

# CEN 5016: Software Engineering

Spring 2026

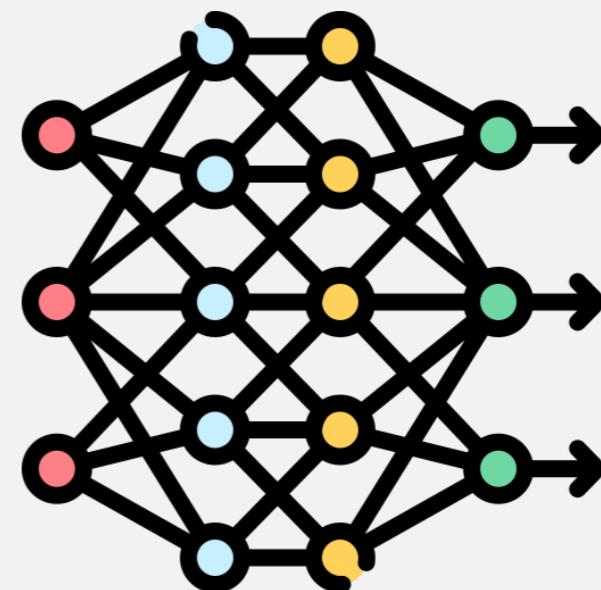


University of  
Central Florida

---

Dr. Kevin Moran

## Week 6 - Class I: A Software Engineer's Guide to LLMs





- *SDE Project Part 1*
  - Due Tomorrow!
  - Two parts:
    - Team Contract
    - Initial Project Backlog
- *Lecture Recordings*
  - Up to Date on Course Webpage

# Software QA: Static & Dynamic Analysis



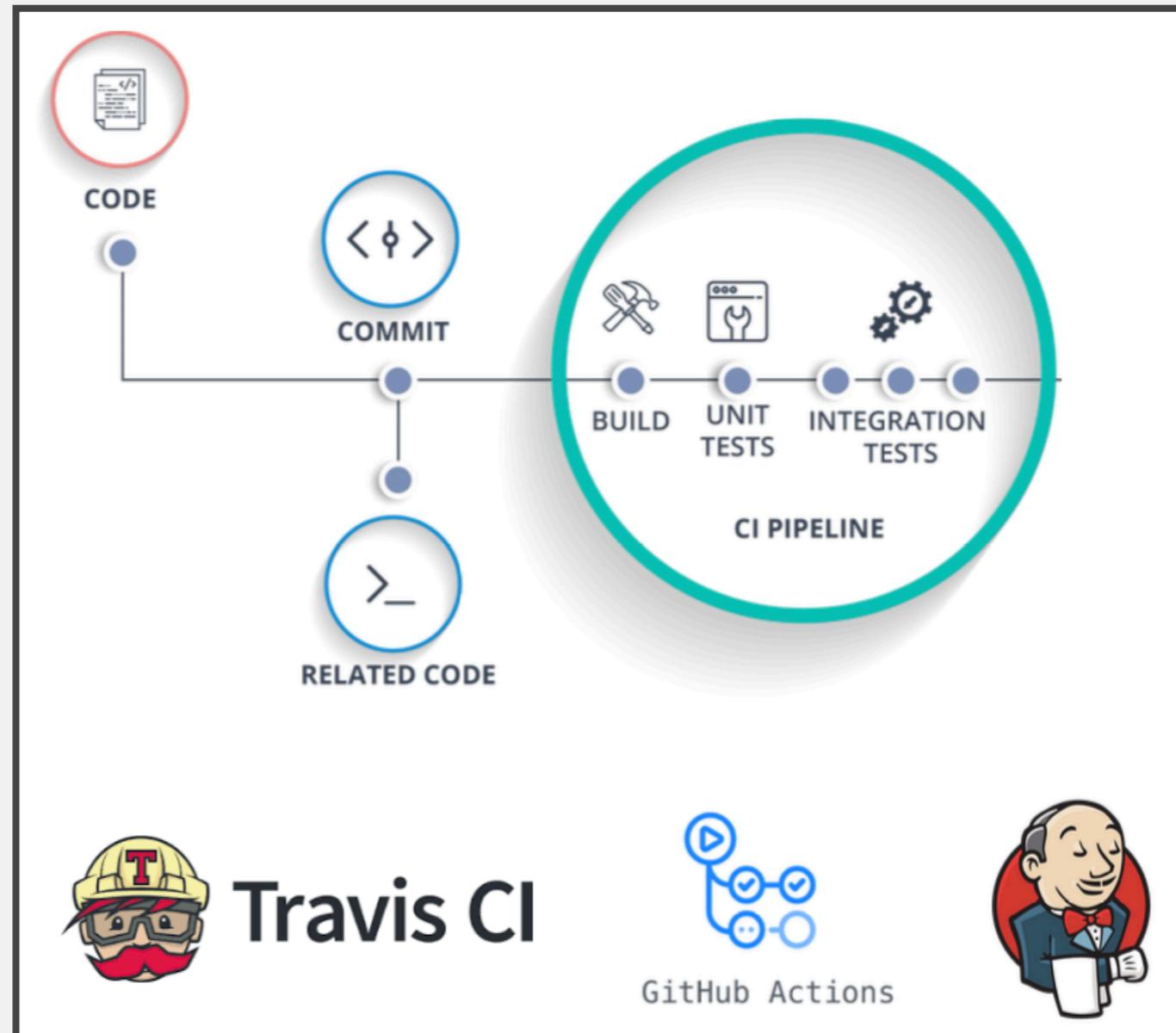
# Static Analysis



# Tools for Static Analysis



# Static Analysis is a Key Part of CI



# Static Analysis used to be Purely Academic...



**GitHub acquires code analysis tool Semmle**

Frederic Lardinois @frederic 1:30 pm EDT • September 18, 2019



comment

**Marketplace** Search results

Types

Apps **Actions**

Categories

- API management
- Chat
- Code quality
- Code review
- Continuous integration
- Dependency management
- Deployment
- IDEs
- Learning
- Localization
- Mobile
- Monitoring
- Project management
- Publishing

Search for apps and actions

Apps

Build on your workflow with apps that integrate with GitHub.

306 results filtered by Apps

App	Description
Zube	Agile project management that lets the entire team work with developers on GitHub
WhiteSource Bolt	Detect open source vulnerabilities in real time with suggested fixes for quick remediation
Crowdin	Agile localization for your projects
BackHub	Reliable GitHub repository backup, set up in minutes
GitLocalize	Continuous Localization for GitHub projects
Codacy	Automated code reviews to help developers ship better software, faster
Code Climate	Automated code review for technical debt and test coverage
Semaphore	Test and deploy at the push of a button
Flaplastic	Manage flaky unit tests. Click a checkbox to instantly disable any test on all branches. Works with your current test suite.
DeepScan	Advanced static analysis for automatically finding runtime errors in JavaScript code
Deptu	Automated dependency updates done right

