

Student ID: 12693547

Module: MSc Computing Project

Assignment: Dissertation

Research Topic: AI-Enhanced Static Code Analysis for Fintech Applications: A Hybrid Approach for Elevated Code Quality, Security, and Developer Productivity

Contents

[Contents 1](#_Toc205838919)

[**Abstract** 6](#_Toc205838920)

[**Chapter 1 - Introduction, Background. Aims and Objectives** 6](#_Toc205838921)

[**1.1 Introduction** 6](#_Toc205838922)

[**1.2 Background** 7](#_Toc205838923)

[**1.3 Aims and Objectives** 7](#_Toc205838924)

[**1.3.1 Aim** 7](#_Toc205838925)

[**1.3.2 Success criteria & evaluation metrics** 8](#_Toc205838926)

[**1.3.3 Objectives** 8](#_Toc205838927)

[**1.4 Scope** 9](#_Toc205838928)

[**1.5 Chapter roadmap** 9](#_Toc205838929)

[**1.6 Alignment with MSc outcomes** 10](#_Toc205838930)

[**Chapter 2 - Literature Review** 10](#_Toc205838931)

[**2.1 Overview and scope** 10](#_Toc205838932)

[**2.2 Rule-based static analysis: strengths, limits, and lessons from industry** 11](#_Toc205838933)

[**2.3 Machine learning for code: from representations to defect signals** 12](#_Toc205838934)

[**2.4 Multi-label learning for overlapping code concerns** 14](#_Toc205838935)

[**2.5 Probability calibration: making scores usable in gates** 15](#_Toc205838936)

[**2.6 Fusion strategies: combining deterministic rules with probabilistic models** 15](#_Toc205838937)

[**2.7 Critique and gaps in the literature (and how this study responds)** 16](#_Toc205838938)

[**2.8 Limitations of the current evidence base** 17](#_Toc205838939)

[**2.9 Summary and implications for this study** 18](#_Toc205838940)

[**Chapter 3 – Methodology, Research Design and Ethical and Professional Considerations** 19](#_Toc205838941)

[**3.1 Overview and aims** 19](#_Toc205838942)

[**3.2 Research design** 19](#_Toc205838943)

[**3.2.1 Comparative conditions and variables** 19](#_Toc205838944)

[**3.2.2 Dataset construction (241 balanced snippets)** 20](#_Toc205838945)

[**3.2.3 Splits and protocol** 20](#_Toc205838946)

[**3.3 Data provenance and licensing** 21](#_Toc205838947)

[**3.4 Labelling scheme and quality assurance** 21](#_Toc205838948)

[**3.5 Rule set and tools** 22](#_Toc205838949)

[**3.6 Model and training** 22](#_Toc205838950)

[**3.7 Calibration and threshold selection** 23](#_Toc205838951)

[**3.8 Gated-fusion decision policy** 23](#_Toc205838952)

[**3.9 Evaluation protocol** 24](#_Toc205838953)

[**3.9.1 Metrics** 24](#_Toc205838954)

[**3.9.2 Procedure** 24](#_Toc205838955)

[**3.9.3 Error analysis and ablations** 25](#_Toc205838956)

[**3.10 Ethical & professional considerations** 25](#_Toc205838957)

[**3.10.1 Licensing and IP compliance** 25](#_Toc205838958)

[**3.10.2 Secure handling and data protection** 25](#_Toc205838959)

[**3.10.3 Bias and false-positive risk** 26](#_Toc205838960)

[**3.10.4 Responsible disclosure** 26](#_Toc205838961)

[**3.10.5 Reproducibility and transparency** 26](#_Toc205838962)

[**3.11 Implementation details** 27](#_Toc205838963)

[**3.11.1 Software and hardware** 27](#_Toc205838964)

[**3.11.2 Runtime measurement** 27](#_Toc205838965)

[**3.11.3 Decision logging** 27](#_Toc205838966)

[**3.12 Threats to validity** 27](#_Toc205838967)

[**3.13 Summary** 28](#_Toc205838968)

[**Chapter 4 – Implementation** 28](#_Toc205838969)

[**4.1 Overview** 28](#_Toc205838970)

[**4.2 System architecture** 28](#_Toc205838971)

[**4.3 FastAPI model microservice** 29](#_Toc205838972)

[**4.3.1 Endpoints** 29](#_Toc205838973)

[**4.3.2 Lifecycle, credentials, and concurrency** 29](#_Toc205838974)

[**4.3.3 Input normalisation and limits** 30](#_Toc205838975)

[**4.3.4 Output schema (abridged)** 30](#_Toc205838976)

[**4.4 Rule engine** 30](#_Toc205838977)

[**4.4.1 AST heuristics** 30](#_Toc205838978)

[**4.4.2 External tools** 31](#_Toc205838979)

[**4.4.3 Deduplication and mapping** 31](#_Toc205838980)

[**4.4.4 Sandboxing and timeouts** 31](#_Toc205838981)

[**4.5 Model serving** 31](#_Toc205838982)

[**4.5.1 Loading and calibration** 31](#_Toc205838983)

[**4.5.2 Inference routine** 31](#_Toc205838984)

[**4.5.3 Caching** 31](#_Toc205838985)

[**4.6 Gated-fusion logic** 31](#_Toc205838986)

[**4.7 GUI/annotation path** 32](#_Toc205838987)

[**4.8 Training and inference pipelines** 32](#_Toc205838988)

[**4.8.1 Training** 32](#_Toc205838989)

[**4.8.2 Inference** 32](#_Toc205838990)

[**4.9 Configuration** 32](#_Toc205838991)

[**4.9.1 YAML structure (excerpt)** 32](#_Toc205838992)

[**4.9.2 Credentials & environment file** 33](#_Toc205838993)

[**4.10 Deployment (local, no Docker)** 33](#_Toc205838994)

[**4.10.1 Local run** 33](#_Toc205838995)

[**4.10.2 CI/CD note** 34](#_Toc205838996)

[**4.11 Reproducibility checklist** 34](#_Toc205838997)

[**4.11.1 Environment** 34](#_Toc205838998)

