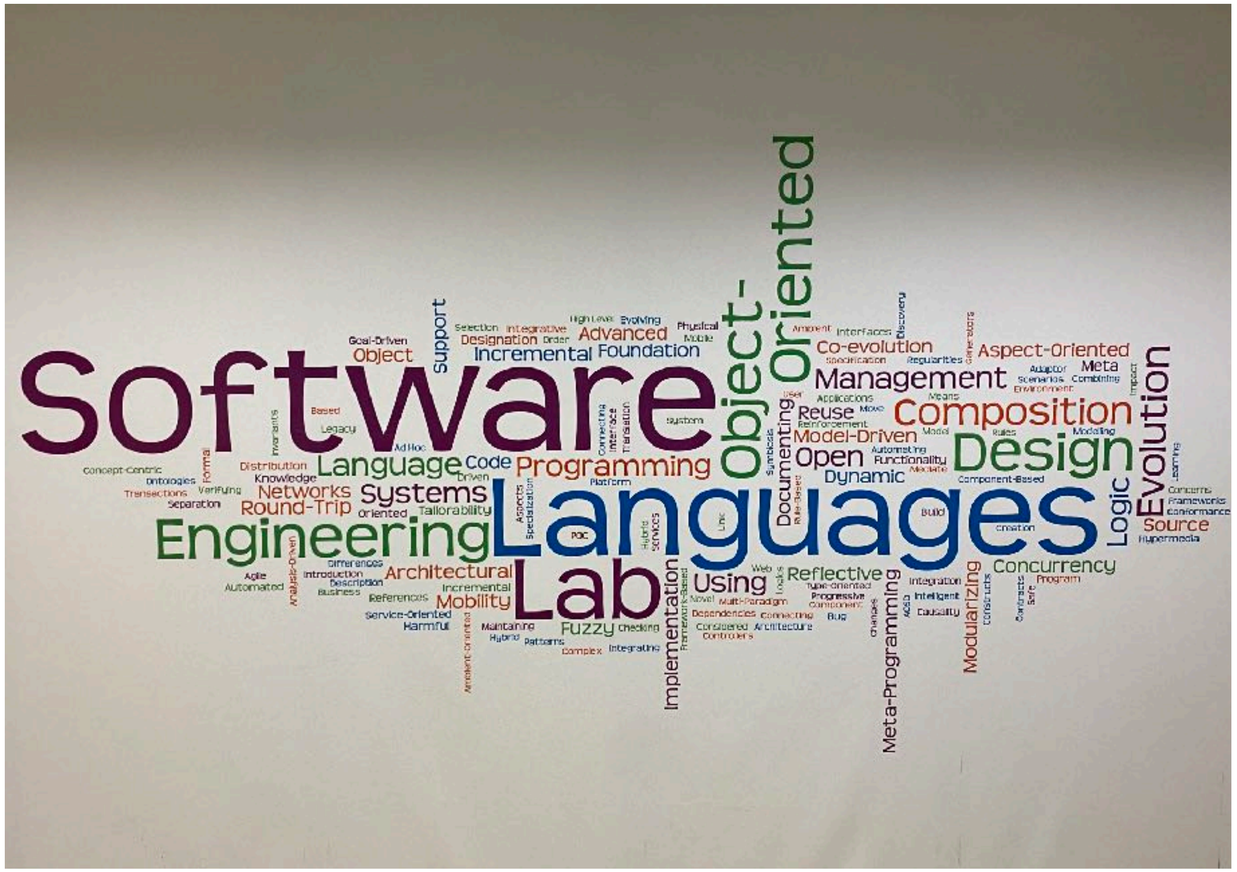
Code Smells in AI-Supported Workflows

Coen De Roover
coen.de.roover@vub.be

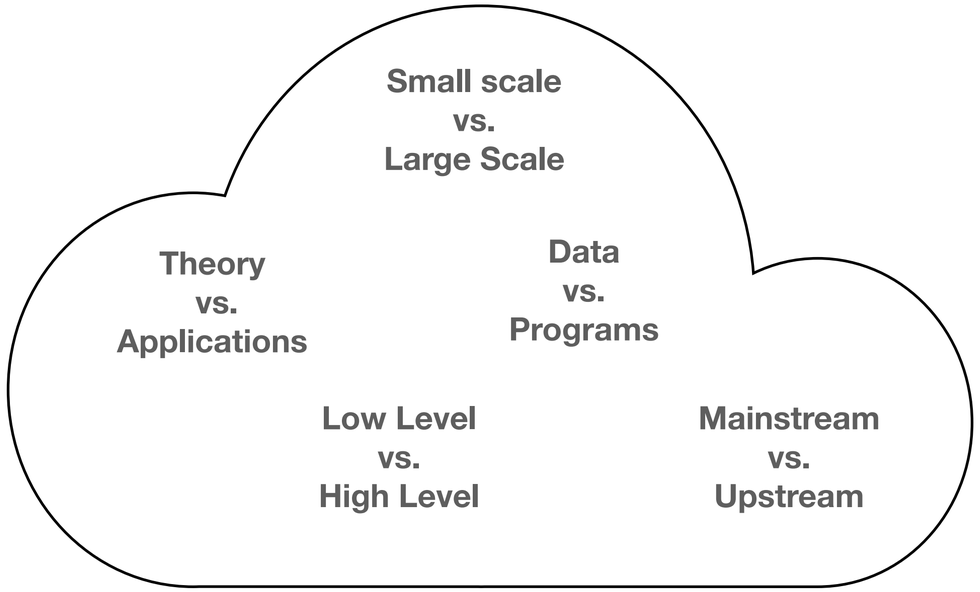
Gilberto Recupito
gilberto.recupito@vub.be

About VUB-SOFT



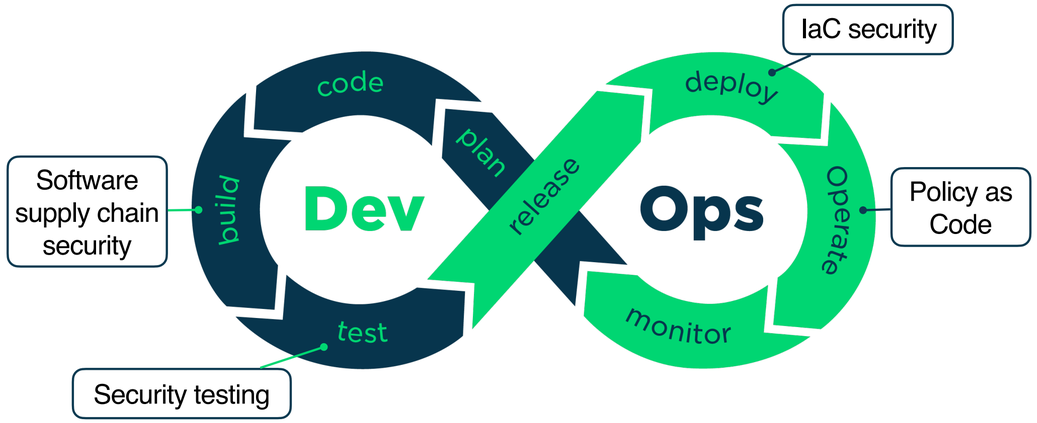
- 1 of 2 **large research groups** (other being AI) within the **Departement of Computer Science** (DINF) at **Vrije Universiteit Brussel** (VUB) in the capital of Belgium
- originating from the **2008 merger** of SSEL and PROG, both **founded in 1987** by Prof. emeriti V. Jonckers and T. D'Hondt
- currently counting about **30 pre-doctoral researchers, 11 post-docs, and 1 valorisation manager –led by 6 full-time faculty members**
- supported by portfolio covering entire **spectrum from fundamental over strategic basic to applied research**

VUB-SOFT and Security Research



Research Diversity Dimensions at SOFT

“ SOFT researches **theories**, **technologies** and **methods** that are at the basis of, or help to improve, the **construction** **of** **software** at all layers of the **software stack** that **begins** where **computer engineering ends** and **goes up** to and includes **application construction**. ”



- empirical research into **secure SE practices**
- **reactive security** & incident management
- dynamic and static program analyses
- tools for SCA, SAST, DAST, IAST, RASP
- secure programming **abstractions**
- secure programming **languages**

VUB-SOFT in the room

SOFTWARE LANGUAGES LAB

come and talk to us!