**GitHub**

**News**

**Snyk Secures \$150M, Snags \$1B Valuation**

Sydney Sawaya | Associate Editor

January 21, 2020 1:12 PM

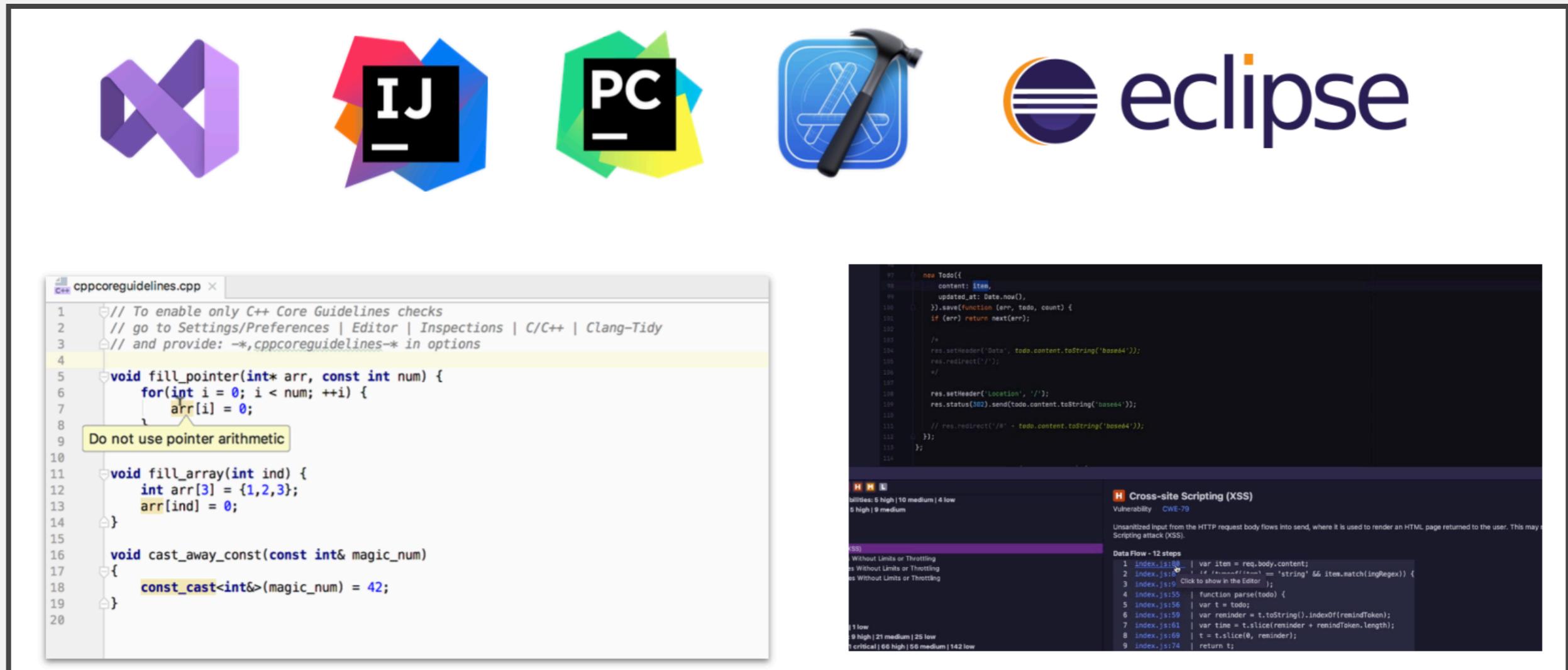
Share this article:





Snyk, a developer-focused **security** startup that identifies vulnerabilities in **open source** applications, announced a \$150 million Series C funding round today. This brings the company's total investment to \$250 million alongside **reports** that put the company's valuation at more than \$1 billion.

# Static Analysis is Also Integrated into IDEs



# What Makes a Good Static Analysis Tool?

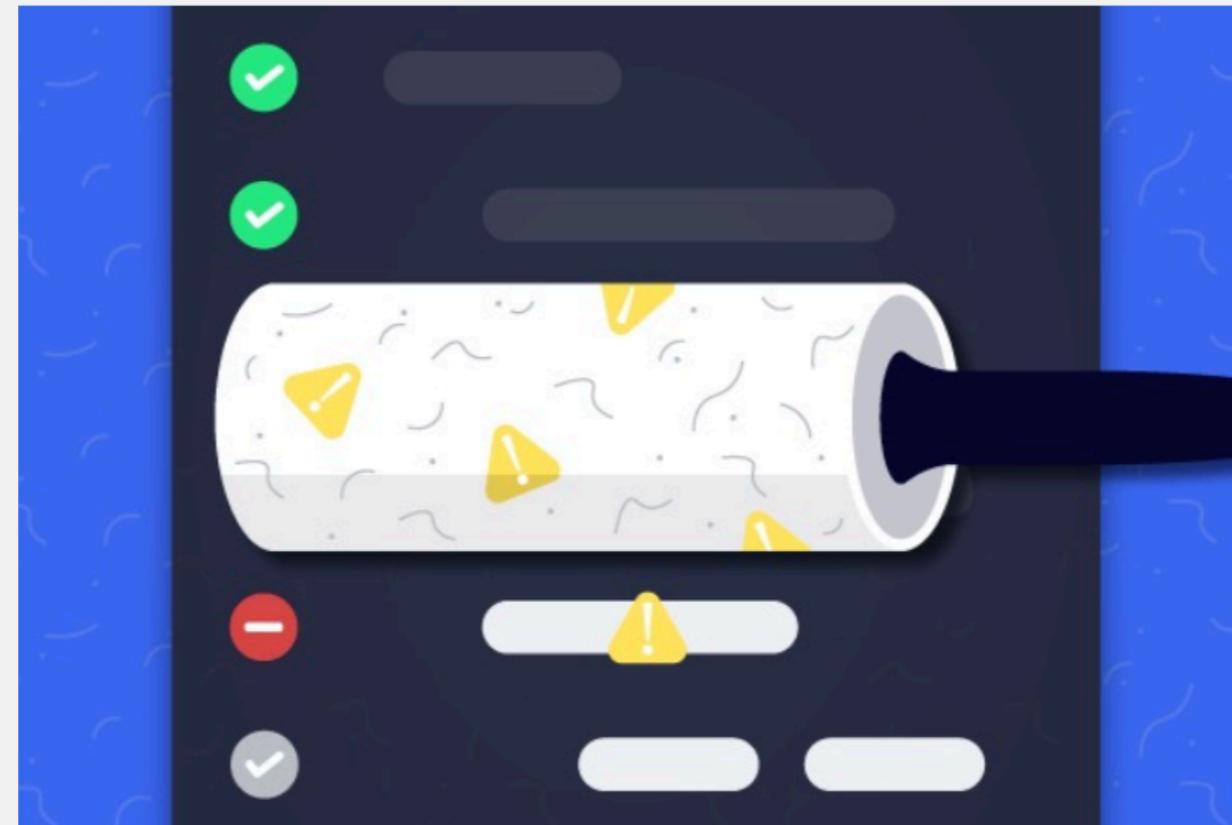


- Static analysis should be fast
  - Don't hold up development velocity
  - This becomes more important as code scales
- Static analysis should report few false positives
  - Otherwise developers will start to ignore warnings and alerts, and quality will decline
- Static analysis should be continuous
  - Should be part of your continuous integration pipeline
  - Diff-based analysis is even better -- don't analyse the entire codebase; just the changes
- Static analysis should be informative
  - Messages that help the developer to quickly locate and address the issue
  - Ideally, it should suggest or automatically apply fixes

# (I) Linters



- Cheap, fast, and lightweight static source analysis



# Use Linters to Enforce Style Guidelines



- Don't rely on manual inspection during code review!

Don't rely on manual inspection during code review!



# Linters Use Very “Shallow” Static Analysis



- Ensure proper indentation
- Naming convention
- Line sizes
- Class nesting
- Documenting public functions
- Parenthesis around expressions
- What else?

# Use Linters to Improve Maintainability



- Why? We spend more time reading code than writing it.
  - Various estimates of the exact %, some as high as 80%
  - Code ownership is usually shared
  - The original owner of some code may move on
  - Code conventions make it easier for other developers to quickly understand your code

# UseStyle Guidelines to Facilitate Communication



The image shows two side-by-side screenshots of style guides. On the left is the Python PEP 8 -- Style Guide for Python Code page, which includes a sidebar with tweets from the Python Software Foundation (@ThePSF) and a footer for The PSF. On the right is the Style Guidelines page for Rust, which includes sections on Guideline statuses, Guideline stabilization, and What's in this document.

**PEP 8 -- Style Guide for Python Code**

**Style Guidelines**

This document collects the emerging principles, conventions, abstractions, and best practices for writing Rust code.

Since Rust is evolving at a rapid pace, these guidelines are preliminary. The hope is that writing them down explicitly will help drive discussion, consensus and adoption.

Whenever feasible, guidelines provide specific examples from Rust's standard libraries.