[**4.11.2 One‑shot setup** 34](#_Toc205838999)

[**4.11.3 Re‑running experiments** 34](#_Toc205839000)

[**4.11.4 Logging, monitoring, and QA** 35](#_Toc205839001)

[**4.12 Security and professional considerations in deployment** 35](#_Toc205839002)

[**4.13 Limitations and future engineering work** 35](#_Toc205839003)

[**4.14 Summary** 35](#_Toc205839004)

[**Chapter 5 – Results** 35](#_Toc205839005)

[**5.1 Overview of evaluation outputs** 35](#_Toc205839006)

[**5.2 Main results (aggregate micro/macro)** 36](#_Toc205839007)

[**5.3 Per-label performance** 36](#_Toc205839008)

[**5.4 K-fold stability** 37](#_Toc205839009)

[**5.5 Confusion matrices (error structure)** 37](#_Toc205839010)

[**5.6 ROC curves and threshold sweeps** 38](#_Toc205839011)

[**5.7 Runtime analysis** 38](#_Toc205839012)

[**5.8 Summary of findings** 38](#_Toc205839013)

[**5.9 Reproducibility and artefact availability** 39](#_Toc205839014)

[**Chapter 6 – Discussion** 39](#_Toc205839015)

[**6.1 Interpreting the results** 39](#_Toc205839016)

[**6.2 Ablations and policy sensitivity** 40](#_Toc205839017)

[**6.3 Error analysis (representative failure modes)** 40](#_Toc205839018)

[**6.4 Threats to validity (revisited)** 41](#_Toc205839019)

[**6.5 Relationship to prior work** 41](#_Toc205839020)

[**6.6 Implications for CI/CD adoption** 42](#_Toc205839021)

[**6.7 Limitations and opportunities** 43](#_Toc205839022)

[**6.8 Concluding perspective** 44](#_Toc205839023)

[**Chapter 7 – Conclusion & Future Work** 44](#_Toc205839024)

[**7.1 Answers to the research questions** 44](#_Toc205839025)

[**7.2 What the hybrid approach improves** 45](#_Toc205839026)

[**7.3 Limitations** 46](#_Toc205839027)

[**7.4 Recommendations for practitioners** 46](#_Toc205839028)

[**7.5 Future work** 46](#_Toc205839029)

[**7.6 Final remarks** 47](#_Toc205839030)

[**References** 48](#_Toc205839031)

[**Appendices** 51](#_Toc205839032)

[**Appendix A)** **Evaluation summary (Rules vs Model vs Gated Fusion)** 51](#_Toc205839033)

[**Appendix B)** **K-fold macro-F1 (mean ± sd)** 52](#_Toc205839034)

[**Appendix C)** **Environment (reproducibility)** 57](#_Toc205839035)

[**Appendix D)** **Overall Micro/Macro** 59](#_Toc205839036)

[**Appendix E)** **Objective Traceability** 59](#_Toc205839037)

[**Appendix F)** **System architecture (full diagram)** 61](#_Toc205839038)

[**Appendix G)** **Confusion matrices per label** 61](#_Toc205839039)

[**Appendix H)** **ROC curves per label (Model)** 66](#_Toc205839040)

[**Appendix I)** **Threshold sweep plots** 67](#_Toc205839041)

[**Appendix J)** **Calculating optimal thresholds** 69](#_Toc205839042)

[**Appendix K)** **Sanity Tests** 70](#_Toc205839043)

[**Appendix L)** **Starting the microservice** 70](#_Toc205839044)

[**Appendix M)** **Repository Structure & Reproducibility Readme** 71](#_Toc205839045)

[**M.1 Summary** 71](#_Toc205839046)

[**M.2 Repository structure (abridged, from repo root)** 71](#_Toc205839047)

[**M.3 Environment** 72](#_Toc205839048)

[**M.4 Repro steps (outline)** 72](#_Toc205839049)

[**M.5 CI usage (minimal)** 72](#_Toc205839050)

[**M.6 Models & checkpoints** 73](#_Toc205839051)

[**M.7 Versioning and integrity** 73](#_Toc205839052)

[**M.8 Notes** 73](#_Toc205839053)

# **Abstract**

This dissertation evaluates a hybrid static analysis approach that combines rule-based checks with a lightweight neural classifier under a gated-fusion scheme. The rules layer aggregates signals from AST heuristics and established tools (e.g., SonarLint and Pylint), while the machine-learning layer is a multi-label RoBERTa-base model served via a FastAPI microservice.

To ensure domain relevance, the training data is composed of short Python snippets and code drawn from leading open-source fintech GitHub projects (selected by stars, recent activity, and permissive licences). Calibrated probabilities feed a simple decision gate designed to reduce false positives.

On a balanced dataset of 241 short code snippets, the rule-based checker achieved very high recall but low precision (micro precision/recall/F1 = 0.50/1.00/0.67; macro-F1 = 0.72). The learned model performed better overall (0.81/0.86/0.84; macro-F1 = 0.84). The gated fusion of rules and model was best (0.88/0.84/0.86; macro-F1 = 0.85), with per-label F1 of 1.00 (SAST risk), 0.79 (ML signal), and 0.90 (best practice). Five-fold cross-validation indicates the fusion result is stable (macro-F1 0.894 ± 0.031), and, once probabilities are computed, the additional runtime for fusion is negligible. These findings suggest a calibrated hybrid analyser can raise precision while maintaining high recall. The limitations of the hybrid analyser include a Python-only scope and the need for threshold tuning.

# **Chapter 1 - Introduction, Background. Aims and Objectives**

## **1.1 Introduction**

While the goal of static analysis is to improve software quality and security, purely rule-based tools often generate high volumes of warnings and false positives, which can hinder adoption (Johnson et al., 2013). Recent advances in code representation with transformer models suggest that learned signals can complement hand-engineered rules by capturing context beyond syntactic patterns (Liu et al., 2021). This dissertation investigates a hybrid static analysis approach that combines a rule layer with a multi-label RoBERTa-based classifier, fused via a simple gate to reduce false positives. The model is domain-oriented through training on code drawn from popular open-source fintech repositories, and its output probabilities are calibrated to support thresholding decisions (Guo et al., 2017). The study evaluates whether such a calibrated hybrid can raise precision while maintaining high recall, improving the actionability of findings for developers.