Coen De Roover

professor

program analysis
design and
applications in
software quality

Gilberto Recupito

post-doc researcher

software agents
design smells

Giacomo Benedetti

post-doc researcher

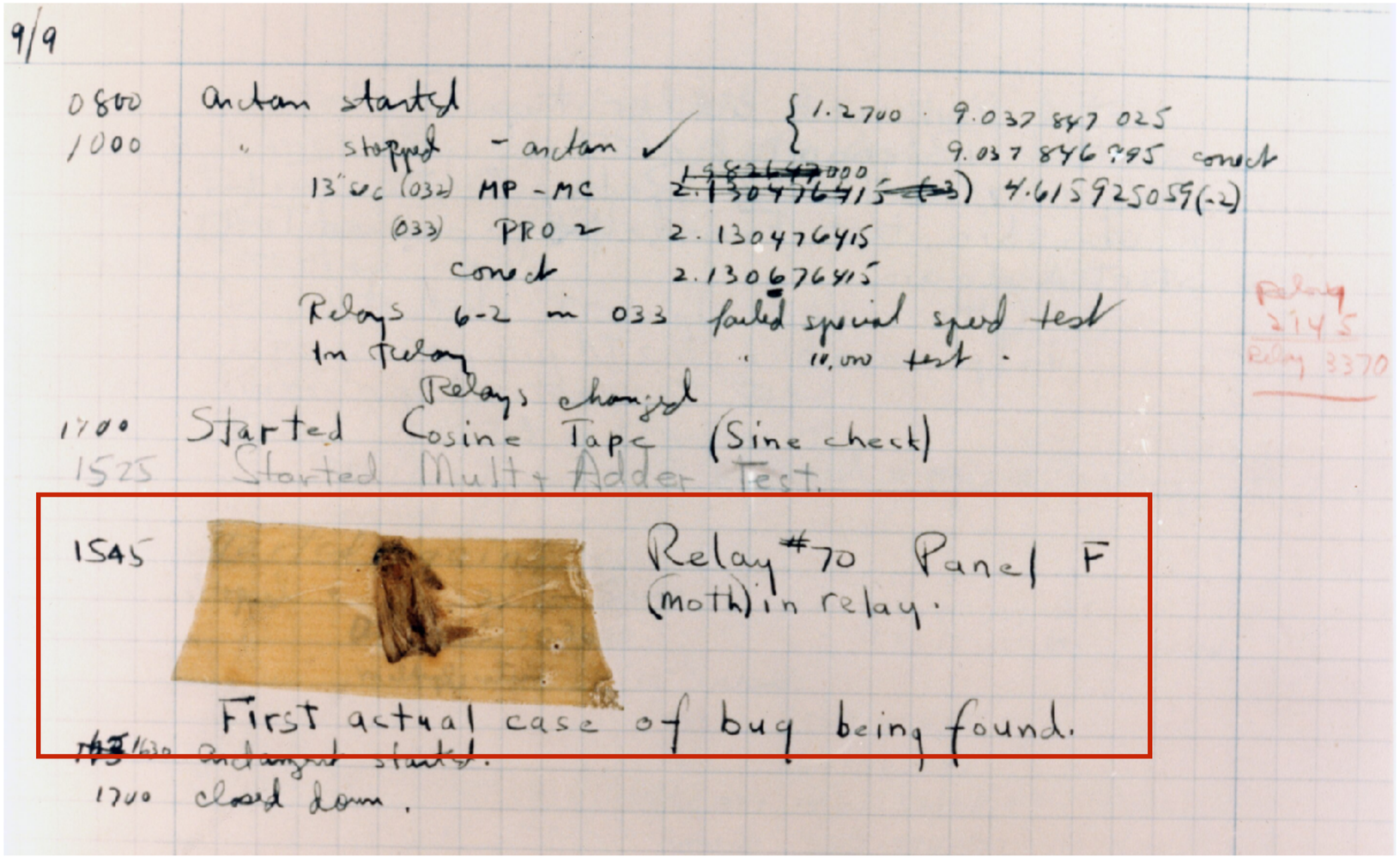
software supply
chain security
CI/CD

Nivedita Yadav

business developer

outreach
R&D funding expert

Famous defects .. in relays

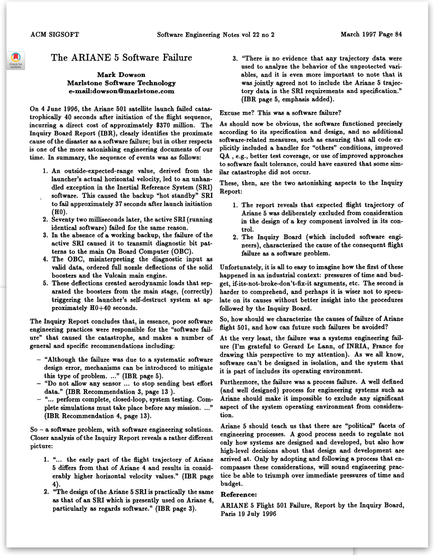


Discovered by **Grace Murray Hopper** on September 9th of 1947

Famous defects .. in space

On June 4th of 1996, the **Ariane 5 maiden flight** ended up a 370 billion dollar disaster.

Integer conversion from 64-bit floating point number to 16-bit signed integer caused an **overflow** and a hardware exception.
The code was reused from Ariane 4.



ACM SIGSOFT Software Engineering Notes vol 22 no 2 March 1997 Page 84

The ARIANE 5 Software Failure

Mark Dowson
Marlstone Software Technology
e-mail:dowson@marlstone.com

On 4 June 1996, the Ariane 501 satellite launch failed catastrophically 40 seconds after initiation of the flight sequence, incurring a direct cost of approximately $370 million. The Inquiry Board Report (IBR), clearly identifies the proximate cause of the disaster as a software failure; but in other respects is one of the more astonishing engineering documents of our time. In summary, the sequence of events was as follows:

1. An outside-expected-range value, derived from the launcher's actual horizontal velocity, led to an unhandled exception in the Inertial Reference System (SRI) software. This caused the backup "hot standby" SRI to fail approximately 37 seconds after launch initiation (H0).
2. Seventy two milliseconds later, the active SRI (running identical software) failed for the same reason.
3. In the absence of a working backup, the failure of the active SRI caused it to transmit diagnostic bit patterns to the main On Board Computer (OBC).
4. The OBC, misinterpreting the diagnostic input as valid data, ordered full nozzle deflections of the solid boosters and the Vulcain main engine.
5. These deflections created aerodynamic loads that separated the boosters from the main stage, (correctly) triggering the launcher's self-destruct system at approximately H0+40 seconds.

The Inquiry Report concludes that, in essence, poor software engineering practices were responsible for the "software failure" that caused the catastrophe, and makes a number of general and specific recommendations including:

- "Although the failure was due to a systematic software design error, mechanisms can be introduced to mitigate this type of problem. ..." (IBR page 5).
- "Do not allow any sensor ... to stop sending best effort data." (IBR Recommendation 2, page 13).
- "... perform complete, closed-loop, system testing. Complete simulations must take place before any mission. ..." (IBR Recommendation 4, page 13).

So – a software problem, with software engineering solutions. Closer analysis of the Inquiry Report reveals a rather different picture:

1. "... the early part of the flight trajectory of Ariane 5 differs from that of Ariane 4 and results in considerably higher horizontal velocity values." (IBR page 4).
2. "The design of the Ariane 5 SRI is practically the same as that of an SRI which is presently used on Ariane 4, particularly as regards software." (IBR page 3).
3. "There is no evidence that any trajectory data were used to analyse the behavior of the unprotected variables, and it is even more important to note that it was jointly agreed not to include the Ariane 5 trajectory data in the SRI requirements and specification." (IBR page 5, emphasis added).

Excuse me? This was a software failure?

As should now be obvious, the software functioned precisely according to its specification and design, and no additional software-related measures, such as ensuring that all code explicitly included a handler for "others" conditions, improved QA , e.g., better test coverage, or use of improved approaches to software fault tolerance, could have ensured that some similar catastrophe did not occur.

These, then, are the two astonishing aspects to the Inquiry Report:

1. The report reveals that expected flight trajectory of Ariane 5 was deliberately excluded from consideration in the design of a key component involved in its control.
2. The Inquiry Board (which included software engineers), characterized the cause of the consequent flight failure as a software problem.

Unfortunately, it is all to easy to imagine how the first of these happened in an industrial context: pressures of time and budget, if-its-not-broke-don't-fix-it arguments, etc. The second is harder to comprehend, and perhaps it is wiser not to speculate on its causes without better insight into the procedures followed by the Inquiry Board.

So, how should we characterize the causes of failure of Ariane flight 501, and how can future such failures be avoided?

At the very least, the failure was a systems engineering failure (I'm grateful to Gerard Le Lann, of INRIA, France for drawing this prespective to my attention). As we all know, software can't be designed in isolation, and the system that it is part of includes its operating environment.

Furthermore, the failure was a process failure. A well defined (and well designed) process for engineering systems such as Ariane should make it impossible to exclude any significant aspect of the system operating environment from consideration.

Ariane 5 should teach us that there are "political" facets of engineering processes. A good process needs to regulate not only how systems are designed and developed, but also how high-level decisions about that design and development are arrived at. Only by adopting and following a process that encompasses these considerations, will sound engineering practice be able to triumph over immediate pressures of time and budget.

References

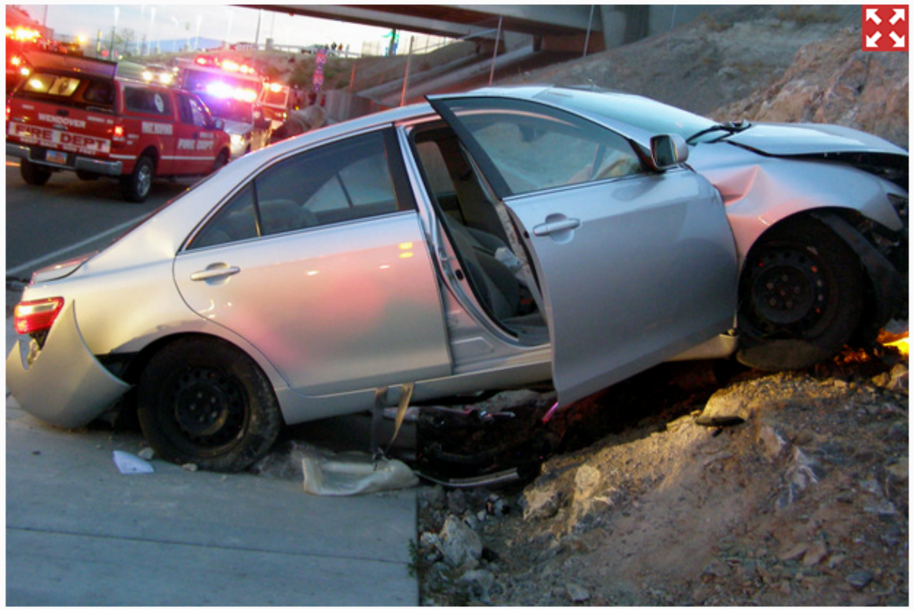
ARIANE 5 Flight 501 Failure, Report by the Inquiry Board, Paris 19 July 1996

```
...
declare
  vertical_veloc_sensor: float;
  horizontal_veloc_sensor: float;
  vertical_veloc_bias: integer;
  horizontal_veloc_bias: integer;
  ...
begin
declare
pragma suppress(numeric_error, horizontal_veloc_bias);
begin
  sensor_get(vertical_veloc_sensor);
  sensor_get(horizontal_veloc_sensor);
  vertical_veloc_bias := integer(vertical_veloc_sensor);
  horizontal_veloc_bias := integer(horizontal_veloc_sensor);
  ...
exception
  when numeric_error => calculate_vertical_veloc();
  when others => use_irs1();
end;
end irs2;
```

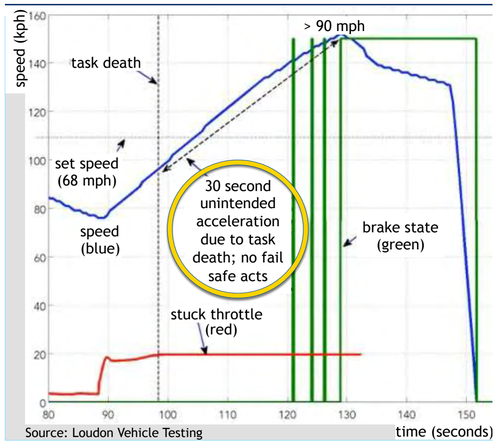
Famous defects .. in cars

Toyota to pay $1.2B settlement in vehicle acceleration lawsuit

By Bob Fredericks and Post Wires March 19, 2014 | 9:19am



A Toyota Camry that crashed as it exited Interstate 80 in Wendover, Utah, in 2010.
Photo: AP



On October 24th 2013, a jury ruled against Toyota and found that **unintended acceleration incidents** could have been caused by deficiencies in its electronic throttle control system. A single **bit flip could kill a software task monitoring the accelerator pedal**.

