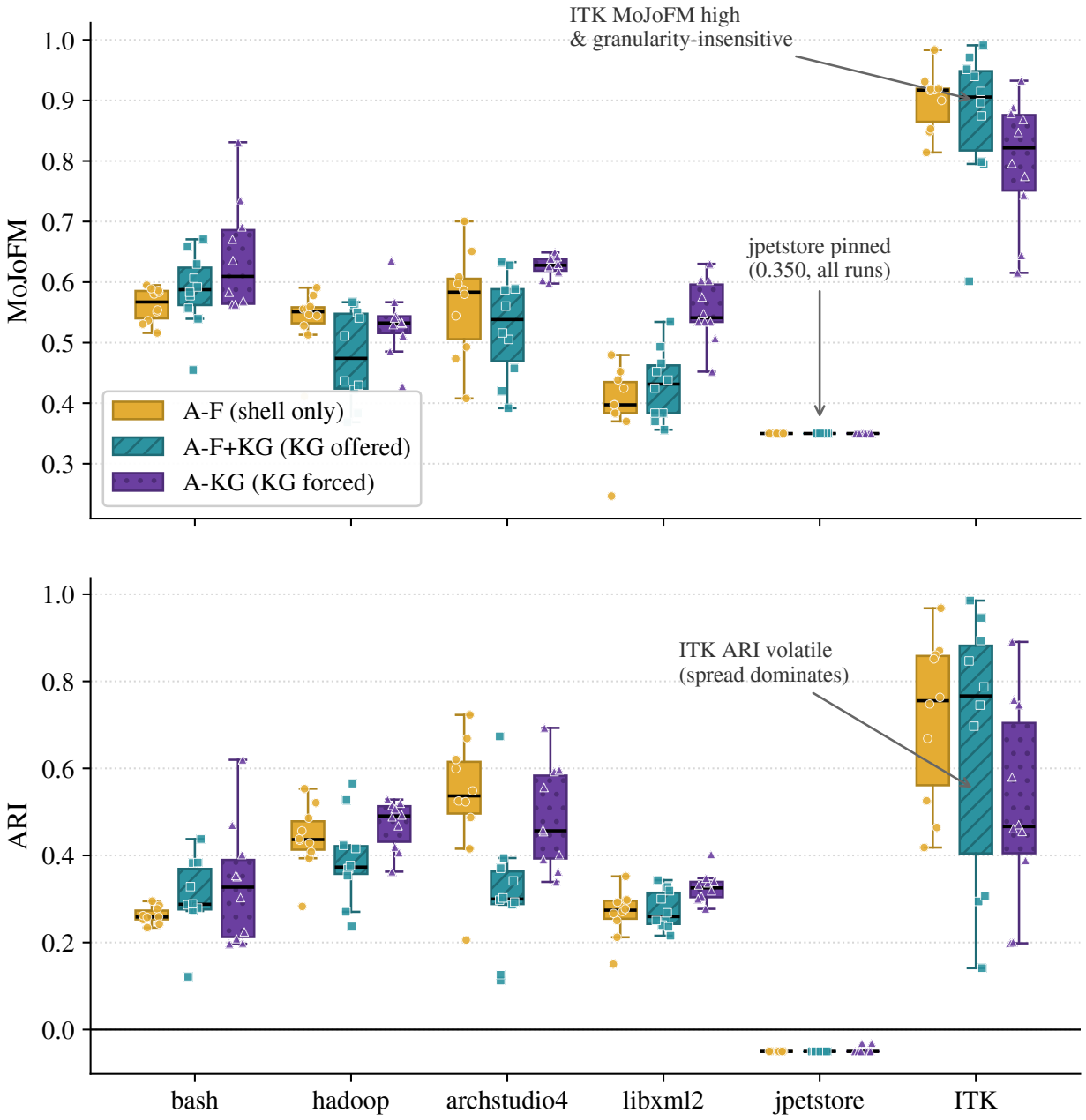


Figure 7: Per-configuration MoJoFM (top) and ARI (bottom) distributions (n=10) across the six codebases. Run-to-run spread dominates configuration effects. Jpetstore is pinned. ITK is granularity-insensitive on MoJoFM and is read on ARI. Boxes give the median and inter-quartile range, points the individual runs; source: the scored grid (A-F, A-F+KG, and A-KG across the six codebases).

The single largest *positive* delta in Table 14 is libxml2's MoJoFM lift, and libxml2 is the codebase whose gold has the *least* directory alignment (`ARI(dir@2, gold)` $\approx$ `0`, Section 4.8.1 / threat T1; the flat-variant ablation of §4.6 confirms it). **Where directory layout carries no gold information, the KG-only configuration produces the biggest quality lift over the shell baseline. Where directory layout IS the gold (ITK, `ARI(dir@2, gold) = 1.000`), the shell-only configuration parrots that layout for free and the KG-only configuration cannot match it on this corpus.** This is the prediction of the structure-riding mechanism §4.3.5 surfaced on bash: where `find` returns no useful gold signal the KG's edges provide an alternative source of structure. Where `find` already returns the gold, a configuration that cannot run `find` is at a structural disadvantage.

4.5.3 Behavioural KG Usage

With the shell removed, cypher volume rises as expected. All 60 scored A-KG runs are matched 1:1 to a trajectory file by the proposed-component-set procedure of Section 3.5.1, so the behavioural corpus is the full scored corpus at n=10 on every codebase. Across the 60 runs the agent issued **725 cypher_query calls in total**: a median of 11 calls per run (range 7 to 25), versus a median of two on A-F+KG. Non-cypher tool calls are exactly 1 per run (the final `submit`), confirming the shell is disabled. Table 15 reports the per-codebase activity under the four-category scheme of Section 3.5.1.

TABLE 15
Classification of all 725 cypher_query calls in the 60 scored A-KG runs.

codebase	runs	queries	structural	relational	hybrid	rel+hyb %
ITK	10	127	94	33	0	26%
archstudio4	10	148	133	12	3	10%
bash	10	103	68	12	23	34%
hadoop	10	153	129	23	1	16%
jpetstore	10	99	78	20	1	21%
libxml2	10	95	57	17	21	**40%**
TOTAL	60	725	559	117	49	**23%**

Three observations.

First, the relational+hybrid rate rises from 3.7% (A-F+KG) to 23% (A-KG). Every codebase now issues relational+hybrid queries, where on A-F+KG they were concentrated in bash: *the agent's under-use of relational edges on A-F+KG was a behavioural choice, not a capability constraint.* Execution outcomes confirm the capability reading: across the calls classified in Tables 12 and 15, fewer than two percent failed, four of the five failures were surface-syntax slips (`CypherSyntaxError`) and the fifth was the harness read-only guard firing on an `OFFSET` clause; none was a schema error, and the relational and hybrid calls executed essentially without error (none failed on A-F+KG, at most one on A-KG). As with the other behavioural claims in this chapter, this observes what the agent's queries *did*, not its stated reasoning, which the trajectory

format does not preserve (Threat T6).

Second, the bulk of A-KG cypher activity (77%) remains structural, much of it the shell's directory-prefix enumeration ported into Cypher. **46.5% of all A-KG cypher calls** (337 of 725) are path-string operations on `File.path` or `Directory.path` (`STARTS WITH`, `CONTAINS`, `split(path,'/')`), returning information equivalent to `find` or `grep`. Hadoop reaches 58%. **Removing the shell does not close the structure-riding leak. The agent translates path-riding into Cypher.** The lifts in Tables 13 and 14 are therefore not "the agent stopped riding structure" but "the agent rides structure via Cypher *and additionally* exercises relational edges."

Third, the per-codebase relational+hybrid distribution is suggestive but not conclusive. libxml2 has both the highest relational+hybrid rate (40%) and the largest MoJoFM lift over A-F (+0.144). Archstudio4 has the lowest rate (10%) and an ARI drop vs A-F (−0.080). The link is consistent, but the sample is six codebases. We note the pattern without claiming the mechanism. The HYBRID category is dominated by `(Container)-[:CONTAINS]->(Unit)-[:CALLS]->(Unit)` co-traversals computing per-file or per-class call counts as a coupling proxy, queries the KG was designed to enable.

4.5.4 What A-KG Establishes

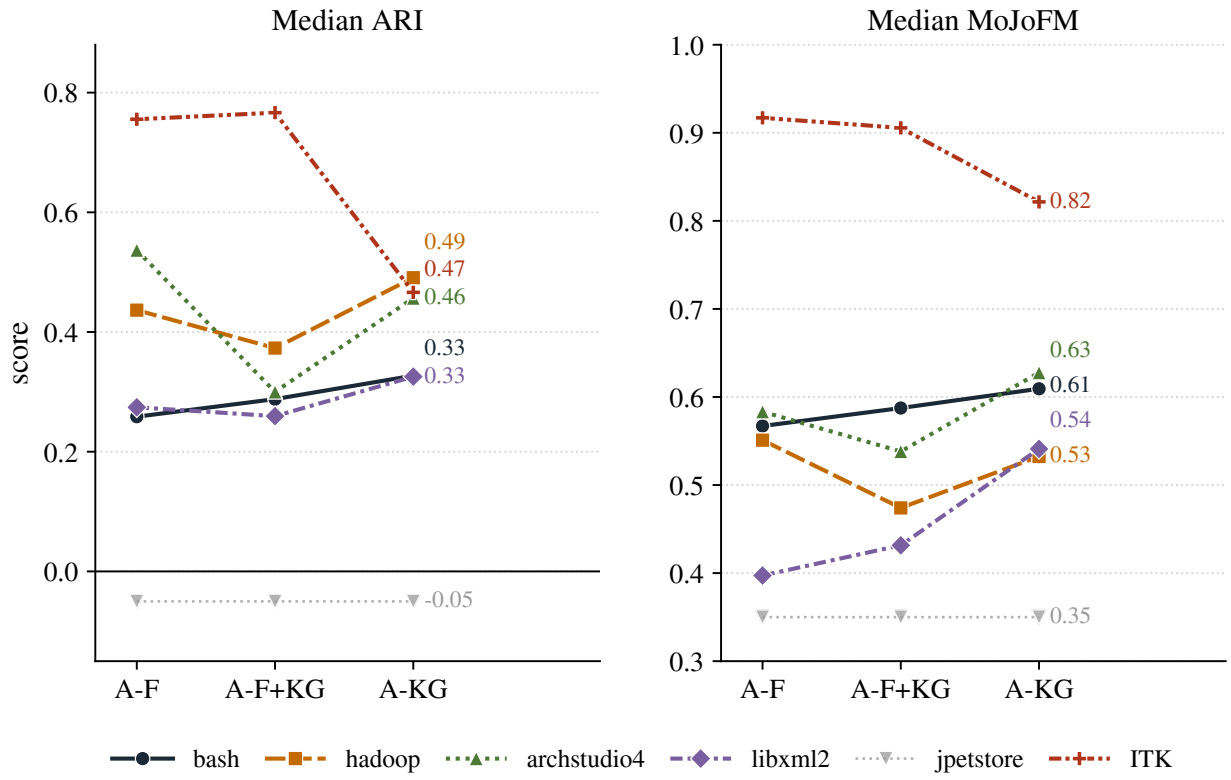


Figure 8: Median ARI across the A-F to A-F+KG to A-KG ladder, one line per codebase: offering a KG gives no consistent lift, forcing it lifts ARI on four of five informative codebases but regresses ITK. jpetstore (greyed) is degenerate. Each point is a per-codebase median over ten runs.

Three claims follow from the corpus. First, **forcing the KG produces closer gold-aligned partitions than**

offering it on four of the five informative codebases (Table 13; the per-codebase A-F to A-F+KG to A-KG ARI trajectories are traced in Figure 8), direct evidence that the under-use documented in §4.4.3 was behavioural, not a fundamental limit. Second, **the KG is modestly load-bearing relative to the shell-only baseline** (Table 14): it helps where the gold is not directly recoverable from directory layout (libxml2 is the strongest positive case) and loses where the gold IS the directory layout (ITK), a partial answer, not a clean yes. Third, **removing the shell does not close the structure-riding leak**: the agent ports filesystem-shape enumeration into Cypher (§4.5.3), so the lifts are "structure-riding plus relational edges," not the absence of structure-riding.

A fourth observation concerns reproducibility rather than quality. archstudio4's ten A-KG partitions agree with *one another* at median pairwise ARI 0.576, and its MoJoFM distribution is the tightest of any non-degenerate ten-run cell, its standard deviation 5.2 times smaller than A-F's on the same codebase. (Three jpetstore cells have zero MoJoFM variance, for the structural reason of §4.3.4, and are not informative here.) Agreement between runs is not evidence of agreement with the gold (archstudio4's A-KG median ARI stays below A-F's), but it does show that the forced-KG configuration converges on the same decomposition far more reliably on some codebases than on others, which bears on the run-to-run variance question rather than on whether the KG helps.

ITK organizes both exceptions at once: its A-KG ARI distribution is bimodal (four runs at ARI ≥ 0.580, six at ≤ 0.471, and none in between), and its gold is literally the depth-2 directory partition (`ARI(dir@2, gold) = 1.000`, §4.8.1, threat T1), which a shell-driven configuration recovers for free while path-string Cypher recovers it less reliably. The only configuration that closes the structure-riding leak is the flat-KG ablation (§4.6).

4.5.5 Probing the Structure-Riding Mechanism: the Flat-Variant Ablation

The three findings of §4.5.4 leave one question the four-rung ladder cannot answer: does the KG carry semantic information that helps recover gold, or directory information the agent re-uses as a substitute for `find`? **§4.6 introduces two structure-riding probes, A-F-flat and A-KG-flat**, which close the shell and Cypher pathways at their source and ask whether recovered quality drops in proportion to how directory-aligned each codebase's gold already was.

4.6 Structure-Riding: the Flat-Variant Ablation

The four-rung ladder of §§4.2 to 4.5 produced two observations it cannot itself judge. On A-F, the bash partition was a fixed coarsening of the filesystem layout (§4.3.5). On A-KG, removing the shell did not stop the agent from riding directory structure, since it ported the path-string enumeration from `find` into Cypher (§4.5.3). The load-bearing question of the §4.5 synthesis is therefore open in two directions: does the KG carry semantic architectural information, or directory information that the agent re-uses as a substitute for the shell?

This section closes both directions with a controlled ablation: two new configurations re-run A-F and A-KG against a substrate from which directory structure has been removed at its source, both in the source tree the shell reads and in the knowledge graph Cypher queries against. The ablation is run on the three C/C++

codebases (bash, libxml2, ITK). The Java codebases are treated through a separate, indirect lens that does not require flattening (§4.6.3).

4.6.1 The Two Pathways

The structure-riding leak surfaces through two distinct interfaces. **The shell pathway:** A-F's only exploration tool is a shell, and on every canonical A-F run the agent's first command is a `find /work -maxdepth 2`-style listing. On bash, where `ARI(dir@2, gold) = 0.817`, the resulting partition's agreement with the directory partition essentially equals its agreement with the gold (§4.3.5). **The Cypher pathway:** A-KG removes the shell, but the KG carries the canonical `File.path` string in every File node and a full `(:Directory)-[:CONTAINS]->(:File)` hierarchy, so the agent recovers the same directory-prefix enumeration through Cypher on every codebase (§4.5.3). Neither configuration on its own separates "the tooling carries semantic architectural information" from "the tooling re-presents the directory tree, which on this gold is highly aligned to the answer." To separate them, both pathways have to be closed at the substrate. The two flat-variant configurations do that (their pathway-closing logic and outcome readings are laid out in Table 2):

- **A-F-flat** runs A-F against a directory-flattened copy of each codebase: every file sits in a single root under a unique flat name, so `find` still returns the file universe but the paths carry no directory segments to group by. The shell pathway is closed.
- **A-KG-flat** runs A-KG against a knowledge graph extracted *from the flat codebase*: `File.path` is the flat filename and `:Directory` collapses to a single trivial root, so path-string Cypher returns nothing useful, while the relational edges (`:CALLS`, `:DEPENDS_ON_FILE`, `:INCLUDES`, `:INHERITS`) are preserved by the Stage-2 reference rewriter (Section 3.3.4). The Cypher pathway is closed. The relational signal is not.