## **1.2 Background**

Static analysis can uncover defects and security issues early; however, adoption is often limited by prioritisation overheads, developer trust, and fit with everyday workflows (Bessey et al., 2010). Empirical studies show that high false-positive rates and low per-warning utility suppress sustained use, leading to alert fatigue—especially in security-sensitive domains such as fintech (Johnson et al., 2013). Recent advances in code representation with Transformer-based models indicate that learned signals can complement hand-engineered rules by capturing long-range dependencies and semantics beyond surface syntax (Devlin et al., 2019), and code-specialised pre-training further improves performance on code understanding and defect detection (Feng et al., 2020). Because a single snippet may simultaneously exhibit security risks, maintainability smells, and stylistic issues, multi-label formulations and metrics tailored to overlapping labels are appropriate (Tsoumakas & Katakis, 2007). A practical complication is that learned models emit probabilities requiring thresholding, which—without calibration—can inflate false positives or suppress true positives (Guo et al., 2017).

These considerations motivate a gated-fusion design that preserves the high recall and explainability of rules while using a calibrated classifier to confirm or filter findings. In this setup, rules provide deterministic signals and a lightweight RoBERTa-based multi-label classifier supplies calibrated probabilities that drive per-label gates (Guo et al., 2017). This approach suits engineering teams because it adds a tunable confirm/veto layer without discarding existing rule outputs, and operating points can be selected with ROC-style analysis (Fawcett, 2006). The present work is scoped to Python only and trains/evaluates on snippets curated from popular open-source fintech GitHub repositories released under permissive licences. Permissively licensed code was used to ensure ethical and legal compliance.

Automated evaluations run in offline CI with reason-generation disabled; no third-party API calls are made during assessment.

CI/network controls. To ensure determinism and eliminate external dependencies, all Continuous Integration (CI) jobs run in offline mode. In CI, the reason-generation component is explicitly disabled, no outbound network calls are permitted, and environment variables for external providers (e.g., OPENAI\_API\_KEY, HUGGINGFACE\_HUB\_TOKEN) are not loaded. Local inference paths and cached artefacts are used exclusively, which reproduces the main pipeline without network access. The interactive service can enable reason-generation for manual experiments on a developer workstation, but this is not part of the CI evaluation and is disabled in all automated runs.

## **1.3 Aims and Objectives**

**1.3.1 Aim**This dissertation aims to determine whether a calibrated hybrid static analyser—combining rules with a multi-label RoBERTa classifier via gated fusion—can increase precision without materially reducing recall on fintech-relevant Python code.

### **1.3.2 Success criteria & evaluation metrics**

* **Primary criterion:** relative to a rules-only baseline, achieve ≥10 percentage-point absolute gain in precision with ≤5 percentage-point absolute loss in recall on a held-out, domain-oriented dataset.
* **Secondary criteria:** improved macro-F1; stability under 5-fold cross-validation (standard deviation ≤0.04); and negligible runtime overhead once model scores are available.

### **1.3.3 Objectives**

1. Design and implement the hybrid architecture that aggregates rule signals (AST heuristics and established linters such as SonarLint/Pylint) with a RoBERTa-based multi-label classifier served via FastAPI, with configuration, logging and reproducibility.
2. Build a domain-oriented evaluation set of short Python snippets labelled {sast\_risk, ml\_signal, best\_practice}, including samples from popular open-source fintech projects, with recorded provenance and permissive licensing.
3. Train and calibrate the classifier using BCEWithLogitsLoss and per-label temperature scaling; select thresholds that align scores with observed frequencies.
4. Implement gated fusion with tunable per-label gates; run ablations (Rules, Model, Fusion).
5. Evaluate rigorously using micro/macro Precision/Recall/F1, confusion matrices, ROC curves, and 5-fold cross-validation; report confidence intervals or bootstrap where practical.
6. Analyse failure modes and validity threats, including false positives/negatives, label noise, domain shift and tool availability; discuss engineering trade-offs (latency, memory, explainability, integration).
7. Provide deployment guidance and artefacts, including thresholds, monitoring and retraining cadence, and scripts/configs for full reproducibility.

**Traceability note.** A full objective–evidence mapping (methods, datasets, scripts, figures/tables) is provided in Appendix E (ref. Table E‑1. Objective Traceability) for quick cross-reference to the results that satisfy each objective.

Unless stated otherwise, 95% confidence intervals (CIs) for macro-F1 are computed from k-fold estimates as x̄ ± t\_{k-1,0.975}·s/√k, where x̄ is the fold mean, s the sample standard deviation, and t\_{k-1,0.975} the two-sided t-critical value with k−1 degrees of freedom.

Per-fold macro-F1 values are visualised in Appendix B (Figures B1–B3); the main tables report the fold mean with confidence intervals.

This appendix provides the per-fold macro-F1 trajectories and confusion matrices for each model variant (Figures B1–B3), complementing the aggregated results in Chapter 5.

**1.4 Scope**  
The work is limited to Python and uses code curated from popular open-source fintech GitHub repositories released under permissive licences**.** Ethical and professional considerations such as licence compliance, dataset provenance, and reproducibility are detailed in Chapter 3.