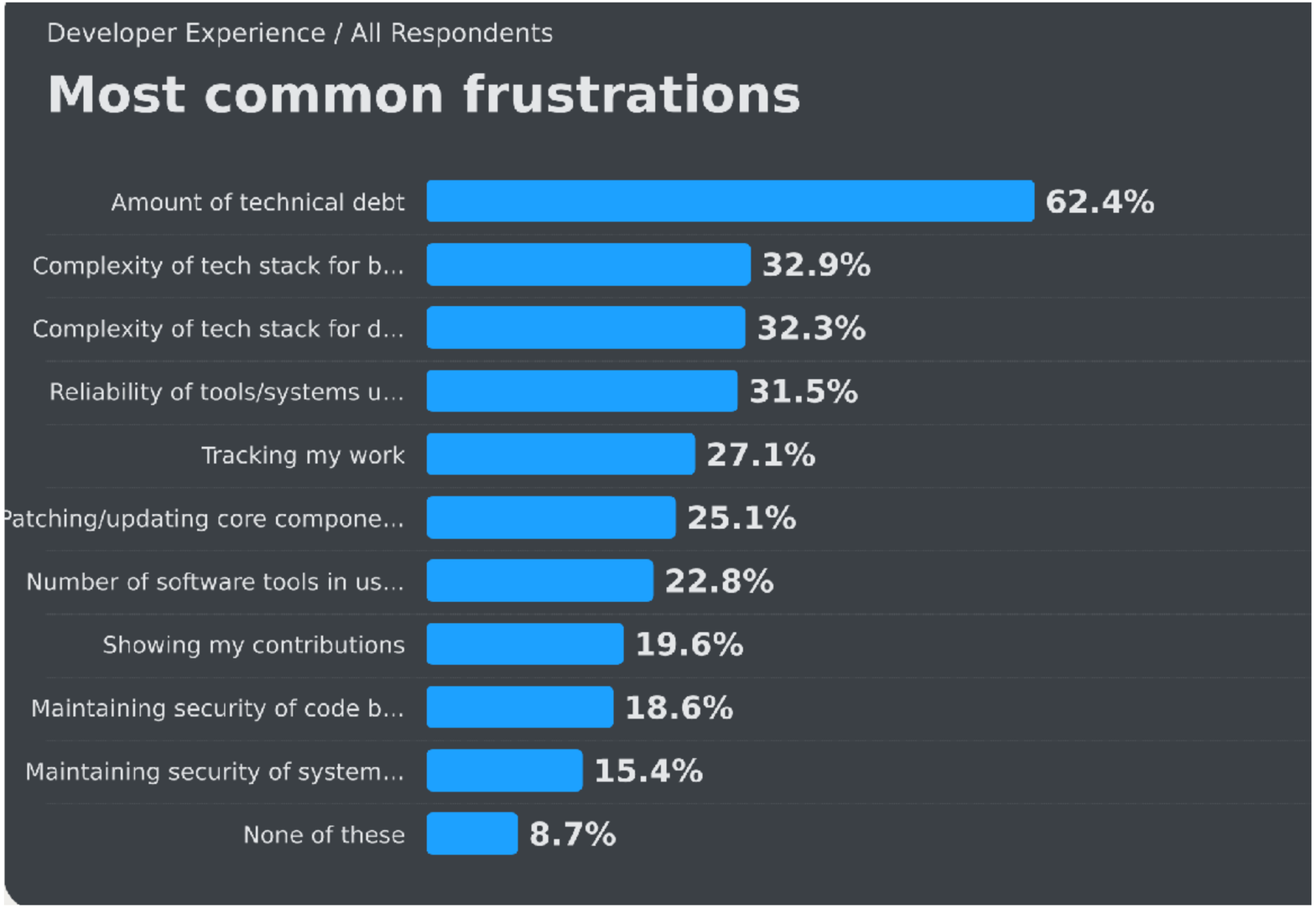
important factor hypothesised in court: “technical debt”

- **data flow hard to understand**
  - > 11.000 global variables
- **control flow hard to understand**
  - > 100 independent paths through throttle function
  - 67 functions > 50 paths
  - “Some of them are even so complex that they are what is called unmaintainable, which means that if you go in to fix a bug or to make a change, you're likely to create a new bug in the process.” [Barr]

Technical debt

The **implied cost** of **future rework** when choosing **quick but suboptimal solutions** during software development.

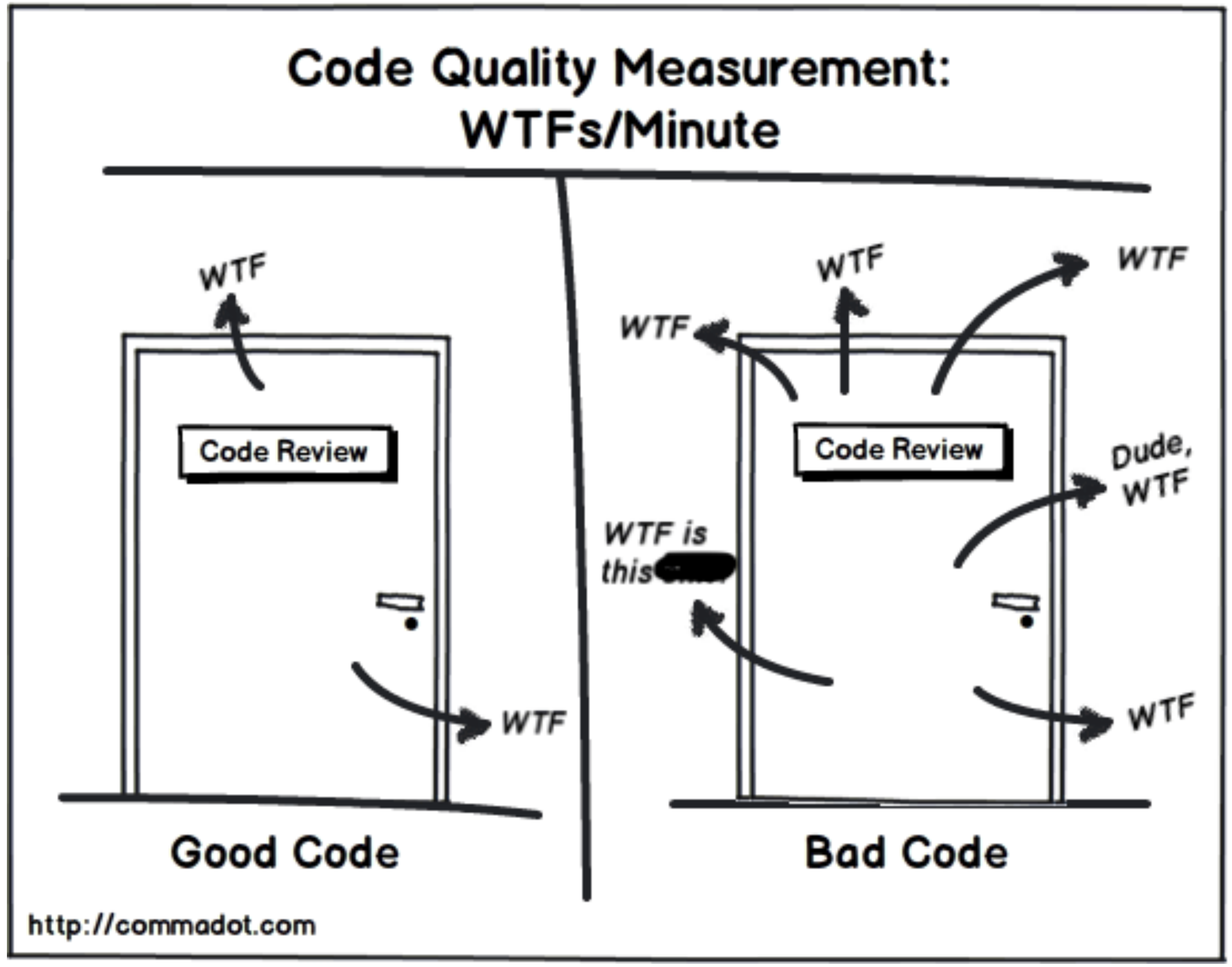
Technical debt leads to frustration



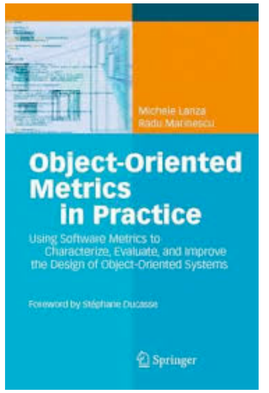
StackOverflow developers survey reports Technical Debt as the **most common cause** of **developer frustration [1]**.

[1] https://survey.stackoverflow.co/2024/professional-developers#developer-experience

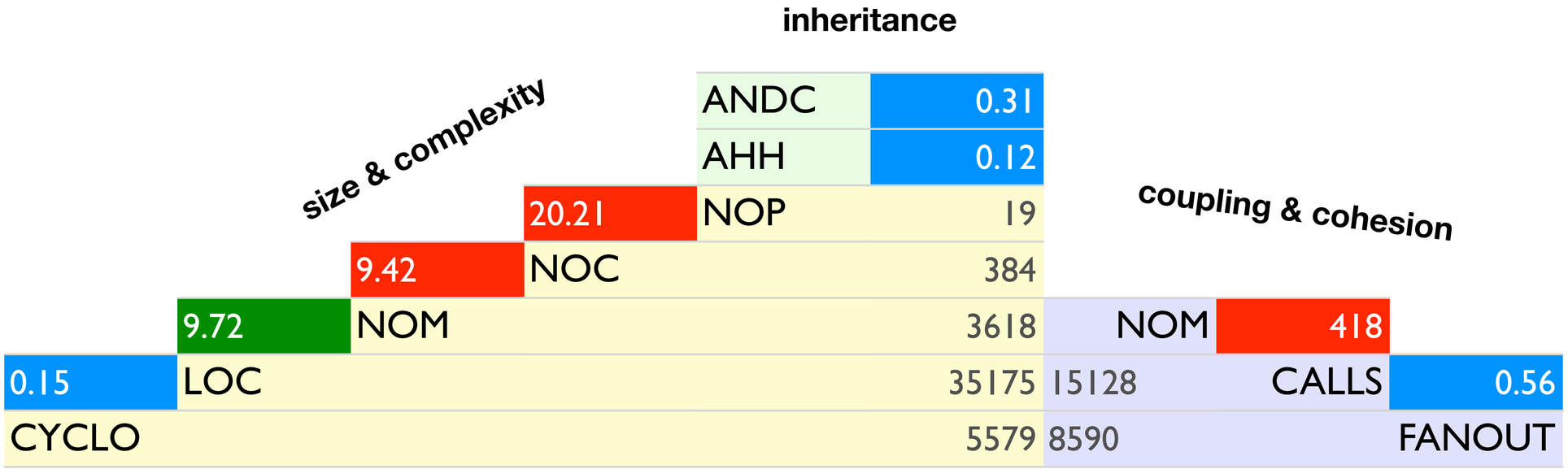
One way to measure technical debt



Automated way: code metrics

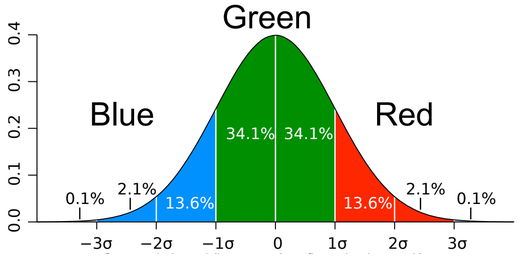


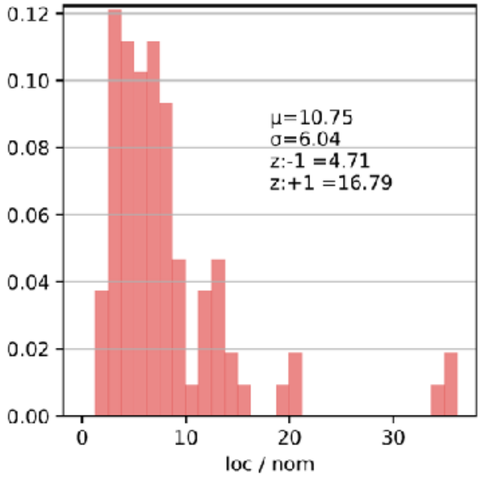
[Lanza, Marinescu 2006]



overview pyramid

- computes a selection of code metrics
- computes meaningful ratios
  - e.g., LOC/NOM is more meaningful than LOC or NOM alone
- colors them according to empirically-established thresholds
  - determined from distribution of ratios computer for other systems
  - e.g., LOC/NOM is “normal” for this system





