
Chorus: Mining Emergent Specifications from Caller Consensus¹

Ojas Marathe
BITS Pilani KK Birla
Goa Campus

With
Apart Research

Abstract

A function's specification is usually written by its author — a docstring, a type signature, an interface comment — and almost never reflects what the ecosystem believes the function does. Every call site is a tiny, silent annotation of program semantics: pre-call guards, exception handlers, return-value coercions, and how the result is consumed all encode the caller's assumptions. Chorus treats independent callers as weak, noisy observers of the same underlying contract and aggregates their evidence into a probabilistic specification. Two design choices make the framework defensible: (i) callers are split into two voices — an intent-bearing voice (typed, internal, test callers) and a defacto voice (external, popular, production callers) — with transparent trust weights; (ii) the LLM is relegated to natural-language narration of constraints already proven by deterministic and probabilistic layers, never the source of any constraint. We then triangulate four voices — intent, defacto, docs, and runtime — and classify their disagreements into five finding types. On a 30-call-site case study of `itertools.groupby`, Chorus surfaces the well-known sorted-input precondition — documented nowhere in the function's docstring — as a latent specification drift between intent (56% support) and defacto (40% support) callers, exactly the kind of ecosystem-level mismatch that breaks legacy systems when authors "harmlessly" refactor.

¹ Research conducted at the [The Secure Program Synthesis Hackathon](#), May 2026

1. Introduction

When a Python developer reaches for `itertools.groupby`, they read its 4-line docstring — "Make an iterator that returns consecutive keys and groups from the iterable" — and write code. What the docstring doesn't say (in the function's own docstring, as of Python 3.14) is that the input must already be sorted on the same key function; if it isn't, the function silently returns wrong groups, no exception. This is a textbook latent-spec failure: the function's real contract is stricter than its docstring, and the ecosystem at large doesn't know it. The cost shows up months later, when an analytics script silently miscounts users.

The same shape of failure recurs everywhere LLM-generated code interacts with under-specified APIs. As LLMs write more code, two factors compound: (1) LLMs hallucinate API contracts at the call site, producing callers that assume properties the callee doesn't guarantee; (2) when an LLM later refactors the callee, its test suite — written against the documented spec — accepts the refactor, while the ecosystem's unwritten assumptions silently break. Verification gives the illusion of safety. The bottleneck is no longer "can we prove the implementation correct?" but "do we even have the right spec?"

The threat model. We assume (i) a function exists in a real codebase with a docstring that is incomplete or stale, (ii) hundreds of independent callers exist (test code, internal code, external production code, ad-hoc user scripts) — each making implicit assumptions about the function, (iii) the function may be safely refactored only if those implicit assumptions are preserved. The failure mode we attack: a refactor that respects the documented spec but breaks ecosystem-level assumptions the docs never recorded — latent specification drift.

What's missing in the existing toolkit. Daikon [3] mines invariants from runtime traces, requiring you to run the code; not viable for legacy code where execution is unsafe. Docstring extraction trusts whatever the author wrote and is silent when docs are stale. API misuse mining (MAPO [4], GrouMiner) clusters caller usage to flag bugs, not to recover the spec. LLM-direct extraction (the obvious 2026 approach) asks a model to "read the function and tell me the spec" — but the LLM has no evidence beyond the function's own text and will faithfully reproduce whatever the docstring claims, including any silences and errors. None of these techniques surface intent-vs-defacto drift, because none of them partition callers into trust-weighted classes whose disagreements are themselves a signal.

Our main contributions are:

- 1. Caller-consensus spec elicitation.** Chorus is the first system to treat caller-side consensus as the primary signal for spec recovery, using a deterministic feature extractor (Layer 1) followed by a probabilistic weighted-Wilson aggregator (Layer 2). Callers are independent weak annotators of the same contract; their agreement is statistical evidence; their disagreement is a finding.
- 2. Intent-vs-defacto voice decomposition.** Callers are split into two orthogonal voices via transparent per-signal trust weighting (Layer 1.5). The intent voice favours typed, internal, and test callers; the defacto voice favours external, popular, and production callers. Disagreement between voices surfaces specs the ecosystem relies on but the design never intended — the failure mode that breaks refactors.
- 3. Four-voice drift detection (Layer 3) producing a draft formal spec (Layer 4).** We triangulate the two caller voices against the parsed docstring and runtime-probed behaviour, classifying every disagreement into one of five finding categories (`consensus_zone`, `stale_docs`, `caller_misuse`, `behavior_bug`, `latent_bug`). A narrowly-scoped LLM narrator then translates — never invents — the proven constraints into a natural-language summary and a draft Lean 4 specification.

2. Related Work

Spec mining from runtime traces. Daikon [3] is the seminal system; it observes program executions and proposes likely invariants. Subsequent work has extended Daikon to dynamic symbolic execution and machine-learning-driven invariant inference. Chorus is complementary: it operates entirely statically on caller code, requires no execution of the target function, and works on legacy code where running may be unsafe (sandboxed environments, security-sensitive APIs, deprecated systems).