**Guideline statuses**

Every guideline has a status:

- [FIXME]: Marks places where there is more work to be done. In some cases, that just means going through the RFC process.
- [FIXME #NNNN]: Like [FIXME], but links to the issue tracker.
- [RFC #NNNN]: Marks accepted guidelines, linking to the rust-lang RFC establishing them.

**Guideline stabilization**

One purpose of these guidelines is to reach decisions on a number of cross-cutting API and stylistic choices. Discussion and development of the guidelines will happen primarily on <http://discuss.rust-lang.org>, using the Guidelines category. Discussion can also occur on the [guidelines issue tracker](#).

Guidelines that are under development or discussion will be marked with the status [FIXME], with a link to the issue tracker when appropriate.

Once a concrete guideline is ready to be proposed, it should be filed as an [FIXME: needs RFC]. If the RFC is accepted, the official guidelines will be updated to match, and will include the tag [RFC #NNNN] linking to the RFC document.

**What's in this document**

This document is broken into four parts:

- **Style** provides a set of rules governing naming conventions, whitespace, and other stylistic issues.
- **Guidelines by Rust feature** places the focus on each of Rust's features, starting from expressions and working the way out toward crates, dispensing guidelines relevant to each.
- **Topical guidelines and patterns**. The rest of the document proceeds by cross-cutting topic, starting with **Ownership and resources**.
- **APIs for a changing Rust** discusses the forward-compatibility hazards, especially those that interact with the pre-1.0 library stabilization process.

- Guidelines are inherently opinionated, but consistency is the important point. Agree to a set of conventions and stick to them.

# Take Home Message: Style is an Easy Way to Improve Readability!



- Everyone has their own opinion (e.g., tabs vs. spaces)
- Agree to a convention and stick to it
  - Use continuous integration to enforce it
- Use automated tools to fix issues in existing code

# (2) - Pattern-based Static Analysis Tools



- Bad Practice
- Correctness
- Performance
- Internationalization
- Malicious Code
- Multithreaded Correctness
- Security
- Dodgy Code

FindBugs Bug Descriptions	
This document lists the standard bug patterns reported by <a href="#">FindBugs</a> version 3.0.1.	
Description	Category
BC: Equals method should not assume anything about the type of its argument	Bad practice
BIT: Check for sign of bitwise operation	Bad practice
CN: Class implements Cloneable but does not define or use clone method	Bad practice
CN: clone method does not call super.clone()	Bad practice
CN: Class defines clone() but doesn't implement Cloneable	Bad practice
CNT: Rough value of known constant found	Bad practice
Co: Abstract class defines covariant compareTo() method	Bad practice
Co: compareTo()/compare() incorrectly handles float or double value	Bad practice
Co: compareTo()/compare() returns Integer.MIN_VALUE	Bad practice
Co: Covariant compareTo() method defined	Bad practice
DE: Method might drop exception	Bad practice
DE: Method might ignore exception	Bad practice
DMI: Adding elements of an entry set may fail due to reuse of Entry objects	Bad practice
DMI: Random object created and used only once	Bad practice
DMI: Don't use removeAll to clear a collection	Bad practice
Dm: Method invokes System.exit(...)	Bad practice
Dm: Method invokes dangerous method runFinalizersOnExit	Bad practice
ES: Comparison of String parameter using == or !=	Bad practice
ES: Comparison of String objects using == or !=	Bad practice
Eq: Abstract class defines covariant equals() method	Bad practice
Eq: Equals checks for incompatible operand	Bad practice
Eq: Class defines compareTo... and uses Object.equals()	Bad practice
Eq: equals method fails for subtypes	Bad practice
Eq: Covariant equals() method defined	Bad practice
FI: Empty finalizer should be deleted	Bad practice
FI: Explicit invocation of finalizer	Bad practice
FI: Finalizer nulls fields	Bad practice
FI: Finalizer only nulls fields	Bad practice
FI: Finalizer does not call superclass finalizer	Bad practice
FI: Finalizer nullifies superclass finalizer	Bad practice
FI: Finalizer does nothing but call superclass finalizer	Bad practice
FS: Format string should use %n rather than %n	Bad practice
GC: Unchecked type in generic call	Bad practice
HE: Class defines equals() but not hashCode()	Bad practice
HE: Class defines equals() and uses Object.hashCode()	Bad practice
HE: Class defines hashCode() but not equals()	Bad practice
HE: Class defines hashCode() and uses Object.equals()	Bad practice
HE: Class inherits equals() and uses Object.hashCode()	Bad practice
IC: Superclass uses subclass during initialization	Bad practice
IMSE: Dubious catching of IllegalMonitorStateException	Bad practice
ISC: Needless instantiation of class that only supplies static methods	Bad practice
It: Iterator next() method can't throw NoSuchElementException	Bad practice
JZEE: Store of non serializable object into HttpSession	Bad practice
JCIP: Fields of immutable classes should be final	Bad practice
ME: Public enum method unconditionally sets its field	Bad practice

# SpotBugs can be Extended with Plugins



The image shows two screenshots. On the left is a screenshot of the Find Security Bugs plugin within an IDE (Android Studio). The main window shows Java code in a file named `DoTransfer.java`. A sidebar on the left displays the project structure and a 'FindBugs:IDEA FindBugs Analysis Results' panel. This panel lists 'InsecureBankv2' with 14 bug items in 59 classes, categorized under 'Security' (14 items). One item is highlighted: 'External File Access (Android)'. A tooltip for this item provides a detailed description: 'The application write data to external storage (potentially SD card). There are multiple security implication to this action. First, file store on SD card will be accessible to the application having the READ\_EXTERNAL\_STORAGE permission. Also, if the data persisted contains confidential information about the user, encryption would be needed.' Below the description is a 'Code at risk:' section with code snippets and a 'Better alternative:' section with code snippets. On the right is a screenshot of the 'Find Security Bugs' GitHub page. The page title is 'Find Security Bugs' and it describes the plugin as 'The SpotBugs plugin for security audits of Java web applications'. It features sections for 'Features' (138 bug patterns, OWASP TOP 10 and CWE coverage), 'Screenshots' (Eclipse, IntelliJ / Android Studio, SonarQube), and 'Downloads' (version 1.11.0). The GitHub page also shows project statistics: 1,504 stars, 354 forks, and a link to the GitHub repository.



- The analysis must produce zero false positives
  - Otherwise developers won't be able to build the code!
- The analysis needs to be really fast
  - Ideally < 100 ms
  - If it takes longer, developers will become irritated and lose productivity
- You can't just "turn on" a particular check
  - Every instance where that check fails will prevent existing code from
  - There could be thousands of violations for a single check across large codebases

### (3) -Use Type Annotations to Detect Common Errors



- Uses a conservative analysis to prove the absence of certain defects
  - Null pointer errors, uninitialized fields, certain liveness issues, information leaks, SQL injections, bad regular expressions, incorrect physical units, bad format strings, ...
- C.f. SpotBugs which makes no safety guarantees
- Assuming that code is annotated and those annotations are correct
- Uses annotations to enhance type system
- Example: Java Checker Framework or MyPy