From code metrics to design smells

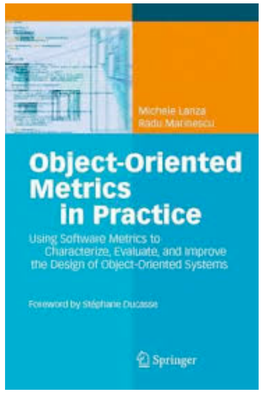
[Lanza, Marinescu 2006] formulate smells in terms of metrics and thresholds

e,g., **God class** “Centralizes intelligence, does everything, uses data from other data classes”

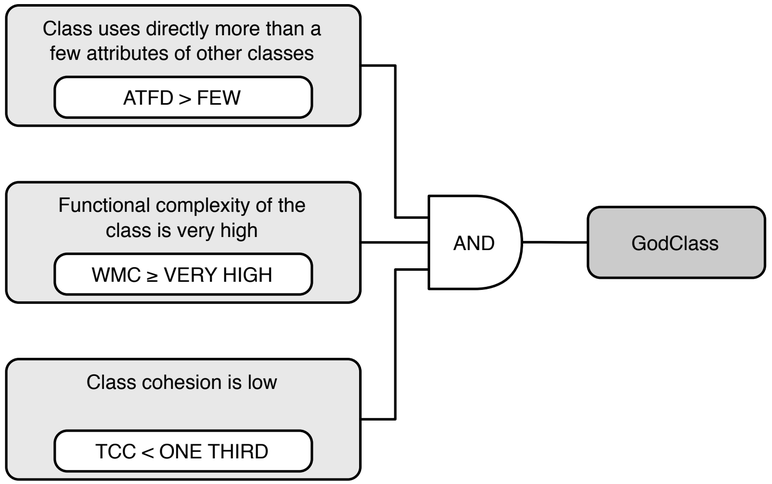
top-level classes should share work uniformly

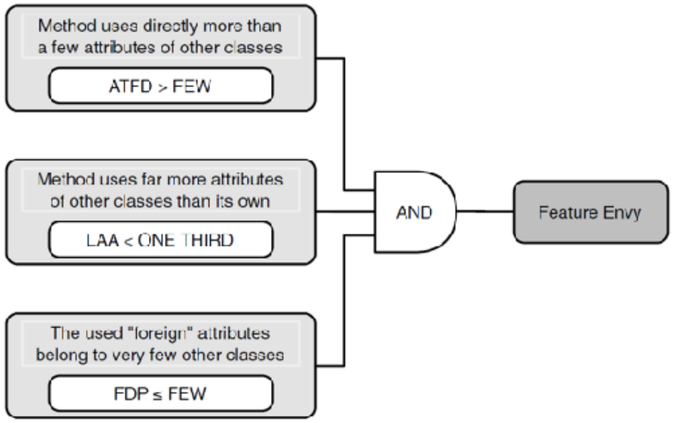
beware of classes with much non-communicative behavior

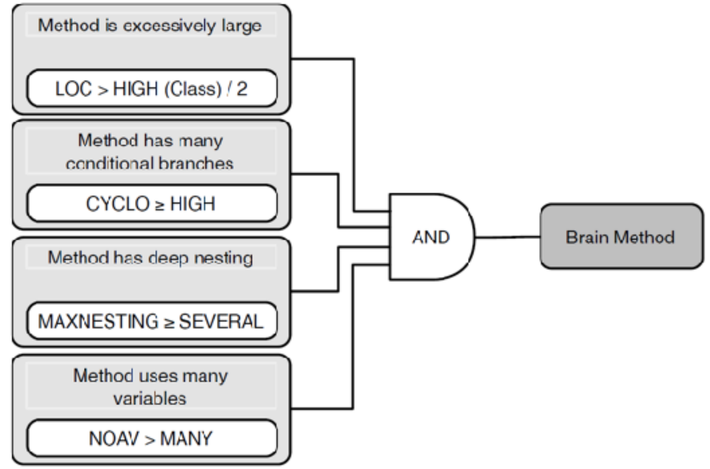
beware of classes that directly access data from other classes



[Lanza, Marinescu 2006]







Which smells to smell

Assessing technical debt by identifying design flaws in software systems

R. Marinescu

Tough time-to-market constraints and unanticipated integration or evolution issues lead to design tradeoffs that usually cause flaws in the structure of a software system. Thus, maintenance costs grow significantly. The impact of these design decisions, which provide short-term benefits at the expense of the system's design integrity, is usually referred to as technical debt. In this paper, we propose a novel framework for assessing technical debt using a technique for detecting design flaws, i.e., specific violations of well-established design principles and rules. To make the framework comprehensive and balanced, it is built on top of a set of metrics-based detection rules for well-known design flaws that cover all of the major aspects of design such as coupling, complexity, and encapsulation. We demonstrate the effectiveness of the framework by assessing the evolution of technical debt symptoms over a total of 63 releases of two popular Eclipse projects. The case study shows how the framework can detect debt symptoms and past refactoring actions. The experiment also reveals that in the absence of such a framework, restructuring actions are not always coherent and systematic, not even when performed by very experienced developers.

Introduction

Software systems must evolve continuously to cope with requirements and environments that are always changing [1]. Stringent time to market constraints and fast emerging business opportunities require swift, but profound, changes of the original system. Therefore, in most software projects, the focus is on immediate completion and on choosing a design or development approach that is effective in the short term, even at the price of increased complexity, and higher overall development cost in the long term [2]. Cunningham introduced the term *technical debt* [3] to describe this widespread phenomenon. According to this metaphor, *interest payments* represent the extra maintenance effort that will be required in the future, due to the hasty, inappropriate design choices that are made today, and *paying the principal* means the effort required to restructure the part of the systems that were improperly designed in the past. As in financial debt, there are two options: continuously paying the interest or paying down the principal, by refactoring the design affected by flaws into a better one and consequently gaining through reduced future interest payments [4].

When debt is incurred, at first, there is a sense of rapid feature delivery, but later, integration, testing, and "bug" (i.e., flaw) fixing become unpredictable, to the point where the effort of adding features becomes so high that it exceeds the cost of writing the system from the ground up [5]. The cause of this unfortunate situation is usually less visible: The internal quality of the system design is declining [6], and duplicated code, overly complex methods, noncohesive classes, and long parameter lists are just a few signs of this decline [7]. These issues, and many others, are usually the symptoms of higher-level design problems, which are usually known as design flaws [8], design smells [7], or anti-patterns [9].

Technical debt is not always bad. Often, the decision to incur debt is the correct one, especially when considering nontechnical constraints [10], but most of the time, incurring debt has a negative impact on the quality of a system's design. In this paper, we propose a framework for assessing technical debt by exposing debt symptoms at the design level. Our framework is based on detecting a set of relevant

Digital Object Identifier: 10.1147/JRD.2012.2204512

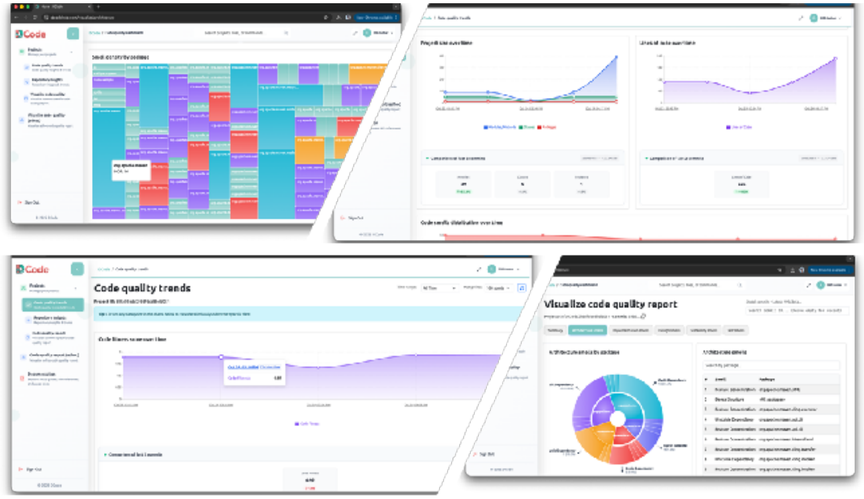
0018-8646/12/$5.00 © 2012 IBM

[Marinescu 2012]

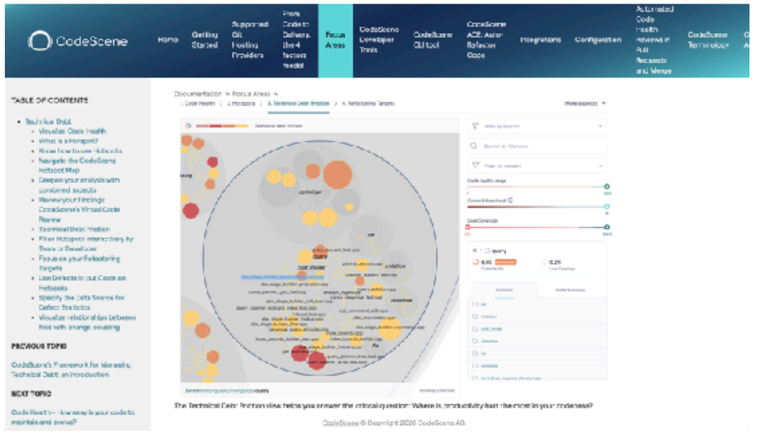
- empirical studies show these **8 smells** have
  - high occurrence rate
  - significant negative impact on quality
- cover four **major characteristics** of **good design**
  - low coupling
  - high cohesion
  - moderate complexity
  - proper encapsulation
- **automatically detectable** with
  - good precision
  - good recall
- **typical detection methods** include
  - **metrics-based**
    - product metrics:
      e.g., CK suite, …
    - process metrics:
      e.g., churn = frequency & volume of code changes
  - **rule-based** by pattern matching against AST, heuristic thresholds
  - through **ML model** trained on examples

Design flaw	Description
God Class	An excessively complex class that breaks encapsulation by directly using data from other classes.
Schizophrenic Class	A class that has a very low cohesion, as it defines a large interface that is used by disjoint groups of clients.
Refused Parent Bequest	A subclass that shows a weak connection to its superclass, by insufficiently using the hierarchy-specific methods and data inherited from the superclass.
Data Class	A class that does not encapsulate its data and usually does not provide significant functionality, allowing other classes to use its data directly.
Code Duplication	A method that has a significant number of code lines duplicated with at least another method, from a different class.
Brain Method	A method that is overly long, with statements showing a deep nesting level and an over-complex branching structure.
Data Clumps	A long parameter list, which appears over and over again in many other methods throughout the system.
Intensive Coupling	A method that calls an excessive number of other method from one or several other classes.

