ML ANOMALY DETECTION · STATIC ANALYSIS (SAST)

Analysis Anomaly Detection System

A machine-learning system that predicts, detects, and alerts on abnormally long static-analysis runs in Fasoo's SPARROW SAST

Yeongseok Lim · SP Development Team 1, Fasoo · 10-week project

ABSTRACT

SPARROW is Fasoo's static application security testing (SAST) product. Occasionally the same project took drastically longer to analyze for no obvious reason, with no way to tell such a run from a normal one, so engineers had to wait until the job finished. I analyzed about 700 GitHub projects to collect metadata and runtimes, then trained an XGBoost regressor on 28 features to predict the expected analysis time. The model reached a coefficient of determination (R^2) of about 0.90 on average, and any run that falls outside a ±20% band around the prediction is detected automatically and alerted via Discord.

0.90	0.96	~700	28	5
Mean R^2	Best R^2	Repositories	Features	Languages

1. Background & Problem

SPARROW is Fasoo's static application security testing (SAST) product. Developers submit source code and SPARROW runs static analysis to surface security and quality issues.

Most analyses finish quickly, but occasionally **the same project would take drastically longer for no obvious reason**. With no criterion to separate a normal run from a stuck, abnormal one, engineers had to sit and watch the job until it finished.

Table 1. Analysis runtime for identical source code. The inputs were the same, yet some runs were abnormally inflated.

Target	Start	End	Duration
Source A	2025-06-26 13:46:58	2025-06-26 13:52:13	5m 15s
Source A	2025-06-09 08:01:56	2025-06-09 13:04:56	⚠ **5h 03m**
Source B	2025-06-02 09:58:38	2025-06-02 09:59:51	1m 13s
Source B	2025-06-02 09:02:23	2025-06-02 12:17:23	⚠ **3h 15m**

Goal. Build a system that learns the expected analysis time from a project's characteristics and automatically detects and alerts on any run that falls far outside that range.

2. Approach & System

I owned the project end to end: data collection, storage, exploratory analysis, model development, and integration into the analysis service. The full pipeline is as follows.

```
Collect GitHub repos → Run static analysis (runtime + metadata) → S3 (source) · PostgreSQL (metadata) → Train XGBoost → Predict runtime · detect deviations → Discord alert
```

Tech stack

Python · pandas · NumPy · scikit-learn · XGBoost · joblib · PostgreSQL · AWS S3 · GitHub Actions · SPARROW SDK · Discord Webhook

3. Data Collection

I collected about 700 public GitHub repositories across the languages SPARROW supports, deliberately spread across size bands (0–50 MB: 30, 50–100 MB: 30, 100–200 MB: 40) to avoid size bias.

Each repository went through static analysis and its runtime recorded with structural metadata. Per language I captured code size (MB), lines of code, cyclomatic complexity sum (CCN), and file count, then added aggregate totals and derived ratios (lines per MB, complexity per line, files per MB, language count) for a **total of 28 features**.

Table 2. Engineered feature set (28 total).

Group	Features
Per language (5 langs)	`mb` · `line` · `ccn_sum` · `file_count`
Aggregate totals	`total_line` · `total_mb` · `total_ccn` · `total_file_count` · `lang_count`
Derived ratios	`line_per_mb` · `ccn_per_line` · `file_per_mb`

Target languages: Java · C / C++ · Python · JavaScript / TypeScript · C#. Both single-language and multi-language (extension-combination) projects were included.

4. Factor Analysis

I ran a controlled-variable analysis to isolate each factor and see how it related to runtime.

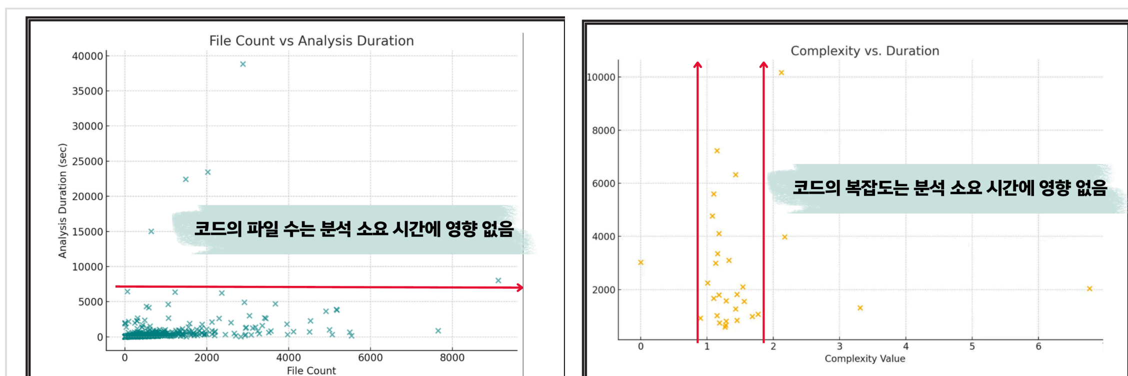

Figure 1. **File count** and **average complexity** showed no clear relationship with runtime.