### (3) -Use Type Annotations to Detect Common Errors



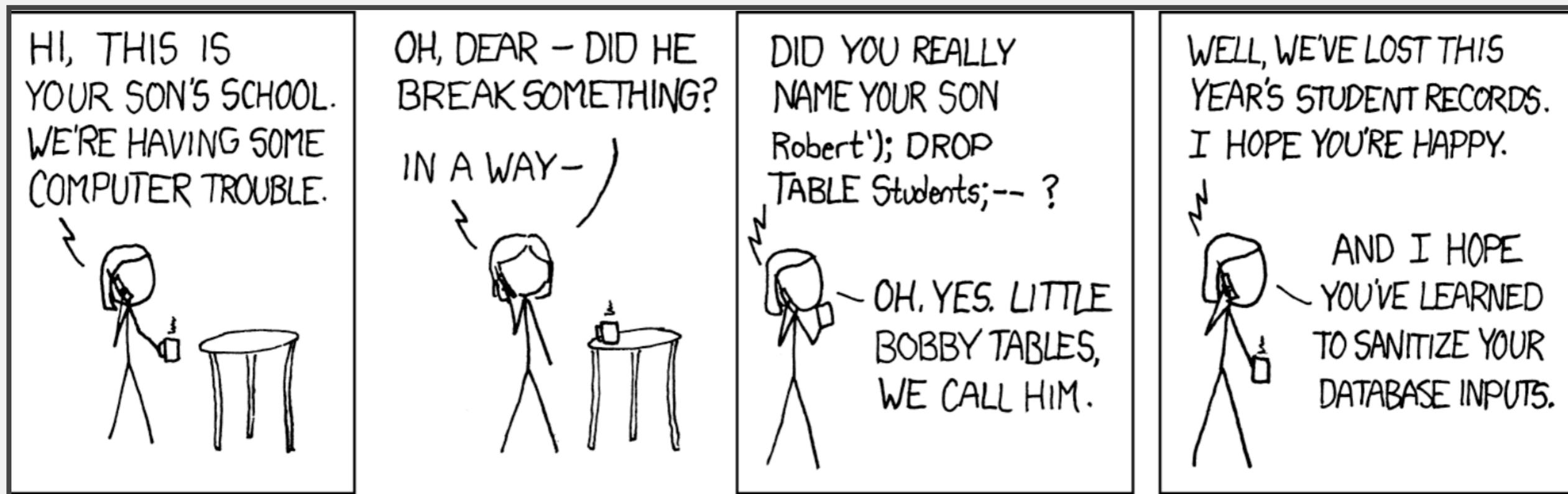
- Uses a conservative analysis to prove the absence of certain defects
  - Null pointer errors, uninitialized fields, certain liveness issues, information leaks, SQL injections, bad regular expressions, incorrect physical units, bad format strings, ...
- C.f. SpotBugs which makes no safety guarantees
- Assuming that code is annotated and those annotations are correct
- Uses annotations to enhance type system
- Example: Java Checker Framework or MyPy





- Tracks flow of sensitive information through the program
- Tainted inputs come from arbitrary, possibly malicious sources
  - User inputs, unvalidated data
- Using tainted inputs may have dangerous consequences
  - Program crash, data corruption, leak private data, etc.
- We need to check that inputs are sanitized before reaching sensitive locations

# Classic Example: SQL Injection



# Classic Example: SQL Injection



```
void processRequest() {  
    String input = getUserInput();  
    String query = "SELECT ... " + input;  
    executeQuery(query);  
}
```

# Classic Example: SQL Injection



```
void processRequest() {  
    String input = getUserInput();  
    String query = "SELECT ... " + input;  
    executeQuery(query);  
}
```

Tainted input arrives from untrusted source

Tainted output flows to a sensitive sink

# Classic Example: SQL Injection



```
void processRequest() {  
    String input = getUserInput();  
    input = sanitizeInput(input);  
    String query = "SELECT ... " + input;  
    executeQuery(query);  
}
```

Taint is removed by sanitizing data

We can now safely execute query on untainted data

# Unit Catastrophe



**Remember the Mars Climate Orbiter incident from 1999?**

**SIMSCALE** | [Blog](#) [Product](#) [Solutions](#) [Learning](#) [Public Projects](#) [Case Studies](#) [Careers](#) [Pricing](#) [Log In](#) [Sign Up](#)

## When NASA Lost a Spacecraft Due to a Metric Math Mistake



WRITTEN BY [Ajay Harish](#) UPDATED ON [March 10th, 2020](#) APPROX. READING TIME: 11 Minutes

[Blog](#) > [CAE Hub](#) > When NASA Lost a Spacecraft Due to a Metric Math Mistake

[f](#) [in](#) [t](#)

In September of 1999, after almost 10 months of travel to Mars, the Mars Climate Orbiter burned and broke into pieces. On a day when NASA engineers were expecting to celebrate, the ground reality turned out to be completely different, all because someone failed to use the right units, i.e., the metric units! The Scientific American Space Lab made a brief but interesting video on this very topic.

**NASA'S LOST SPACECRAFT**  
**The Metric System and NASA's Mars Climate Orbiter**

The Mars Climate Orbiter, built at a cost of \$125 million, was a 338-kilogram robotic space probe launched by NASA on December 11, 1998 to study the Martian climate, Martian atmosphere, and surface changes. In addition, its function was to act as the communications relay in the Mars Surveyor '98 program for the Mars Polar Lander. The navigation team at the Jet Propulsion Laboratory (JPL) used the metric system of millimeters and meters in its calculations, while

**NASA's Mars Climate Orbiter (cost of \$327 million) was lost because of a discrepancy between use of metric unit Newtons and imperial measure Pound-force.**



- Guarantees that operations are performed on the same kinds and units
- Kinds of annotations
  - @Acceleration, @Angle, @Area, @Current, @Length, @Luminance, @Mass, @Speed, @Substance, @Temperature, @Time
- SI unit annotation
  - @m, @km, @mm, @kg, @mPERs, @mPERs2, @radians, @degrees, @A, ...



- Can only analyze code that is annotated
  - Requires that dependent libraries are also annotated
  - Can be tricky, but not impossible, to retrofit annotations into existing codebases
- Only considers the signature and annotations of methods
  - Doesn't look at the implementation of methods that are being called
- Dynamically generated code
  - Spring Framework
- Can produce false positives!
  - Byproduct of necessary approximations

# Infer: What if we didn't want Annotations



- Focused on memory safety bugs
  - Null pointer dereferences, memory leaks, resource leaks, ...
- Compositional interprocedural reasoning
  - Based on separation logic and bi-abduction
- Scalable and fast
  - Can run incremental analysis on changed code
- Does not require annotations
- Supports multiple languages
  - Java, C, C++, Objective-C
  - Programs are compiled to an intermediate representation





## **NULLPTR\_DEREFERENCE**

Reported as "Nullptr Dereference" by [pulse](#).

Infer reports null dereference bugs in Java, C, C++, and Objective-C when it is possible that the null pointer is dereferenced, leading to a crash.

### **Null dereference in Java**

Many of Infer's reports of potential Null Pointer Exceptions (NPE) come from code of the form

```
p = foo(); // foo() might return null
stuff();
p.goo(); // dereferencing p, potential NPE
```





## Examples

Infer's cost analysis statically estimates the execution cost of a program without running the code. For instance, assume that we had the following program:

```
void loop(ArrayList<Integer> list){  
    for (int i = 0; i <= list.size(); i++){  
    }  
}
```

For this program, Infer statically infers a polynomial (e.g.  $8|list| + 16$ ) for the execution cost of this program by giving each instruction in Infer's intermediate language a symbolic cost (where  $|. |$  refers to the length of a list). Here---overlooking the actual constants---the analysis infers that this program's asymptotic complexity is  $O(|list|)$ , that is loop is linear in the size of its input list. Then, at diff time, if a developer modifies this code to,

# Beware of Inevitable False Positives



Consider using Facebook's "infer" static analysis tool #6968 [New issue](#)

Open richsalz opened this issue on Aug 28, 2018

dot-asn commented on Sep 2, 2018

Contributor

I'm not impressed. Majority, >2/3 of reports are DEAD\_STORE and most common reason is last `*ptr++`. More specifically `++` is viewed problematic because `pointer` is not used anymore. The post-increment is also customarily part of macro, so that in order to address this, one would have to have two macros, one that leaves pointer post-incremented and one that doesn't. It would be excessive and doesn't help readability.

Majority of MEMORY\_LEAK reports is because it fails to recognize for example `EVP_MD_CTX_free` as resource freeing. This is counter-productive, one has to work too hard look for real ones. There seem to be couple in `test/*...` Then there is some hairy stuff in `o_names.c:236`, maybe false positive... Oh! There seem to be real leak in `ssl3_final_finish_mac()`, multiple logical errors...