## **1.5 Chapter roadmap**

* **Chapter 1 – Introduction:** Motivation, research question, scope (Python-only), contributions, success criteria (ref. Table E‑1. Objective Traceability), and chapter outline.
* **Chapter 2 – Literature Review:** Rule-based static analysis, ML for code, multi-label learning, probability calibration, and fusion strategies; identifies gaps this study addresses.
* **Chapter 3 – Methodology:**
  + **Research Design:** comparative evaluation of Rules vs Model vs Gated Fusion; datasets, splits (incl. 5-fold), variables, and analysis plan.
  + **Data provenance & licensing:** GitHub selection, permissive licences; domain relevance to fintech.
  + **Ethical & professional considerations:** responsible use, IP/licensing compliance, reproducibility.
  + **Labelling scheme:** {sast\_risk, ml\_signal, best\_practice}, guidance and QA.
  + **Rule set & tools:** AST heuristics; SonarLint/Pylint configuration.
  + **Model & training:** RoBERTa-base, loss, hyperparameters.
  + **Calibration & thresholds:** temperature scaling, threshold sweeps.
  + **Gated-fusion policy:** decision logic and defaults.
  + **Evaluation protocol:** micro/macro P/R/F1, confusion matrices, ROC, runtime; report CIs/variance where practical.
* **Chapter 4 – Implementation:** FastAPI microservice, model serving, rule engine, SonarLint/Pylint integration, configuration, tooling; system architecture diagram (ref Figure F‑1. System architecture (hybrid rules + model + gated fusion))
* **Chapter 5 – Results:** Metrics for Rules/Model/Fusion, confusion matrices, ROC curves, threshold sweeps, 5-fold CV with variance, and runtime analysis.
* **Chapter 6 – Discussion:** Interpretation, ablations, failure modes, threats to validity (internal/external/construct), comparison with prior work, and implications for CI/CD adoption.

The gated fusion model’s macro-F1 remains superior to either component alone, with non-overlapping CIs against the rules baseline and a small but consistent margin over the model-only variant, indicating a real (though modest) improvement rather than fold noise.

External validity. To gauge generalisability beyond the curated dataset, a practical extension is a hold-out repository test from the same financial domain but unseen during development (e.g., a fintech OSS service with a comparable Python stack). We would (i) sample ~200–300 files across modules, (ii) run the frozen pipeline end-to-end, and (iii) compare macro-F1/precision/recall to in-sample results, reporting a generalisation gap (Δ macro-F1). If a material gap appears (>2–3 points), we would perform a lightweight calibration refresh: fit temperature scaling and decision thresholds on a small labelled slice of the new repo (≈10% of files), then re-evaluate on the remaining 90%. We would also report calibration metrics (ECE/Brier) before/after refresh to show whether miscalibration, rather than model mismatch, drives any degradation. This plan is low-cost, respects licensing constraints, and yields an actionable deployment recipe: evaluate frozen, refresh calibration only if needed, and document Δ and ECE as part of onboarding a new codebase.

* **Chapter 7 – Conclusion & Future Work:** Findings, limitations (Python-only, threshold tuning), and extensions (other languages, uncertainty-aware explanations).
* **References**
* **Appendices:** Success-criteria table; system architecture; plots (confusion matrices, ROC, threshold sweeps); dataset schema/provenance; any ethics documents retained.

## **1.6 Alignment with MSc outcomes**

This project demonstrates advanced research design and implementation via a comparative evaluation of rules, model, and gated fusion; it evidences critical analysis through method selection, calibration, and ablation; it addresses legal/ethical/professional considerations (licensing, reproducibility, responsible use) and provides an original engineering contribution (a calibrated fusion policy and microservice integration) consistent with the MSc programme learning outcomes.

# **Chapter 2 - Literature Review**

### **2.1 Overview and scope**

Static analysis aims to detect defects, vulnerabilities, and maintainability issues without executing code. The promise is early detection and reduced downstream cost, yet long-standing adoption challenges persist—especially high false-positive rates, uneven actionable value per warning, and workflow friction (Bessey et al., 2010; Johnson et al., 2013; Sadowski et al., 2018). In parallel, advances in machine learning (ML) for code suggest learned models can complement rules by modelling context that exceeds syntactic patterns (Allamanis et al., 2018; Liu et al., 2019; Feng et al., 2020). This review synthesises five strands essential to a hybrid approach: (i) rule-based static analysis, (ii) ML for code, (iii) multi-label learning, (iv) probability calibration, and (v) fusion strategies. Throughout, implications are framed for a fintech setting where data sensitivity and regulatory expectations raise the cost of both missed issues and false alarms.

Scope & provenance. This project targets Python. Data are composed of short, curated snippets plus popular open-source fintech GitHub repositories under permissive licences; provenance and licensing records are maintained to support reproducibility and ethical reuse.

### **2.2 Rule-based static analysis: strengths, limits, and lessons from industry**

#### **2.2.1 Capabilities and engineering trade-offs**

Rule-based static analysis encodes bug patterns, code smells, and security checks as declarative rules or dataflow analyses. Mature tools (e.g., FindBugs/SpotBugs) and industrial frameworks at Google and Facebook have demonstrated real-world defect discovery at scale (Ayewah et al., 2008; Sadowski et al., 2018). Strengths include explainability (a concrete rule fires for a concrete reason), determinism, and typically low per-file latency compared with heavyweight interprocedural analyses. Security-oriented static application security testing (SAST) often augments AST checks with taint analysis to track untrusted sources to sinks; however, interprocedural taint is notoriously hard to scale without over- or under-approximation (Livshits et al., 2015).

A fundamental tension is the soundness vs. completeness trade-off: stricter analyses reduce false negatives but tend to increase false positives or require annotations developers rarely provide (Livshits et al., 2015). Many rules operate locally (AST or intra-procedural), and so they miss context spanning call graphs, configuration, and data provenance. Configuration burden—suppressions, baselines, severity tuning—accumulates cost (Bessey et al., 2010), and misconfigured rulesets can swamp teams with non-actionable warnings.

#### **2.2.2 Empirical evidence on developer uptake**

Empirical studies consistently report warning overload and limited trust as barriers to sustained use (Johnson et al., 2013). Developers prioritise a signal that is fix-ready, specific, and actionable; generic or noisy warnings are often ignored (Christakis & Bird, 2016). Google’s experience suggests success hinges on integrating checks into developer workflows with triage, ownership, and feedback loops (Sadowski et al., 2018). In security-sensitive domains (e.g., fintech), alert fatigue is particularly harmful: teams either ignore a large fraction of warnings or disable noisy rules, reducing coverage precisely where it is needed most.