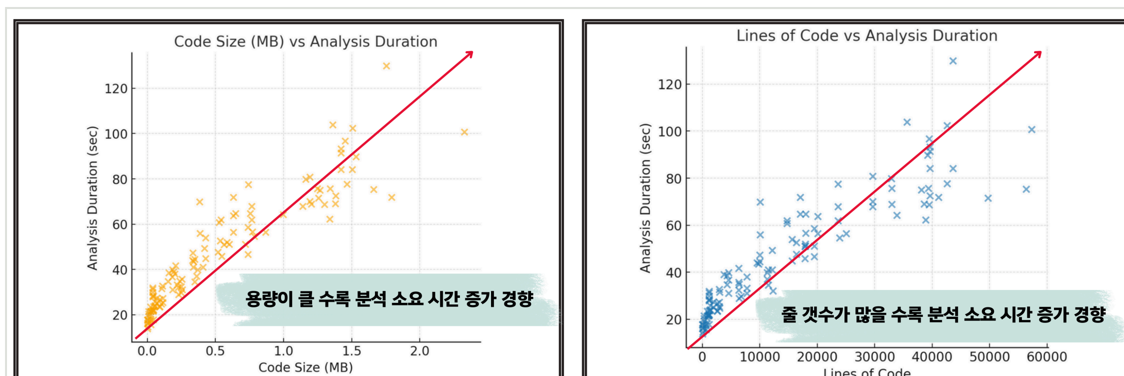

Figure 2. In contrast, **code size (MB)** and **total lines of code** both increased runtime in a roughly linear fashion.

In short, file count and average complexity had little effect, while code size and lines of code drove runtime. I used these two as the primary input signals.

5. Modeling: From Linear Regression to XGBoost

I first tried a per-language linear regression on code size and lines of code. It captured the broad per-language trend (Java R^2 0.93, Python 0.93), but a single straight line could not express the interactions between factors, so its predictions were unreliable.

I switched to **XGBoost gradient boosting**, which learns the interactions across all 28 features. The target was log-transformed runtime, and predictions were inverse-transformed back to seconds.

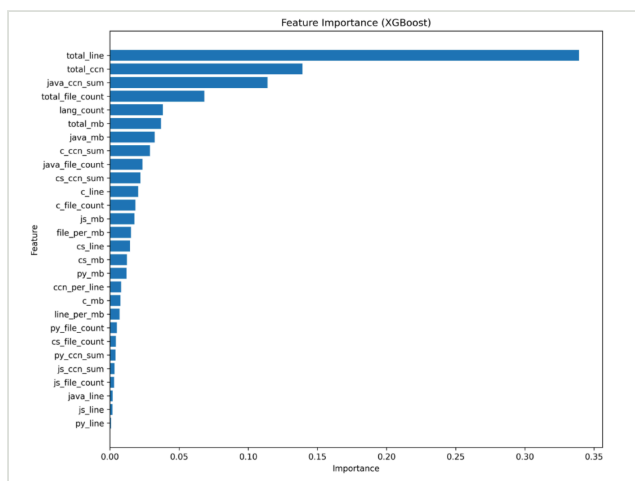

Figure 3. Feature importance. **total_line** had by far the largest impact, followed by **total_ccn** (total complexity sum). The model captured the non-linear interactions the linear model had missed.

```
# model configuration
XGBRegressor(n_estimators=100, max_depth=5)
# target: log-transformed duration · features: 28 · split: train 90% / test 10% · metric: R²
y_pred = np.expm1(model.predict(X))   # → seconds
```

6. Results

The XGBoost model reached a coefficient of determination (R^2) averaging **0.8963** (min 0.7321, max 0.9610, std 0.0392). Where linear regression could not capture the interactions between factors and predicted unreliably, XGBoost, trained on all 28 features jointly, achieved consistently high explanatory power.

Table 3. XGBoost predictions (sample): actual vs predicted.

Analysis ID	Actual time (s)	Predicted time (s)
10039	84	86.34
10038	76	76.02
10037	62	66.66
10036	69	78.57
10035	64	89.86
10033	104	65.02
10032	102	91.55
10031	90	83.68
10030	93	81.40
10029	97	83.72
10022	1017	940.32
10021	764	727.84

Table 4. Per-language linear R^2 (reference).

Language	R^2
Java	**0.93**
Python	**0.93**
C / C++	**0.83**
C#	**0.79**
JavaScript / TS	**0.70**

Anomaly criterion. A run is normal if its actual time falls within ±20% of the predicted time, and flagged as an anomaly otherwise.

7. Productionization

I integrated the detector into the static analysis service and built a self-improving retraining loop. GitHub Actions queries PostgreSQL for the latest analyses on a fixed schedule and retrains the model on accumulated data, so accuracy keeps improving over time.

When a new run's actual time falls outside the predicted ±20% band, a Discord bot (Captain Hook) posts an alert with a link to the analysis and the predicted versus actual times. The feature was built for the **SPARROW 2508.1 release**.

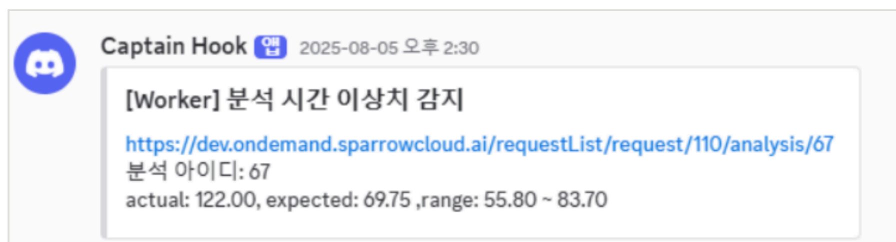

Figure 4. Real alert example. Predicted 69.75s (normal range 55.80–83.70) versus an actual 122.00s, flagged as an anomaly, with a direct link to the analysis page.

8. Conclusion & Future Work

Before, abnormal runs went unnoticed until someone checked manually. Now they are flagged automatically in near real time, and the model keeps improving as more data accumulates.

Future work

- Collect more data, especially from large projects, to push accuracy higher.
- Expand detection beyond the current five languages.
- Deploy to a cloud environment for operational stability.