# The Best QA Strategies use Multiple Tools



## How Many of All Bugs Do We Find? A Study of Static Bug Detectors

Andrew Habib  
andrew.a.habib@gmail.com  
Department of Computer Science  
TU Darmstadt  
Germany

### ABSTRACT

Static bug detectors are becoming increasingly popular and are widely used by professional software developers. While most work on bug detectors focuses on whether they find bugs at all, and on how many false positives they report in addition to legitimate warnings, the inverse question is often neglected: How many of all real-world bugs do static bug detectors find? This paper addresses this question by studying the results of applying three widely used static bug detectors to an extended version of the Defects4J dataset that consists of 15 Java projects with 594 known bugs. To decide which of these bugs the tools detect, we use a novel methodology that combines an automatic analysis of warnings and bugs with a manual validation of each candidate of a detected bug. The results of the study show that: (i) static bug detectors find a non-negligible amount of all bugs, (ii) different tools are mostly complementary to each other, and (iii) current bug detectors miss the large majority of the studied bugs. A detailed analysis of bugs missed by the static detectors shows that some bugs could have been found by variants of the existing detectors, while others are domain-specific problems that do not match any existing bug pattern. These findings help potential users of such tools to assess their utility, motivate and outline directions for future work on static bug detection, and provide a basis for future comparisons of static bug detection with other bug finding techniques, such as manual and automated testing.

Michael Pradel  
michael@binaervarianz.de  
Department of Computer Science  
TU Darmstadt  
Germany

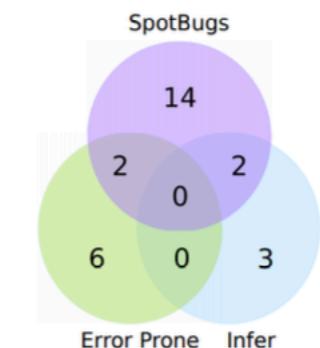
*International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3238147.3238213>*

### 1 INTRODUCTION

Finding software bugs is an important but difficult task. For average industry code, the number of bugs per 1,000 lines of code has been estimated to range between 0.5 and 25 [21]. Even after years of deployment, software still contains unnoticed bugs. For example, studies of the Linux kernel show that the average bug remains in the kernel for a surprisingly long period of 1.5 to 1.8 years [8, 24]. Unfortunately, a single bug can cause serious harm, even if it has been subsisting for a long time without doing so, as evidenced by examples of software bugs that have caused huge economic losses and even killed people [17, 28, 46].

Given the importance of finding software bugs, developers rely on several approaches to reveal programming mistakes. One approach is to identify bugs during the development process, e.g., through pair programming or code review. Another direction is testing, ranging from purely manual testing over semi-automated testing, e.g., via manually written but automatically executed unit tests, to fully automated testing, e.g., with UI-level testing tools. Once the software is deployed, runtime monitoring can reveal so far missed bugs, e.g., collect information about abnormal runtime

Tool	Bugs
Error Prone	8
Infer	5
SpotBugs	18
<i>Total:</i>	<b>31</b>
<i>Total of 27 unique bugs</i>	