If A-F-flat collapses while A-F survives, the A-F recovery was largely shell-driven directory parroting. If A-KG-flat collapses while A-KG survives, the A-KG quality was largely Cypher-driven directory parroting. If neither collapses (if the flat configurations recover comparable quality to their canonical counterparts), then the underlying signal each configuration is using is not directory-aligned.

4.6.2 The Flat-KG Construction

The flat substrate is built by the four-stage pipeline of Section 3.3.4. Stage 1 is a verified bijective rename map. Stage 2 (threat T7) rewrites every `#include "..."` directive to its target's flat name through a resolver that does not silently guess ambiguous basenames, with residual directives left canonical and disclosed as real codebase ambiguities rather than rewriter errors (88% non-residual on bash, 96% on libxml2, 97% on ITK). Stage 3 re-extracts the KG from the flat codebase under an integrity gate (dependency-edge counts within tolerance of the canonical KG, exactly one `:Directory` node, no `/` in any `File.path`). Stage 4 translates recovered flat partitions back to canonical paths so they are scored against the same canonical gold.

4.6.3 Why C/C++ Only

The ablation is run on the three C and C++ codebases and excluded on the three Java codebases. In C and C++ the dependency-resolution key is *lexical and file-local* (every inter-file reference is an `#include "..."` path the Stage-2 rewriter can rewrite verifiably). In Java it is *semantic and binding-driven* (packages, imports, same-package implicit references), so flattening destroys the resolver's input, a purely lexical rewriter cannot reconstruct it, and even a symbol-solver Java-flat KG would be provably edge-incomplete under reflection-driven loading. Threat T9 records the exclusion (full rationale in Section 3.3.4). Threat T7 records the C/C++ residual policy. The C/C++ result is the strongest *clean* test of the structure-riding hypothesis the design can produce. The Java codebases are probed indirectly through the §4.8 directory-lens covariate (threat T1), which needs no flattening.

4.6.4 Canonical vs Flat Quality

Tables 16 and 17 report the per-codebase quality deltas. A-F and A-KG are at n=10 on every codebase, as are both flat configurations on bash and libxml2. On ITK the flat cells are smaller: A-F-flat is at n=7 and A-KG-flat at n=3. ITK costs roughly $2.60 per run against $0.14 for bash and $0.32 for libxml2, so a full ten-run flat pair on ITK would have cost more than the rest of the ablation combined. Every flat run is `exit_status=success` with full coverage of the authoritative file universe.

TABLE 16

A-F vs A-F-flat per codebase. Median MoJoFM and ARI computed from the scored-run ledger after Stage-4 flat→canonical scoring translation.

codebase	n (F / F-flat)	A-F MoJo / ARI	A-F-flat MoJo / ARI	Δ MoJoFM	Δ ARI
bash	10 / 10	0.567 / 0.258	0.519 / 0.212	−0.048	−0.047
libxml2	10 / 10	0.397 / 0.274	0.452 / 0.279	+0.055	**+0.005**
ITK	10 / 7	0.917 / 0.756	0.388 / 0.065	**−0.529**	**−0.691**

TABLE 17

A-KG vs A-KG-flat per codebase.

codebase	n (KG / KG-flat)	A-KG MoJo / ARI	A-KG-flat MoJo / ARI	Δ MoJoFM	Δ ARI
bash	10 / 10	0.609 / 0.327	0.481 / 0.151	**−0.128**	**−0.176**
libxml2	10 / 10	0.541 / 0.325	0.520 / 0.297	−0.021	−0.028
ITK	10 / 3	0.822 / 0.466	0.491 / 0.114	**−0.330**	**−0.353**

The three codebases respond in three qualitatively different ways. Libxml2 is essentially unchanged on both metrics under both configurations, with an ARI delta of +0.005 on A-F and −0.028 on A-KG. Bash drops moderately on both, with the larger drop on A-KG. ITK collapses on both, most severely on A-F (Δ

ARI −0.691), with a bimodal failure mode discussed in §4.11.1. The per-codebase magnitudes are *paired*: codebases that drop little on A-F drop little on A-KG, and vice versa, so the two pathways appear to be reaching for the same structural signal. The next subsection makes that claim quantitative.

4.6.5 The Quantitative Prediction

The pattern in Tables 16 and 17 is not "all codebases drop under flattening": libxml2 does not. It is "codebases drop in proportion to how directly directory layout already encodes the gold" (Figure 9). The diagnostic that makes the proportion explicit is the per-codebase covariate `ARI(directory_at_depth_K, gold)` (Section 3.6, threat T1), a one-number summary of how directly directory layout alone predicts the gold's component assignments. Computed against the cleaned gold RSF (Section 3.7) at depth 2, the depth at which each codebase's gold is most informatively aligned:

TABLE 18

Per-codebase directory covariate `ARI(dir@2, gold)`: how directly depth-2 directory layout alone predicts the gold's component assignments.

codebase	`ARI(dir@2, gold)`	what it means
libxml2	**0.018 (≈ 0)**	gold groupings essentially independent of depth-2 directories
bash	**0.817**	gold aligns strongly, though not perfectly, with depth-2 directories
ITK	**1.000**	the 11 gold components ARE the 11 `Modules/<X>/` depth-2 subdirectories

The covariate predicts the flat-vs-canonical drop magnitudes in Tables 16 and 17 directly: libxml2, whose directories carried no gold information to lose, is essentially unaffected (Δ ARI +0.005 / −0.028 on A-F / A-KG). Bash, whose directories carried partial gold information, drops moderately (Δ ARI −0.047 / −0.176). ITK, where removing the directory partition is mathematically removing the gold partition itself, drops most severely (Δ ARI −0.691 on A-F and −0.353 on A-KG), with the A-F-flat case additionally exhibiting the mode-collapse failure of §4.11.1.

This is the headline finding of the structure-riding ablation. The canonical configurations were riding directory structure to a degree whose magnitude this covariate, computed before any agent run, predicts *in advance*. A reader interpreting a high MoJoFM or ARI on any of these benchmarks without `ARI(dir@2, gold)` reported alongside cannot distinguish "the configuration recovered semantic architecture" from "the configuration parroted a directory tree that the benchmark gold equates with the answer." The methodological recommendation that follows from the result is recorded in threat T1. `ARI(dir@K, gold)` should be reported for every benchmark codebase, as a benchmark-design hygiene metric, alongside any aggregate recovery score. Methods that recover well on a codebase whose `ARI(dir@K, gold)` is high are at most demonstrating *(score − directory-baseline-contribution)* of semantic recovery. The rest is directory-readable for free. The flat-variant ablation is the empirical instantiation of that decomposition.

The flat-variant ablation locates where the *file groupings* come from. §4.7 takes up where the *component*

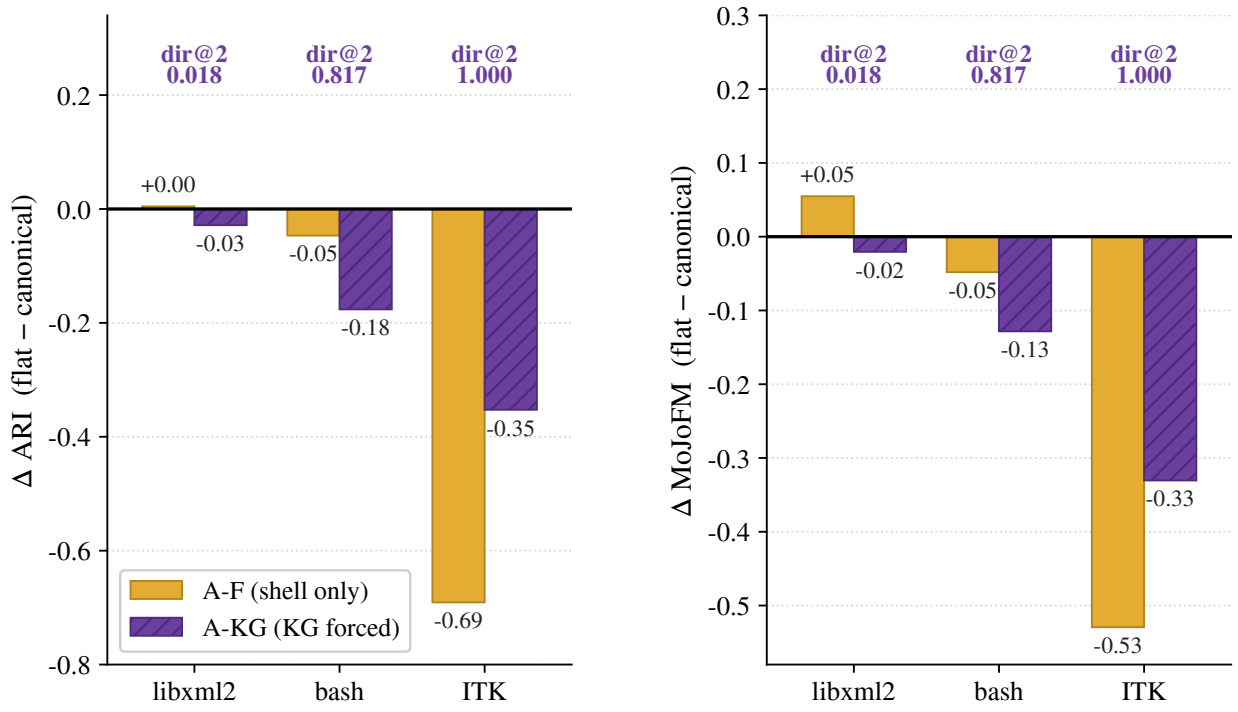


Figure 9: Flat-versus-canonical recovery drop, ordered by the directory covariate ARI(dir@2, gold) (libxml2, bash, ITK): flattening degrades recovery in proportion to how directly directory layout encodes the gold. The three codebases are laid out in ordinal covariate order (libxml2 dir@2 $\approx$ 0, bash 0.817, ITK 1.000); the spacing carries no fitted relationship. Flat runs number five for bash and libxml2 and three for ITK, against ten canonical runs; source: the scored-run ledger.

names come from, with a zero-substrate elicitation probe and an identity-stripping transform that composes with the flattening used here. The Java codebases, for which a binding-correct flat ablation is infeasible, are taken up through the directory-lens covariate in §4.8.

4.7 Recitation: Component Naming Under Identity-Stripped Substrates

4.7.1 The Question the Structure-Riding Result Leaves Open

The flat-variant ablation (§4.6) concerns where the *file groupings* come from. It leaves a deeper question about where the *component names* come from: when a method labels a cluster "XPath Engine" or "Job Control," is it reasoning from the artifact in front of it, or reproducing the well-documented architecture of a famous open-source system it has seen many times in pre-training? The benchmark codebases (bash, libxml2) are among the most-described software in existence, their architectures almost certainly present in the model's training corpus (the contamination construct of [T2]). A zero-substrate elicitation probe makes the prior concrete: prompted only with a codebase *name* and no source access, the model reproduces a recognizable canonical decomposition of each study system. The architecture is in the weights. The question for this section is whether that prior still asserts itself when the codebase's *identity* is removed from the substrate the agent reads.

4.7.2 The Zero-Substrate Elicitation Probe

The probe establishes, before any substrate is queried, that the model already holds a canonical decomposition of each subject system in its weights. It is run for three independent trials per codebase at `temperature 0` with the primary model (`gpt-5.4-mini-2026-03-17`), with nothing supplied beyond the prompt below: no source, no knowledge graph, no tools, not even a file listing.

> *Recover the architecture of the [codebase] codebase (language: [L]). It is a real software project; group its source files into coherent architectural components. Submit your proposed components: one JSON object per line, with keys "name" and "purpose" only (no files). Do not output anything else.*

Eighteen trials (six codebases × three) ran to completion at a combined cost of **$0.0354**. The probe transcripts are persisted per trial. Every trial on every codebase produces a recognizable canonical decomposition: components are domain-specific (not generic "Core", "Utilities"), match the subject system's documented architecture, and sit at a granularity appropriate to each codebase's scale. For the two in-scope study codebases, a representative trial of each is as follows:

- **bash** (trial 1, 8 components): *Core shell runtime; Builtins and shell commands; Parser and syntax handling; Expansion and word processing; Execution and process management; Job control and signals; Environment and variables; Memory and utility infrastructure.*
- **libxml2** (trial 1, 20 components): the trial enumerates libxml2's full documented subsystem inventory (core tree and document model, parser and tokenizer, SAX, validation and schema processing, XPath, XPointer, XInclude, serialization, I/O and catalogs, encoding, Python bindings, and the remaining canonical modules).

The four out-of-scope codebases (ITK, archstudio4, hadoop, jpetstore) reproduce the same architecture at the same fidelity. Trials of the same codebase agree on the decomposition: they vary in component count and wording while naming the same subsystems.

The probe pins the central baseline of this section: the canonical decomposition is recoverable from the codebase's name alone, with no source artifact, so any decomposition in the substrate-driven configurations that *matches* the probe is, at minimum, *consistent* with reproduction from the prior, regardless of what the agent did or did not query. The rest of the section asks how much of the substrate's identity must be removed before the model stops reproducing it.

4.7.3 Identity Stripping as a Substrate Transformation

Obfuscating real code to defeat a memorization advantage is established practice in the code-generation benchmark literature. OBFUSEVAL rebuilds five real C projects with renamed symbols, restructured control flow and semantics-preserving rewrites, then re-runs code generation on the result [51]; CODEMORPH applies dependency-preserving transformations across files for the same purpose [52]; and a parallel line isolates the naming channel specifically by stripping identifiers and measuring what comprehension survives

[53]. This thesis does not invent that transformation. What it changes is the scope of the stripping and the question asked of the result.

Those studies rename *identifiers* and measure whether the model can still *perform the task*. Three channels they leave open are decisive when the object of study is architecture rather than code completion: file and directory names, comments and copyright headers, and identity strings embedded in literals and compiler directives. Each of those names the project outright. The transformation used here closes all three at repository scale and verifies the closure directly rather than inferring it from a performance drop (Table 19). It is important to be clear that this is a *different* transform, not a stronger one: prior work rewrites the identifiers this thesis retains, so the substrate used here preserves more project identity than theirs, and nothing established on it transfers to their setting.

TABLE 19

The identity channels closed by the stripping transform, and how closure is verified. Applied to the whole repository, so the result remains an agent-explorable substrate rather than a set of isolated snippets.

Channel	Transformation	Verification
project token	literal substitution of the project name throughout source and paths	zero case-insensitive occurrences in the extracted graph
file and directory names	every path segment replaced by `f_0097.c` / `dir_0008`, root normalized	no residual canonical filename in any node property
`#include` graph	directives rewritten to the opaque names	dependency-edge parity gate against the canonical extraction
comments and headers	all comment blocks removed, copyright and authorship stripped	mean file-contents length falls by the comment mass
directives and literals	`#line` directives genericized, residual identity strings substituted	zero occurrences of a fixed residual-token list

Two channels are deliberately **retained**, because closing them would destroy the task rather than anonymize it: **code identifiers** (`xmlXPathEval`, `reap_dead_jobs`) and **file extensions** (`.dtd`, `.rng`). A recovery method that cannot read function names is not performing architecture recovery, so these constitute an irreducible floor. Any residual recitation measured below is therefore an *upper bound* on how much identity stripping can remove, not evidence of a clean substrate.