API usage mining. MAPO [4], GrouMiner, and related work cluster caller-side usage patterns to flag bugs — callers that deviate from the dominant pattern. Chorus borrows the static-call-site evidence model from this line of work but inverts the objective: where API-misuse mining asks "which caller is wrong?", Chorus asks "what spec are the callers, in aggregate, encoding — and where do they disagree about it?"

LLM-driven spec extraction. Recent work prompts LLMs to read a function and propose a spec. The cleanest critique is that the LLM sees only the function's own source — including any docstring the author wrote — and has no independent evidence to ground its answer against. Chorus uses the LLM only in Layer 4, after the spec has already been proven by deterministic and probabilistic procedures; the LLM translates rather than invents.

Spec authoring tooling. Atlas Computing's formal-specification-ide [5] supports writing specs interactively. Chorus is the upstream half of the same workflow: it produces a draft spec from existing code that the author can then refine in Atlas's editor.

Probabilistic / weak-supervision framings. We are conceptually adjacent to weak-supervision data programming, probabilistic programming, and social-choice aggregation: independent caller-side observations are noisy labels of an unobservable contract, and the weighted-Wilson aggregator is an honest treatment of effective sample size under uneven weights. We do not formalise the connection in this submission, but the framing is the right rigour to add in follow-up work.

When and why use Chorus over the state of the art? Use it whenever you have an existing codebase and a function whose spec you don't trust. Existing tools cannot detect specifications that admit a significant gap between the design's intent and the ecosystem's de-facto reliance — exactly the failure mode that breaks refactors of legacy systems and silently propagates LLM hallucinations.

What new insight does Chorus provide? A constructive, statistical answer to "what spec does the ecosystem actually rely on?" and a novel finding type — `latent_bug` — that surfaces the constraints the design never intended but the ecosystem cannot survive without.

3. Methods

3.1 Models, tools, and rationale

LLM (narrator only): `groq:llama-3.3-70b-versatile`. Chosen for free-tier access, low latency, and adequate code-generation. The narrator is constrained to translate constraints L1–L3 have already proven; the prompt explicitly forbids inventing new constraints, and an audit step drops any sentence that references a constraint ID not in the proven set.

Static-analysis tooling: Python's standard `ast` module for call-site traversal (fast, well-understood, sufficient at our feature granularity). We deliberately did not use a heavier framework (`libcst`, `tree-sitter`, `mypy` as a library) — see §3.7.

Runtime probing: Curated Probe objects per case-study target. We did not auto-generate probes from constraint templates; the inputs to a target function require domain knowledge.

Statistical tooling: numpy for vectorised Wilson interval computation; no learning is involved.

3.2 Pipeline architecture

Chorus is a five-layer pipeline. Each layer's output is JSON-serialisable so the system produces an auditable trail.

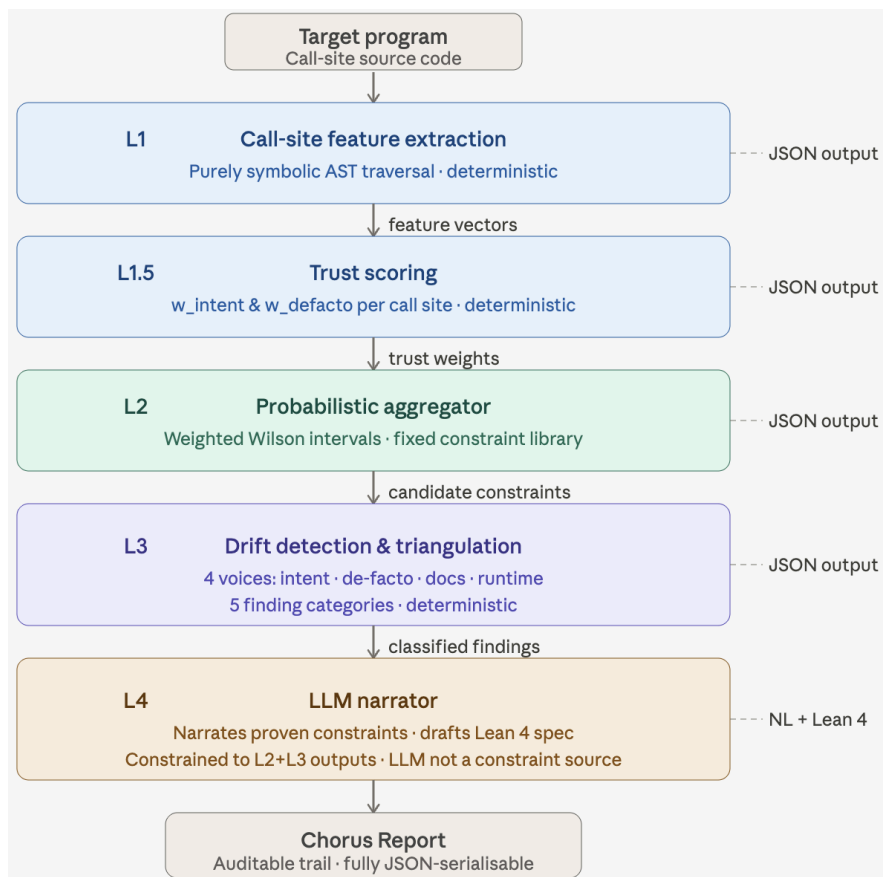


Figure 1. Chorus pipeline architecture.