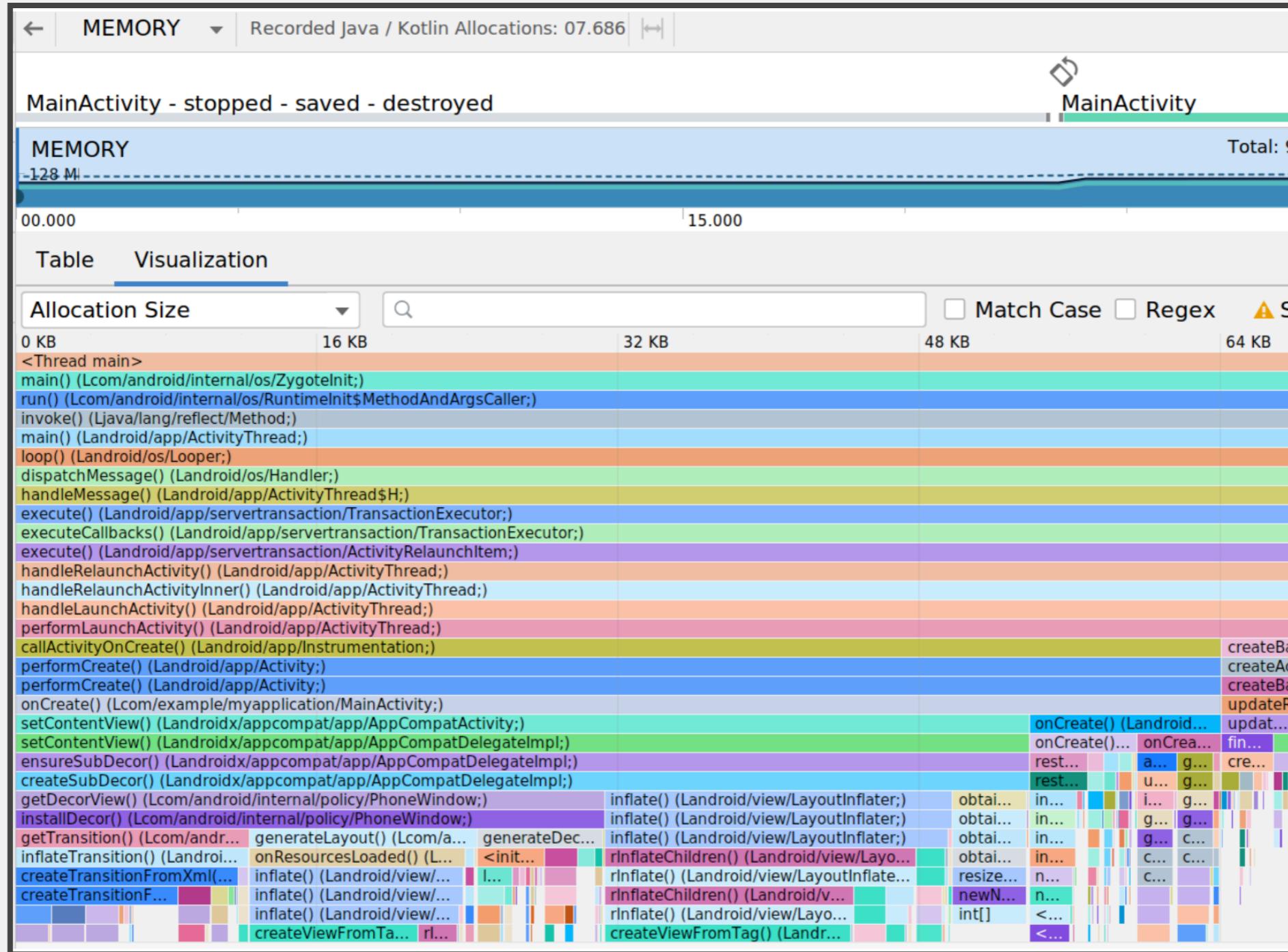


**Figure 4: Total number of bugs found by all three static checkers and their overlap.**

# Dynamic Analysis

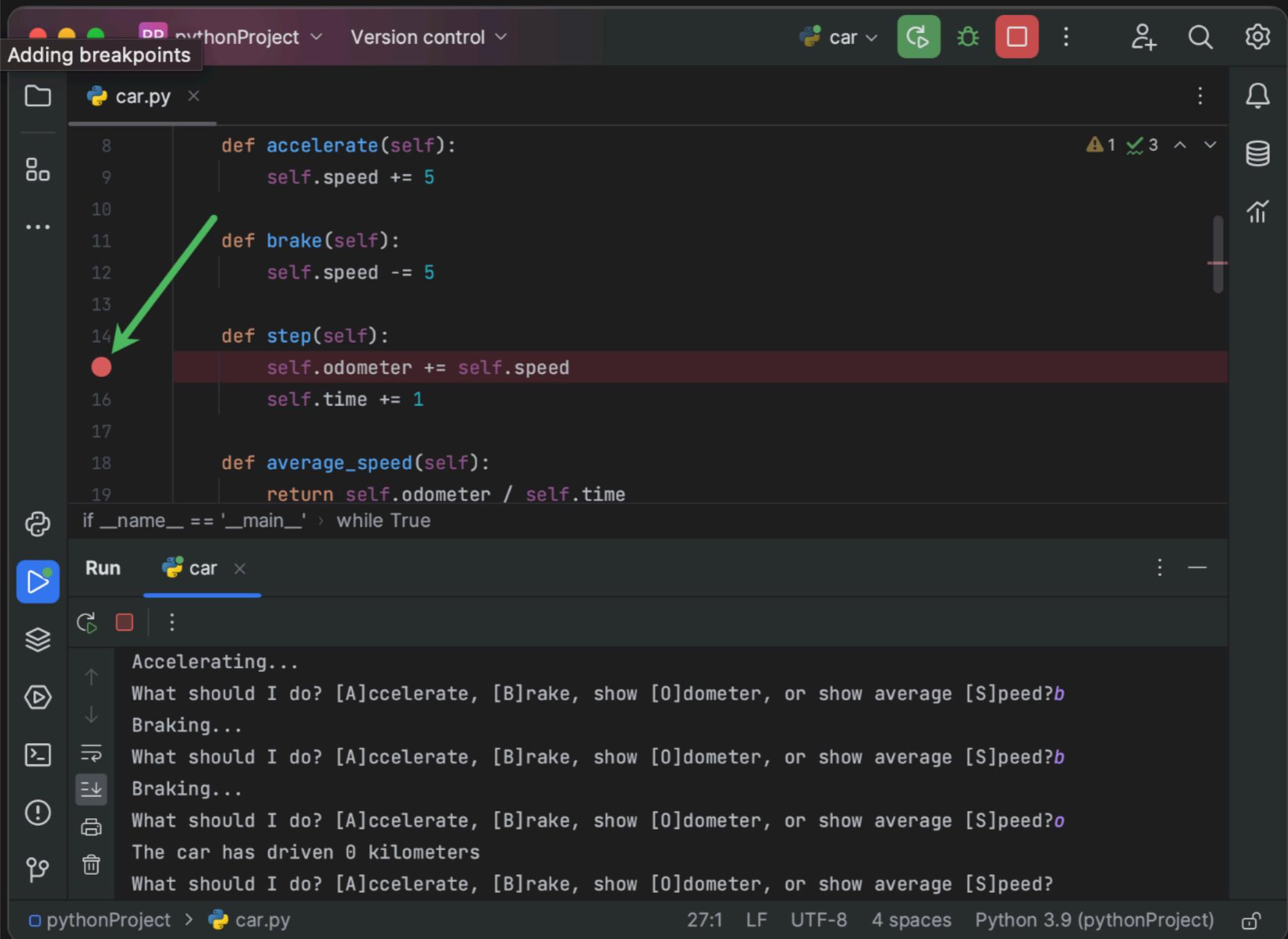


# Android Memory Profiler



<https://developer.android.com/studio/profile/memory-profiler>

# Pycharm Debugger



Adding breakpoints

```
8     def accelerate(self):
9         self.speed += 5
10
11    def brake(self):
12        self.speed -= 5
13
14    def step(self):
15        self.odometer += self.speed
16        self.time += 1
17
18    def average_speed(self):
19        return self.odometer / self.time
if __name__ == '__main__':
    while True
```

Run car

```
Accelerating...
What should I do? [A]ccelerate, [B]rake, show [0]dometer, or show average [S]peed?b
Braking...
What should I do? [A]ccelerate, [B]rake, show [0]dometer, or show average [S]peed?b
Braking...
What should I do? [A]ccelerate, [B]rake, show [0]dometer, or show average [S]peed?o
The car has driven 0 kilometers
What should I do? [A]ccelerate, [B]rake, show [0]dometer, or show average [S]peed?
```

pythonProject > car.py

27:1 LF UTF-8 4 spaces Python 3.9 (pythonProject)

# Valgrind Dynamic Analysis Library



**Valgrind**

Current release: [valgrind-3.23.0](#)

Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

The Valgrind distribution currently includes seven production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and two different heap profilers. It also includes an experimental SimPoint basic block vector generator. It runs on the following platforms: X86/Linux, AMD64/Linux, ARM/Linux, ARM64/Linux, PPC32/Linux, PPC64/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, X86/Solaris, AMD64/Solaris, ARM/Android (2.3.x and later), ARM64/Android, X86/Android (4.0 and later), MIPS32/Android, X86/FreeBSD, AMD64/FreeBSD, ARM64/FreeBSD, X86/Darwin and AMD64/Darwin (Mac OS X 10.12).

Valgrind is [Open Source / Free Software](#), and is freely available under the [GNU General Public License, version 2](#).



- Linters are cheap, fast, but imprecise analysis tools
  - Can be used for purposes other than bug detection (e.g., style)
- Conservative analyzers can demonstrate the absence of particular defects
  - At the cost of false positives due to necessary approximations
  - Inevitable trade-off between false positives and false negatives
- The best QA strategy involves multiple analysis and testing techniques
  - The exact set of tools and techniques depends on context

# A Software Engineer's Guide to LLMs





- Part I: How to Effectively use an modern LLM agent framework for Software Development
- Part II: How to Integrate LLM-powered reasoning into an application.

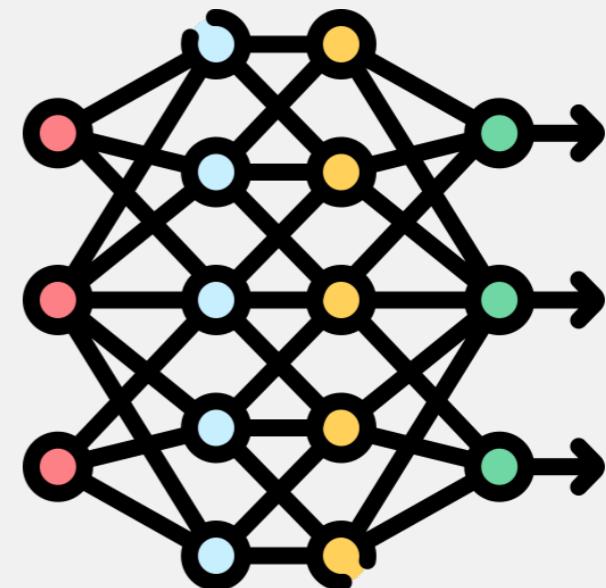
# What even is an LLM?



# Large Language Models



- Language Modeling: Measure probability of a sequence of words
  - Input: Text sequence
  - Output: Most likely next word
- LLMs are... large
  - GPT-3 has 175B parameters
  - GPT-4 is estimated to have ~1.24 Trillion
- Pre-trained with up to a PB of Internet text data
  - Massive financial and environmental cost



*\*Not actual size*

# Large Language Models are Pre-trained



- Only a few people have resources to train LLMs
- Access through API calls
- OpenAI, Google Vertex AI, Anthropic, Hugging Face
- We will treat it as a *black box that can make errors!*