A non-exhaustive selection of tools

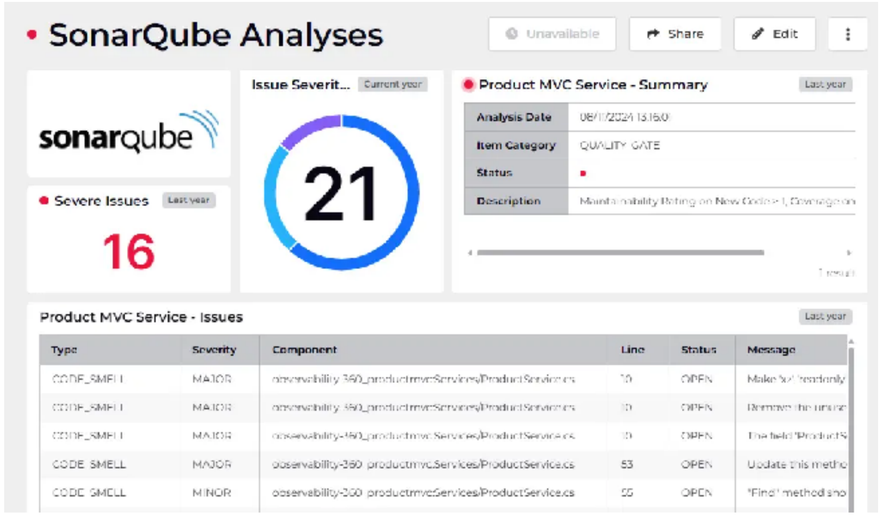


Designite & DCode



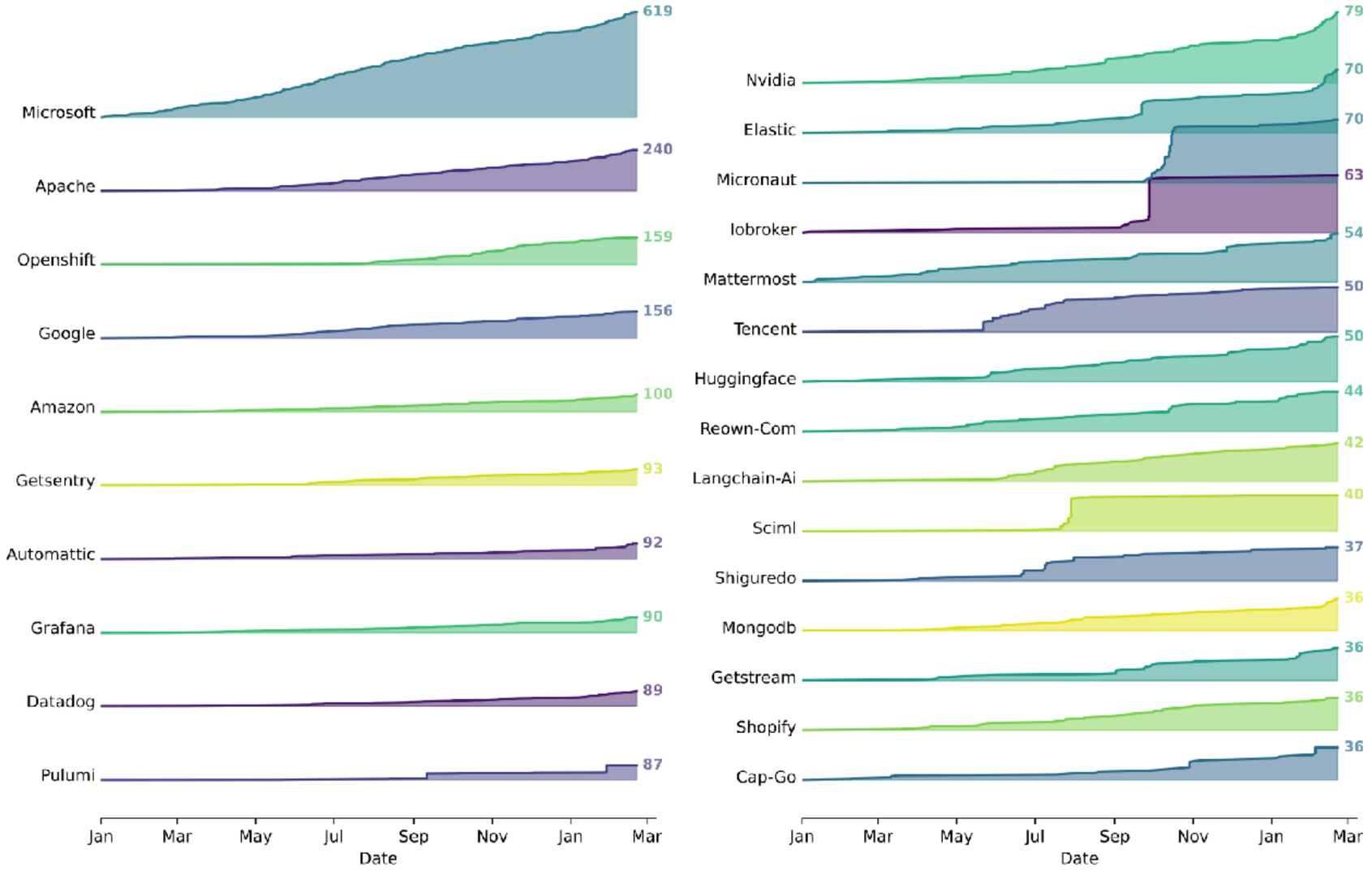
CodeScene

SQAle



SonarQube

Times are changing: agentic activity on GitHub



[3] Robbes, R., Matricon, T., Degueule, T., Hora, A., & Zacchiroli, S. (2026). Agentic Much? Adoption of Coding Agents on GitHub. *arXiv preprint arXiv:2601.18341*.

The AI PR Scenario

"An AI coding agent opens a pull request. The code compiles. Tests pass. The feature works. Should we merge it?"

Functional correctness is not the only quality dimension.

What's missing from "tests pass":

- Readability for reviewers
- Maintainability and changeability
- Observability and testability

The AI PR Scenario

"An AI coding agent opens a pull request. The code compiles. Tests pass. The feature works. Should we merge it?"

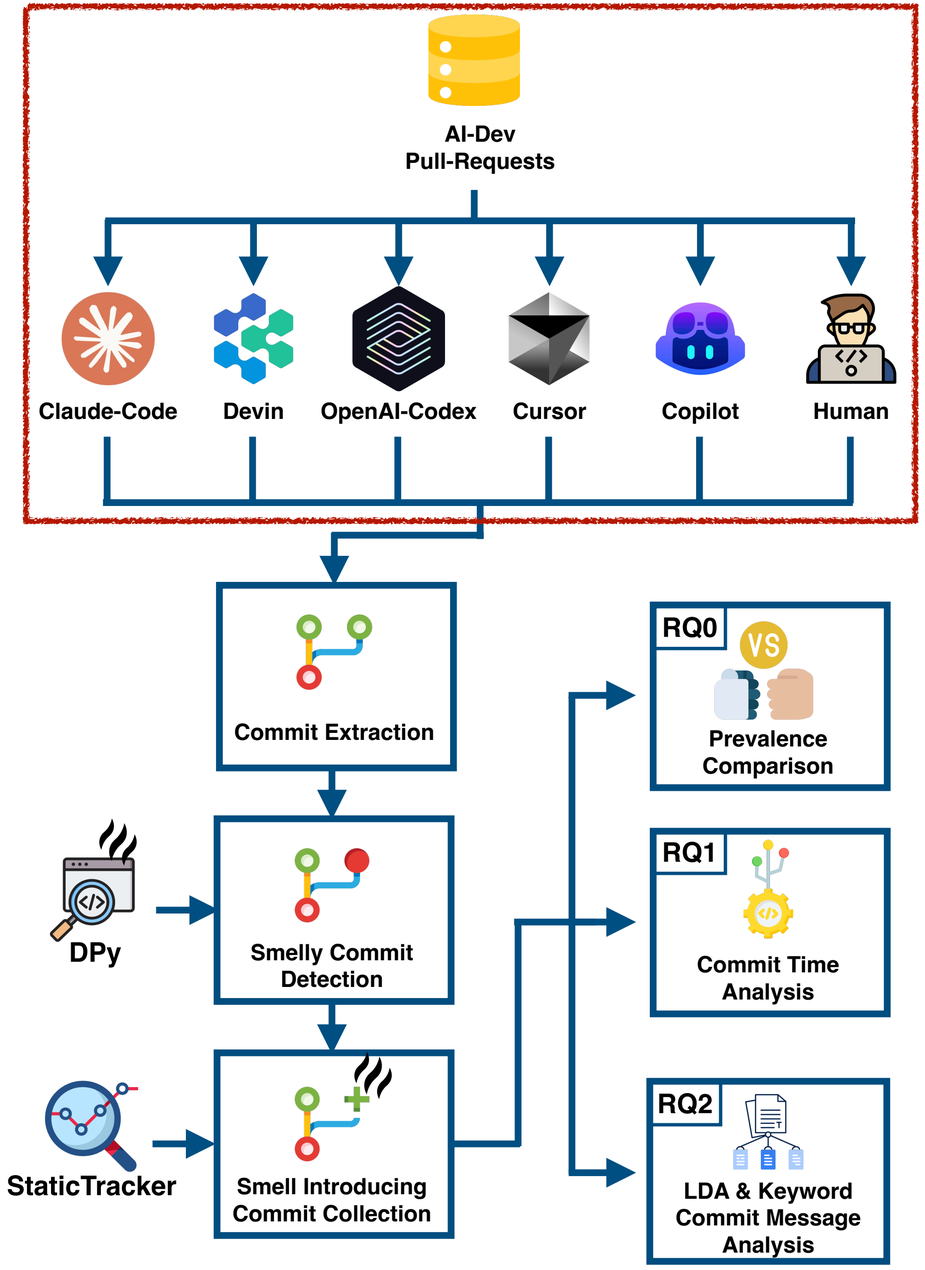
How do LLMs behave from the perspective of code quality?

Functional correctness is not the only quality dimension.

What's missing from "tests pass":

- Readability for reviewers
- Maintainability and changeability
- Observability and testability

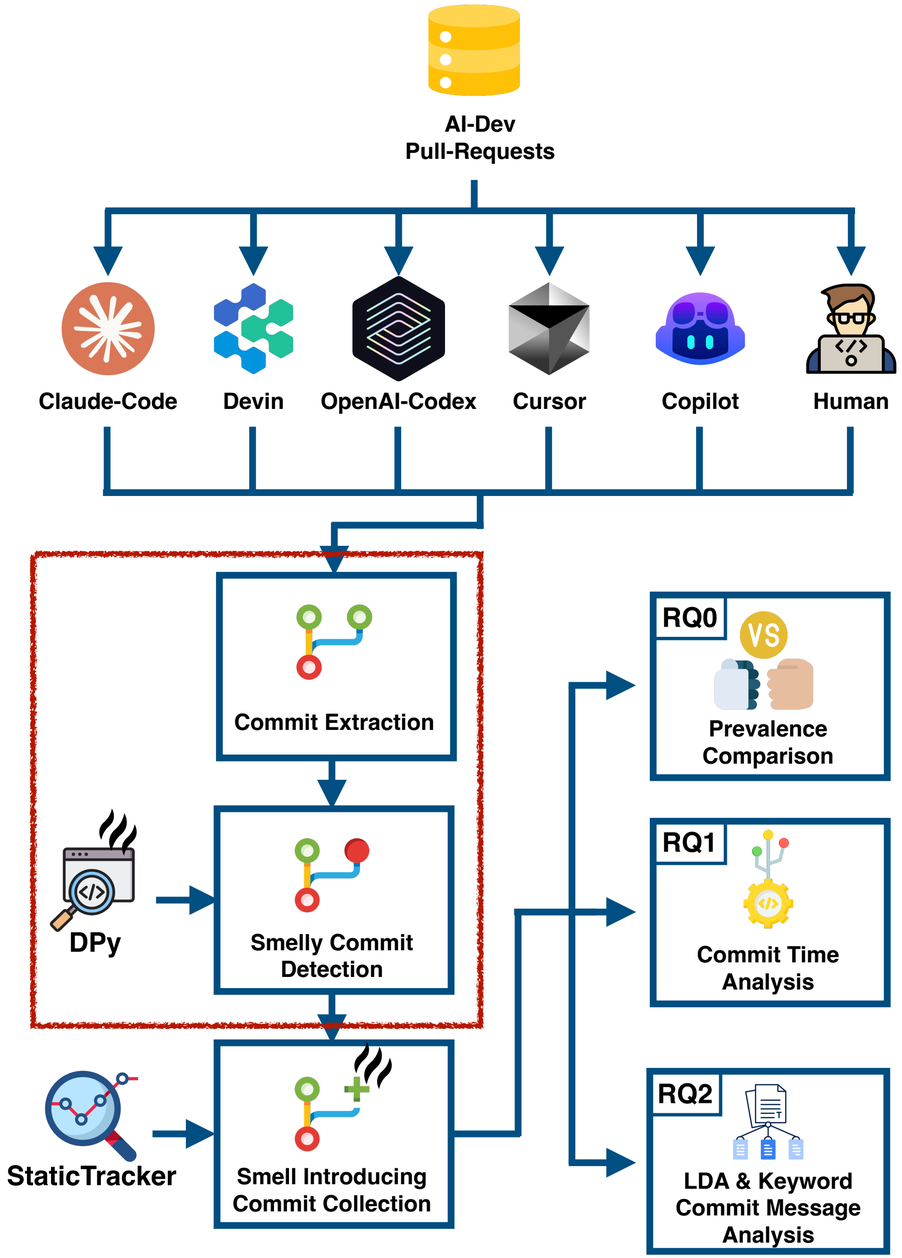
To answer this question, we looked at **the smells introduced by agents in software repositories.**