Because the transform operates on names and contents, and the flattening of §4.6 operates on the directory tree, the two are orthogonal and compose. That composition is the design of the next subsection: it separates "the model is reading the folder layout" from "the model is reading the identifiers" by removing each channel

independently and together.

4.7.4 Design: Identity Stripping Crossed with Flattening

The two transformations are crossed on both C study systems, with the explorer held constant: the knowledge-graph-only configuration of §4.5, whose only view of the codebase is Cypher over the extracted graph. Holding the explorer fixed matters, because configurations differ systematically in how they name things, and a substrate comparison that also varies the configuration cannot attribute a difference to the substrate.

The measure is **probe-containment** (Method §3.5.2): the fraction of a run's distinctive component-name tokens that also appear in that codebase's zero-shot probe. A run scoring high is producing names the model can already generate with no code at all. Containment is computed by a deterministic scorer over the run snapshots and is stable under removing the stopword list.

A deeper stripping build was also run in the unflattened arm on both systems, applying the comment removal and literal scrub of Table 19 in full. It is reported alongside the crossed design as a test of whether the residual can be pushed lower.

4.7.5 Neither Transformation Removes Canonical Naming

TABLE 20

Probe-containment and probe-coverage (median per cell) under identity stripping crossed with directory flattening, knowledge-graph-only explorer, both C study systems. *Containment* is the share of the run's own naming that is already producible from the codebase name alone; *coverage* is the share of the probe's canonical vocabulary the run reproduced.

Substrate	bash, dirs intact	bash, flattened	libxml2, dirs intact	libxml2, flattened
	cont / cov	cont / cov	cont / cov	cont / cov
real tree	0.42 / 0.46 (n=10)	**0.53 / 0.54** (n=10)	0.76 / 0.38 (n=10)	**0.78** / 0.34 (n=10)
identity-stripped	0.41 / 0.30 (n=4)	0.48 / 0.39 (n=5)	0.53 / 0.12 (n=5)	0.77 / 0.17 (n=5)
identity-stripped, deep	**0.43** / 0.39 (n=10)	n/a	**0.58** / 0.16 (n=10)	n/a

Four readings follow, and none of them is a decay.

Removing the directory tree does not reduce recitation; it raises it. On the real tree, flattening moves bash from 0.42 to 0.53 containment and 0.46 to 0.54 coverage, and libxml2's containment from 0.76 to 0.78, with the same ordering on the stripped substrate. When the folder layout is taken away, the naming falls back *harder* on recalled architecture. This is the direct link between the two findings of this chapter: the structure-riding channel of §4.6 and the recitation channel measured here are substitutes, and closing one drives naming onto the other.

Identity stripping does not change the *character* of the naming. Containment is the share of a run's own vocabulary that the model could have produced from the codebase name alone, and it barely moves: bash runs 0.42 on the real tree, 0.41 stripped and 0.43 under the deeper strip; libxml2 falls once, from 0.76 to 0.53, and then stops. Whatever a run names on the stripped substrate, it names in the study system's own terms.

What stripping reduces is how much of the architecture is recovered, not whether the system is recognized. Coverage, the share of the probe's canonical vocabulary a run reproduces, does fall: bash 0.46 to 0.30, libxml2 0.38 to 0.12. The runs enumerate fewer of the documented subsystems once the file names, comments and headers are gone, because there are fewer cues to enumerate from. They do not become generic. The two measures moving in different directions is the finding: **the substrate transformation removes evidence, not identifiability.**

Both transformations together still leave the system recognizable, bash at 0.48 containment, libxml2 at 0.77. There is no cell in Table 20 in which the naming reverts to generic architecture vocabulary.

A per-run reading of the deeply stripped substrate agrees. Taking any cut inside the gap that separates the two modes of the coverage distribution on each system (bash 0.130 to 0.174, libxml2 0.104 to 0.208), **8 of 10** bash runs and **5 of 10** libxml2 runs reproduce a substantial share of the canonical vocabulary with no codebase name present anywhere in the prompt or the graph. Appendix B.6 lists every run, so each of the twenty can be checked.

The mechanism this points to is not that canonical vocabulary is emitted regardless of the substrate. It is that **the system remains identifiable from what the transformation cannot remove.** A run that names *Job Control and Process Management*, *Readline Editing* and *Builtins* on a substrate containing no occurrence of "bash" has not guessed those names: it has inferred, from `reap_dead_jobs` and its neighbours, which system it is looking at, and recalled that system's published decomposition. Identity stripping in the sense of the literature, renaming what can be renamed, is therefore not anonymization on this task.

The reason sits in the retained floor of Table 19. Both systems carry domain-bearing identifiers (libxml2's `xml-` and `xpath`-prefixed function names, bash's `builtin`, `shell` and `readline` names) and telling file extensions (`.dtd` and `.rng` are XML on sight), and those are exactly the tokens a recovery method must read. The honest conclusion is therefore not that recitation decays under stripping, but that **we could not construct a substrate on which the system stops being recognizable, short of removing the information that makes the task recoverable at all.**

4.7.6 Did the Explorer Know What It Was Reading?

Everything measured so far compares a run's component names against names the model can produce from the project name alone. High overlap is consistent with two different stories. The model may have recognized the system and recalled its published architecture. Or it may have analysed the code correctly and arrived at the same names independently, because "Job Control" is the right label for code that reaps dead jobs whether or not one has heard of bash. Name overlap cannot separate them, because both stories predict it.

Asking a fresh agent to identify the codebase does not separate them either. An agent told to find out what a project *is* explores differently from one told to recover its architecture, and its conclusion rests on different observations than the ones that produced the component names. Run that way on these substrates, a separate

agent named the correct project in **0 of 20** attempts: on libxml2 it inspected the opaque file paths, concluded from them that the graph was anonymized, and in 8 of 10 trials never queried a node name at all; on bash it did query names and answered **GNU Readline** in 6 of 10 trials, naming the line-editing library bash bundles rather than bash itself. Neither outcome speaks to what the recovery runs knew.

The question is therefore put to the recovery runs themselves. Each saved A-KG trajectory is replayed verbatim up to, but not including, the turn in which the agent submitted its components, and the conversation is continued with one further turn asking whether the model knows which specific project it has been exploring, with an explicit statement that an honest "unknown" is a correct answer. The naming branch and the identification branch share an identical evidence base, message for message. No new exploration is performed and no substrate is touched: this is one additional question put to evidence already on disk.

TABLE 21

Project identification obtained by forking each A-KG run on the deeply identity-stripped substrate at its final turn. The context is the run's own exploration, unchanged.

Substrate	correctly named	unknown	wrongly named	context length (messages)
bash	**7 / 10**	3	0	16 to 34
libxml2	**5 / 9**	4	0	14 to 36

Twelve of the nineteen runs named the system correctly, every one at high confidence, and no run named a wrong project. The evidence the model cites is the retained identifier floor: for bash, *"job control and pipelines (`start_pipeline`, `wait_for`), readline integration, shell builtins (`shopt_builtin`, `getopts`)"*; for libxml2, *"`xmlParserHandlePEReference`, `xmlSAXParseDTD`, `xmlSchemaValidateFacet`, `xmlXPathEvalExpression`"*.

Two consequences follow. First, for those runs the competing explanation is closed: the model that wrote *XPath Engine* and *Job Control and Process Management* knew, from the same observations, that it was reading libxml2 and bash. The naming is recall. Second, the standing agent probes *understate* what the recovery runs knew: 0 of 20 against 12 of 19 on the same substrates. An agent asked to identify a codebase performs worse at it than an agent that was asked to do something else and identified it along the way, which is a caution about probing for recognition directly.

4.7.7 Canonical-Overlap Across the Full Suite

The canonical-overlap scorer (Method §3.5.2) applies the same measure to the whole six-system suite on the real substrate, comparing each run's proposed component *names* against two references: the zero-shot elicitation probe (recitation) and the directory vocabulary of the real tree (structure-riding). Both views are regenerable from the deterministic scorer and the run snapshots, and the cross-codebase ranking is stable under removing the stopword list.

TABLE 22

Canonical-overlap by channel per codebase, averaged over A-F, A-F+KG, and A-KG on the real substrate (n=30 runs per row). The recitation channel is reported as both containment and coverage against the probe; the structure channel is reported as containment only, because the directory vocabulary ranges from 13 tokens (bash) to 213 (ITK), so coverage against it is not comparable across codebases.

Codebase	recitation (cont)	recitation (cov)	structure-riding (cont)	Dominant channel
bash	0.46	0.43	0.13	recitation
libxml2	0.74	0.27	0.24	recitation
hadoop	0.65	0.25	0.46	recitation (both high)
ITK	0.44	0.14	0.90	structure-riding
archstudio4	0.28	0.15	0.65	structure-riding
JPetStore	0.45	0.33	0.55	structure-riding

The two shortcuts are separable and codebase-specific. The famous, extension- or identifier-rich systems (libxml2, hadoop, bash) name from recalled architecture, whereas the systems whose gold *is* their module tree (ITK, archstudio4) name from folder vocabulary. Read against Table 20, this is the same substitution seen there: where the directory channel is weak the recitation channel carries the naming, and where it is strong the naming rides it instead.

Cue-presence: the in-run mechanism is identifier-riding. For each canonical (probe-matching) name a run emits, we check whether any tool result the agent saw contained a supporting token: across the trajectory configurations, **73 to 100% of canonical names had such a cue present**, so identifier-riding, not blind template recall, is how the memorized architecture reaches the output within a run.

4.7.8 What This Section Establishes

The canonical architecture of each study system is resident in the model's weights and is recoverable with no code at all: eighteen probe trials, six systems, every trial producing the documented decomposition from the project name alone, for three and a half cents.

On a substrate from which the project name, every file and directory name, every comment and copyright header, and every residual identity literal have been removed and verified absent, the knowledge-graph-only explorer still names the study system's own subsystems. The two scorer outputs show two different effects of the transformation. Coverage falls: the runs enumerate fewer of the documented subsystems once the removable cues are gone. Containment holds steady: what they do name is still cast in that system's terms. The transformation removed evidence without removing identifiability. Within a run, 73 to 100% of the canonical names had a supporting token in a tool result the agent had already seen, so the channel is the retained identifier vocabulary rather than blind template emission.

The competing explanation, that the model simply analysed the code correctly and arrived at the same names, is closed for the runs we could test. Forking each run at its final turn and asking the *same context* which project it had been exploring, twelve of nineteen answered correctly at high confidence and none named a

wrong project (§4.7.6). The model that wrote those component names knew, from those same observations, which system it was reading.

Removing the directory tree does not suppress this; it amplifies it. Across the wider suite the two channels trade off by codebase: recitation dominates where identifiers carry the domain, structure-riding dominates where the gold is the module tree. The conclusion is narrower and sharper than "recitation is robust": **renaming what can be renamed does not anonymize a system for this task.** The identifiers a recovery method must be able to read are sufficient to tell the model which system it is reading, and once the system is identified its published decomposition is already available. Component naming on these benchmarks is therefore not separable from memorized-architecture recall.

This is a descriptive finding about what these methods do, not a claim that any configuration is better or worse. No configuration in this section is scored against the gold.

4.7.9 Threats Specific to This Section

- **Scope.** The stripping transform rewrites `#include` graphs and file names, which is sound for C and C++ and unsound for languages whose dependency resolution is name-bound (threat T9). Every claim here is bounded to bash and libxml2, two C systems, on one model at temperature zero.
- **The retained floor is the limit of the technique, not a clean substrate.** Code identifiers and file extensions remain by design (Table 19). The measured residual is an upper bound on what identity stripping can remove, and a substrate that also removed identifiers would no longer support architecture recovery.
- **The scrub is incomplete in file contents, which the reported runs did not read.** The residual-token audit of Table 19 checked a fixed list of project-identifying phrases; it did not check for bare `GNU`, `GNOME` or `W3C`, for the author name `daniel`, or for original identity strings surviving inside `File.contents`. All of these remain: on a case-insensitive substring match, 14% of bash's file nodes contain `GNU`, `Free Software Foundation` or `DISTVERSION`, and 20% of libxml2's contain `GNOME`, `W3C`, `daniel` or `Veillard`. The verification claim is therefore narrower than a clean substrate: it establishes that a specific list of phrases was removed, not that the substrate is anonymous. What licenses the results reported here is behavioural rather than constructive. Across all forty A-KG runs on these substrates, **no run queried `File.contents` at any point**, so no leaked string entered an exploration context. The allocator does read contents, by design, but it runs after naming and takes no part in it. A reader who does not accept that behavioural argument should treat §4.7.5 and §4.7.6 as bounded to the runs whose trajectories are on disk and checkable.
- **The transformation is detectable, and detection changes behaviour.** Renaming files to `f_0097.c` and directories to `dir_0008` announces that the substrate has been altered. A separate agent asked to identify the codebase remarked on this in 10 of 10 libxml2 trials and, in 8 of those, stopped investigating without ever querying a node name. A substrate renamed to plausible but incorrect names would not signal interference in this way, and would be the better instrument.
- **Containment is lexical.** It counts distinctive token overlap against the probe, not semantic equivalence, and it is asymmetric by construction. It supports "these names are already producible without

the code"; it does not support a claim about whether the decomposition is *correct*.

- **The per-run counts depend on a cut in a continuous measure.** The 8-of-10 and 5-of-10 figures apply a coverage threshold, chosen to fall inside a gap that is clear on both systems but is not itself a natural boundary (Appendix B.6). The underlying measure is continuous, and the counts should be read as a summary of the distribution rather than as a count of two discrete kinds of run. Every per-run value is published so the cut can be moved.
- **Unequal cell sizes.** The crossed design in Table 20 is n=10 on the real-tree row and n=5 on the stripped row, with one cell at n=4 where a run recorded no component names.
- **No causal manipulation.** This section establishes that the canonical naming is not removable by any substrate transformation available to us. It does not establish that the behaviour can be switched off and on, and no claim of that form is made.
- **Run-to-run variance applies here too.** Every cell is a distribution over repeated runs at temperature zero, reported as a per-cell median with its n.

4.8 Directory Lens and Deterministic Reference Partitions

§4.6 used the covariate `ARI(directory_at_depth_K, gold)` (threat T1) as a *predictor* of how far each flat configuration would drop. This section treats the directory partition as a standing **diagnostic lens** applied to every codebase: for any recovered partition, how much is already readable from the folder tree before any agent, model, or knowledge graph is invoked? The lens serves as a yardstick for *attribution*, separating the fraction of an output that is folder-explained for free from the fraction that requires content-level or relational reasoning. It is a diagnostic instrument, and the agentic configurations are not ranked against it.

4.8.1 The Immediate-Parent and dir@K Construction

The base lens is the immediate-parent directory partition already defined and scored in Method §3.6/§3.8, abbreviated here as **IP**: group every gold-labeled file by its immediate parent directory and score that partition against the gold (canonical `mojo.jar`, sklearn ARI). This section extends it with the **dir@K sweep**, generalizing the partition to directory prefixes truncated to depth `K = 1, 2, 3, full-dirname`, reading off the depth at which the gold is most directory-aligned (§4.6.5, threat T1). Both are deterministic functions of the filesystem, computed once per codebase with no run-to-run variance.

Table 23 reports both lenses on the four codebases with LOCKED gold/universe/baseline counts, plus the two supplementary systems (libxml2, jpetstore) at the partial scope available for them. That scope is gold sizes only, with no universe/distractor rebuild (see §4.9 and the open-numbers note in the provenance appendix).