# LLMs are Far from Perfect



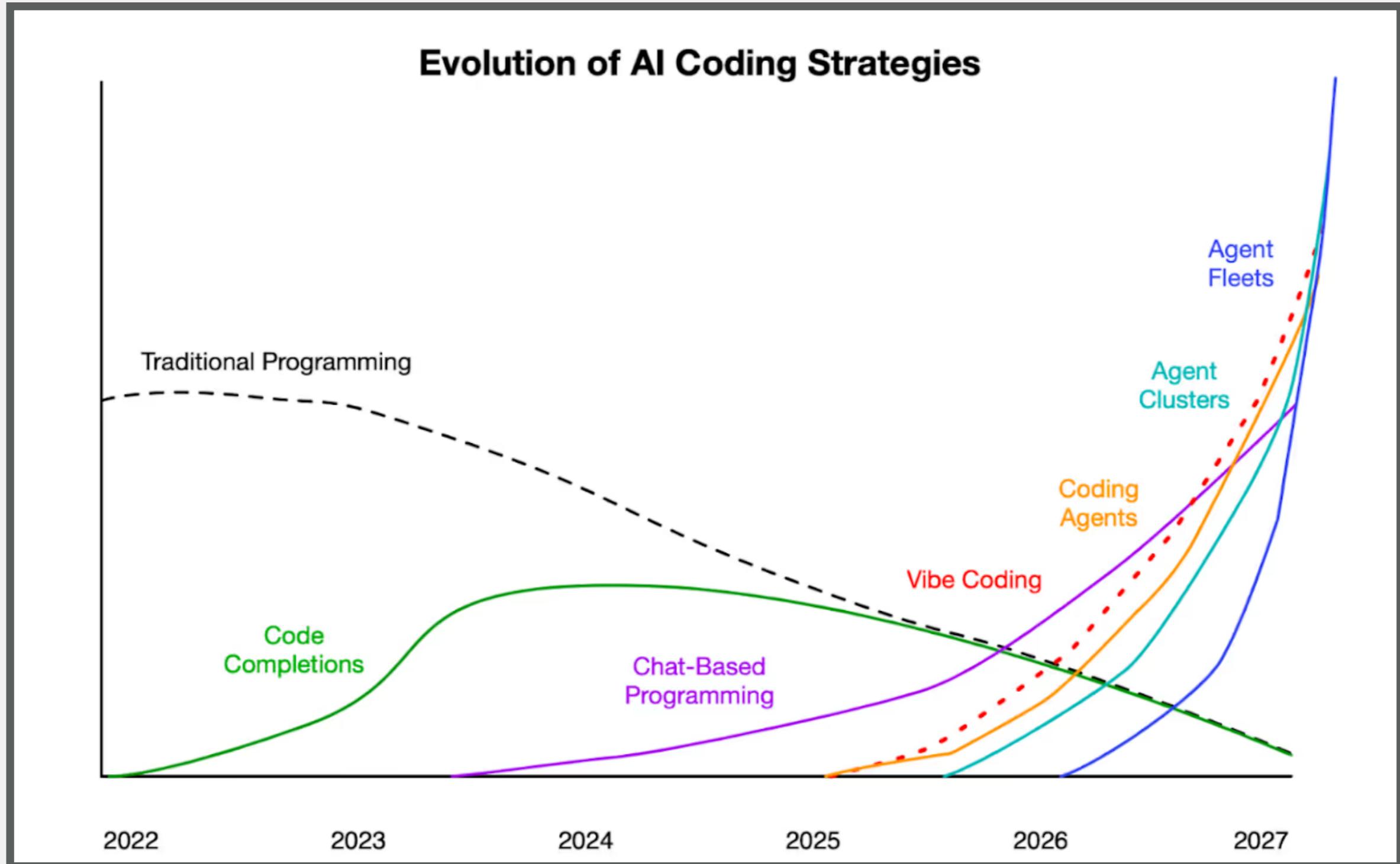
- Hallucinations
  - Factually Incorrect Output
- High Latency
  - Output words generated one at a time
  - Larger models also tend to be slower
- Output format
  - Hard to structure output (e.g. extracting date from text)
  - Some workarounds for this (later)

USER	print the result of the following Python code: ``` def f(x): if x == 1: return 1 return x * (x - 1) * f(x-2)  f(2) ```
ASSISTANT	The result of the code is 2.

# The AI Coding Hype Cycle

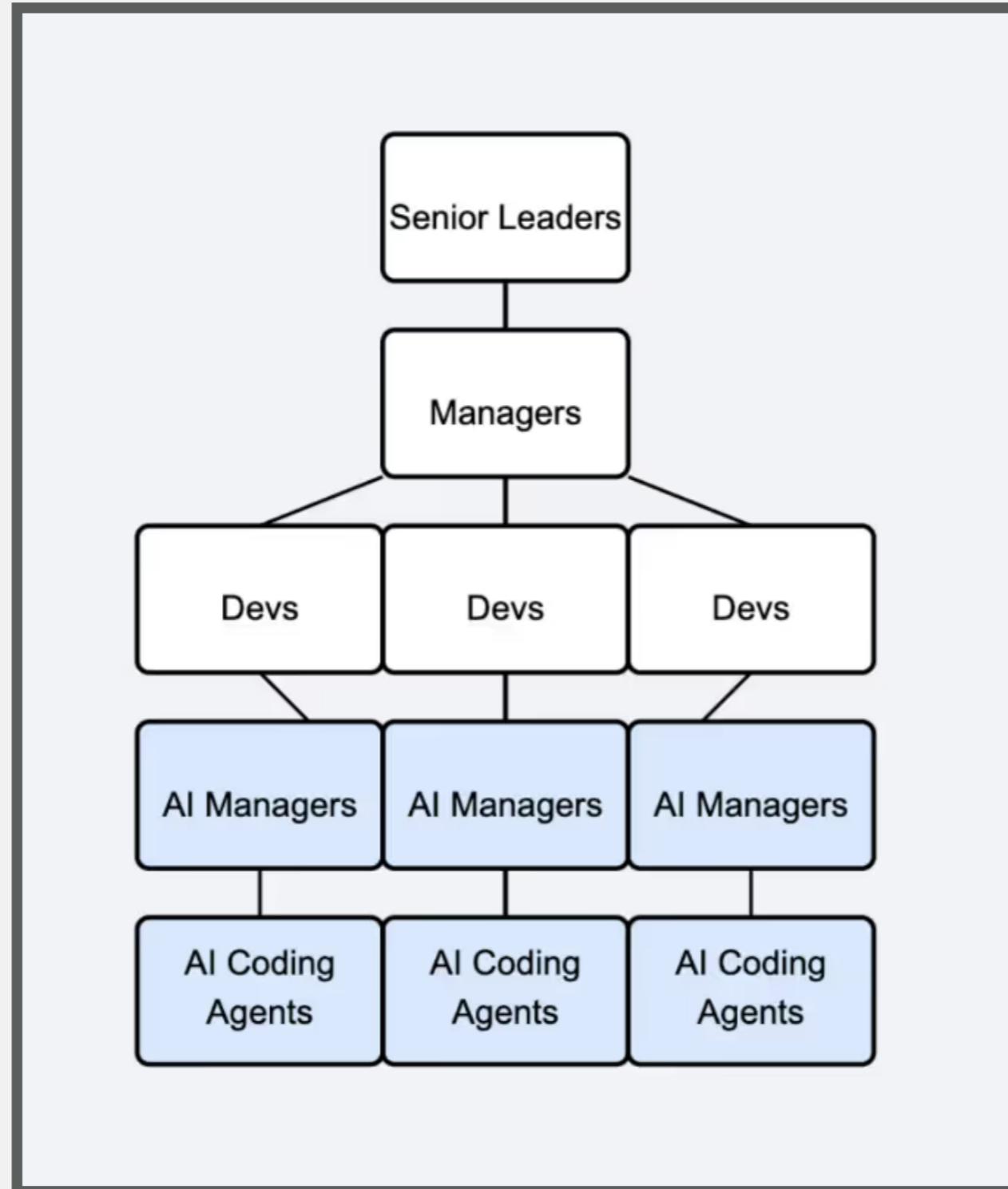


# The Rise of LLMs for Coding



Credit Steve Yegge - "The Revenge of the Junior Developer"

# The Shifting Landscape



Credit Steve Yegge - "The Revenge of the Junior Developer"

# Industry Anecdotes



**Andrey Karpathy**  @karpathy · Jan 26



A few random notes from claude coding quite a bit last few weeks.

Coding workflow. Given the latest lift in LLM coding capability, like many others I rapidly went from about 80% manual+autocomplete coding and 20% agents in November to 80% agent coding and 20% edits+touchups in December. i.e. I really am mostly programming in English now, a bit sheepishly telling the LLM what code to write... in words. It hurts the ego a bit but the power to operate over software in large "code actions" is just too net useful, especially once you adapt to it, configure it, learn to use it, and wrap your head around what it can and cannot do. This is easily the biggest change to my basic coding workflow in ~2 decades of programming and it happened over the course of a few weeks. I'd expect something similar to be happening to well into double digit percent of engineers out there, while the awareness of it in the general population feels well into low single digit percent.

