Tech Titans - Milestone 2

Chut Giet rbj863
Chibuikem Emeka-Nwuba Esq498
Samuel Olukuewu nds091
Ademola Obaleye zyq049
Haidari Alhaidari vrl968

OVERVIEW OF STATIC ANALYSIS

Static analysis is the systematic examination of an abstraction of a program's state space. The analysis is performed on representations, typically a finite one, such as source code, abstract syntax trees, or compiled code. Static analysis does not rely on: specific input values, runtime execution or observed failures. Instead, it reasons about the structure and potential behavior of the program. Some static analyses, such as type checking, are precise and focus on language correctness.

Static analysis operates on
- Program structure (methods, classes, modules)
- Control flow and data flow
- Syntactic and semantic patterns
- Relationships between program elements

Why static analysis is valuable
- Can be applied early, before code is run
- Scales to large codebases
- Can detect classes of issues that are rare or hard to reproduce at runtime

PURPOSE

By completing this milestone, we are:
- Understanding how static analysis tools differ in scope, assumptions, and outputs
- Gaining hands-on experience configuring and running static analysis tools for multiple programming languages
- Learning to extract comparable metrics (counts, categories, severity, runtime, and complexity) across tools
- Practicing interpreting static analysis warnings in terms of actionability, noise, and maintainability implications
- Relating tool outputs to software engineering concepts such as coding standards, correctness, and structural complexity
- Producing a professional technical report with tables, charts, and evidence-based discussion

Our objective is to run these analyses, collect consistent and reproducible metrics, and interpret what each perspective reveals about the structure, practices, and potential maintainability risks of the code. Rather than relying on a single notion of "code quality," this report emphasizes comparing different static analysis signals, understanding their limitations, and reasoning about how such tools can (and cannot) inform real software engineering decisions.

GUAVA

Google Guava is a large, mature Java library focused on providing core utilities that extend the standard Java SDK. Its domain is general-purpose infrastructure: collections (immutable collections, multimaps, multisets), concurrency utilities, caching, I/O helpers, hashing, primitives, strings, and graph data structures. It is designed for production-grade use, with separate JRE and Android variants, strong backward-compatibility guarantees, and extensive testing. In terms of size, Guava is a very large codebase (hundreds of classes across many packages, tens of thousands of lines of code), actively maintained, and widely used both inside Google and across the Java ecosystem.

TOOLS USED

Semgrep is a lightweight, rule-based static analysis tool that supports multiple programming languages *(including Java)*. It detects bug patterns, insecure practices, and maintainability issues using syntactic and semantic rules. Semgrep serves as the shared cross-language analysis backbone in this Milestone.
Checkstyle enforces coding standards and structural conventions in Java source code. It helps identify stylistic issues and maintainability concerns related to code organization and formatting.
CK computes classical object-oriented metrics for Java source code. In this milestone, only the Weighted Methods per Class (WMC) metric is used as a class-level complexity indicator.

EXPERIMENTAL SETUP

We'll use a shared Dockerized repo for Milestone 2. The container standardizes all versions, and static-analysis tools. For lack of content to work with, we analyze the entire repository structure. Everyone runs tools through the same scripts so results are reproducible. The repository link is: https://github.com/hxdxri/M2-470.

RESULTS

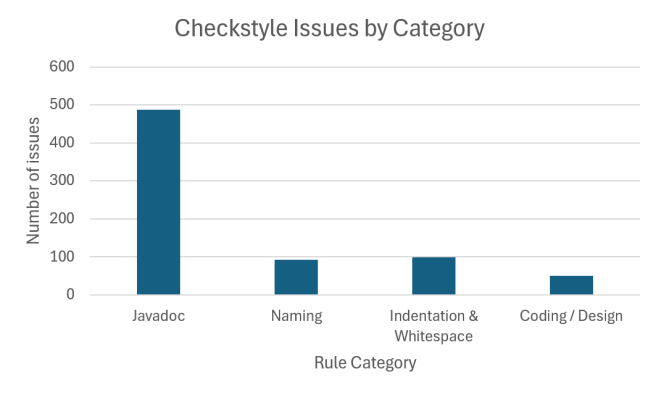
Checkstyle

Most frequent rule or message IDs:

1. source="com.puppycrawl.tools.checkstyle.checks.javadoc.SummaryJavadocCheck" - (221)
2. source="com.puppycrawl.tools.checkstyle.checks.javadoc.JavadocParagraphCheck" - (110)

Execution time: real: 2m3.780s user: 2m39.690s, sys: 0m19.784s

Category	Example Rules	# of Issues
Javadoc	SummaryJavadocCheck (221), JavadocParagraphCheck (110), AtclauseOrderCheck	488
Naming	AbbreviationAsWordInNameCheck, LocalVariableNameCheck (31), MemberNameCheck	92
Indentation & whitespace	IndentationCheck (70), EmptyLineSeparatorCheck	99
Coding / Design	MissingSwitchDefaultCheck (30), VariableDeclarationUsageDistanceCheck (20)	50
Total:		804