TABLE 23

Deterministic directory lens per codebase. *IP MoJoFM / IP ARI* are the immediate-parent partition scored against the cleaned gold (Method §3.8). The dir@K columns are `ARI(dir@K, gold)` over the canonical sweep. **Best-depth** names the depth at which the gold is most directory-aligned and the group count at that depth. Numbers transcribed from the canonical consolidated scorer, fidelity-checked 36/36 against the scored-run ledger.

codebase	IP MoJoFM	IP ARI	dir@1	dir@2	dir@3	full-dirname	best-depth (groups)
bash	0.939	0.817	0.406	**0.817**	0.817	0.817	0.817 @ dir$\geq$2 (11)
hadoop	0.691	0.392	0.444	0.370	0.390	0.392	0.444 @ depth 1 (14)
archstudio4	0.832	0.252	0.057	**0.853**	0.276	0.252	0.853 @ depth 2 (43)
ITK	0.945	0.151	0.000	**1.000**	0.290	0.151	1.000 @ depth 2 (11)
libxml2	n/a	n/a	0.015	**0.018**	0.018	0.018	0.018 @ dir (5)
jpetstore	n/a	n/a	0.000	0.000	0.000	-0.050	0.000 @ depth 1 (1)

The table sorts the suite onto a single axis (*how much architecture is folder-readable for free*), and the six codebases land at distinct points on it.

ITK is, at depth 2, its own gold. `ARI(dir@2, gold) = 1.000`: the eleven gold components are the eleven `Modules/<X>/` subdirectories bit-for-bit. For ITK the directory lens is the whole construct (threat T1), which is why ITK collapsed hardest under flattening (§4.6.4).

bash's gold is *almost* its depth-2 directory tree, saturating at depth 2 (dir@2 = dir@3 = full-dirname) at ARI 0.817, reproduced exactly by the immediate-parent partition. The §4.3.5 finding that bash A-F partitions agree with the directory partition and with the gold to within 0.02 ARI is therefore read against a tree that is itself 0.817 against gold: a bash A-F run at ARI $\approx$ 0.26 is a *fixed coarsening* of an already strongly-aligned tree.

archstudio4 is depth-sensitive. Its gold is readable from the folder tree, but *only* at exactly depth 2 (0.853): the top-level grouping is uninformative and the deepest grouping over-fragments. This matters for reading §4.5: A-KG's recurring subsystem names on archstudio4 (Archipelago, TraceLink, XArchCS, ...) coincide with the depth-2 directory names, so what could be read as the KG recovering documented subsystems is equally consistent with a codebase whose depth-2 folders already encode them. The lens records that both descriptions point at the same depth-2 partition.

hadoop is the one codebase the directory lens does *not* explain. Its best directory alignment is 0.444 at depth 1 and it does not climb at depth 2: the gold cuts across the folder tree at every depth, so hadoop is the codebase where gold agreement is *least* attributable to folder-reading, consistent with the genuine A-KG ARI lift there (§4.5.2, Δ ARI +0.118 vs A-F+KG).

The two supplementary systems bracket the floor. libxml2 (`ARI(dir@2, gold) = 0.018` $\approx$ `0`) and jpetstore (0.000) carry essentially no directory signal. libxml2 is the substantive case, and it is where the KG-only configuration produced its largest MoJoFM lift over the shell baseline (§4.5.2, +0.144 vs A-F) and where flattening did *not* degrade recovery (§4.6.4). jpetstore's 0.000 is degenerate for the orthogonal layer-versus-domain reason documented at §4.3.4, and we do not read its lens as a quality signal.

4.8.2 Louvain-on-Dependency-Graph and Louvain-on-KG Reference Partitions

The Method chapter specifies two *algorithmic* deterministic reference partitions as additional non-LLM lenses: Louvain community detection on the basic file-level dependency graph (the `DEPENDS_ON_FILE` edges), and Louvain on the enriched KG projected to file level (all file-file relational edge types plus the entity-level relational edges lifted to their containing files). Both were executed once per codebase with the seeded, reproducible runner of Section 3.6 (seed 42, with isolated files forming a single orphan bucket) and scored through the same consolidated scorer as every other partition. Table 24 reports both lenses per codebase.

TABLE 24

Deterministic Louvain reference partitions per codebase: Louvain on the file-level dependency graph (dep) and on the enriched, file-projected KG (enriched), scored against the cleaned gold. *Isolated* counts files with no edge in the variant's graph (pooled into one orphan bucket). Libxml2's dependency graph is sparse (254 edges over 1,159 File nodes), so its lens is dominated by that bucket and read as weak evidence only.

codebase	variant	edges	isolated	comps	MoJoFM	ARI
bash	dep	2,125	37	6	0.516	0.220
bash	enriched	2,125	37	6	0.498	0.198
hadoop	dep	5,192	7	8	0.374	0.218
hadoop	enriched	5,495	6	7	0.330	0.210
archstudio4	dep	3,181	14	9	0.511	0.445
archstudio4	enriched	3,575	11	8	0.461	0.449
ITK	dep	20,435	1,828	33	0.337	0.083
ITK	enriched	26,644	1,819	32	0.384	0.092
libxml2	dep	254	1,098	4	0.164	0.033
libxml2	enriched	254	1,098	4	0.151	0.009
jpetstore	dep	118	94	5	0.900	0.758
jpetstore	enriched	122	94	5	0.700	0.388

Three readings follow, each answering a question the directory lens alone could not.

Dependency cohesion and directory cohesion come apart, in both directions. The sharpest case is jpetstore: the directory lens scores 0.000 there (§4.8.1) and every LLM configuration is pinned at or near ARI -0.05, below chance (§4.3.4), yet Louvain on the bare dependency graph recovers the three business services at ARI 0.758 / MoJoFM 0.900 (inspection of the partition confirms near-pure order, catalog, and account clusters). The gold that layer-mirroring methods cannot see is nearly free to a method that clusters by dependency instead. The mirror image is ITK: its gold *is* its depth-2 directory tree (`ARI(dir@2, gold) = 1.000`), and dependency-graph Louvain reaches only ARI 0.083. Where the answer is the folder tree, dependency cohesion does not reconstruct it.

A deterministic clusterer matches the forced-KG agent on archstudio4. Louvain-on-dependencies scores ARI 0.445 there, essentially the A-KG median of 0.457 (§4.5.2), and both sit below A-F's 0.537 on the same codebase. A clusterer with no LLM at all reaches the forced-KG agent's quality, so whatever signal the KG supplies on archstudio4 is already available from the plain dependency graph. On hadoop, by contrast, A-KG (median ARI 0.491) sits above Louvain (0.218), so the agent contributes something beyond mechanical graph cohesion on the one codebase whose gold the directory lens explains least.

The enriched projection adds nothing consistent. Across all six codebases the enriched variant moves ARI by at most a few hundredths relative to the dependency-only variant, and on jpetstore it *degrades* it (the lifted call-graph edges pull the partition back toward technical layers), so the answer to "is the richer KG doing hidden work?" is no: whatever load the graph bears is already in the plain file-dependency edges.

4.8.3 What the Lens Establishes

The directory lens sorts the suite onto a single folder-readability axis (Table 23) against which every aggregate recovery score must be read, and the Louvain lens (Table 24) adds the complementary axis of how much of each gold is dependency-readable without any LLM: where `ARI(dir@K, gold)` is high, a high recovery score is at most *(score − directory-readable contribution)* of genuine content- or relation-driven recovery (the decomposition §4.6.5 instantiated, with the T1 reporting recommendation recorded there).

4.9 External Comparators (Qualitative Positioning)

SARIF [11], the strongest non-LLM multi-source fusion recovery of the 2020s, and SemArc [4], the strongest LLM-augmented recovery, both of which evaluate on benchmark suites overlapping this study's core systems (SARIF on all four, SemArc on bash, hadoop, and archstudio4).

There is no quantitative head-to-head in this study. No SARIF, SemArc, or related recovery has been rescored under this study's metrics and golds, so we present *no* comparator numbers. Transcribing a SARIF / SemArc figure beside our own would be the metric-convention error this study avoids: MoJoFM direction and granularity, gold-truth version, and codebase-version drift (e.g. hadoop 0.19.0 vs our 0.19.1-dev) each move the score independently of method quality. The comparison here is qualitative and methodological, and the absence of a rescore is stated openly as a scope decision.

Positioned qualitatively, the prior LLM-SAR work and this study sit on different axes. SemArc uses the LLM *non-agentically* (two batch annotation passes feeding a classical anchor-guided community-detection clustering, single-run numbers at temperature 1). SARIF uses no LLM, fusing typed dependencies, LDA-topic text, and quality-filtered folder structure with per-project adaptive weights. This thesis instead treats recovery as an agentic loop over a knowledge-graph substrate and measures *what the methods do* (§§4.4 to 4.8). The two bodies of work ask different questions of the same systems.

The contamination / recitation construct applies to the comparators too, and this is the load-bearing methodological point. The benchmark suite is, by construction, drawn from the most-documented software in existence, and is therefore in the pretraining corpus of any modern LLM (threat T2). The §4.7 result is a property of the *task and the model class*, not of our pipeline alone. SemArc's LLM passes summarize the files whose architecture the model has read about. Its bash case study reports that the tool "yields even more precise recovery results than those labeled by humans", a claim that, viewed through §4.7 and T2, is equally consistent with the model reciting bash's well-documented decomposition as with it discovering one. SemArc's own ablation sharpens the exposure: removing its code-level semantic channel (the LLM's per-file summaries) costs roughly twenty points, the single largest factor in its reported gains. The component most driven by the model's reading of these famous files is therefore the one most open to recall, and contamination is not measured. A SARIF-style non-LLM tool is immune to recitation but not to the directory-lens artifact of §4.8: the Garcia group reports that SARIF must have its absolute-file-path input disabled before a valid comparison is possible, because that path encodes the ground-truth module boundaries [49], and a non-LLM graph-representation recovery likewise gains gold alignment mainly by drifting toward folder structure [48]. Neither comparator was evaluated with the diagnostic tools this study builds. Their reported headline gains carry the same unmeasured structure-riding and contamination components this thesis makes explicit. The

contribution of this study is those tools.

A quantitative rescore is out of scope and is future work. Both tools ship public artifacts, and bringing them onto this study's golds, metrics, and diagnostic tools would convert the qualitative positioning above into a head-to-head measurement of how much of each comparator's reported gain is folder-readable or documentation-recalled. The identifier-floor caveat bounds even that rescore: on a memorized system the in-code identifiers any semantic method must read are the same cue recitation rides (§4.7), so the clean test is a codebase the model has not memorized.

4.10 Cost

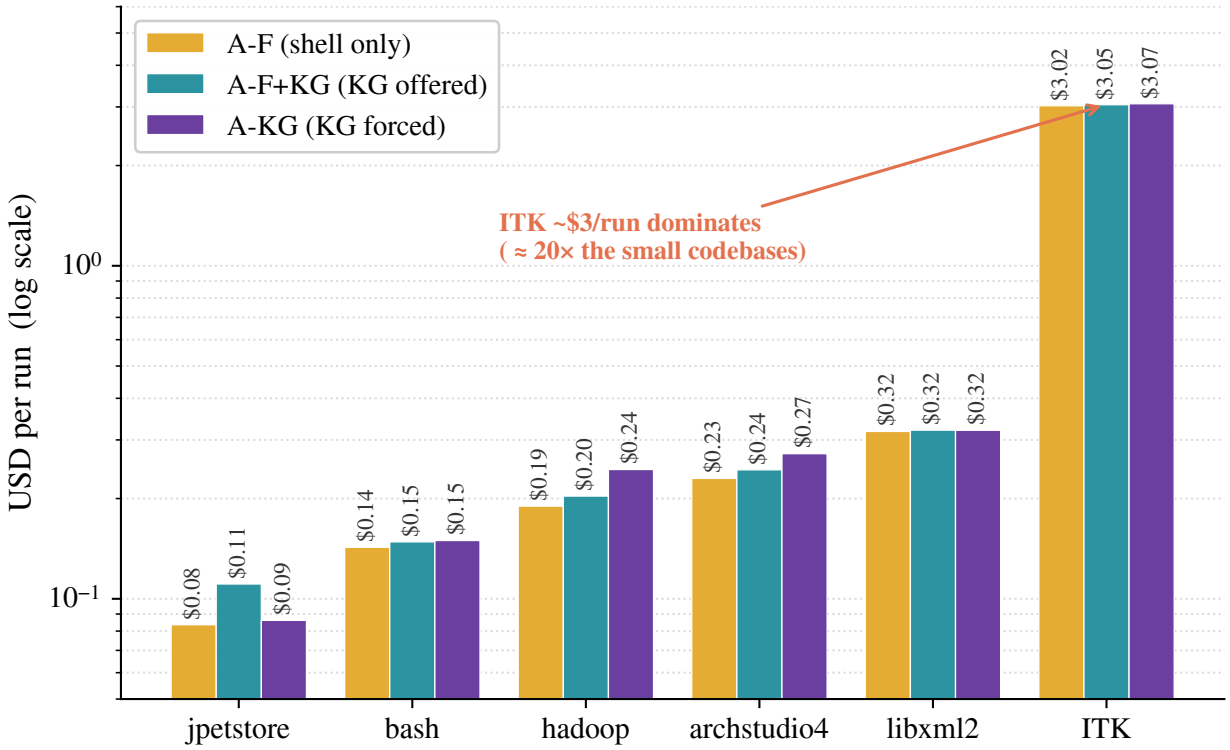


Figure 10: Mean USD per run by codebase and configuration (log scale): cost is set by codebase size, not configuration, with ITK dominant at roughly $3/run. Mean USD per run on gpt-5.4-mini, n=10 per cell; source: the per-run logged cost over the scored grid.

The costs in this section are reported as a reproducibility disclosure, not as a study axis: readers who wish to reproduce or extend the experiments can use them to budget the work. Runs were logged with a per-run USD `cost` field (model `gpt-5.4-mini-2026-03-17`). The costs reported here are correct *internally* (directly comparable across this study's configurations) first and cross-paper-comparable second. For the latter, raw token counts are the durable artifact, and the elicitation-probe counts below are reported raw for that reason.

4.10.1 Phase-1, Phase-2, and Probe Cost

The scored Phase-1 grid (the 230 success-gated runs underlying §§4.3 to 4.6) cost **$156.96** in total. Tables 25 and 26 break it down by configuration and by codebase.

TABLE 25
Phase-1 scored grid cost by configuration (per-run cost as recorded in the scored-run ledger, 230 rows).

config	n rows	total USD	mean USD/run
A-F	60	39.87	0.665
A-F+KG	60	40.73	0.679
A-KG	60	41.39	0.690
A-F-flat	27	22.55	0.835
A-KG-flat	23	12.41	0.540

Cost is roughly flat across the three main ladder configurations (mean $0.66 to $0.69/run). The flat-variant cells differ from each other only because their codebase mix differs: A-F-flat includes seven ITK runs and A-KG-flat three. The dominant cost driver is not the configuration but the **codebase** (Figure 10): ITK alone accounts for roughly $2.93/run versus $0.08 to $0.32 for the small codebases (Table 26).

TABLE 26
Per-codebase Phase-1 cost (USD/run by configuration; A-F n=10 total).

codebase	A-F $/run	A-F+KG $/run	A-KG $/run	A-F total (n=10)
ITK	3.023	3.045	3.066	30.23
libxml2	0.318	0.321	0.321	3.18
archstudio4	0.230	0.244	0.273	2.30
hadoop	0.190	0.203	0.244	1.90
bash	0.143	0.148	0.150	1.43
jpetstore	0.084	0.111	0.086	0.84

ITK's cost dominance reflects its scale (5,228 gold files, 8,547-file universe). Across *all* logged runs (including failed runs, dead substrate variants, and the Phase-2 recitation arms) the grand total is **$419.85** over 630 logged runs, of which the $156.96 scored figure is the defensible per-paper cost. The itemized figures in this section are components of that grand total: the total also covers runs not itemized here, including the identity-stripped substrate cells.