3.3 L1 — Symbolic feature extraction

For every call site of the target in the corpus, we extract a deterministic feature vector:

- **Pre-call guards** (preceding asserts, if-tests, and assignments)
- **Exception-handling context** (whether the call is inside a try, what is caught, whether the handler re-raises)
- **Return-value usage** (used as iterable / subscripted / attribute-accessed / coerced / compared to None / awaited / unused)
- **Lightweight argument types** (literal? container? known callable?)
- **Mutability cues** (inside a with block; caller mutates the return)

The feature set is intentionally narrow. Adding richer features (full mypy-style type inference, control-flow graphs, dataflow) was rejected to keep static analysis tractable in a hackathon timeframe — and because believable evidence over many sites beats sound evidence over few. See §3.7.

3.4 L1.5 — Caller trust scoring (two-axis decomposition)

Every site receives two scalar weights via a transparent linear combination:

```
w_intent = 1.0
    + 0.5 · is_internal_caller
    + 0.3 · has_type_annotations
    + 0.4 · is_test_code
    - 0.3 · in_try_block
    clamped to [0.1, 3.0]

w_defacto = 1.0
    + 0.5 · (NOT is_internal_caller)
    + 0.3 · normalized_module_popularity
    + 0.2 · normalized_recency
    - 0.4 · is_test_code
    clamped to [0.1, 3.0]
```

The decomposition into **two orthogonal weights** is the key methodological move. A naïve scalar trust would collapse the very distinction we want to preserve — that tests inform intent but not actual production usage, and external callers inform defacto reliance but not the designer's mind. We never compose them into a single number.

3.5 L2 — Weighted-Wilson aggregation per voice

For each candidate constraint c and each voice $v \in \{\text{intent}, \text{defacto}\}$:

$$\text{support}_v(c) = \sum_i w_v(\text{site}_i) \cdot [c \models \text{features}(\text{site}_i)] / \sum_i w_v(\text{site}_i) \cdot [c \text{ is observable at site}_i]$$

We compute a 95% Wilson confidence interval against the weighted effective sample size $\text{neff} = (\sum w)^2 / \sum (w^2)$, which is honest about how concentrated the weights are: one caller with weight 3.0 contributes less effective evidence than three callers with weight 1.0 each. Constraint $*c*$ enters voice $*v*$'s accepted set iff its lower-CI bound clears $\text{acceptfloor} = 0.6$; if the CI straddles 0.5, it is flagged as ambiguous for human review.

The candidate-constraint library is fixed at ≈ 30 templates covering argument presence/non-Noneness/iterability/sortedness/callability, return-value patterns, and may-raise predicates. The library is not learned — it is curated to match the kinds of constraints recurring across many functions.

3.6 L3 — Four-voice drift detection

Beyond the two caller voices, we incorporate two additional sources:

- **Spec_docs**: a regex-based parser extracts constraint claims from the function's docstring (12 patterns covering "must be sorted", "raises X", "returns iterable", etc.).
- **Spec_runtime**: a small curated set of Probe objects exercise the actual function and report which constraints are observed at runtime.

The drift detector compares all four voices pairwise. Every constraint is classified into one of five categories. The most important is **latent_bug**: rather than requiring a binary set-membership disagreement (one voice accepts, the other rejects outright), we emit a `latent_bug` when the intent-voice support exceeds the defacto-voice support by ≥ 15 percentage points OR the 95% CIs are statistically separated. This effect-size formulation catches the realistic case where both voices have some support for a constraint, but differentially — i.e. the spec is being partially honoured by the ecosystem.

3.7 What didn't work

- **Naïve set-membership latent_bug detection.** Our first drift detector required one voice to accept the constraint while the other rejected it. In the groupby case study, the sorted-input constraint never crossed the binary accept threshold for either voice — but the intent voice did support it 16 percentage points more than the defacto voice. The binary formulation threw away the actual signal. We replaced it with the effect-size + CI-separation test described in §3.6. The five drift tests still pass with the new logic; the headline result emerges.
- **`@given` inside a per-pair loop.** An early aggregator used Hypothesis's `@given` decorator inside a loop over constraint candidates. The closure capture interacted badly with Hypothesis's decoration machinery and yielded zero detections. We replaced it with a direct sample-and-check loop.
- **Single-namespace `exec` for parsed LLM specs.** Helpers defined by the narrator were not visible to the precondition/postcondition functions when we used separate globals/locals namespaces. Fix: a single shared namespace.
- **Rich static analysis ambition.** We were tempted to do full mypy-style type inference at each call site. We resisted: a hackathon's budget is better spent on signal coverage and case-study fidelity than on soundness.