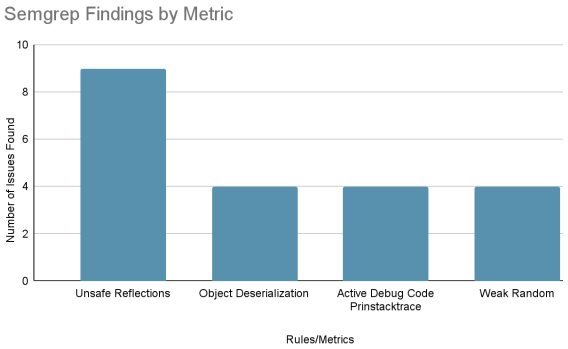
Semgrep

Most frequent rule or message IDs:

1. Unsafe Reflection: This means the program is using reflections, a feature that lets the program inspect and use classes/methods at runtime, where the calls can be influenced by other inputs, thus risking allowing hackers to inject their own dependencies at runtime. This could let them send execution in unexpected paths, bypass authorization or cause unintended behavior.
2. Object Deserialization (Tied #2): This means the code is using ObjectInputStream to rebuild java objects from streams. This is risky because hackers can create malicious serialized objects of their own that trigger unexpected behavior when read, leading to security risks.

Execution time: 0:00:22h, i.e. 22s

Key Analysis: Since Semgrep were mainly false positives, there would not be any high impact findings from this analysis. However, in real development settings, we could prioritize semgrep findings over CK and Checkstyle because Semgrep rules seem to be more focused on vulnerabilities, particularly security code patterns, thus high impact compared to checkstyle code maintainability and CK complexity hotspots.
The huge number of false positives is a limitation because of the rules' lack of context as it did a generic scan thus reporting lots of issues from test and benchmark code, which were intentionally included in the codebase.

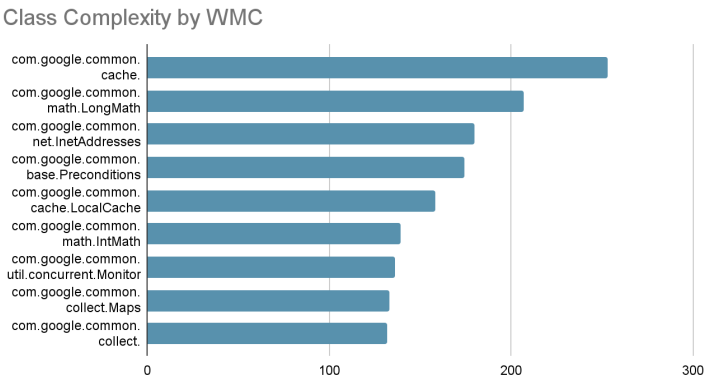


Category	Reasoning	# of Issues
Unsafe Reflection (False Positive)	They seem to be legitimate use cases, i.e. benchmark & internal utility classes are system controlled, reducing user injections.	9
Object Deserialization (False Positive)	These are in test utilities, for verifying serialization correctness, suggesting they are not used in production code.	4
Active Debug Code Prinstacktrace (False Positive)	The issues are in utility classes that intentionally handle traces for error logging, thus having a whole dedicated class implemented to do that.	4
Weak Random (False Positive)	Found in benchmark files, suggesting they are used for generating test data used in benchmarks, not security sensitive production data like tokens or passwords.	4
Total:		21

CK

Most frequent rule or message IDs (structural complexity hotspots by WMC):

- *com.google.common.cache.LocalCache$Segment - 253*
- *com.google.common.math.LongMath - 207*
- *com.google.common.net.InetAddresses -180*
- *com.google.common.base.Preconditions - 174*
- *com.google.common.cache.LocalCache - 158*
- *com.google.common.math.IntMath - 139*
- *com.google.common.util.concurrent.Monitor - 136*
- *com.google.common.collect.Maps - 135*
- *com.google.common.collect.MapMakerInternalMap$Segment - 132*
- *com.google.common.reflect.TypeToken - 129*
- *com.google.common.collect.CompactHashMap - 115*
- *com.google.common.collect.Iterators - 110*
- *com.google.common.base.CharMatcher - 110*
- *com.google.common.collect.TreeMultiset$AvlNode - 108*
- *com.google.common.util.concurrent.AbstractFuture - 106*
- *com.google.common.collect.ImmutableSortedMap - 106*
- *com.google.common.cache.CacheBuilder - 100*
- *com.google.common.collect.CompactHashSet - 99*
- *com.google.common.collect.Range - 88*
- *com.google.common.collect.TreeMultiset - 81*



CK

Key Analysis: These classes all have high complexity methods, increasing cost of maintenance due to the large number of complex methods. They all have a large number of lines of code making them more difficult to read through and optimize. The high number of static invocations also points to a high degree of coupling with these classes, which makes them difficult to change or update due to their multiple dependencies.