The Phase-2 recitation work is comparatively cheap, and all three of its instruments together cost under half a dollar. The zero-substrate elicitation probe of §4.7.2 cost **$0.0354**: 18 calls (6 codebases × 3 trials), 9,482 tokens (1,950 prompt + 7,532 completion). The identification fork of §4.7.6 cost about **$0.15**: 19 calls

replaying a saved trajectory apiece, roughly 194,000 tokens, dominated by the replayed context. A separate standing identification agent, reported in §4.7.6 for contrast, cost **$0.21** over 40 trials (253,437 tokens). Token counts are given raw for cross-paper comparability; the fork's are recounted offline from the saved contexts rather than metered at call time.

4.10.2 Cost Versus What Is Learned

The most expensive artifact in the study (ITK, roughly $3/run) is the codebase that teaches the *least* about semantic recovery and the *most* about benchmark hygiene (§4.3.4, §4.8), while the cheapest instruments are among the more informative: the $0.0354 elicitation probe establishes that the architectures are already in the model's weights, and the $0.15 identification fork supplies the evidence that the recovery runs knew which system they were reading. We draw no cost-per-quality claim from this, since "quality vs gold" largely re-measures structure-riding (§4.6, §4.8). The dollar cost of these runs tracks file count; it does not track how much each run reveals.

4.11 Qualitative Analysis

The aggregate tables of §§4.3 to 4.7 establish *what* the configurations do on average. They do not show *how* a single run arrives at its number. This section reads individual runs. The cases are not chosen to illustrate a thesis. They are selected mechanically from the scored corpus under four locked criteria, and only then read. Each criterion's disposition:

1. the **worst MoJoFM** cell across all config×codebase pairs. The worst *cell* is jpetstore, pinned at exactly 0.350 / −0.050 in all three configurations by the degenerate layer-versus-domain mismatch of §4.3.4. A pinned constant carries no qualitative mechanism, so we read the worst *single run* instead: A-F / libxml2 (`run_id 308610c3-ab19-413c-97a6-4586175235ad`, MoJoFM 0.2466, ARI 0.150, 5 components), an under-segmented partition whose mega-buckets fold six or seven gold components together, the §4.8.1 libxml2 mechanism (`ARI(dir@2, gold)` $\approx$ `0.018`, no directory signal to ride) exhibited in a single run.
2. the **highest run-to-run IQR** (the config×codebase whose ten runs disagree most): A-F+KG / ITK, read in §4.11.2.
3. the **largest gap against the directory (immediate-parent) baseline** of §3.8 / Table 23. Furthest below on MoJoFM is the flat-ITK collapse read in §4.11.1. Furthest above is inside the A-F+KG ITK cell of §4.11.2. Furthest below on ARI is A-F+KG / bash (`run_id d9645db5-ad52-4e30-9418-14efbac70611`, ARI 0.121 against the 0.817 baseline, $\Delta = -0.696$), the §4.3.5 coarsening dynamic at its most extreme: a five-component run whose 238-file utility catch-all absorbs the four support-library trees the gold keeps separate.
4. the **largest two-run disagreement** within one cell: the same A-F+KG ITK cell as criterion 2, deduped into §4.11.2.

For each case read in full we state the selecting number and the `run_id`, then read the partition artifacts

against the gold. Three of the four criteria converge on the same ITK A-F+KG cell and are read together. The cases are descriptive, each illustrating a mechanism already characterized in the preceding sections.

4.11.1 Largest gap below the directory baseline: A-F-flat ITK, run 345d5821, flattening collapses the partition

Selection. Against the per-codebase immediate-parent baseline of Table 23 (ITK IP MoJoFM = 0.945), the run furthest **below** baseline on MoJoFM is **A-F-flat / ITK, run index 0**, `run_id 345d5821-a4f5-4891-a0b6-acb529d129f6`: MoJoFM 0.2822 ($\Delta = -\mathbf{0.663}$), ARI 0.0046, 5 components, cost $2.59. This is a structure-riding-probe run (A-F against the directory-flattened ITK substrate of §4.6), and it is a single-run illustration of the flat-ablation result.

What happened. With every file relocated to a single root under a unique flat name, `find` returns the file universe but no directory segments to group by. The resulting partition has five components, and **7,513 of the 8,547 files (88%) land in one bucket**, *"ITK Utility and Example Programs"*. The only structure recovered is the vendored third-party libraries still identifiable from in-file content and identifiers (*"GDCM DICOM Toolkit"*, two HDF5 buckets, *"DICOM Parser Support"*). Against the 11-component gold this scores ARI 0.0046, essentially the no-information partition. Canonical A-F ITK medians 0.917 MoJoFM / 0.756 ARI by reading the directory tree. Flattening that tree removes 0.66 MoJoFM and almost all the ARI, most severely on ITK because its gold *is* the directory tree (`ARI(dir@2, gold) = 1.000`). The flat substrate removes the one signal the configuration was riding, and the partition collapses.

4.11.2 Highest run-to-run IQR and largest two-run disagreement: A-F+KG ITK, depth-1 vs depth-2

Selection. Computing the inter-quartile range of MoJoFM and ARI over the ten `run_i` of every cell, the **largest IQR on both metrics is A-F+KG / ITK**: IQR(MoJoFM) = 0.131 (range [0.601, 0.991]) and IQR(ARI) = 0.477 (range [0.141, 0.986]). No other cell's ARI IQR exceeds 0.31. The same cell wins criterion 5: over every pair of runs within a cell, the largest absolute ARI difference is **0.844**, between **run index 3** (`run_id c7b7fbd1-6461-4e0b-b49a-59e8ef73a896`, ARI 0.986, 12 components) and **run index 7** (`run_id f496d893-f126-41e8-b66b-95a175e40ac6`, ARI 0.141, 46 components), both `exit_status=success` at temperature 0, both cost ˜$3. Run index 3 is additionally the corpus's furthest-*above*-baseline run (criterion 3: ARI 0.986 vs the IP-baseline 0.151, Δ +0.835).

What happened. ITK MoJoFM is granularity-insensitive (§4.3.4), so the MoJoFM spread is modest. The *ARI* spread is the real story, and it is **bimodal, not noisy**. The ten runs fall into two families distinguished entirely by the **directory depth** at which the agent cut the tree: the high-ARI family (ARI 0.986, 0.946, 0.894) cuts at ITK's module boundary and proposes ˜11 to 12 components, while the low-ARI family (ARI 0.141, 0.294, 0.307) cuts one level deeper and proposes 30 to 48. Because ITK's gold is exactly its 11 depth-2 modules (`ARI(dir@2, gold) = 1.000`, Table 23), the choice of cut depth is *decisive*. The extreme pair makes this concrete: run_i 3 cuts at **depth 1** and proposes the twelve ITK top-level modules verbatim (*Core*, *Filtering*, *IO*, *Numerics*, ..., *ThirdParty*), an almost exact match to the gold's 11 `group_*` clusters (ARI 0.986), while run_i 7 cuts one level **deeper**, expanding *Core* into 22 sub-buckets and *Filtering* into ˜64 for 46 leaf components. Every sub-bucket is a real ITK subdirectory, so the partition is internally coherent. It is

simply at the *wrong granularity* relative to the 11-component gold, and ARI falls to 0.141. Same input, same configuration, same temperature: the only thing that varies between the corpus's two most-disagreeing runs is the level of the directory tree the agent chose to stop at. On a perfectly directory-aligned gold, recovered "quality" is essentially a referendum on cut depth.

4.11.4 What the Cases Show

Read together, the cases turn on two mechanisms already named quantitatively in this chapter, and nothing else: granularity indeterminacy on directory-aligned golds (§4.11.2, where depth choice *is* the score), and structure-riding and its removal (§4.11.1 and the below-baseline dispositions in the selection list). None of the cases shows a configuration recovering architecture better than another in a way that survives the caveats of §4.6 and §4.7. The qualitative record corroborates the diagnostic reading of the aggregate tables by exhibiting the partitions, with their `run_id`s, on which the numbers were computed.

4.12 Summary of Results

This chapter characterizes *what LLM-driven architecture recovery actually does* on a suite of four core canonical benchmark systems plus two supplementary exhibits. Read end to end, the results converge on a single diagnostic headline with three components, each established on a different part of the apparatus and each qualified codebase-by-codebase:

- **Structure-riding.** Much of every "recovery" is directory-readable for free, in a proportion predicted in advance by the covariate `ARI(dir@K, gold)` (§4.6, §4.8). At the extreme, ITK's gold *is* its depth-2 directory tree (`ARI(dir@2, gold) ~ 1.000`).
- **Recitation.** The component *names* are reconstructed memorized architecture, resting on a code-identifier floor that no available substrate transformation removes (§4.7). Stripping the codebase's identity from every name, comment and literal leaves probe-containment unchanged on bash and plateaued well above zero on libxml2, and removing the directory tree *raises* it.
- **KG under-use and variance.** Offered alongside the shell, the KG is used as a fancier `find`. Forced, it lifts quality on most informative codebases but reverses where the gold is the directory tree, and large run-to-run variance persists throughout (§§4.4 to 4.5).

The diagnostic contribution. These results decompose a single recovered partition into three attributable parts: a folder-readable part (predicted by the directory lens), a memorized-and-recited part (the component names, resting on the identifier floor), and a modest, behaviourally-contingent part that the relational KG actually moves. On this suite, the first two parts dominate. The contribution is the apparatus that makes the decomposition observable: the directory lens of §4.8, the flat-variant ablation of §4.6, and the identity-stripping transform of §4.7. Prior LLM-SAR and classical comparators were evaluated without such tools (§4.9).

A common cause with three signatures. The directory-aligned golds and the pre-training contamination (threats T1 and T2) are one common cause with three visible signatures. Together they predict that recovery reproduces the directory groupings (structure-riding) and recites memorized names (recitation), the RQ3

headline. They predict that a relational knowledge graph adds little on this suite (RQ1, read as a symptom scoped to the suite, not a general claim about KGs). And they predict that run-to-run variance is suppressed (RQ2: recall and folder-riding both land in the same place, so the reported variance is likely an underestimate). Reading the findings this way removes any appearance of self-undermining: each survivor is either presence-of-pathology or a contamination-independent measurement. Two results are not folded into the common cause and stand on their own, that the KG under-use is a behavioural choice rather than a competence failure (§4.5.3), and the distributions-over-point-estimates methodology.

5. Threats to Validity

This chapter lists the residual threats that constrain the conclusions, grouped into construct, internal, and external validity. Each entry states the threat, its mitigation, and what risk remains despite the mitigation. The two most serious are the first two: the reference architectures on this suite are themselves largely directory groupings (T1), and the benchmark systems are in the model's training data (T2).

Construct Validity

T1: directory-aligned golds. The RSF gold truths, though expert-authored, are substantially recoverable from the directory tree: ITK's gold maps essentially one-to-one onto `Modules/<group>` (`ARI(dir@2, gold)=1.000`), archstudio4 and bash reproduce a depth-two grouping at high agreement, and hadoop is the least directory-aligned but still moderate. Any gold-based metric on this suite therefore primarily measures **directory reconstruction**: a method that mirrors folders is rewarded, one that recovers a non-directory but architecturally sensible decomposition is penalized, and a raw score is uninterpretable without knowing the gold's directory alignment. Prior acknowledgement of RSF directory bias exists [1]. The golds are not themselves folder-derived, and their alignment varies (roughly 0.02 on libxml2 to 1.00 on ITK) [3]. *Mitigation:* report `ARI(dir@K, gold)` beside every recovery score as a benchmark-hygiene covariate (a method scoring X where the gold is directory-readable at Y demonstrates at most the residual recovery beyond Y), report the covariate across depths with the immediate-parent value locked in advance and not tuned against the gold, and read every gold score relative to the directory reference rather than as a standalone win or loss. *Residual risk:* even with the covariate reported, no gold-based number on this suite can certify semantic recovery, and scores in this literature overstate it.

T2: the benchmark codebases are in the model's pre-training corpus. bash, libxml2, hadoop, ITK, and archstudio4 are among the most-documented open-source systems, their architectures appearing in material almost certainly present in the training data of any modern code LLM, and the elicitation probe shows the same recall for jpetstore. Recall of this kind is strongest for such popular, heavily duplicated systems [46]. The zero-substrate elicitation probe, the identity-stripped substrates, and the single-token flip establish the effect empirically on this suite (§4.7). A high score on these benchmarks can therefore reflect memorized-architecture recall as much as analysis of the artifact in hand, and the same critique applies to prior comparators evaluated on the same suite without a contamination check. *Mitigation:* report each codebase's elicitation-probe coverage beside its recovery score, pair every score with the directory covariate (T1), and bound recitation claims to the studied codebases. *Residual risk:* on these systems, recall and analysis cannot be fully separated by any score, and generalization to non-memorized codebases is untested (External Validity).

T3: gold provenance and construction. The core golds are inherited from the Lutellier-derived corpus, so its construction choices propagate into every score, as do our own cleaning decisions (Section 3.7, full change log in Appendix A). The two supplementary golds are of separate provenance and different kinds (libxml2 is a basename-keyed group/item hierarchy, jpetstore a class-level service decomposition, Appendix A.3.5 and A.3.6). They run through the full grid and are gold-scored, but they are read only as disclosed

structure-riding exhibits and are not pooled with the core systems (Appendix A.6). ITK's gold equals its developer module structure, an easy special case, so conclusions are weighted toward bash, hadoop, and archstudio4. *Residual risk:* the golds remain a single lineage, and any bias in their construction is inherited by every gold-scored result.

Internal Validity

T4: run-to-run variance, small N, and variance as a compressed signal. LLM non-determinism means a single run is not representative, and with ten runs per cell the estimated distributions are themselves uncertain. Six codebases, two of them structurally degenerate for quality scoring, are too few for significance tests (Section 3.9), so paired deltas are read against within-cell spread. The variance is also biased in a known direction: a run that recalls the same canonical decomposition, or rides the same directory tree, lands in the same place regardless of sampling, so recall and folder-riding compress variance, and the reported variance likely understates the instability of genuine analysis, a condition this study did not measure. *Residual risk:* low variance in any configuration should not be read as stability of recovery. It may be stability of recall.

T5: coverage confounds the gold metrics where the agent self-allocates. MoJoFM and ARI are computed on the intersection of the recovered partition with the gold, which rewards under-coverage. The fixed-allocator configurations reached 100 percent coverage in every reported run, but A-Solo under-covers, sharply on ITK, so its scores and part of its apparent run-to-run variance are coverage effects. *Mitigation:* coverage is reported as a first-class column beside both metrics (Section 3.4). *Residual risk:* Tier-1 comparisons across codebases still mix coverage and quality.

T6: agent-behaviour claims rest on observable tool calls, not the agent's stated reasoning. The KG-under-use and no-fabrication findings are operationalized by what the agent *did* (its tool calls, queries, and allocations): the trajectory format logs only tool commands, assistant reasoning is not persisted, and the agent's *intent* is therefore not auditable with this harness. *Mitigation:* scope the claims as behavioural ("the agent behaviourally under-uses the offered KG", "no *observable* fabrication"), not introspective. *Residual risk:* behavioural framing can still be over-read as a claim about what the agent understood.