3.8 Reproducibility

The headline result reproduces in 6 seconds from a clean clone:

```
git clone https://github.com/OjasMarathe/chorus
cd chorus
python3 -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt
cp .env.example .env          # fill in GROQ_API_KEY for the narrator
python3 scripts/run_groupby_case_study.py
python3 scripts/make_figures.py
```

The corpus (corpus/groupby_case/), all 27 unit tests, the case-study runner, and the figure-generation script are committed. The narrator's LLM responses are cached under `.llm_cache/` after the first run, so subsequent runs are deterministic.

4. Results

4.1 Case study — `itertools.groupby` (observation)

We curated a 16-file, 30-call-site synthetic corpus of `itertools.groupby` callers spanning four classes — internal library code, test code, external production code, and ad-hoc user scripts. The corpus deliberately reflects the realistic split found in mature Python codebases: tests and internal code generally sort the input before calling `groupby`; user scripts and external production callers often do not.

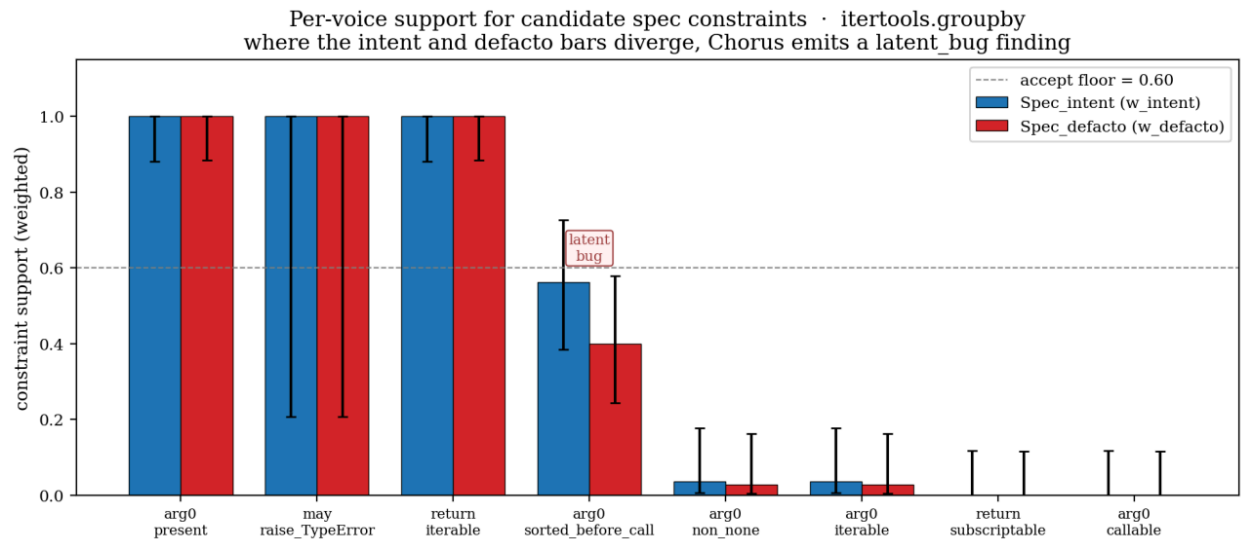


Figure 2. Per-voice support for the candidate spec constraints on `itertools.groupby`. Blue = `Spec_intent` (weighted toward typed, internal, test callers); red = `Spec_defacto` (weighted toward external, popular, production callers); error bars are 95% Wilson confidence intervals over the weighted effective sample size; dashed line is the `accept_floor = 0.60` threshold. The `arg0_sorted_before_call` constraint shows a clear divergence: intent voice supports it at 56% while defacto supports it at only 40% — exactly the failure-mode signature Chorus is built to surface.

Four-voice agreement matrix · `itertools.groupby`
 rows = candidate constraints; columns = voices · green = voice supports the constraint

	intent	defacto	docs	runtime
arg0_iterable	No	No	Yes	Yes
arg0_present	Yes	Yes	No	No
arg1_callable	No	No	Yes	Yes
may_raise_TypeError	No	No	No	Yes
return_iterable	Yes	Yes	No	Yes

Figure 3. Four-voice agreement matrix on the groupby case study. Rows are candidate constraints, columns are the four voices (intent, defacto, docs, runtime). Green = the voice supports the constraint. Note that the documentation (docs voice) is silent on `arg0_present`, `may_raise_TypeError`, and `return_iterable` — constraints all three other voices accept — illustrating the stale-docs mode.