Implication → Method (§3). A hybrid approach that retains rules’ explainability but filters or confirms them with learned context targets the core pain-point—precision—without discarding deterministic signals. We therefore introduce a per-label “gate” that only elevates rule findings (or model findings) when calibrated probabilities clear tuned thresholds.

#### **2.2.3 Techniques to reduce false positives (and why they fall short)**

Industrial practice employs several mitigations:

Baselining (accept current warnings as a baseline and alert only on deltas) tames volume but allows legacy issues to linger and offers no semantic filtering.

Ownership/triage workflows reduce stray warnings but incur ongoing human cost (Sadowski et al., 2018).

Heuristic ranking (e.g., by code churn, past fix history) helps prioritise, yet the underlying precision remains unchanged (Rahman & Devanbu, 2013).

Hardening rules (extra guards to reduce spurious triggers) can inadvertently suppress true positives or narrow applicability.

Why hybridisation? The above reduce symptoms, not causes. A learned model can supply contextual evidence—lexical, structural, and semantic—that is difficult to encode as static rules, offering principled support for confirm/veto decisions.

### **2.3 Machine learning for code: from representations to defect signals**

#### **2.3.1 Representation learning for source code**

Neural models treat code as both text and structure. Transformer encoders (BERT, RoBERTa) supply strong token-level representations (Devlin et al., 2019; Liu et al., 2019). Code-specialised pre-training incorporates identifiers, comments, and bimodal tasks (NL↔code) and yields consistent gains across code understanding tasks (Feng et al., 2020; Kanade et al., 2020). Beyond sequence models, graph and path-aware approaches capture AST and flow relationships that plain token models miss (Wang et al., 2021). A broad survey documents these trends and cautions about dataset artefacts and label quality (Allamanis et al., 2018).

#### **2.3.2 Learning signals relevant to static analysis**

Vulnerability detection, security pattern recognition, and code-smell classification have seen promising—but variable—results (Russell et al., 2018; Wang et al., 2021). Gains typically arise from contextual cues (e.g., how strings are composed before reaching sensitive APIs; authentication branches; error-handling idioms). However, label noise (when labels derive from tools or weak heuristics) can entrench the very biases hybrid systems aim to correct (Heckman & Williams, 2011), and domain shift (training projects vs. target codebases) is a persistent threat to external validity (Rahman & Devanbu, 2013).

Fintech relevance. Domain-specific idioms (payment flows, crypto handling) and libraries differ from generic corpora. We therefore curate a fintech-oriented subset of permissively licensed repositories and short snippets to reduce shift and align signals with the target domain (§3).

#### **2.3.3 Practical constraints and explainability**

Compared to rules, ML inference adds latency and operational complexity (serving, drift monitoring, retraining). Explainability remains harder: feature attributions (e.g., token importance) help but rarely reach the concise clarity of rule messages. Consequently, this work keeps the model lightweight (RoBERTa-base, multi-label head) and serves it behind a narrow FastAPI interface. In the UI, rule reasons remain first-class and all reasons per line are shown; the model acts as a confirm/veto signal rather than replacing explanations (§4).

#### **2.3.4 Domain adaptation and dataset curation**

To counter domain shift, two pragmatic levers recur in the literature: (i) domain-specific sampling (select projects representative of the target setting) and (ii) task-aligned supervision (labels that reflect operational categories). Both apply here: fintech-weighted sampling plus the triad {sast\_risk, ml\_signal, best\_practice}. Curation pitfalls include duplicate snippets, near-duplicates across splits, and ambiguous licences; we mitigate these with provenance tracking, permissive-licence filtering, and balanced label counts (§3).

While code-pretrained Transformers and graph models report strong gains on curated, controlled splits (e.g., CodeBERT-style encoders and GNN variants), evidence on external validity cautions that performance can decay under repository/process drift and label noise—precisely the conditions in real fintech codebases. Consequence for fintech: we treat model scores as evidence to be calibrated and thresholded, not ground truth, and we bias data sampling toward fintech-proximal repositories to reduce shift (see §3).

Rules offer line-level, inspectable rationales; model attributions are helpful but typically less decisive for reviewers. Consequence for fintech: we keep rule reasons primary in the UI and deploy ML as a confirm/veto signal behind a narrow service boundary to control operational risk (see §4).

Recent benchmark work on code intelligence corroborates this pattern of gains alongside sensitivity to dataset design and splits (Lu et al., 2021).

Implication → Method (§3–4). We use a multi-label RoBERTa-base with BCE loss, trained on fintech-oriented data, and expose it as a microservice to isolate operational risk while preserving explainability through rules.

Operationally, we implement this as a lightweight RoBERTa-base multi-label head served via FastAPI, used only to confirm or veto rule signals and to add labels the rules miss (see §3–4).

### **2.4 Multi-label learning for overlapping code concerns**

Static analysis findings often overlap: a single snippet may present a security risk, a maintainability smell, and style issues. Multi-label learning explicitly models such overlap, producing a label vector rather than a single class (Tsoumakas & Katakis, 2007; Zhang & Zhou, 2014). A simple and effective baseline is Binary Relevance (independent logits per label) trained with BCEWithLogitsLoss. More expressive methods (classifier chains, label-graph models) can capture inter-label dependencies but add complexity that is not always justified relative to a strong shared encoder.

#### **2.4.1 Evaluation metrics and pitfalls**

Evaluation should reflect both overall and per-label performance.

Micro-averaged precision/recall/F1 pools TP/FP/FN across labels; it emphasises frequent labels.

Macro-averaged metrics treat labels uniformly and are sensitive to low-support categories.

Operationally, we treat each label as an independent logit (Binary Relevance with BCE) and report micro/macro P/R/F1 plus per-label F1, with multilabel-stratified 5-fold evaluation to reflect overlap and imbalance (see §3, §5).

Example-based F1 and Hamming loss capture per-snippet correctness but are harder to interpret for stakeholders (Tsoumakas & Katakis, 2007).

Subset accuracy (exact match of the entire label set) is overly strict for noisy, overlapping tasks.

Reporting both micro and macro scores, plus per-label F1, is therefore recommended—particularly when class supports differ (Zhang & Zhou, 2014). Stratified evaluation matters: k-fold splits should preserve multi-label frequencies to reduce variance; multilabel-stratified k-folds are preferable where possible (Kohavi, 1995).