T7: the flat substrate is "flat enough", not a verbatim re-extraction. The Stage-2 rewriter resolves project includes where it can and leaves ambiguous or unresolved directives intact as disclosed residuals (roughly 88 to 97 percent of project includes resolve non-residually per codebase), so some flat-KG dependency edges differ from the canonical extraction. The drift can run upward (bash `INCLUDES` 1,376 flat versus 1,165 canonical) because the rewriter resolves includes that canonical extraction, lacking compiler search paths, leaves unresolved. Every flat build is gated on at least 95 percent edge-count parity with the canonical KG (Section 3.3.4). The direction of the drift favours the flat substrate, so it cannot manufacture the flat-variant collapse of §4.6. *Residual risk:* small edge-set differences between the substrates remain in every flat-versus-canonical comparison.

T8: the C extractor does not capture K&R function definitions. The C tree-sitter query does not match old-style (K&R) function definitions (about 56 in bash) and logs per-entity extraction errors without failing. *Mitigation:* none required at the granularity reported here: the file-level findings draw on substrates containing every source file with full contents, and no reported metric is computed over functions. *Residual risk:* the

omission would bias any function-level metric, so entity-level work on this substrate would need the query extended first.

External Validity

Limited model set. Every executed configuration uses the single primary model. No second model was run. Conclusions are scoped to the tested model, pipeline, and prompts, not to "LLMs in general". Whether the diagnostic findings (KG under-use, run-to-run variance, structure-riding, recitation) are model-class properties is an open question, and re-running the ladder under a contrasting model family is future work (Section 6.3).

File-level granularity. The study operates at file level, which collapses much of the KG's largely entity-level relational power. This is disclosed both as a scope decision and as a partial confound on the KG-under-use reading: entity-level results might differ.

T9: the flatten probe is C/C++-only. The flatten ablation excludes the Java codebases because in Java the directory is the package is the dependency-resolution key (the mechanism is given in Section 3.3.4 and the handling in §4.6.3). The flatten result therefore transports to include-based dependency systems, not to namespace-as-dependency-key systems. The same machinery bounds the recitation ladder to the two C study systems (§4.7.9). *Mitigation:* Java structure-riding is probed indirectly through the directory covariate (T1), which needs no flattening. *Residual risk:* structure-riding in namespace-keyed languages is only indirectly measured.

6. Conclusion

This thesis set out to answer a diagnostic question: *what do LLM, knowledge-graph, and agentic methods for the clustering sub-task of software architecture recovery actually do, and do their outputs carry weight?* It did not ask the competitive question of whether such methods win a benchmark. We answered it internally, by probing a single-variable ladder of our own configurations on four core canonical benchmark systems (bash, hadoop, archstudio4, ITK) plus two supplementary exhibits, libxml2 and JPetStore (six in total). We ran the ladder repeatedly at temperature zero and reported the results as distributions. The picture that emerges is consistent, and it is unflattering in a way that is itself the contribution: on these widely studied systems, what LLM-driven recovery produces is largely readable from the directory layout and named from memory. The recovery method adds little beyond those two priors.

6.1 Summary of Contributions

The study makes three contributions, forming a single diagnostic set: whether the knowledge graph carries weight, how stable the recovery is, and what the method reproduces when it appears to succeed. The third is the load-bearing one and has two layers, structure-riding and recitation.

1. The knowledge graph is under-used, and not load-bearing here. When a KG query tool is offered alongside the shell (A-F vs A-F+KG, the allocator held constant), the agent largely ignores it: about 96% of its Cypher calls are structural enumeration equivalent to `find`, so A-F+KG stays close to A-F. Forcing the KG as the sole lens (A-KG) produces no consistent quality lift across the six-system suite: even the largest positive delta, archstudio4 (+0.157 ARI over A-F+KG), sits inside run-to-run spread and remains below A-F on the same codebase, while ITK *regresses* (ΔARI -0.300). The under-use is a behavioural choice: relational Cypher succeeds when the agent issues it.

2. Run-to-run variance. At temperature zero the recovery varies substantially across identical repeated runs, in coverage (where the agent allocates for itself, its stop-or-continue decision differs on every draw) and in component granularity. A single-run point estimate hides most of the picture, and recovery must be reported as a distribution.

3. Structure-riding and recitation (the headline, in two layers). The third contribution identifies what the method reproduces when it appears to succeed. Its first layer is *structure-riding*. A flatten ablation shows that much of what these methods "recover" is readable from the directory layout: when the directory scaffolding is removed, recovery degrades in proportion to a per-codebase covariate, `ARI(dir@2, gold)`, that can be computed before any run. The example is ITK, where the covariate is 1.000 (the gold *is* the depth-2 module tree), so flattening removes the gold partition itself and the drop is severe. The covariate is best read as an *ordinal* predictor of drop direction, not a linear fit. The behavioural rungs (offering and forcing the KG) modulate quality run-to-run but do not change this dependence on layout (contribution 1).

The second layer is recitation. Where the structure-riding result explains where the file *groupings* come from, the recitation result explains where the component *names* come from. They come from a memorized canonical architecture, reconstructed from residual domain signal (code identifiers, call and dependency structure, file extensions) that survives identity-stripping down to the irreducible *code-identifier floor*. On a

substrate whose project name, file and directory names, comments, copyright headers and residual identity literals have all been removed and verified absent, probe-containment is unchanged on bash and plateaus well above zero on libxml2; a deeper strip lowers neither. A zero-substrate elicitation probe, prompted with only the codebase name and no files or tools, confirms the architectures are already in the model's weights. That the naming is recall rather than coincidence is established directly. Forking each recovery run at its final turn and asking the *same context window* which project it had been exploring, twelve of nineteen runs named the system correctly at high confidence and none named a wrong one. The model that produced those component names knew, from the same observations, which system it was reading.

The two channels are substitutes rather than independent. Removing the directory tree does not suppress recitation but *raises* it, and across the wider suite recitation dominates precisely on the systems whose identifiers carry the domain, while structure-riding dominates where the gold is the module tree. We could not construct a substrate on which the canonical naming stops, short of removing the identifiers that make recovery possible at all.

What this means for obfuscation as a control. Two families of transformation have now been applied to this problem. Prior work rewrites code identifiers and program structure and measures whether the model can still complete a task [51, 52, 53]; the transform used here removes the project name, file and directory names, comments and copyright headers, and measures whether the model can still name the system and reproduce its architecture. Neither family removes both sets of channels, and neither has been shown to prevent recognition: prior work because it never measured recognition, and this study because its transform demonstrably failed to. Our own attempt was also incomplete, leaving identity in file contents that the reported runs did not read. We therefore cannot say whether a sufficiently thorough obfuscation would work. What we can say is that **no obfuscation demonstrated so far is dependable as a decontamination control for architecture-recovery evaluation**, and that the reliable alternative is to evaluate on a system the model has not seen.

What it means for LLM-SAR. The two findings say that on these famous, documentation-rich benchmark systems, an apparently successful LLM recovery is substantially the sum of two priors: *directory-readable-for-free* and *training-data-recallable*. A small to negligible residual is attributable to analysis of the artifact itself. The pattern reaches beyond any one pipeline: it is a property of evaluating memorizing methods on contaminated, directory-aligned benchmarks. A high MoJoFM or ARI on this suite cannot, on its own, distinguish semantic recovery from structure-mirroring plus recitation.

The benchmark-contamination obligation. The practical consequence is a reporting obligation the subfield does not currently meet. Any LLM-SAR result on canonical benchmarks should be disclosed alongside (a) the per-codebase `ARI(dir@K, gold)` covariate, which bounds how much of a score is directory-readable for free, and (b) a contamination measurement. A zero-substrate elicitation probe is a cheap, reproducible instrument for this. This obligation lands at least as hard on prior LLM-driven systems evaluated on the same suite without such apparatus as it does on the configurations studied here.

6.2 Limitations

The findings are bounded, and several scope decisions and threats constrain them. The Threats chapter develops each. The most serious are summarized here.

- **Clustering sub-task only.** Which files enter the partition (filtering build scripts, vendored code, tests, generated artifacts) is out of scope. The study fixes an authoritative file universe and evaluates only how it is partitioned (Section 3.8).
- **File-level granularity.** The entity level was not run end-to-end, which collapses some of the KG's relational power and is disclosed as a confound on the KG under-use reading (Chapter 5, External Validity).
- **Gold-truth construct validity.** The expert-authored RSF golds largely coincide with the ˜depth-2 directory grouping, so gold-based metrics on this suite substantially grade directory reconstruction, compounding with the contamination threat (T1, T2).
- **Metric caveats.** MoJoFM is granularity-insensitive and compressed on this suite, so ARI is the discriminating metric, and jpetstore is structurally pinned by a layer-vs-domain gold mismatch (§4.3.4).
- **Limited model set and codebase scope.** All results are on a single primary model family at temperature zero, on two C systems (bash, libxml2) for the recitation ladder and three C and C++ systems (bash, libxml2, ITK) for the flatten ablation, with the directory-lens covariate covering all six, so generalization to other models and to non-memorized systems is not demonstrated and is the chief external-validity threat (Chapter 5, External Validity).

6.3 Future Work

Several directions follow, kept modest and out of scope for the present study.

- **A non-memorized testbed.** The decisive test of beyond-directory, beyond-recitation recovery is a codebase whose architecture the model has *not* seen and whose gold diverges from its directory tree. Constructing or sourcing such a system, and re-running the ladder on it, is the natural next experiment. We did not construct such a testbed in this study. Such a control is feasible and has precedent. An LLM's near-perfect scores on an in-training system have been shown to collapse on a proprietary, guaranteed-unseen counterpart [58]. Parametric recall falls off sharply for less-popular, long-tail entities [46], the regime a non-memorized subject would occupy. The reason a fresh testbed is necessary, rather than more scrubbing of a known one, is that the in-code identifiers a method reads to analyse a file are the same signal that cues recitation. The effect survives to the identifier floor (§4.7), and only genuine unfamiliarity can separate analysis from recall. The same testbed would settle the recitation question for our own configurations and for the external comparators (SARIF, SemArc).
- **Head-to-head comparator rescoring.** A numeric comparison against external LLM-SAR systems (e.g. SARIF, SemArc) on shared golds would require rescoring those tools under the same apparatus. Rescoring them, and measuring their contamination exposure with the probe we provide, is **future, not present, work**.

- **Reference-free evaluation.** Because the gold truths on this suite are directory-aligned, reference-free intrinsic measures (e.g. Newman modularity, coupling, cohesion) are a more honest primary lens for "is the decomposition any good?" and should be developed as such.
- **Entity-level granularity and broader models.** Running the pipeline end-to-end at entity granularity would let the KG's relational edges carry their weight. Broader model coverage would test whether the recitation and under-use patterns are model-class properties.
- A counter-recitation prompt and the dead pre-pivot directions (a relocation/shift phase, transliteration, code translation/migration) are **not** pursued and are not revived here.

6.4 Concluding Remarks

The contribution of this thesis is a characterization. By instrumenting a ladder of LLM-driven recovery configurations and reporting their results as distributions, we show that on the canonical SAR benchmarks the apparent success of these methods is substantially explained by two things they get for free: the directory layout they ride and the famous architectures they recite from memory. The recovery machinery adds little beyond them on this suite. The practical takeaway is a measurement discipline the subfield should adopt: report run-to-run distributions, report `ARI(dir@K, gold)` so structure-readable scores are visible, and measure benchmark contamination directly. Stated honestly and bounded to the systems studied, that is a more useful result for the field than another headline number.

Bibliography

[1] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (Silicon Valley, CA, USA), pp. 486–496, IEEE, 2013.

[2] T. Lutellier *et al.*, "Comparing software architecture recovery techniques using accurate dependencies," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, (Florence, Italy), pp. 69–78, IEEE, 2015.

[3] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, "Measuring the impact of code dependencies on software architecture recovery techniques," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 159–181, 2018.

[4] W. Zhao, W. Jin, Y. Zhang, M. Fan, H. Wang, L. Li, Y. Liu, and T. Liu, "Software architecture recovery augmented with semantics," *IEEE Transactions on Software Engineering*, vol. 52, no. 1, pp. 338–359, 2025.

[5] S. A. Rukmono, L. Ochoa, and M. Chaudron, "Deductive software architecture recovery via chain-of-thought prompting," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER'24)*, pp. 92–96, ACM, 2024.

[6] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An empirical study of the non-determinism of ChatGPT in code generation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1–28, 2024. arXiv:2308.02828.

[7] B. Atıl, S. Aykent, A. Chittams, L. Fu, R. J. Passonneau, E. Radcliffe, G. R. Rajagopal, A. Sloan, T. Tudrej, F. Ture, Z. Wu, L. Xu, and B. Baldwin, "Non-determinism of "deterministic" LLM system settings in hosted environments," in *Proceedings of the 5th Workshop on Evaluation and Comparison of NLP Systems (Eval4NLP)*, (Mumbai, India), pp. 135–148, Association for Computational Linguistics, Dec. 2025. arXiv:2408.04667.

[8] Y. Song, G. Wang, S. Li, and B. Y. Lin, "The good, the bad, and the greedy: Evaluation of LLMs should not ignore non-determinism," in *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, Association for Computational Linguistics, 2025. arXiv:2407.10457.

[9] O. Sainz, J. A. Campos, I. García-Ferrero, J. Etxaniz, O. Lopez de Lacalle, and E. Agirre, "NLP evaluation in trouble: On the need to measure LLM data contamination for each benchmark," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 10776–10787, Association for Computational Linguistics, 2023.

[10] M. Riddell, A. Ni, and A. Cohan, "Quantifying contamination in evaluating code generation capabilities of language models," in *Proceedings of the 62nd Annual Meeting of the Association for Computational*

Linguistics (Volume 1: Long Papers), pp. 14116–14137, Association for Computational Linguistics, 2024.

[11] Y. Zhang, Z. Xu, C. Liu, H. Chen, J. Sun, D. Qiu, and Y. Liu, "Software architecture recovery with information fusion," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*, pp. 1535–1547, Association for Computing Machinery, 2023.

[12] P. Zaragoza, A.-D. Seriai, A. Seriai, A. Shatnawi, and M. Derras, "Leveraging the layered architecture for microservice recovery," in *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, pp. 135–145, IEEE, 2022.

[13] Y. Wang, S. Bornais, and J. Rubin, "Microservice decomposition techniques: An independent tool comparison," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Association for Computing Machinery, 2024. Artifact: Zenodo 10.5281/zenodo.13855559.

[14] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner, "Bunch: a clustering tool for the recovery and maintenance of software system structures," in *IEEE International Conference on Software Maintenance*, (Oxford, UK), pp. 50–59, IEEE, 1999.

[15] D. Doval, S. Mancoridis, and B. Mitchell, "Automatic clustering of software systems using a genetic algorithm," in *STEP '99: Proceedings Ninth International Workshop Software Technology and Engineering Practice*, (Pittsburgh, PA, USA), pp. 73–81, IEEE, 1999.

[16] V. Tzerpos and R. Holt, "Acdc: An algorithm for comprehension-driven clustering," in *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, p. 258, IEEE, 2000.

[17] M. Shtern and V. Tzerpos, "Clustering methodologies for software engineering," *Advances in Software Engineering*, May 2012.

[18] I. Inayat *et al.*, "Evolution in software architecture recovery techniques - a survey," in *International Conference on Emerging Technologies (ICET)*, 2017.

[19] I. Stavropoulou, M. Grigoriou, and K. Kontogiannis, "Case study on which relations to use for clustering-based software architecture recovery," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1717–1762, 2017.

[20] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: Bridging the gap between design and implementation," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 364–380, 2001.