# Industry Anecdotes



**Andrej Karpathy**  @karpathy · Jan 26



A few random notes from claude coding quite a bit last few weeks.

IDEs/agent swarms/fallability. Both the "no need for IDE anymore" hype and the "agent swarm" hype is imo too much for right now. The models definitely still make mistakes and if you have any code you actually care about I would watch them like a hawk, in a nice large IDE on the side. The mistakes have changed a lot - they are not simple syntax errors anymore, they are subtle conceptual errors that a slightly sloppy, hasty junior dev might do. The most common category is that the models make wrong assumptions on your behalf and just run along with them without checking. They also don't manage their confusion, they don't seek clarifications, they don't surface inconsistencies, they don't present tradeoffs, they don't push back when they should, and they are still a little too sycophantic. Things get better in plan mode, but there is some need for a lightweight inline plan mode. They also really like to overcomplicate code and APIs, they bloat abstractions, they don't clean up dead code after themselves, etc. They will implement an inefficient, bloated, brittle construction over 1000 lines of code and it's up to you to be like "umm couldn't you just do this instead?" and they will be like

# Industry Anecdotes

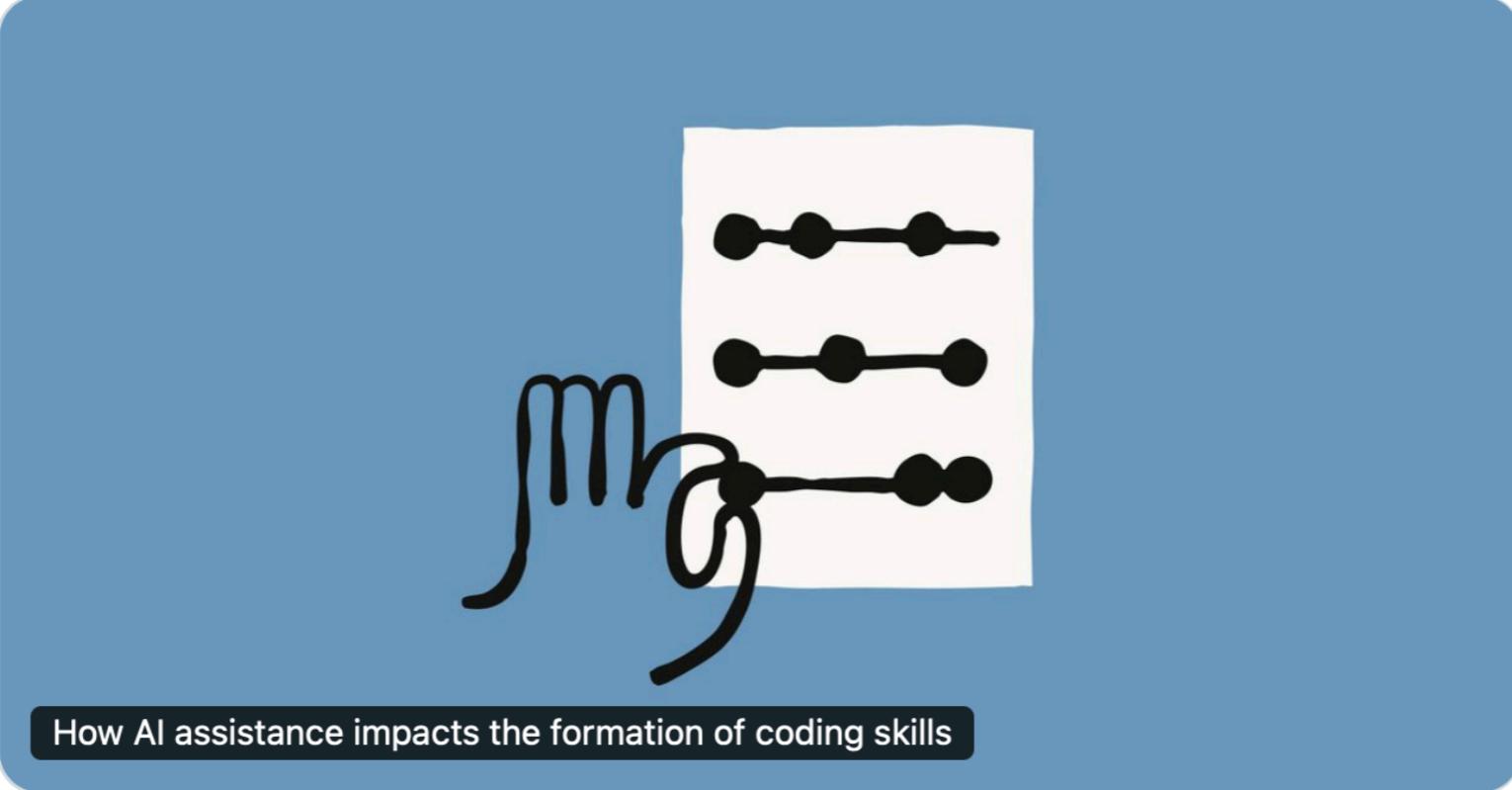


[←](#) **Post** [...](#)

 **Anthropic**   
@AnthropicAI

AI can make work faster, but a fear is that relying on it may make it harder to learn new skills on the job.

We ran an experiment with software engineers to learn more. Coding with AI led to a decrease in mastery—but this depended on how people used it.



How AI assistance impacts the formation of coding skills

From anthropic.com

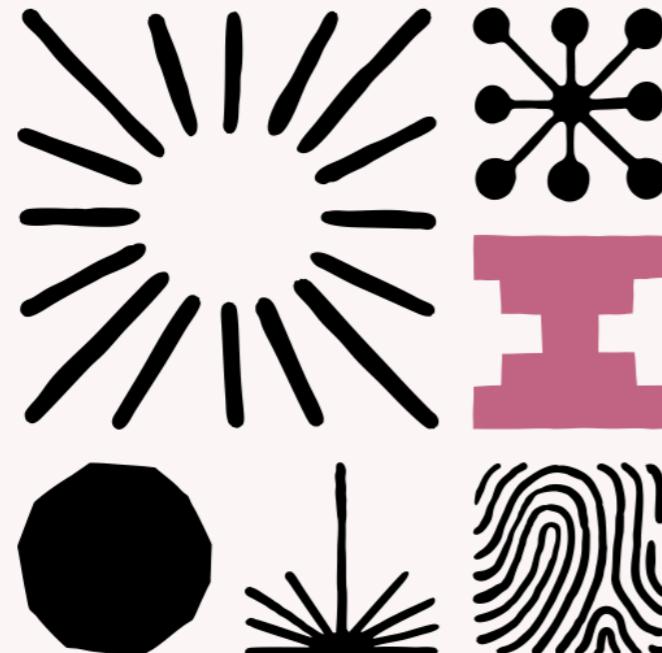
2:43 PM · Jan 29, 2026 · 1M Views



AI

≡

## Engineering at Anthropic



### Building a C compiler with a team of parallel Claudes

We tasked Opus 4.6 using agent teams to build a C Compiler, and then (mostly) walked away. Here's what it taught us about the future of autonomous software development.

# LLM-powered Software Development



# Claude Code - Our Subject



 Claude

Meet Claude ▾ Platform ▾ Solutions ▾ Pricing ▾ Learn ▾ Login Contact sales Try Claude

Product / Claude Code Explore here ▾

\* Debugging...

# Autocomplete finishes lines. Claude Code finishes features.

Work with Claude directly in your codebase. Build, debug, and ship from your terminal, IDE, Slack, or the web. Describe what you need, and Claude handles the rest.

Get Claude Code ▾ curl -fsSL https://claude.ai/install.sh | bash

Or read the documentation



- Getting Started - Codebase Comprehension
  - `cd /path/to/project`
  - `claude`
    - `> give me an overview of this codebase`
    - `> explain the main elements of the architecture used here`