Under distribution shift, calibration properties can degrade; recent analyses emphasise per-class or label-wise treatment and careful validation (Minderer et al., 2021).

Implication → Method (§3, §5). We adopt Binary Relevance with BCE, report micro/macro P/R/F1 and per-label F1, and use 5-fold evaluation with stratification where feasible.

Operationally, we fit per-label temperature scaling on a validation split, then select fixed thresholds via PR sweeps to meet explicit precision/recall targets before testing (see §3, §5).

### **2.5 Probability calibration: making scores usable in gates**

Modern neural networks are often over-confident (Guo et al., 2017). Without calibration, fixed thresholds can be unstable across datasets, inflating false positives or suppressing true positives. Temperature scaling—a single scalar applied to logits—is a simple post-hoc method that reliably improves calibration on held-out data (Guo et al., 2017). Alternatives such as Platt scaling and isotonic regression can be superior in some regimes but risk overfitting with limited validation data (Platt, 1999; Zadrozny & Elkan, 2002). In multi-label settings, per-label calibration is typically needed because the marginals differ; a global temperature may under- or over-correct certain labels (Kull et al., 2017).

#### **2.5.1 Measuring calibration and selecting thresholds**

Calibration can be quantified via Expected Calibration Error and Brier score; however, operationally the key question is which thresholds achieve the desired precision–recall trade-off. Thresholds should be selected on a validation split via ROC/PR sweeps and then fixed for test/evaluation to avoid optimistic bias (Fawcett, 2006). In regulated contexts (e.g., fintech), one might also impose cost-sensitive decision rules (e.g., penalising false negatives for certain risks more than false positives).

Implication → Method (§3, §5). We apply per-label temperature scaling, then sweep thresholds on a validation split to select operating points that meet explicit precision and recall targets before evaluation.

### **2.6 Fusion strategies: combining deterministic rules with probabilistic models**

Operationally, we implement a per-label, threshold-based gate over calibrated probabilities, with ablations for Rules-only, Model-only, and Fusion to quantify marginal utility (see §3–5).

#### **2.6.1 Early vs. late fusion**

“Early fusion” injects model features into rule engines or co-trains with program analysis features; “late fusion” combines decisions (or calibrated probabilities) from separate modules (Kuncheva, 2004). For safety-critical domains, late, decision-level fusion is attractive because each component remains independently testable and degradable. It also allows organisations to preserve investments in existing rules while adding a statistically principled layer.

#### **2.6.2 Gated decision policies**

A gated policy elevates a label if: (i) rules signal it and the model does not veto (model score below a reject threshold), or (ii) the model strongly supports it even when rules are silent. Such policies can be formalised in ROC space and tuned to maximise F1, minimise expected cost, or satisfy domain-specific constraints (Fawcett, 2006). When rules have high recall but low precision (as seen in multiple industrial reports), a calibrated gate typically raises precision with a bounded recall penalty.

#### **2.6.3 Alternatives to gating (and why not chosen)**

Other late-fusion methods include weighted voting, stacking (meta-learners over component outputs), and belief-based schemes (e.g., Dempster–Shafer). These can outperform simple gates in some settings but at the cost of additional supervision, complexity, and explainability. In regulated or safety-conscious settings, simplicity and testability weigh heavily. A gate with per-label thresholds is auditable, tunable, and maps cleanly to operational constraints.

Implication → Method (§3–5). We implement a per-label, threshold-based gated fusion that preserves rule determinism while letting the model confirm or add findings. Ablations (Rules vs. Model vs. Fusion) quantify each component’s marginal utility.

### **2.7 Critique and gaps in the literature (and how this study responds)**

Precision under operational constraints. Many ML-for-code papers show gains on balanced/curated benchmarks, but fewer report operational precision once integrated with noisy, real-world rules (Allamanis et al., 2018; Russell et al., 2018).  
Gap → Method. We evaluate Rules vs. Model vs. Fusion on a balanced, domain-oriented set and use calibrated gates to target precision gains at bounded recall loss (§5).

Calibration for multi-label code under domain shift. Calibration evidence is strong for image/NLP classification (Guo et al., 2017; Kull et al., 2017) but thinner for multi-label code tasks affected by repository/domain drift.  
Gap → Method. We adopt per-label temperature scaling plus threshold sweeps, and we report ROC-style curves to make operating points explicit (§3, §5).

Data provenance and licensing. Public code corpora often mix licences or omit commit-level provenance, complicating ethical reuse and reproducibility (Kocetkov et al., 2022).  
Gap → Method. We restrict to permissively licensed fintech repositories with recorded provenance and supplement with short, synthetic snippets to control confounds (§3).

Explainability and developer trust. ML explanations rarely match rule messages for line-level specificity; developers value justifications tied to code locations and recognised patterns (Christakis & Bird, 2016).  
Gap → Method. We preserve rule messages as first-class reasons and display all reasons per line; the model acts as a confirm/veto signal rather than replacing explanations (§4).

Evaluation breadth and stability. Many studies emphasise accuracy; fewer report micro/macro P/R/F1, confusion matrices, and k-fold stability, which matter when label distributions vary (Tsoumakas & Katakis, 2007; Fawcett, 2006; Kohavi, 1995).  
Gap → Method. We standardise on micro/macro P/R/F1, per-label F1, ROC, threshold sweeps, and 5-fold estimates (§5).

From offline metrics to developer outcomes. Few papers quantify how hybrid systems change fix rates or time-to-triage; most rely on offline metrics.  
Gap → Method. While developer-in-the-loop trials are out of scope, we report metrics that best correlate with actionable precision and outline how to extend to user studies (§6).

### **2.8 Limitations of the current evidence base**

Label quality and construct validity. Many vulnerability/smell datasets derive labels from tools or weak heuristics; these may entrench rule biases and inflate agreement with rule-like models (Heckman & Williams, 2011). Human-validated, multi-label gold standards remain scarce.