Agent-authored commit collection

We collected agent-authored and human-authored Python commits from 434 open-source repositories on Github.

Author	Proj.	PR	Comm.	Avg LOC
Codex	201	2417	2449	+56.2 / -19.5
Devin	44	1819	1849	+76.5 / -21.3
Copilot	99	535	542	+4182.2 / -3961.2
Cursor	33	328	329	+138.8 / -49.1
Claude	17	152	172	+316.2 / -82.0
Human	154	674	2128	+189.4 / -103.5
Total	**434**	**5924**	**7469**	**+406.1 / -332.0**



Code smell detection

We extracted all the commits to evaluate the presence of code smells in the changed files, through DPy.

Code smells:

10 types of **design smells**

11 types of **implementation smells**

Examples of Code Smells

Implementation Smells

Long Statement very long line/expression

Magic Number hardcoded number without meaning

Complex Method too many branches/responsibilities

Empty Catch Block exception swallowed

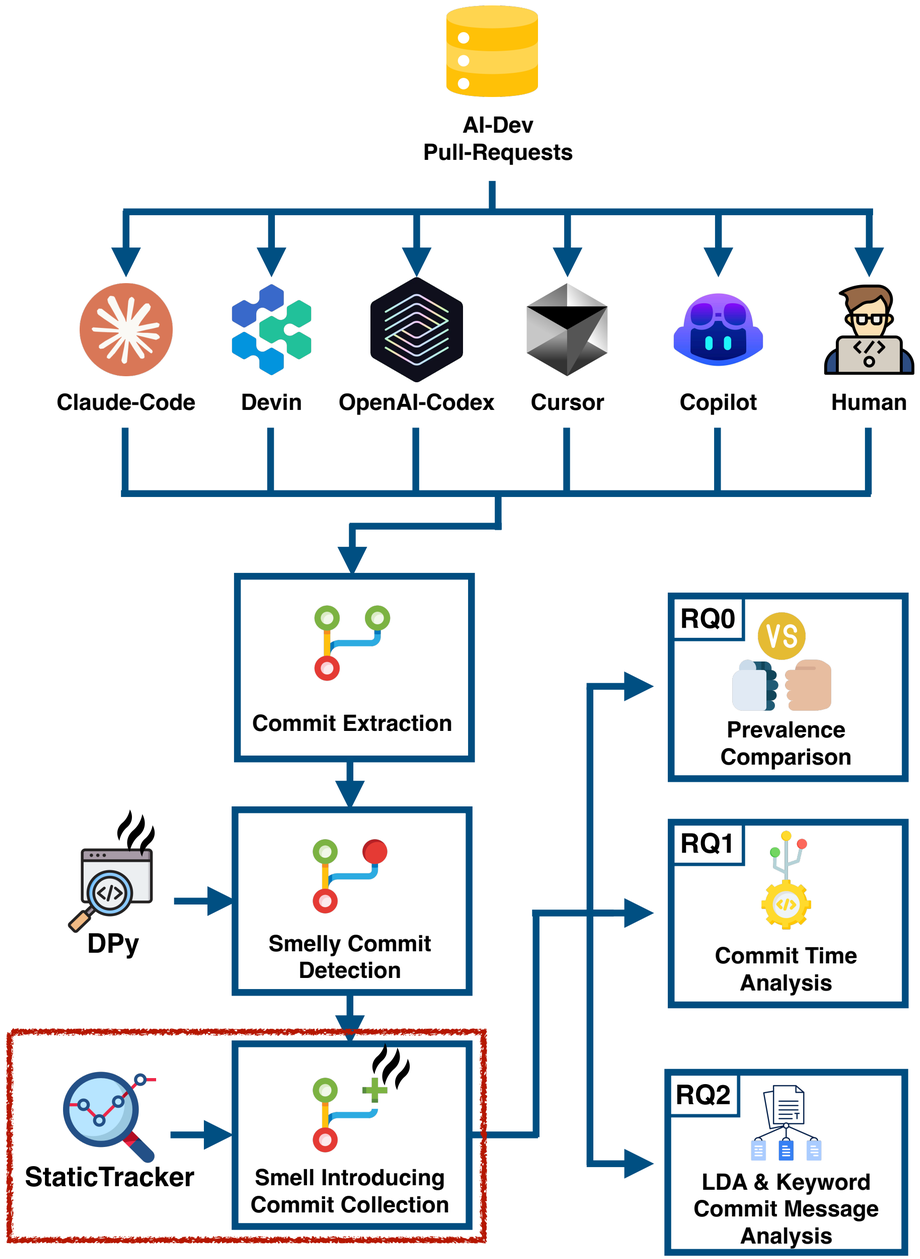
Design Smells

Deficient Encapsulation internals exposed

Feature Envy method uses another class's data heavily

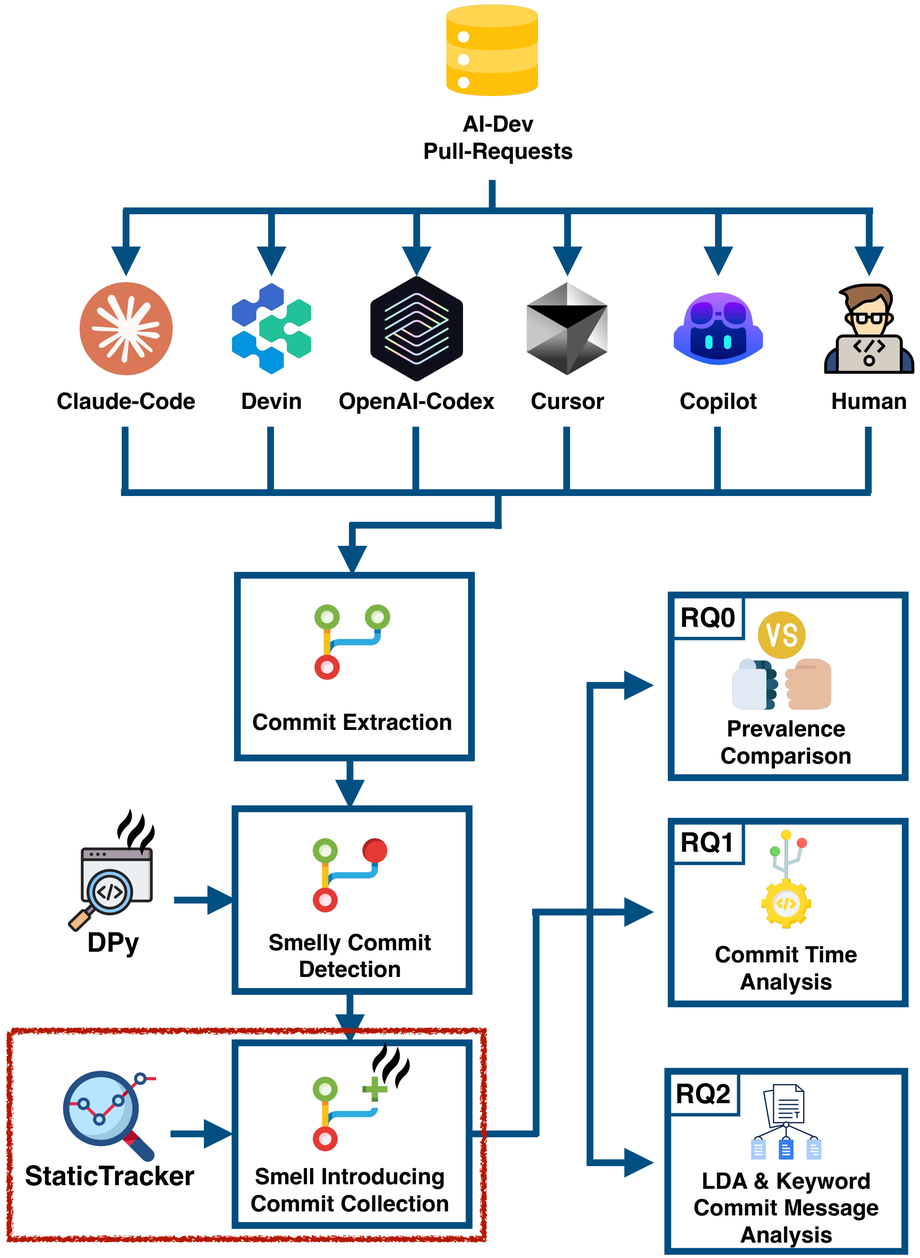
Multifaceted Abstraction class/module does too much