[21] K. Sartipi and K. Kontogiannis, "On modeling software architecture recovery as graph matching," in *International Conference on Software Maintenance (ICSM)*, (Amsterdam, Netherlands), pp. 224–233, IEEE, 2003.

[22] K. Sartipi and K. Kontogiannis, "A user-assisted approach to component clustering," *Journal of Software Maintenance*, vol. 15, no. 4, pp. 265–295, 2003.

[23] M. Soliman and J. Keim, "Do large language models contain software architectural knowledge? : An exploratory case study with gpt," in *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*, (Odense, Denmark), pp. 13–24, IEEE, 2025.

[24] R. Dhar, K. Vaidhyanathan, and V. Varma, "Can llms generate architectural design decisions? - an exploratory empirical study," in *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, (Hyderabad, India), pp. 79–89, IEEE, 2024.

[25] J. Díaz-Pace, A. Tommasel, and R. Capilla, "Helping novice architects to make quality design decisions using an llm-based assistant," in *Software Architecture. ECSA 2024* (M. Galster, P. Scandurra, T. Mikkonen, P. Oliveira Antonino, E. Nakagawa, and E. Navarro, eds.), vol. 14889 of *Lecture Notes in Computer Science*, Cham: Springer, 2024.

[26] N. Johansson, M. Caporuscio, and T. Olsson, "Mapping source code to software architecture by leveraging large language models," in *Software Architecture. ECSA 2024 Tracks and Workshops* (A. Ampatzoglou *et al.*, eds.), vol. 14937 of *Lecture Notes in Computer Science*, Cham: Springer, 2024.

[27] M. Esposito *et al.*, "Generative ai for software architecture: Applications, trends, challenges, and future directions," *arXiv preprint arXiv:2503.13310*, 2025.

[28] L. Schmid *et al.*, "Software architecture meets llms: A systematic literature review," *arXiv preprint arXiv:2505.16697*, 2025.

[29] A. Boronat and J. Mustafa, "Mdre-llm: A tool for analyzing and applying llms in software reverse engineering," in *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, (Montreal, QC, Canada), pp. 850–854, IEEE, 2025.

[30] Z. Chen, X. Tang, G. Deng, F. Wu, J. Wu, Z. Jiang, V. K. Prasanna, A. Cohan, and X. Wang, "Locagent: Graph-guided llm agents for code localization," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL), Volume 1: Long Papers*, pp. 8717–8740, Association for Computational Linguistics, 2025.

[31] X. Liu, B. Lan, Z. Hu, Y. Liu, Z. Zhang, F. Wang, M. Q. Shieh, and W. Zhou, "CodexGraph: Bridging large language models and code repositories via code graph databases," in *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), Volume 1: Long Papers*, 2025. arXiv:2408.03910.

[32] S. Ouyang, W. Yu, K. Ma, Z. Xiao, Z. Zhang, M. Jia, J. Han, H. Zhang, and D. Yu, "Repograph: Enhancing ai software engineering with repository-level code graph," in *International Conference on Learning Representations (ICLR)*, 2025.

[33] D. Edge, H. Trinh, N. Cheng, J. Bradley, A. Chao, A. Mody, S. Truitt, D. Metropolitansky, R. O. Ness, and J. Larson, "From local to global: A graph rag approach to query-focused summarization," *arXiv preprint arXiv:2404.16130*, 2024.

[34] J. Sun, C. Xu, L. Tang, S. Wang, C. Lin, Y. Gong, L. M. Ni, H.-Y. Shum, and J. Guo, "Think-on-graph: Deep and responsible reasoning of large language model on knowledge graph," in *International Conference on Learning Representations (ICLR)*, 2024.

[35] L. Luo, Y.-F. Li, G. Haffari, and S. Pan, "Reasoning on graphs: Faithful and interpretable large language model reasoning," in *International Conference on Learning Representations (ICLR)*, 2024.

[36] L. Hubert and P. Arabie, "Comparing partitions," *Journal of Classification*, vol. 2, no. 1, pp. 193–218, 1985.

[37] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC)*, pp. 194–203, IEEE, 2004.

[38] N. Carlini, F. Tramèr, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, Ú. Erlingsson, A. Oprea, and C. Raffel, "Extracting training data from large language models," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 2633–2650, USENIX Association, 2021.

[39] N. Carlini, D. Ippolito, M. Jagielski, K. Lee, F. Tramèr, and C. Zhang, "Quantifying memorization across neural language models," in *International Conference on Learning Representations (ICLR)*, 2023.

[40] C. Zhang, D. Ippolito, K. Lee, M. Jagielski, F. Tramèr, and N. Carlini, "Counterfactual memorization in neural language models," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[41] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo, "Unveiling memorization in code models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024.

[42] A. Al-Kaswan, M. Izadi, and A. van Deursen, "Traces of memorisation in large language models for code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 1–12, 2024.

[43] U. S. Prashanth, A. Deng, K. O'Brien, J. S. V, M. A. Khan, J. Borkar, C. A. Choquette-Choo, J. R. Fuehne, S. Biderman, T. Ke, K. Lee, and N. Saphra, "Recite, reconstruct, recollect: Memorization in LMs as a multifaceted phenomenon," in *International Conference on Learning Representations (ICLR)*, 2025.

[44] M. Meeus, A. Ramakrishna, M. Grange, Z. Xu, and L. Melis, "Detecting functional memorization in code language models." arXiv preprint arXiv:2606.12764, 2026.

[45] X. Zhou, M. Weyssow, R. Widyasari, T. Zhang, J. He, Y. Lyu, J. Chang, B. Zhang, D. Huang, and D. Lo, "Lessleak-bench: A first investigation of data leakage in llms across 83 software engineering benchmarks," *arXiv preprint arXiv:2502.06215*, 2025.

[46] A. Mallen, A. Asai, V. Zhong, R. Das, D. Khashabi, and H. Hajishirzi, "When not to trust language models: Investigating effectiveness of parametric and non-parametric memories," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Toronto, Canada), pp. 9802–9822, Association for Computational Linguistics, July 2023.

[47] S. Longpre, K. Perisetla, A. Chen, N. Ramesh, C. DuBois, and S. Singh, "Entity-based knowledge conflicts in question answering," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, (Online and Punta Cana, Dominican Republic), pp. 7052–7063, Association for Computational Linguistics, Nov. 2021.

[48] R. Jabeen, M. Ericsson, J. Nordqvist, and A. Wingkvist, "From dependency graphs to modules: Graph representation learning for architecture recovery," *Nordic Machine Intelligence*, 2026.

[49] Y. He, Y. Huai, X. Chen, and J. Garcia, "Embedding software intent: Lightweight java module recovery," *arXiv preprint arXiv:2512.15980*, 2025.

[50] S. Bernstein, D. Beste, D. Ayzenshteyn, L. Schönherr, and Y. Mirsky, "Trust me, i know this function: Hijacking llm static analysis using bias," in *Network and Distributed System Security Symposium (NDSS)*, 2026.

[51] Y. Zhang, Y. Xie, S. Li, K. Liu, C. Wang, Z. Jia, X. Huang, J. Song, C. Luo, Z. Zheng, R. Xu, Y. Liu, S. Zheng, and X. Liao, "Unseen horizons: Unveiling the real capability of LLM code generation beyond the familiar," in *47th IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 604–615, IEEE, 2025.

[52] H. Rao, Y. Zhao, W. Zhu, L. Xiao, M. Wang, and H. Wang, "CODEMORPH: Mitigating data leakage in large language model assessment," in *47th IEEE/ACM International Conference on Software Engineering (ICSE) – Companion Proceedings*, pp. 267–278, IEEE, 2025.

[53] C. C. Le, M. V. T. Pham, C. D. Van, H. N. Phan, H. N. Phan, and T. N. Nguyen, "When names disappear: Revealing what LLMs actually understand about code." arXiv preprint arXiv:2510.03178, 2025.

[54] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *24th USENIX Security Symposium (USENIX Security 15)*, pp. 255–270, USENIX Association, 2015.

[55] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in Android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 356–367, ACM, 2016.

[56] C. Fang, N. Miao, S. Srivastav, J. Liu, R. Zhang, R. Fang, Asmita, R. Tsang, N. Nazari, H. Wang, and H. Homayoun, "Large language models for code analysis: Do LLMs really do their job?," in *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 829–846, USENIX Association, 2024.

[57] Tree-sitter, "Tree-sitter: An incremental parsing system for programming tools." GitHub repository, 2013.

[58] V. Bekmyradov, N. C. Pütz, and T. Bartz-Beielstein, "Llms taking shortcuts in test generation: A study with sap hana and leveldb," *arXiv preprint arXiv:2604.14437*, 2026.

Appendix A. Codebase and Gold-Truth Provenance and Change Log

This appendix is the auditable record of how each of the six subject codebases and their human-authored ground-truth (gold) decompositions were obtained and modified to reach a workable, defensible condition for the study. The four core golds are developer-authored; the two supplementary golds are not, and their authorship is stated in A.3.5 and A.3.6. Every fix is recorded here with its rationale. The Methodology chapter (Sections 3.7 and 3.8) summarizes the governing principle and the headline counts. This appendix is the line-by-line audit trail that those sections refer to. The cleaned golds are written to a cleaned-gold directory, separate from the originals. The original RSF files are left untouched, so every change below is reproducible by re-running the cleaning script against the originals. A general on-disk filter (drop any gold entry that has no source file on disk) is applied across all codebases and is the rule that removes the generated and phantom entries discussed below.

A.1 The Governing Principle: Select the File Universe by File Structure, Not by the Gold

The file universe shown to the recovery pipeline is constructed by a rule about *where files live* (which project or subsystem folders), applied without consulting the ground-truth labels. The non-architectural "noise" (distractor files) is retained in that universe.

Selecting the input files by gold membership (handing the pipeline exactly the files the answer key labels) would be information leakage. The input becomes a function of the answer, the pipeline is spared the real sub-task of distinguishing architectural code from noise, and recovery scores are inflated by a step that was pre-solved with the answer key. The test applied to every codebase is therefore: *could someone who has not seen the gold reconstruct this exact file universe?* If yes, the universe is gold-independent (unbiased). If it required reading the gold's file list, it is biased.

TABLE 27

Universe-selection rules considered, and why gold-membership selection is rejected in favour of a gold-independent path-scoped rule.

Selection rule	Example	Gold needed to build input?	Verdict
By gold membership	"include the 561 files the RSF lists"	Yes	Biased (rejected)
By file structure (ours)	"include every `.java` under `src/core`, `hdfs`, `mapred`"	No	Gold-independent (used)

Scoring is a separate step and introduces no bias. Metrics (the canonical `mojo.jar` MoJoFM and ARI) are computed over the *intersection* of the recovered partition with the gold: the gradebook is restricted to the files for which the gold defines an answer. Distractor files have no "correct component," so they cannot be graded, but the pipeline still saw them and could still mis-group them. Restricting the gradebook to

answerable questions is standard practice and does not simplify the task. Consequently each universe retains real non-architectural files (tests, build helpers, utilities, vendored code) that the pipeline must avoid misassigning. The per-codebase distractor-tail sizes are recorded below.

A.2 Per-Codebase Provenance and Universe-Selection Rule

TABLE 28
Per-codebase file universe (folder rule) and gold-independence verdict.

Codebase	Language	Version	Universe (folder rule)	Gold-independent?
bash	C	4.2	the full bash-4.2 release (all `.c`/`.h`/`.y`)	Yes
hadoop	Java	0.19.0	all `.java` under `src/core` + `src/hdfs` + `src/mapred` (sparse checkout by path)	Yes
archstudio4	Java	ArchStudio4	all `.java` in the `edu.uci.isr.archstudio4.*` source bundles	Yes
ITK	C++	4.5.2	all source under `Modules/` (path-scoped, 8,547 src files, rebuilt from a full re-clone)	Yes
libxml2	C	2.4.22	the full libxml2-2.4.22 release (flat library)	Yes
jpetstore	Java	JPetStore-6	the full Maven project (`src/`)	Yes

Each folder rule corresponds to the subsystem or subsystems that the original ground-truth study covers. One can only score where ground truth exists. Choosing whole folders is coarse and does not pre-select which individual files are architectural. The pipeline still sorts that out.

A.3 Gold-Truth Cleaning: Every Change, with Rationale

Golds are cleaned only to make them valid single-membership partitions of real source files (a requirement for ARI and `mojo.jar`). Every count was verified against the source RSF files and an authoritative file-to-component mapping before being applied.

A.3.1 bash: 356 files, 13 components

- **Dropped seven multi-membership files** (all in `lib/readline/`), each claimed by both `libhistory` and `libreadline`: `histexpand.c`, `histfile.c`, `histlib.h`, `history.c`, `histsearch.c`, `shell.c`, and `history.h`, the last of these also claimed by `general`. Rationale: a partition requires one component per file, and these files are ambiguous.
- **Remapped and deduped** the phantom top-level entry `bash/builtins.c` to its real on-disk path `bash/builtins/builtins.c`, already a gold entry under `libbuiltins`. The remap **collapses into the existing entry**, with no duplicate RSF line. The on-disk filter only drops entries with no source file, so no broad generated-file exclusion is applied: `y.tab.c/.h` and `builtins/builtins.c` are build-generated but present as source and architecturally labelled, so they are kept.

The cleaning is set-based, so the gold contains no duplicate RSF line.

A.3.2 hadoop: 560 files, 62 components

- The inherited (Lutellier-derived) gold is a **cover, not a partition**: 67 groups overlay 53 numbered architectural components (for example `16_Namenode`) and 14 package components (for example `org.apache.hadoop.*`), with 126 files dual-labelled. Resolution (option B): **prefer each file's numbered/architectural label** over its package label. **Keep package-only files as package components**. **Exclude the 31 files that belong to two or more numbered components** (true architectural overlap). This produces a partition while preserving the architectural decomposition and the original gold's package buckets.
- **On-disk filter (general rule).** Dropped all **24** gold entries with no source file on disk: the **23** generated `*_jsp.java` servlet files plus the phantom `mapred/MRSortResultIterator.java` (a ground-truth label for a non-existent file is meaningless). $\rightarrow$ **560 files**.

Recorded characteristics (not defects, disclosed). Nine singleton components. Mixed naming schemes (`NN_`, `NNx_`, `org.apache.*`). A 25-file catch-all `1_Other_utilities`. A 45-file first-party distractor tail.

A.3.3 ITK: 5,228 files, 11 components

- **Merged the typo component** `group_numberics` (518 files) into `group_numerics` (127 files) to form one `group_numerics` (645 files). Rationale: this is the same component, misspelled.
- **Excluded `group_thirdParty`** (2,428 vendored files: GDCM, HDF5, VNL, zlib, and others). Rationale: vendored third-party code is not part of ITK's architecture.

- **On-disk filter.** Dropped one non-source gold entry, `Modules/Bridge/VtkGlue/wrapping/VtkGlue.i`, a SWIG `.i` file not present as a source-extension file in the path-scoped tree. → **5,228 files**.
- **Path-scoped universe.** The universe is a full clone of ITK 4.5.2 plus the four remote modules, keeping **all** source under `Modules/` (path-scoped, gold-independent): **8,547** source files with a **3,319-file distractor tail** (`Modules/ThirdParty/`, tests, and non-gold module source the pipeline must handle).

Recorded characteristics (disclosed). ITK's own `test/` files (1,733) are intentionally in-scope. The distractor tail is large but relatively homogeneous (third-party, tests, non-gold module source). The main grid (§§4.2 to 4.5) and the flatten ablation (§4.6) were run over the cleaned golds plus the path-scoped full universes (ITK included).

A.3.4 archstudio4: 583 files, 57 components

- **No changes.** The gold was already a clean single-membership partition.