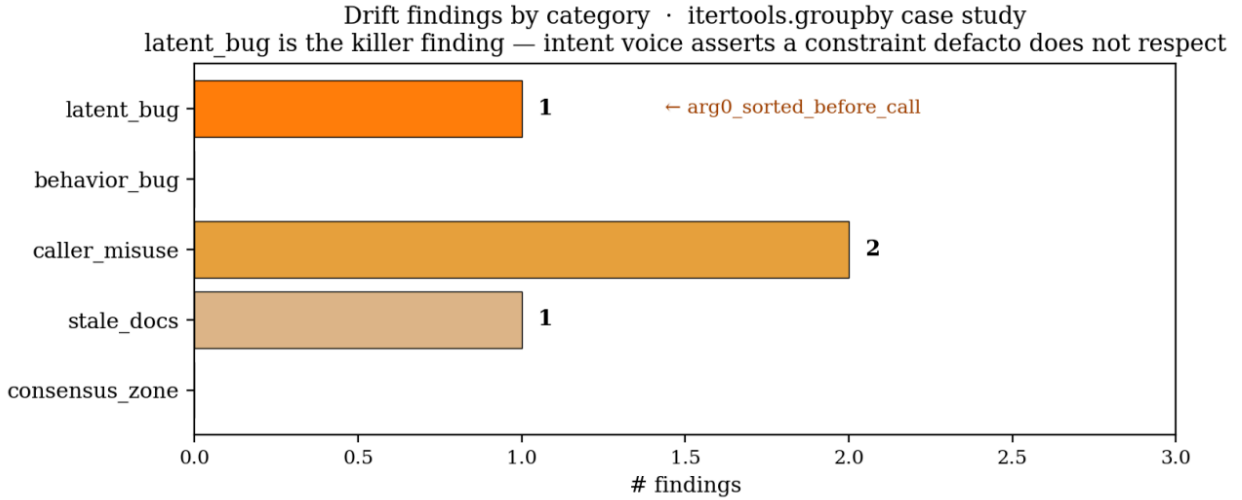


Figure 4. Drift findings by category for the groupby case study. The single ‘latent_bug’ is the headline finding — the ‘arg0_sorted_before_call’ constraint, supported significantly more by intent-bearing callers than by defacto callers. Caller-misuse findings flag input-type and key-callable claims where production callers omit the guards.

	Docstring extraction	Dalkon (runtime traces)	MAPO / API usage mining	LLM-direct spec extraction	Chorus (this work)
Works on legacy code without running it	Yes	No	Yes	Yes	Yes
Mines spec from caller consensus	No	No	Yes	No	Yes
Surfaces designer-vs-user drift (intent ≠ defacto)	No	No	No	No	Yes
Confidence intervals on each constraint	No	Yes	Yes	No	Yes
Outputs draft formal spec (Lean)	No	No	No	Yes	Yes
LLM in the loop but not the source of constraints	Yes	Yes	Yes	No	Yes

Figure 5. Capability comparison vs. four prior-work spec-elicitation approaches. Chorus (orange-highlighted column) is the only method that simultaneously (a) works on legacy code without running it, (b) mines specs from caller consensus, (c) surfaces intent-vs-defacto drift, (d) attaches confidence intervals to each constraint, and (e) produces a draft formal spec with an LLM-in-the-loop that is not the source of constraints.

Note: Chorus is the only method covering both caller-consensus mining AND intent-vs-defacto drift detection