Broken Hierarchy class/subclass do not share conceptually



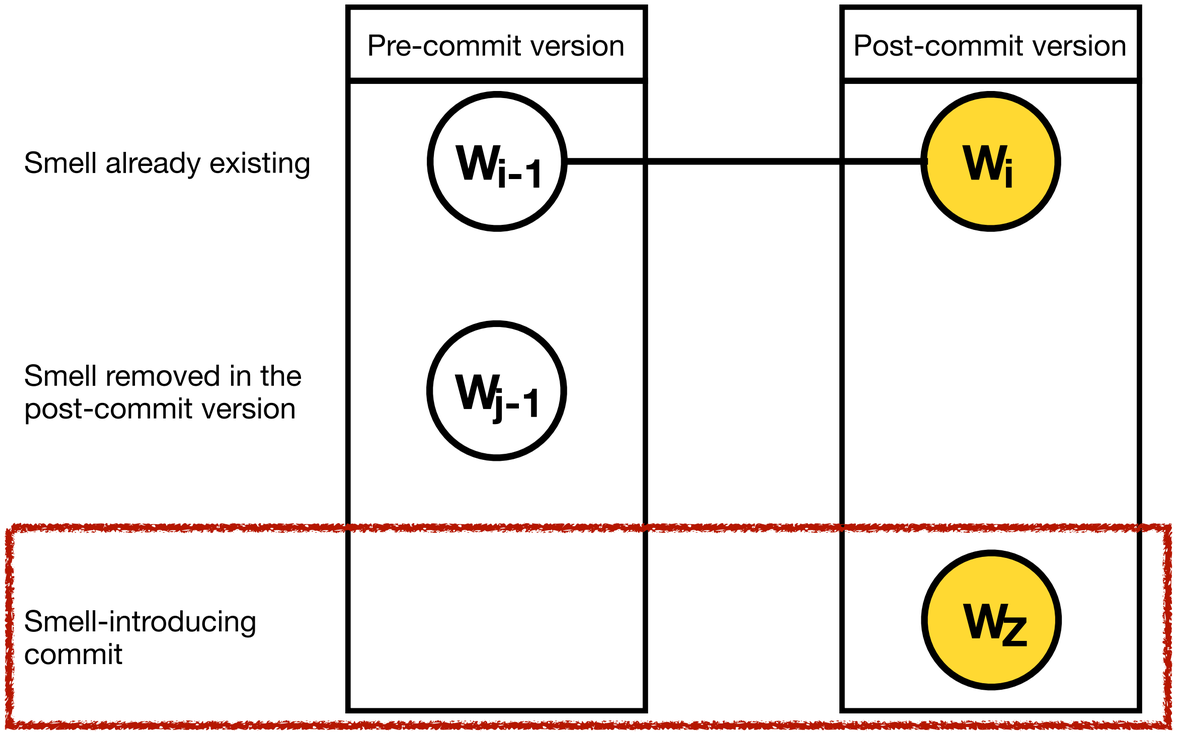
Smell-introducing commit detection

We collected the set of commits that introduced a specific instance of a smell (i.e., **smell-introducing commits**).

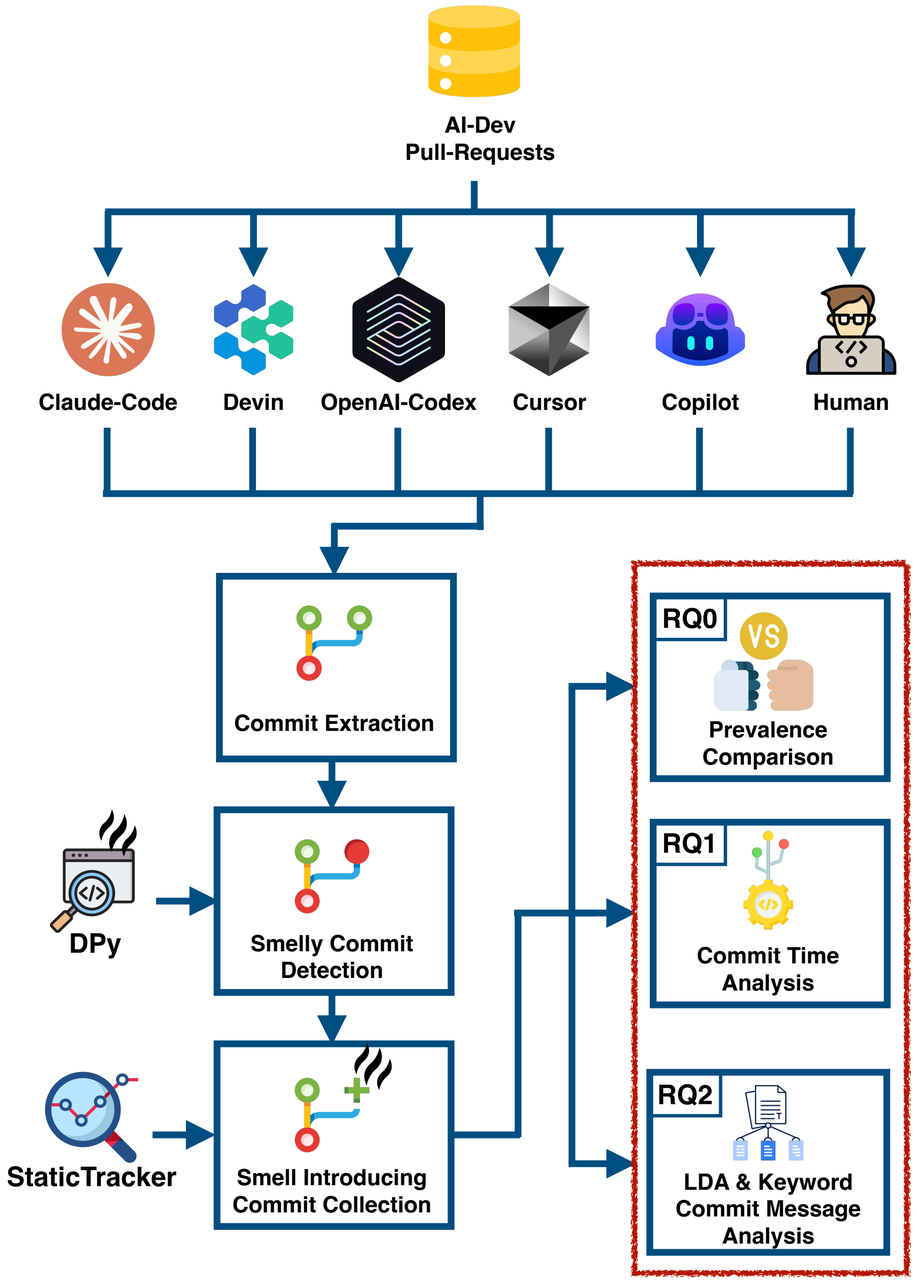


Smell-introducing commit detection

We reused the StaticTracker algorithm, which computes diffs for warnings based on **location, snippet and exact similarity** and **Hungarian matching** [5].



[5] J. Li and J. Yang, "StaticTracker: A Diff Tool for Static Code Warnings," 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME).



Data analysis

RQ0: How do human developers and LLM-based agents compare on the introduction of code smells?

RQ1: When do agents introduce code smells?

RQ2: Why do agents introduce code smells in software systems?

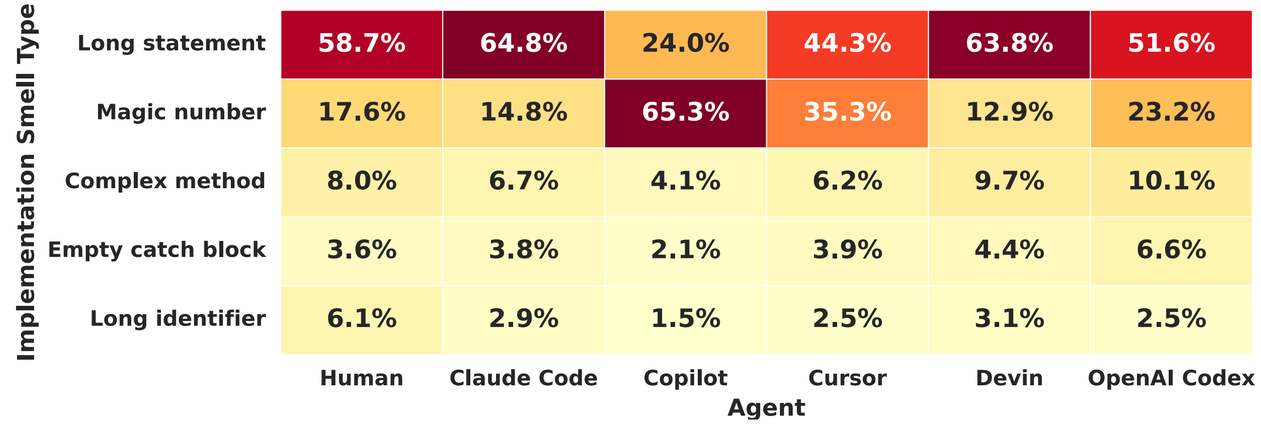
RQ0: How do human developers and LLM-based agents compare on the introduction of code smells?

Table 3: Ratio of Smell-Introducing Commits by Agent.

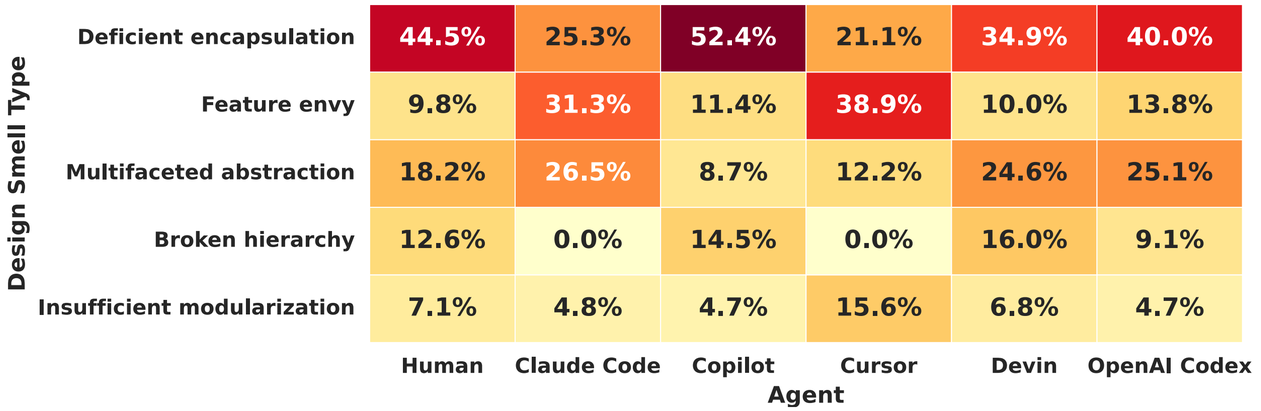
Agent	Any Smells (%)	Impl. Smells (%)	Design Smells (%)
Human	761 (37.5%)	730 (36.0%)	200 (9.9%)
Claude Code	**75 (49.3%)**	**72 (47.4%)**	**30 (19.7%)**
Copilot	214 (40.0%)	209 (39.1%)	54 (10.1%)
Cursor	150 (45.7%)	148 (45.1%)	34 (10.4%)
Devin	647 (35.6%)	628 (34.5%)	119 (6.5%)
OpenAI_Codex	947 (39.2%)	904 (37.4%)	165 (6.8%)

Agents introduce smells, **but not all agents behave the same.**

RQ0: How do humans and LLM-based agents compare on the introduction of code smells?



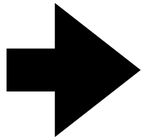
Most common implementation smells are **Long Statement, Magic Number, Complex Method**.



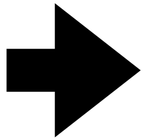
While we notice variability in design smells introduced by LLMs.

RQ0: How do humans and LLM-based agents compare on the introduction of code smells?

While all the agents introduce a significant number of smells, they differ on the type of smells that they introduce.