Domain shift and repository bias. Benchmarks often focus on a few ecosystems or large projects, overstating generalisation to fintech-specific idioms and libraries (Rahman & Devanbu, 2013). Open-source finance code may not reflect regulated enterprise codebases.

Reproducibility and licensing. Some datasets lack precise commit hashes, dependency states, or licence filters; this complicates replication and ethical reuse (Kocetkov et al., 2022). Even when licences are permissive, provenance gaps impede auditability.

Calibrated multi-label evidence. Results from image/NLP do not automatically transfer to multi-label code tasks with imbalanced marginals; robust evidence on per-label calibration efficacy is limited (Kull et al., 2017).

User-centred outcomes. There is limited evidence on how hybrid systems impact developer behaviour (e.g., reduced triage time, increased fix rates) compared with rules alone (Christakis & Bird, 2016; Sadowski et al., 2018).

Consequence for this study. We mitigate these constraints through transparent data curation/licensing, balanced labels, per-label calibration, and multi-metric reporting; nevertheless, we acknowledge residual threats to validity, analysed in §6.

### **2.9 Summary and implications for this study**

The literature paints a consistent picture: rules provide explainable, high-recall signals but often suffer low precision; learned models add context sensitivity yet require calibration and domain-appropriate data. Multi-label framing matches the overlapping nature of static analysis findings, and decision-level fusion—especially gated policies—offers a practical path to combine strengths while containing operational risk.

These insights motivate the following design choices:

Hybrid architecture. Preserve deterministic rule outputs and explanations; add a lightweight RoBERTa-base multi-label classifier to confirm/veto and augment labels.

Domain-grounded data. Train/evaluate on short Python snippets plus permissively licensed fintech repositories to align with target idioms.

Per-label calibration and thresholding. Apply temperature scaling and threshold sweeps to select operating points that meet explicit precision–recall goals.

Transparent evaluation. Report micro/macro P/R/F1, per-label F1, confusion matrices, ROC curves, threshold sweeps, and 5-fold stability to satisfy methodological expectations for fairness and reproducibility.

Method-facing signposts. Operationally, these decisions materialise in Chapter 3 (Methodology) and Chapter 4 (Implementation): §3 details data curation/licensing, model training and per-label temperature scaling, and the gated fusion policy; §4 documents the FastAPI microservice and rule-engine integration. The chosen metrics and validation strategy used in Chapter 5 (Results) directly follow from the literature’s guidance on multi-label evaluation and calibrated decision-making (Tsoumakas & Katakis, 2007; Fawcett, 2006; Guo et al., 2017; Kohavi, 1995).

**Chapter 3 – Methodology, Research Design and Ethical and Professional Considerations**

**3.1 Overview and aims**

This chapter operationalises the design choices motivated in the literature review (§2.3–§2.6). The study compares three analysis conditions for Python code: (i) **Rules** (AST heuristics and external static-analysis tools), (ii) **Model** (a RoBERTa-base multi-label classifier), and (iii) **Gated Fusion** (a calibrated, per-label decision policy that combines Rules and Model). I describe dataset construction (241 balanced snippets and a multi-label scheme), splitting and evaluation protocols (multi-label-stratified 5-fold cross-validation with an inner calibration split), model architecture and training (tokenizer, BCEWithLogits loss), calibration and thresholding (per-label temperature scaling and PR-based threshold sweeps), the rule engine (AST heuristics + SonarLint CLI/Pylint), and the metrics/analysis plan. I conclude with the ethical and professional controls required to work with open-source code (licensing, bias/false positives risk, secure handling, and reproducibility), acknowledging that ethics apply even without human participants.

**3.2 Research design**

**3.2.1 Comparative conditions and variables**

The **independent variable** is the analysis strategy with three levels:

* **Rules-only**: Findings from AST heuristics and external tools are taken as final decisions.
* **Model-only**: Decisions are derived purely from the model’s calibrated probabilities and per-label thresholds.
* **Gated Fusion**: Final decisions are made by a per-label gate that uses (a) rule signals, (b) calibrated model probabilities, and (c) label-specific operating thresholds (see §3.8).

**Dependent variables** are performance measures (micro/macro precision, recall, F1; per-label F1), operating characteristics (ROC/PR), error structure (per-label confusion matrices), and **runtime** (throughput per snippet and wall-clock time). I also record **variance** across folds and report mean ± s.d.; where practical, I include 95% confidence intervals via non-parametric bootstrap on fold scores.

Control settings include the code normalisation pipeline, tokenisation limits, random initialisation seeds, and fixed rule configurations. Ablations include **Rules-only**, **Model-only**, and **Fusion** to quantify marginal utility (as advocated in §2.6).

**3.2.2 Dataset construction (241 balanced snippets)**

I assembled **241 Python snippets** representing typical security, quality and style concerns seen in fintech-adjacent repositories. Snippets were curated to avoid near-duplicates, and to represent a range of idioms (I/O, data parsing, crypto use, web handlers, configuration). The corpus is **multi-label**: each snippet may simultaneously carry multiple concerns. I targeted **approximate balance across labels** by stratified sampling during construction (see §3.4). Each snippet contains between 12 and ~200 lines (median ≈ 60), trimmed to fit within the model’s maximum token length after tokenisation.

Pre-processing removes comments that carry obvious repository-identifying information, normalises whitespace, and redacts literal secrets if present. File-level metadata (path, repo origin, licence) is retained in a manifest but is never provided to the model.

**3.2.3 Splits and protocol**

I evaluate under **multi-label-stratified 5-fold cross-validation** so that each fold preserves label prevalences and co-occurrence patterns. For fold *k*:

* The **test set** is the *k*-th fold.
* The **training set** is the union of the remaining four folds.
* From the training set, I carve out a **calibration split** (20% of the training fold, multi-label-stratified) used solely for **temperature scaling** and **threshold selection** (§3.7). The model’s weights are *not* updated on the calibration split.

This yields five unbiased test evaluations; reported numbers are the mean across folds with dispersion statistics. To reduce randomness, I fix a global seed for fold assignment and initialisation. All folds share the same **rule configurations**, ensuring that differences arise from decision policies rather than moving targets.