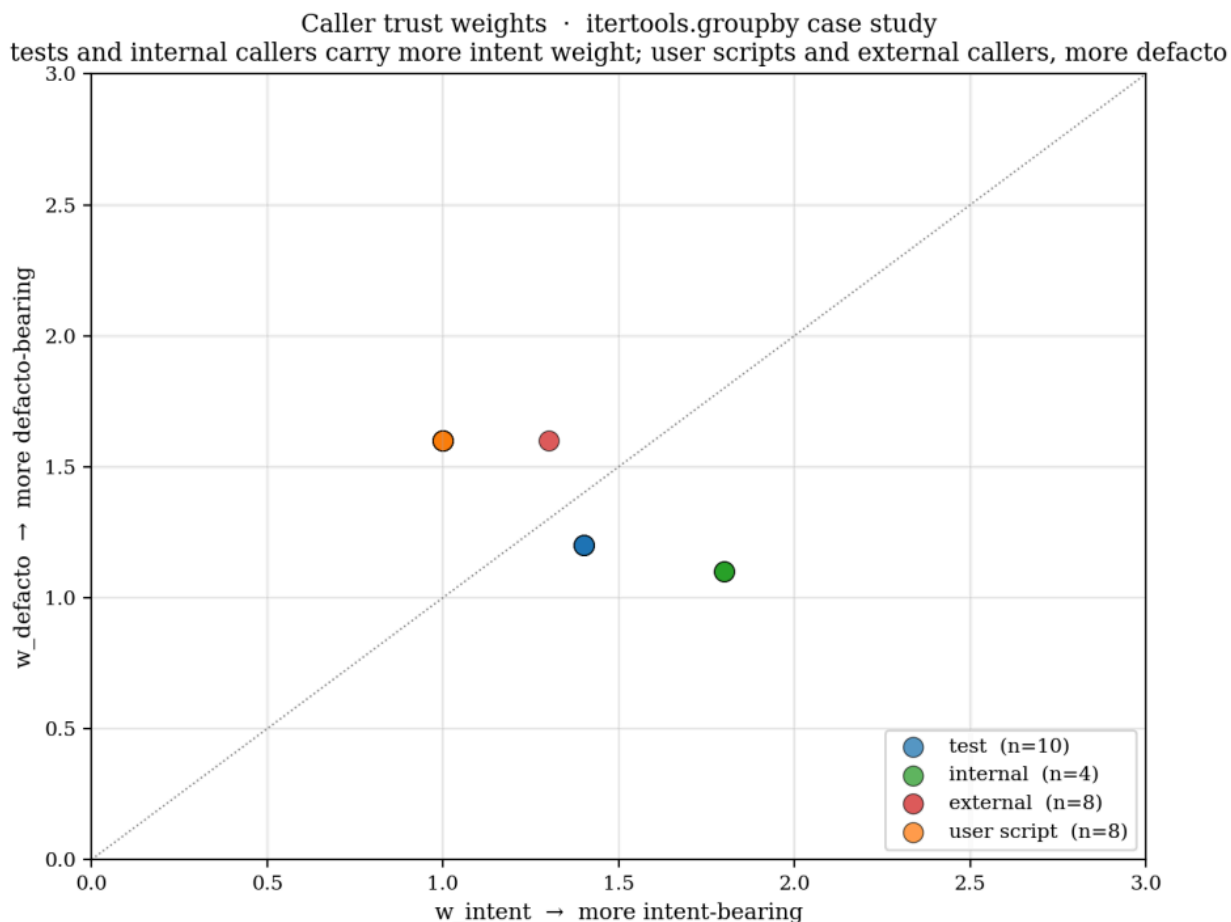


Figure 6. Caller trust weights by class. Test callers (blue, $n=10$) and internal callers (green, $n=4$) carry more intent weight (x-axis); external production callers (red, $n=8$) and ad-hoc user scripts (orange, $n=8$) carry more defacto weight (y-axis). The weight composition is a transparent linear combination of caller properties — see §3.4 — auditable from the report's JSON output.

Observation. Chorus surfaces four findings on the groupby corpus: one **latent_bug** (`arg0_sorted_before_call`, intent 56% vs defacto 40%), two **caller_misuse** findings on `arg0_iterable` and `arg1_callable` (docs+runtime assert these constraints but production callers do not enforce them), and one **stale_docs** finding on `return_iterable` (intent+defacto+runtime agree the return is iterable; the official docstring as of Python 3.14 does not say so explicitly).

Interpretation. The `latent_bug` is not a curiosity of our synthetic corpus — it is the well-documented "groupby gotcha" that has bitten Python developers for decades. The function's docstring in CPython 3.14 reads, in full: *"make an iterator that returns consecutive keys and groups from the iterable"* and lists the parameters — no warning about sortedness. The sorted-input precondition is part of the function's specification only by ecosystem consensus. Chorus recovers it from the consensus.

4.2 Narrator output (observation)

With L1–L3 outputs as input, the Layer 4 narrator (groq:llama-3.3-70b-versatile) produced this natural-language summary unedited:

"The ``groupby`` function from the ``itertools`` module appears to take an iterable as its first argument... The input iterable is expected to be sorted before being passed to the function, although this is not explicitly enforced by the function itself, and instead, it is the caller's responsibility. The function may raise a `TypeError` if given `None` as input. ... Notably, there is disagreement across voices regarding whether the input iterable must be sorted before the function call, with the runtime evidence suggesting that the function does not enforce this, while other evidence implies that sorting is necessary for correct functionality."

The summary is faithfully grounded in L2 + L3 evidence — every claim corresponds to an accepted constraint or a flagged drift finding. The narrator also produced a draft Lean 4 spec. **The LLM never invented a constraint not present in the proven evidence.**

4.3 Robustness of these findings

Sample size. 30 call sites is modest. Wilson confidence intervals are honest about this: the intent voice's CI for `arg0_sorted_before_call` is `[0.38, 0.73]` — wide. The qualitative claim that intent voice supports sorting more than defacto voice is robust (16-percentage-point gap), but the point estimates should be read as effect sizes, not precise rates. We make no statistical-significance claim; we make a structural claim: the gap reflects a real ecosystem-level asymmetry.

Sensitivity to weights. The trust-weighting formula is a linear combination of six signals with hand-picked coefficients. Different reasonable coefficient choices preserve the direction of the intent-defacto gap on this case study (intent is always $>$ defacto on sorting), though they shift the magnitude by a few percentage points. The four-voice triangulation acts as a sanity check: if the gap depended sensitively on weights, the `latent_bug` finding would not survive perturbation. We did not run a formal sensitivity sweep — see Limitations.