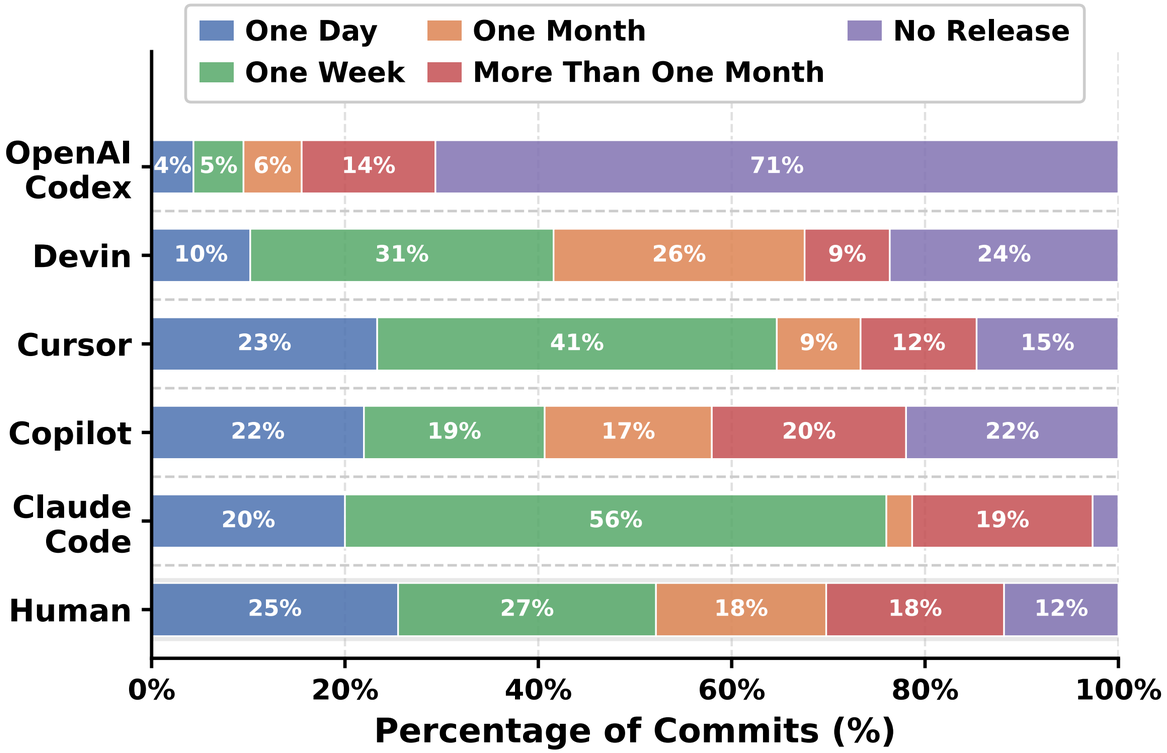


Never trust the LLM. Smelly code is a **warning not to let your guard down**.



Choose agents not only by productivity or benchmark score, but also **by the kind of technical debt they tend to introduce**.

RQ1: When do agents introduce code smells?



A large number of smells is introduced close to release deadlines.

The brief time before a deadline is one of the most important periods **to be careful**.

RQ1: When do agents introduce code smells?

The brief time before a deadline is one of the most important periods **to be careful**.

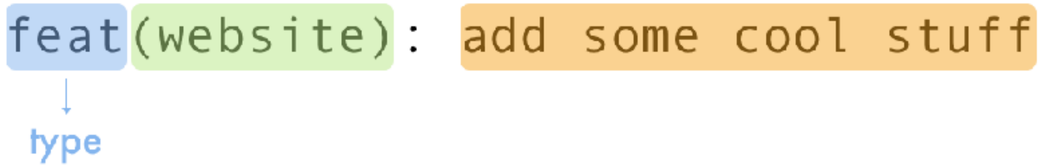
Of course, agents do not feel a sense of “pressure”, but humans may be tempted to let their guard down for the quality of agentic contributions.

RQ2: Why do agents introduce code smells?

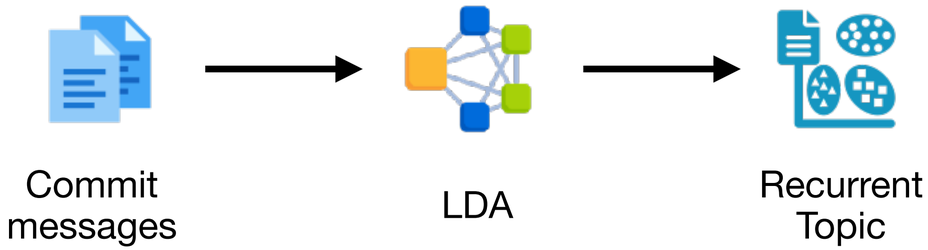
To answer this question, we looked at the activities registered by agents and developers in their commit messages.

We performed two analyses:

Keyword Matching: Structured pattern matching based on keywords of GitHub Commit Conventions (e.g., fix, refactor, feat).

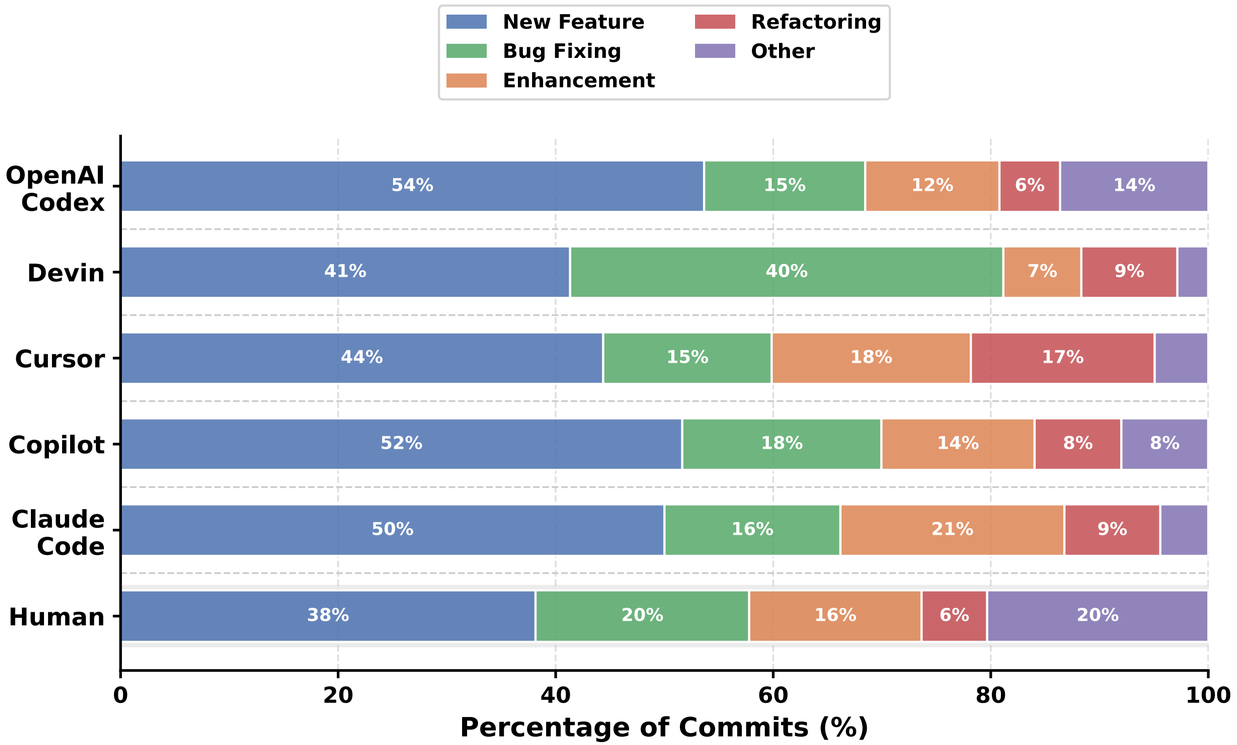


Latent Dirichlet Allocation (LDA): Clustering algorithm that looks at the semantic distance of words. Good for topic modelling problems.



RQ2: Why do agents introduce code smells?

Keyword Matching:



Code smells are mostly introduced when a new feature is checked into the codebase.

To conclude, just like humans may introduce code smells, LLMs may do so too, in ways similar to humans, but with:

Amplified effect. As evidenced by their measured prevalence.

Silent effect. While developers have cognition of their own code, they may miss this for automatically-generated code. **This effect is called cognitive debt.**

Practical suggestions

1. **Don’t Trust LLMs blindly.** LLM-generated code requires deeper inspection than human-written code to ensure maintainability.

2. **When using LLMs, choose your LLM wisely.** The model selection should not be driven by a (single) benchmark score, but should consider all the pros and cons.

3. **React on LLM cons.** Familiarise yourself with LLMs and understand their typical deficiencies. Over time, you’ll know what to look for first.

The AI PR Scenario

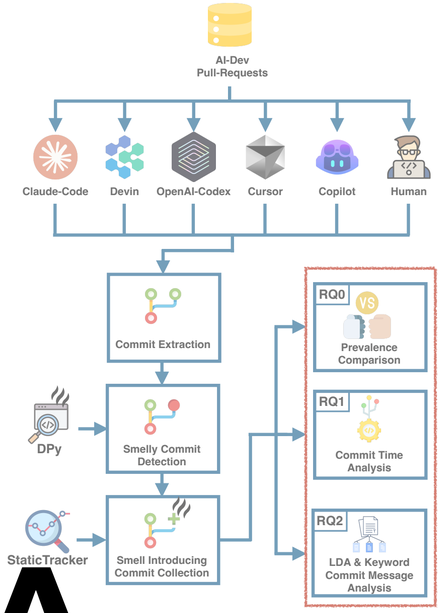
"An AI coding agent opens a pull request. The code compiles. Tests pass. The feature works. Should we merge it?"

Functional correctness is not the only quality dimension.

What's missing from "tests pass":

- Readability for reviewers
- Maintainability and changeability
- Debuggability/observability

How do LLMs behave from the perspective of code quality?



Data analysis

RQ0: How do human developers and LLM-based agents compare on the introduction of code smells?

RQ1: When do agents introduce code smells?

RQ2: Why do agents introduce code smells in software systems?

Q&A

RQ0: How do human developers and LLM-based agents compare on the introduction of code smells?

Implementation Smell Type	Human	Claude Code	Copilot	Cursor	Devin	OpenAI Codex
Long statement	58.7%	64.8%	24.0%	44.3%	63.8%	51.6%
Magic number	17.6%	14.8%	65.3%	35.3%	12.9%	23.2%
Complex method	8.0%	6.7%	4.1%	6.2%	9.7%	10.1%
Empty catch block	3.6%	3.8%	2.1%	3.9%	4.4%	6.6%
Long identifier	6.1%	2.9%	1.5%	2.5%	3.1%	2.5%

Agent

Most common implementation smells are **Long Statement, Magic Number, Complex Method**.

Design Smell Type	Human	Claude Code	Copilot	Cursor	Devin	OpenAI Codex
Deficient encapsulation	44.5%	25.3%	52.4%	21.1%	34.9%	40.0%
Feature envy	9.8%	31.3%	11.4%	38.9%	10.0%	13.8%
Multifaceted abstraction	18.2%	26.5%	8.7%	12.2%	24.6%	25.1%
Broken hierarchy	12.6%	0.0%	14.5%	0.0%	16.0%	9.1%
Insufficient modularization	7.1%	4.8%	4.7%	15.6%	6.8%	4.7%

Agent

While we notice variability in design smells introduced by LLMs.

Practical Suggestions

1. **Don't Trust LLMs blindly.** LLM-generated code requires deeper inspection than human-written code to ensure maintainability.
2. **When using LLMs, choose your LLM wisely.** The model selection should not be driven by benchmark scores, but should consider all the pros and cons.
3. **React on LLM cons.** Familiarise yourself with LLMs and understand their typical deficiencies. Over time, you'll know what to look for first.