Class	WMC	NOSI	LOC
com.google.common.cache.LocalCache$Segment	253	11	1117
com.google.common.math.LongMath	207	78	606
com.google.common.net.InetAddresses	180	100	571
com.google.common.base.Preconditions	174	84	458
com.google.common.cache.LocalCache	158	37	3046

ANALYSIS AND DISCUSSION

Alignment of Signals Across Tools

Structural complexity hotspots identified by CK do not strongly align with the majority of Checkstyle or Semgrep findings.

CK highlights a small set of classes with extremely high Weighted Methods per Class (WMC), such as LocalCache$Segment, LongMath, and InetAddresses. These classes are large, central infrastructure components with dense logic and high coupling, indicating maintenance risk due to size and complexity.

In contrast:

- Checkstyle findings are overwhelmingly concentrated on stylistic and documentation-related issues (e.g., Javadoc formatting, naming conventions, indentation). These issues are distributed broadly across the codebase and do not cluster specifically around CK's complexity hotspots.
- Semgrep findings are sparse and flagged as false positives, often occurring in benchmark or utility code rather than the structurally complex core classes identified by CK.

The limited overlap suggests that:

- Complexity (CK) captures architectural and structural risk.
- Checkstyle captures consistency and readability issues.
- Semgrep captures potential semantic or security patterns, but with limited contextual precision in this repository.

Overall, the tools expose different quality dimensions rather than reinforcing the same signals.

ANALYSIS AND DISCUSSION

Actionability of Findings

Among the three tools, Semgrep provides the most immediately actionable findings, despite the high false-positive rate.

- Semgrep flags concrete patterns (e.g., unsafe reflection, object deserialization, debug code) that developers can directly inspect, assess risk, and fix or justify.
- Checkstyle findings are actionable primarily in terms of style enforcement, but they do not indicate functional errors or correctness issues.
- CK's WMC metric identifies risk hotspots but does not prescribe corrective actions; it requires developer interpretation (e.g., refactoring, decomposition).

Thus, in a real development setting:

- Semgrep findings would be prioritized for security review
- CK hotspots would guide refactoring and architectural attention
- Checkstyle would be used to enforce consistency, not correctness

Limitations of the Tools

None of the tools can directly conclude correctness or functional quality.

- **CK metrics** quantify complexity but do not explain *why* complexity exists or whether it is justified.
- **Checkstyle** enforces predefined standards and does not detect bugs or semantic errors.
- **Semgrep** relies on rule matching and lacks full contextual awareness, leading to false positives when analyzing test or benchmark code.

Additionally, results are highly sensitive to configuration and scope:

- Running tools on incorrect directories (e.g., test or Android variants) significantly increases noise.
- Tool outputs depend on available rule sets (e.g., Semgrep Pro vs default rules).

Static analysis should therefore be viewed as a **best-effort signal generator**, not a definitive measure of software quality or correctness.

Practical Implications

When applied correctly, these tools are most effective when used **together**:

- CK identifies *where* developers should look.
- Semgrep suggests *what might be wrong*.
- Checkstyle ensures how code should be written remains consistent.

Used in isolation, each tool provides an incomplete picture; combined, they support informed engineering decisions about maintenance, refactoring, and risk management.

LESSONS LEARNED

This milestone demonstrates that static analysis tools are most effective when used as **complementary instruments rather than standalone judges of quality**.

CK is effective at highlighting structural risk but requires careful interpretation and does not provide direct remediation guidance. Checkstyle excels at enforcing consistency and readability but generates large volumes of low-impact findings. Semgrep offers the strongest link to correctness and security concerns but suffers from contextual noise and false positives.

A key challenge encountered was **configuration and scope control**. Tool outputs varied significantly depending on directory selection, rule availability, and environment setup. Interpreting results also required understanding each tool's underlying assumptions and intended use.

In practice, static analysis is best applied early and continuously to reduce technical debt, but its outputs must be reviewed critically. High-quality rules, correct configuration, and human judgment are essential for meaningful results.

CONCLUSION

This analysis shows that different static analysis perspectives reveal **complementary but non-redundant insights** about a software system. CK identifies where complexity and maintenance risk are concentrated, Checkstyle enforces consistency and coding discipline, and Semgrep highlights potential semantic and security-related issues. No single tool provides a complete picture of software quality or correctness. When combined, these tools support more informed engineering decisions: identifying risky areas, prioritizing reviews, and maintaining long-term code health. However, static analysis should be treated as a **best-effort signal generator**, not a definitive assessment of correctness.

For real-world development workflows, the most effective approach is to integrate these tools together, apply them consistently, and interpret their outputs with domain knowledge and engineering judgment

REFERENCES

1. Semgrep Documentation: https://semgrep.dev/docs/
2. Checkstyle Documentation: https://checkstyle.sourceforge.io/
3. Maurício Aniche, 2015: Java code metrics calculator (CK): Available in https://github.com/mauricioaniche/ck/