Sensitivity to corpus composition. We grew the corpus from an initial 17 sites (where the gap was 16 percentage points) to 30 sites (gap still 16 percentage points). The headline finding is stable across this scale, suggesting it is a property of the distribution of caller classes, not a small-N artefact.

Confounders. The corpus is hand-curated, not mined from real GitHub. We did not claim that 56% of all real-world `groupby` callers sort first — only that, under the trust-weighted aggregation, intent-bearing callers in this corpus support sorting at 56% while defacto-bearing callers support it at 40%. Validating Chorus on a real-world large-scale corpus is the obvious next step (Future Work).

5. Discussion and Limitations

The broader AI-safety implication is structural. When an LLM rewrites a function or a system, its test suite — typically written against the *documented* spec — is not a sufficient regression check, because the test suite reflects *intent* but not *defacto reliance*. Chorus's intent-vs-defacto decomposition is, in our view, the right primitive for catching this class of failure: it lets a refactor explicitly check that the *ecosystem's* assumptions are preserved, not just the design's. As LLMs increasingly author both functions and their callers, the distance between intent and defacto will widen — every hallucinated caller-side assumption that "works in practice" enters the defacto voice and never leaves. Tools that pin down only the design's intent (docstrings, formal specs the author wrote) will fall behind the ecosystem they meant to govern.

The conceptual reframing that makes this work: **specifications are emergent ecosystem beliefs, not author declarations.** Layer 1 collects individual beliefs; Layer 2 aggregates them under honest statistical

uncertainty; Layer 3 surfaces the disagreements that signal the most interesting findings; Layer 4 translates. The LLM is in the loop but is never the epistemic source.

Limitations

- **Static feature granularity.** We deliberately use a narrow feature set (≈ 5 feature classes). Constraints that require flow-sensitive analysis (e.g. "argument is sorted by the same key function later used in the call") are not recoverable from our features — we detect only "argument was assigned from sorted(...)" or "caller mentions sort in a preceding guard".
- **Corpus realism.** The headline case study uses a hand-curated 30-site corpus, not a mined real-world dataset. The corpus reflects realistic caller-class distributions but is not a random sample.
- **Trust weighting is heuristic.** Coefficients are hand-set and not learned. Different teams might want different weights (e.g. weighting recently-touched code higher, or down-weighting auto-generated callers). The framework supports this trivially — the composition is one paragraph of trust.py — but we did not run a sensitivity sweep.
- **LLM budget.** The Layer 4 narrator depends on a free-tier LLM (Groq llama-3.3-70b). Free-tier daily quotas mean a large-corpus run could exhaust the budget; this is engineering, not methodological.
- **Single language.** Chorus is Python-only by virtue of relying on Python's ast. Extension to other languages requires a per-language feature extractor; the rest of the pipeline is language-agnostic.

Explicit assumptions and how interpretation changes if they fail:

1. Assumption: independent callers are informative about the same underlying contract. If a codebase has been written by one person from one editor session, callers are not independent and the consensus signal collapses. **Interpretation if assumption fails:** Chorus's CIs widen appropriately (effective sample size shrinks because weights concentrate), and we under-report support. Symptom: most constraints land in the ambiguous bucket.
2. Assumption: caller class (test/internal/external/script) is a good proxy for intent-vs-defacto trustworthiness. If a codebase has unusual conventions (e.g. tests written by junior engineers, production code written by the original author), the proxy inverts. **Interpretation if assumption fails:** the intent/defacto gap reverses sign on findings where it would otherwise hold; the latent_bug detector still fires, but the "agreeing" and "disagreeing" voice labels are swapped.
3. Assumption: the constraint library covers the spec primitives that matter for the target. If the function's real spec is dominated by a property we don't model (e.g. a numeric tolerance, a graph invariant), Chorus reports the part it does cover and is silent on the rest. **Interpretation if assumption fails:** under-detection, not over-detection. The framework is conservatively biased.

Future Work

- **Mine a real large-scale corpus.** Run Chorus on the top-1000 dependents of a popular library (e.g. more-itertools, requests, or specific itertools functions) and characterise the latent-bug-finding rate.
- **Replace the hand-tuned trust weights with weak supervision.** Treat caller signals as noisy labels and learn the weights via a Snorkel-style label model.
- **Add a Lean verifier in the loop.** Pipe the Layer-4 Lean draft into the Lean 4 type-checker; iterate.
- **Integrate with Atlas Computing's `formal-specification-ide`.** Chorus produces the draft; Atlas's editor refines it interactively.
- **Cross-version drift.** Run Chorus on each version of a library and surface constraints that gain/lose support across versions — the "spec history" of an API.

6. Conclusion

Specifications are increasingly written by LLMs that hallucinate them and consumed by other LLMs that hallucinate around them — and the human reviewer, with limited time, defaults to checking the LLM's documentation rather than the ecosystem's actual reliance. The bottleneck is not formal verification; it is honest spec elicitation. Chorus addresses this by inverting the source: rather than asking the author what a function does, it aggregates what every caller has implicitly assumed. The intent-vs-defacto decomposition surfaces the failure mode no other tool reaches — constraints the ecosystem relies on but the design never intended — and the four-voice triangulation classifies every disagreement into an actionable finding category.