Recorded characteristics (disclosed). Five singleton components. A 26-file distractor tail.

A.3.5 libxml2 (supplementary): basename-keyed JSON gold (82 items)

- **Source.** The gold is the labelled libxml2 decomposition released in the SARIF artifact repository [11] (`libxml2-2.4.22-GT.json`); our copy is byte-identical to the published file. SARIF's authors report that this decomposition was labelled by their industrial collaborator, not by libxml2's developers.
- **No cleaning needed.** The gold is a clean group/item hierarchy. The universe is the full flat release (approximately 100 `.c`/`.h` files) with an approximately 18-file distractor tail. libxml2 is the non-directory-aligned exhibit.

A.3.6 jpetstore (supplementary): class-level CSV gold (23 classes to 3 services, from 26 CSV rows)

- **Source.** The decomposition originates with Zaragoza et al. [12], a manual expert decomposition of MyBatis JPetStore-6 into account, catalog, and order services. It was digitized as a class-to-service CSV in the tool-comparison artifact of Wang et al. [13]; our copy is byte-identical to that published file. The gold is therefore researcher-authored rather than developer-authored.
- **No cleaning needed** beyond the already-disclosed exclusion of the duplicate base class `AbstractActionBean` (handled by the scoring script). The universe is the full Maven project (41 `.java` files). Eighteen distractor classes, on the universe-minus-gold definition of A.5. The gold cuts across the layer directories.

A.4 Independent Verification (Three-Way Audit)

Three independent runs of an LLM auditing agent, each given the same computation-driven task against the cleaned golds and the staged universes, returned byte-for-byte identical quantitative summaries. The agreement is evidence that the counts are reproducible from the artifacts, not that three people checked them. The only true data defects all three surfaced were the two flagged above (the bash duplicate line and

the hadoop missing file), both **now fixed** as part of the cleaning (A.3.1, A.3.2). Everything else (the ITK third-party tail, the hadoop mixed naming, the duplicate basenames) is intentional or benign.

A.5 Locked Directory Baseline (Cleaned Golds × Path-Scoped Universes)

The locked directory-baseline characterization table (per-codebase MoJoFM and ARI over the cleaned golds and the path-scoped universes) is given once, in Section 3.8 of the method chapter. For all four core systems the baseline is unchanged to four decimal places between the gold-only and path-scoped constructions, ITK included despite its far larger universe, because distractors do not alter how the gold's own files group by parent directory. The pipeline nonetheless faces those distractors: 3,319 of them on ITK. Per-codebase distractor tails (source files in the universe not in the cleaned gold): bash 60, hadoop 45, archstudio4 26, ITK 3,319.

A.6 Cross-Codebase Consistency and the Two-Tier Design

The input-construction principle (folder-based, gold-independent, distractors retained) and the scoring procedure (intersection with the gold via `mojo.jar` and ARI) are consistent across all six systems. The golds themselves differ in *kind*: file-level multi-component (the four core systems), basename-keyed (libxml2), and class-level service (jpetstore). The four core systems anchor the conclusions. libxml2 and jpetstore run through the same quality grid at n = 10 per configuration cell, but they are read as supplementary structure-riding exhibits of different gold provenance and are not pooled with the core systems. A single metric is not pooled across all six systems.

A.7 Cypher Query Corpus: Worked Examples

The Cypher-query classifier of Section 3.5.1 is illustrated in the method chapter with two poles (the bootstrap STRUCTURAL enumeration and the canonical HYBRID co-traversal). Two further verbatim examples are collected here. The full query log is preserved with the per-run artifacts: bracketed identifiers give the codebase and run short-id, and each trajectory file is named by the pattern `<codebase>_<config>_<short>.json` within a per-configuration trajectory folder.

STRUCTURAL: function-count per file (uses :CONTAINS, a structural containment edge):

```
[bash/b1e460c7]    MATCH (f:File)-[:CONTAINS]->(fn:Function) WITH f, count(fn) AS n
                   RETURN f.path, n ORDER BY n DESC LIMIT 30
```

RELATIONAL: dependency-outdegree probe, restricted to a subtree by WHERE filter:

```
[ITK/8964618a]     MATCH (f1:File)-[:DEPENDS_ON_FILE]->(f2:File)
                   WHERE f1.path CONTAINS '/Modules/IO/'
                   RETURN split(f1.path,'/')[2] AS top,
                          split(f2.path,'/')[2] AS dep_top,
                          count(*) AS edges
                   ORDER BY edges DESC LIMIT 20
```

(Despite the path-string `CONTAINS` filter in WHERE, this is RELATIONAL: the main operation is the dependency traversal, computing cross-module edge density.)
The fourth category, SCHEMA, is defined by the classifier for completeness but has no example: no schema-introspection call (`db.labels`, `db.relationshipTypes`, `db.schema`) occurs in any trajectory in the corpus, and the SCHEMA column is zero in every behavioural table. The agents queried the graph's contents throughout and never asked what shape it was.

A.7.1 Matching trajectories to scored runs

The agent writes each trajectory to a UUID-suffixed filename that does not itself carry the canonical `run_id` recorded in the run ledgers, so trajectories are joined to runs on content.
Timestamps are not a usable key for this corpus. Grid runs launch in parallel (four bash A-F runs share the start second `03:53:08`), and a trajectory is written roughly forty seconds *before* its run ends, because allocation continues after exploration finishes. Every one of those runs' time windows therefore contains every one of those trajectories, so no time-based rule can separate them.
The join is the **set of component names the explorer proposed**, which the trajectory transcript and the run's allocation log record independently of one another. Matching is exclusive, one trajectory per run, greedy by descending Jaccard overlap. On this key every scored run in the A-F, A-F+KG and A-KG corpora matches a distinct trajectory, so the behavioural corpus and the quality corpus are the same 60 runs per configuration and no behavioural table needs an exclusion count. Where two runs of the same configuration proposed identical component sets the pairing is ambiguous by construction; such ties are broken by file modification time and recorded as ties. Aggregate per-codebase counts are invariant to how a tie is broken, since both trajectories belong to the cohort either way. Trajectory files carry their `run_id` in the filename, and the mapping is produced by `scripts/map_trajectories_to_runs.py`.

Appendix B. Algorithm Listings

This appendix gives language-neutral pseudocode for the five algorithmic components of the pipeline that the body of the thesis describes in prose. Each listing is drawn from the implementation and notes any simplification it makes.

B.1 The Fixed Batch Allocator

One pass allocates the files of a parent component to its proposed children. No relational edge (`CALLS`, `DEPENDS_ON`, `INCLUDES`) is queried. Coverage is enforced by bounded retries, and every reported run reached 100 percent coverage.

```
INPUT  parent component P, knowledge graph KG (used as storage only)
C <- child components of P, each as (name, purpose)
IF C is empty THEN RETURN
F <- files allocated to P that are not yet allocated to any child of P
SHUFFLE F                                  (batches must not mirror directory order)
SPLIT F into batches of size 30
FOR EACH batch B:
    FOR EACH file f in B:
        fetch (path, name, contents) of f, TRUNCATE contents to 1000 characters
    prompt <- the component tree, the per-file (path, name, truncated contents),
              and the per-component (name, purpose) list
    instruct the LLM to map EACH file to EXACTLY ONE listed component
    A <- parse the file-to-component lines of the reply
    DROP any component not in C and any file not in B      (hallucination filter)
    WRITE each surviving (file, component) edge immediately (idempotent, crash-safe)
FOR retry round r = 1 TO 3:
    R <- files of P still unallocated to any child
    IF R is empty THEN BREAK
    SHUFFLE R and submit the whole set as one batch, as above
IF files remain unallocated THEN log a warning and leave them unallocated
```

The retry bound means full coverage is enforced rather than guaranteed. In the reported corpus the warning branch did not fire.

B.2 Flat-Substrate Construction

The four gated stages of Section 3.3.4.

```
STAGE 1  (rename map)
    FOR EACH file under the source tree, in deterministic order:
```

```
        flat <- basename of the file, add "__N" before the extension on collision
        RECORD map entry flat -> canonical path
    VERIFY the map is bijective, complete, count-matched, and slash-free
STAGE 2  (include rewriting, C and C++ only)
    FOR EACH #include directive (quoted or angle form) in each flat file:
        resolve the target IN ORDER: (a) relative to the including file's canonical
            directory, (b) relative to the source root, (c) unique-basename fallback
        IF resolved THEN rewrite the include path to the flat name
        ELSE leave the directive untouched and record a residual
            (reason: system header, unresolved, or ambiguous basename)
STAGE 3  (re-extraction)
    run the canonical extractor of Section 3.2 on the flat tree, snapshot the KG
STAGE 4  (integrity gate, flat versus canonical snapshots)
    FOR EACH tracked dependency-edge type present in the canonical KG
            (INCLUDES, DEPENDS_ON_FILE, DEPENDS_ON, CALLS):
        REQUIRE flat count / canonical count >= 0.95 (warn below 0.98)
    REQUIRE no File path carries a directory segment beyond its constant prefix
    ACCEPT the flat snapshot only if no requirement failed
```

B.3 Consolidated Scoring

Every recovered partition, flat or canonical, is scored by the same procedure.

```
INPUT  run snapshot S, gold RSF G, optional flat rename map
GOLD:
    FOR EACH "contain component path" line in G:
        normalize the path, DROP its leading root segment, RECORD gold[path] <- component
PREDICTED:
    leaves <- Component nodes in S with no incoming SUBCOMPONENT_OF edge
    FOR EACH File f with an ALLOCATED_TO edge:
        pred[path of f] <- the leaf component among f's allocations
IF the run used the flat substrate:
    replace every flat key in pred by its canonical path via the rename map
ALIGN:
    build three key variants of pred: as-is, drop-first-segment, basename
    keep the variant whose key set overlaps the gold's key set the most
SCORE on common <- keys(pred) INTERSECT keys(gold):
    IF common is empty THEN RETURN (n_components, 0, none, none)
    MoJoFM <- canonical mojo.jar on pred versus gold restricted to common
    ARI    <- adjusted Rand index over the label pairs on common
RETURN (number of predicted components, size of common, MoJoFM, ARI)
```

The alignment step chooses among three fixed key-namespace transforms by key-set overlap with the gold's file keys. It aligns namespaces and does not see the scores.

B.4 The Canonical-Overlap (Recitation) Scorer

```
INPUT  proposed component names M_names of one run,
       probe reference names R_names (union over the elicitation trials),
       directory vocabulary of the real source tree
TOKENIZE(name): lowercase, split on non-alphanumerics, drop tokens shorter than 3,
    drop a short generic stopword list (system, core, util, manager, and the like),
    fold one trailing plural s when safe, use no synonym or alias table
M <- union of TOKENIZE(n) over M_names
R <- union of TOKENIZE(n) over R_names
containment <- size(M INTERSECT R) / size(M)     (the recitation signal)
coverage    <- size(M INTERSECT R) / size(R)     (canonical vocabulary reproduced)
DIRECTORY-RIDING: fraction of M matching a directory token of the real tree
CUE PRESENCE (trajectory runs only):
    seen <- lowercased concatenation of all tool results the agent received
    FOR EACH proposed name: cue-present IF any of its tokens occurs in seen
    canonical names split into rode-a-cue versus cold (recited with no supporting token)
RE-RUN everything with the stopword list disabled as a sensitivity check
```

B.5 The Louvain Reference Partitions

```
INPUT  codebase KG snapshot, gold RSF, universe U <- all File paths
FOR variant IN (dep, enriched):
    IF variant = dep:
        edges <- file pairs joined by DEPENDS_ON_FILE, weights summed
    ELSE:
        edges <- file pairs joined by DEPENDS_ON_FILE, INCLUDES, or IMPORTS
        PLUS entity edges (CALLS, USES, DEPENDS_ON, EXTENDS, IMPLEMENTS, INHERITS)
            lifted to their containing files via CONTAINS, weights summed per pair
    build an undirected graph: nodes <- sorted U, edges inserted in sorted order
    isolated <- degree-zero nodes, placed in one ORPHAN component
    relabel the remaining nodes to integers in sorted order (defeats hash randomization)
    communities <- Louvain(integer graph, edge weights, seed = 42)
    K <- 5 percent of size(U)
    FOR EACH community c, in sorted order:
        IF size(c) >= K: keep c as its own component
        ELSE IF c has no external edge weight: keep c as its own small component
        ELSE: merge c into the kept community receiving its largest external weight,
```

```
            promoting c to its own component if that neighbour was itself unkept
    score the resulting file-to-component map with the procedure of Listing B.3
```

K is computed over the full file universe, including the isolated files.

B.6 Per-Run Evidence on the Identity-Stripped Substrate

§4.7.5 reports how many identity-stripped runs still name the study system's documented subsystems. The count uses the canonical-overlap scorer of B.4, with no additional instrument, and the per-run evidence is printed here so that every verdict can be checked.

Which of the scorer's two outputs, and why. B.4 reports a companion pair over the same intersection: *containment* $|M \cap R| \,/\, |M|$ is the fraction of a run's own name tokens that appear in the probe, and *coverage* $|M \cap R| \,/\, |R|$ is the fraction of the probe's vocabulary the run reproduced. The question here is how much of the documented architecture a run recovered, which is coverage. Containment answers a different question and is the wrong instrument for it: a run that names six genuine bash subsystems verbosely ("History and Search Subsystem", "Terminal and TTY Support", "Platform and Compatibility Utilities") is penalised, because containment is a fraction and the extra vocabulary enlarges its denominator, while a wholly generic run picks up containment for free because words like "parsing", "execution" and "core" appear in the probe's own names too. Run 32 and run 4 of bash below are that inversion: 0.190 containment for the run naming six subsystems against 0.182 for the run naming none, while coverage orders them 0.174 against 0.087.

The cut. Coverage is bimodal on both systems with a clear gap: on bash the runs fall into 0.087 to 0.130 and 0.174 to 0.522, on libxml2 into 0.000 to 0.104 and 0.208 to 0.247. Any cut inside both gaps produces the same split; **0.15** is used. No value in either gap changes any verdict.

TABLE 29

Per-run scorer output on the deeply identity-stripped substrate, knowledge-graph-only explorer, n=10 per system. *Components* is the number of components the run proposed. The final column applies the coverage cut described above.

Codebase	Run	Components	Containment	Coverage	Names the documented subsystems?
bash	run_4	5	0.182	0.087	no
bash	run_2	7	0.562	0.391	yes
bash	run_3	5	0.333	0.130	no
bash	run_1	9	0.423	0.478	yes
bash	run_5	8	0.409	0.391	yes
bash	run_29	6	0.438	0.304	yes
bash	run_30	6	0.556	0.435	yes
bash	run_28	10	0.500	0.522	yes
bash	run_31	8	0.579	0.478	yes
bash	run_32	8	0.190	0.174	yes
libxml2	run_1	2	0.000	0.000	no

Codebase	Run	Components	Containment	Coverage	Names the documented subsystems?
libxml2	run_2	6	0.429	0.078	no
libxml2	run_3	5	0.214	0.039	no
libxml2	run_4	9	0.857	0.234	yes
libxml2	run_5	5	0.500	0.104	no
libxml2	run_29	4	0.273	0.039	no
libxml2	run_28	9	0.792	0.247	yes
libxml2	run_30	8	0.889	0.208	yes
libxml2	run_31	10	0.654	0.221	yes
libxml2	run_32	11	0.704	0.247	yes

Totals. bash **8 of 10**, libxml2 **5 of 10**.

Regenerated by `scripts/identity_flatten_2x2.py --per-run`, which reads the run snapshots and the elicitation probe and calls the same scorer as B.4.