**🔎 How SonarQube’s Code Smell Engine Works**

**1. Static Code Analysis Core**

* SonarQube doesn’t execute the program.
* It performs **static analysis** (reads code → parses into an Abstract Syntax Tree (AST) → applies predefined rules).
* The AST is essentially a structured tree representation of your source code (e.g., class → methods → variables → statements).

**2. Rules Engine**

* SonarQube defines **rules** (e.g., “A method should not exceed 100 lines,” “A class should not have >20 fields”).
* Each rule has:
  + **Key/ID** (e.g., java:S1448 → “Methods should not have too many parameters”).
  + **Implementation** (a Java class that traverses AST nodes and checks conditions).
  + **Severity** (Info, Minor, Major, Critical).

A screenshot of a computer code

AI-generated content may be incorrect.

**3. Rule Sets and Profiles**

* A **Quality Profile** is a collection of rules.
* Default profiles: “Sonar way,” “Sonar way recommended.”
* You can activate/deactivate rules or set thresholds.

**4. Scanner Workflow**

When you run SonarQube on a project:

1. **SonarScanner** parses your Java files using its analyzers (SonarJava plugin).
2. Each file → converted to AST.
3. Active rules → applied to AST nodes.
4. Issues found → reported to the **SonarQube server**.
5. Results → visible in GUI (Dashboard → Code Smells, Bugs, Vulnerabilities).

**5. Detection Limitations (Where You Can Add Value)**

* SonarQube’s rules are **static and heuristic-based**:
  + No learning from project context.
  + Limited cross-file reasoning (some rules check only within a file/class).
  + Often leads to **false positives** (“This is a smell, but in this project it’s intentional”) and **false negatives** (“Missed smell because rule is too rigid”).

**🚀 Where Your AI/ML Fits In**

Now, connecting to your earlier question 👇

“But my data will take SonarQube output as input but it also has to do its own research between the output and the repo…”

Exactly ✅

* **SonarQube Output = baseline labels.**
  + Example: SonarQube says “This is a Long Method.”
* **Your Agent = cross-checker + learner.**
  + It re-analyzes code (AST, embeddings, or NLP) and decides:
    - Confirm → true positive.
    - Reject → false positive.
    - Discover new → false negative fixed.

So your pipeline becomes:

Repo Source Code → SonarQube Rules → Issues → Export Issues (JSON) → ML Agent Re-checks Against Code → Improved Issue Set.

⚡ Example Cycle 1 Experiment:

* Rule: “God Class” (too many methods/fields).
* SonarQube flags 50 classes.
* Your ML model (trained with AST embeddings + metrics like cohesion, coupling) says:
  + 35 are true smells.
  + 15 are false positives.
* Plus, it finds 5 more classes that SonarQube missed.

This “post-processor” can later evolve into a **custom SonarQube plugin**, but early cycles can just run it externally.