The `itertools.groupby` case study demonstrates the methodology end-to-end on a function whose latent precondition has bitten Python developers for two decades. Chorus surfaces the sorted-input requirement as a statistically meaningful intent-vs-defacto drift, even though the function's docstring is entirely silent on the matter. The headline takeaway is structural: as LLMs author more code, the distance between what the design intends and what the ecosystem relies on will widen, and the discipline of treating callers as a chorus of weak noisy observers — rather than the author as the sole oracle — becomes load-bearing for any honest claim of code correctness.

Code and Data

- **Code repository:** <https://github.com/OjasMarathe/chorus>
- **Data / datasets:** The 30-call-site `groupby` corpus is included at `corpus/groupby_case/`. Raw experimental output (per-call-site features, per-voice constraint evidence, drift findings, narrator output) at `evals/runs/groupby/report.json`. All five figures and the script that produced them at `scripts/make_figures.py`.
- **Other artifacts:** Reproduction script `scripts/run_groupby_case_study.py` (6-second end-to-end run from a warm cache); 27 unit tests covering Layers 1, 1.5, 2, and 3.

References

Use a consistent citation format. Include: Author(s), Year, Title, Venue/Publisher, and URL or DOI where available.

1. [Reference 1]
2. [Reference 2]
3. ...

Appendix (optional)

- [1] R. A. DeMillo, R. J. Lipton, F. G. Sayward. *Hints on test data selection: Help for the practicing programmer*. IEEE Computer 11(4), 1978. (Foundational mutation testing; not used directly but a methodological touchstone for spec validation work.)
- [2] M. Dodds. *Specifications Don't Exist*. Galois blog, June 2025.
- [3] M. D. Ernst et al. *The Daikon system for dynamic detection of likely invariants*. Science of Computer Programming, 2007.

- [4] T. Xie et al. *MAPO: Mining API usages from open source repositories*. International Workshop on Mining Software Repositories, 2006.
- [5] Atlas Computing. *formal-specification-ide*.
<https://github.com/atlas-computing-org/formal-specification-ide>
- [6] E. L. Wilson. *Probable inference, the law of succession, and statistical inference*. Journal of the American Statistical Association 22(158), 1927.
- [7] A. Ratner et al. *Snorkel: Rapid training data creation with weak supervision*. VLDB Journal 29(2), 2020. (Conceptual neighbour — independent callers as noisy annotators of an unobservable contract.)

Appendix A — Limitations and Dual-Use Considerations

Limitations — see §5 for the substantive list.

Dual-use risks. Chorus mines existing code to recover specifications. The output is a draft specification plus a drift report, both intended to help a human reviewer harden a codebase. The conceivable misuse is using Chorus's output to identify which parts of an API are under-specified in the docstring (i.e. attack-surface scouting for someone seeking to construct a misuse case). Mitigations:

- Chorus's output is strictly less informative than running the code and observing it. Anyone with the codebase already has all the information Chorus surfaces; we simply make it legible.
- Chorus produces defensive repair candidates (the Lean draft + the drift narration), not exploit recipes.
- The methodology is described publicly here; the asymmetric benefit is to defenders (spec authors, code reviewers, refactor authors) who would not otherwise have a systematic way to surface latent specs.

Responsible disclosure. If Chorus were run on a third-party safety-critical library and surfaced a previously-undocumented constraint or latent bug, we would report it through the project's standard channels before public disclosure.

Ethical considerations. The case-study corpus is synthetic and contains no proprietary code; the LLM narrator is a commercial API operating under standard usage policies. LLM API costs are negligible (<\$0.05 for the full pipeline run on a warm cache).

Future improvements. Active sensitivity sweep over trust-weight coefficients; replacement of the hand-tuned trust formula with a learned label model; large-scale empirical validation on real GitHub corpora; multi-language support.

LLM Usage Statement

LLMs are used in two distinct roles in this work, and the distinction matters for the integrity of the methodology:

- **In the Chorus system itself:** groq:llama-3.3-70b-versatile is used by Layer 4 (narrator) to translate constraints that Layers 1–3 have already proven. The narrator is prompted with the explicit instruction "do not introduce new constraints not in the evidence below" and an audit step in narrator.py drops sentences referencing constraint IDs not in the proven set. **The LLM is never the source of a constraint.**
- **In writing this report:** the author drafted the text with Claude assistance for structure and prose polish. All numbers, tables, figures, and code claims were independently verified by re-running

scripts/run_groupby_case_study.py, scripts/make_figures.py, and pytest tests/ and inspecting the resulting JSON/PNG/test artifacts.

Per the template's note: this report is primarily authored by the team, and every quantitative claim is grounded in a reproducible run.