**🔹 How It Works (Cycle 1, Domain 1 – Detection Engine Augmentation)**

**1. Run SonarQube Normally**

* Install SonarQube CE locally.
* Scan your chosen Java + Spring Boot/Apache repos.
* SonarQube will generate a set of *detected code smells/issues* (e.g., God Class, Long Method, Duplicated Code).
* You can view these in the GUI **and** fetch them programmatically with the Web API (/api/issues/search).

👉 This is your **baseline detector**.

**2. Export SonarQube Results**

* Use the Web API to dump results in JSON.
* Each issue has:
  + Rule ID (e.g., java:S1186 for “Empty Method”).
  + File + line numbers.
  + Type (Bug, Vulnerability, Code Smell).
  + Message + severity.

👉 This is your **SonarQube-detected dataset**.

**3. Align Issues with Repository Source**

Here’s where your **AI/ML angle** comes in:

* Fetch the **actual source code snippet** for each flagged location (e.g., AST subtree or function body).
* For comparison, also gather **unflagged code** (SonarQube didn’t mark it, but maybe it *should* have).

👉 You now have **pairs**:

* (SonarQube label, actual code snippet).

**4. Identify Weaknesses**

* **False positives** → SonarQube flagged an issue, but manual/benchmark check says it’s fine.
* **False negatives** → SonarQube missed real smells (you can benchmark against datasets like **MORO, PROMISE, or manual expert labeling**).

👉 This becomes your **ground truth dataset**.

**5. Build Your ML-Augmented Detector**

You add an **AI/ML agent** that works *on top of SonarQube output*:

**Two Strategies:**

1. **Post-Processor Classifier (short-term, easier)**
   * Input: SonarQube’s reported issues + code context.
   * ML model: binary classifier (True issue vs False alarm).
   * Goal: reduce false positives.
2. **Independent Detector (long-term, harder)**
   * Input: raw code snippets (with AST/NLP embeddings).
   * ML model: learns to detect God Class, Long Method, etc.
   * Goal: catch issues SonarQube *missed* (reduce false negatives).
   * You compare overlap with SonarQube → that’s your improvement metric.

👉 In Cycle 1, you probably stick with **Post-Processor** because it’s feasible and publishable within 1 year.

**6. Integration Options**

* **Option B (short-term, external agent)**:  
  Run SonarQube → export JSON → process with your ML pipeline (Python, TensorFlow/PyTorch, HuggingFace for code embeddings).  
  Output: “Refined Issue List” (less noisy than SonarQube’s).
* **Option A (long-term, plugin)**:  
  Implement a **SonarQube plugin** in Java that hooks into the detection engine, calling your ML model directly.  
  This makes it part of the actual SonarQube workflow.

**🔹 Analogy**

Think of it like a **spam filter**:

* Gmail (SonarQube) already flags spam.
* You train your own AI (ML agent) to double-check Gmail’s spam classification.
* At first, you only say “yes/no” to Gmail’s flags.
* Later, you can build your own independent spam detector.

**🔹 Why This Is Valid Research**

* **Industry gap**: Tools like SonarQube are deterministic/static → no ML adaptivity.
* **Novelty**: You create a hybrid pipeline that reduces false alarms or augments detection.
* **Feasibility**: You don’t need to “rewrite SonarQube,” you build on top of it.
* **Scalability**: Later, you can migrate into plugin form.

✅ So in Cycle 1, yes:  
You **take SonarQube’s output as input**, align it with repo code, build a small ML model that reduces false alarms or finds missing smells, and benchmark your “refined detection engine” against raw SonarQube.