**3.3 Data provenance and licensing**

In line with the ethical bridge in §2.7/§2.8, I restrict source repositories to **permissive licences** (MIT, Apache-2.0, BSD-2/3-Clause) and record provenance (repo URL, commit hash, file path, licence file snapshot) in a manifest. I favour repositories with domain proximity to fintech (e.g., payment processing utilities, data validation libraries, API clients), identified via topic tags and README keywords.

To respect authorship and minimise redistribution risk, the dataset is used **internally** for evaluation; if an external release is desired, I provide (i) the **manifest** (licences, exact locations) and (ii) a reproducible **harvesting script** so that others can reconstruct the corpus from its sources rather than receiving redistributed code. Where a repository’s licence is ambiguous or non-permissive, files are excluded. Any snippet that appears to contain credentials or sensitive operational details is removed or redacted before inclusion (§3.10).

**3.4 Labelling scheme and quality assurance**

Labels capture complementary aspects surfaced in §2.3–§2.4:

* **sast\_risk**: A pattern that static-analysis practice typically treats as a **potential vulnerability or security risk** (e.g., unsafe deserialisation, insecure randomness, command construction with user input).
* **ml\_signal**: A **statistically learnable pattern** linked to problems that models often detect (e.g., risky API usage context, suspicious data-flow hints), which may not be codified as a rule. This label acknowledges that learned signals can complement rules (§2.3).
* **best\_practice**: **Quality/maintainability** issues that influence operational risk or review burden (e.g., broad exceptions, insecure defaults, hard-coded configuration) that are valuable to flag in CI.

Each snippet is considered independently. Labelling guidance defines positive examples and counter-examples; ambiguous cases are resolved using a **conservative policy** (only label positive when the intent is clear). Quality assurance consists of a second-pass review to check consistency, plus a rule-assisted sanity check: if a snippet triggers a rule for, say, insecure randomness, I verify that **sast\_risk** is set appropriately. Inter-annotator agreement is planned for future scaling; for this study, I rely on the guidance and the second-pass QA to maintain consistency.

**3.5 Rule set and tools**

The **Rules** condition aggregates findings from:

1. **AST heuristics**: Lightweight Python visitors that match structural patterns (e.g., ast.Call to random.random() in cryptographic contexts; subprocess.Popen with shell=True; pickle.loads over tainted inputs; exception handlers catching Exception with no action). Heuristics include limited **data-flow checks** (argument origin, literal vs variable, membership in allow-lists) to reduce trivial false positives.
2. **External tools**: **SonarLint CLI** and **Pylint**, executed with pinned versions in a containerised environment. I enabled rulesets aligned with security and reliability and **normalise severities** to a common scale.
3. **Normalisation and de-duplication**: Findings are merged by (file, line span, rule ID). Overlapping or near-duplicate findings are coalesced into a single alert per label category. Each finding maps to one or more of the study labels via a simple rule-to-label table (e.g., insecure randomness → sast\_risk; broad except → best\_practice).

The Rules pipeline outputs **per-label binary signals** (present/absent) and retains tool metadata (rule IDs, messages) for explainability.

**3.6 Model and training**

The **Model** condition uses a **RoBERTa-base** encoder with a **multi-label classification head** (three logits). Input consists of the snippet’s text, truncated/padded to the model’s maximum sequence length. I use the **RoBERTa tokenizer** with standard normalisation; no repository metadata is included.

**Loss and optimisation.** I use **BCEWithLogits** over the three labels. Let and be the logit for label . The loss is

I train with **AdamW**, fixed learning rate (e.g., 2e-5), weight decay (e.g., 0.01), batch size sized to GPU memory (or CPU if applicable), and **early stopping** on the training fold using the calibration split as a **stop-criterion guard** to discourage overfitting to the training split. Label imbalance is modest by design; if any fold shows skew, I allow **per-label class weights** in the loss (default 1.0).

**Regularisation.** I applied dropout in the classifier head and gradient-clipping. I fixed random seeds and log versions (transformer library, tokeniser, Python, OS) to support reproducibility (§3.10).

**Outputs.** The model produces **per-label probabilities** after per-label **temperature scaling** ​ (§3.7).

**3.7 Calibration and threshold selection**

As argued in §2.5, raw scores are poorly calibrated and calibration can drift under shift. I therefore applied **per-label temperature scaling** learned on the **calibration split** of each fold. Temperatures are optimised to minimise negative log-likelihood on the calibration split **without** changing the logits otherwise.

After calibration, I choose **per-label decision thresholds** ​ by sweeping thresholds on the calibration split to meet **explicit operating targets** (e.g., prioritising higher precision for sast\_risk to reduce false alarms, balanced F1 for best\_practice). Concretely, for each label, I compute the PR curve and select ​ that maximises F1 **subject to** a minimum precision constraint if specified. These thresholds are then **frozen** and applied to the unseen test fold. I report both the **constrained-F1** operating point and the **unconstrained best-F1** point as a sensitivity check.

**3.8 Gated-fusion decision policy**

Fusion reflects the literature’s recommendation to combine explainable rule signals with calibrated model probabilities (§2.6). I implemented a **per-label gate**:

* If a rule fires for label , emit a **positive** only if the calibrated model probability . (The model **confirms** the rule.)
* If **no rule** fires for , emit a **positive** if ​ and the model’s **margin** over the next alternative is above a small safety margin (default =0.05). This allows the model to **add** findings that rules might miss, while avoiding low-confidence drifts.
* If a rule fires but the model probability is **very low** (e.g., ), I **suppress** the alert as likely spurious (a **veto**). The veto path is logged for audit.

This policy yields **three interpretable regimes**: *confirm*, *add*, and *veto*, each auditable with reasons (rule ID, probability, threshold, margin). Defaults are =0.05,=0.15; ablations in §5 examine sensitivity.

**3.9 Evaluation protocol**

**3.9.1 Metrics**

I report:

* **Micro-averaged precision/recall/F1** over all labels and instances.
* **Macro-averaged precision/recall/F1** (unweighted mean across labels).
* **Per-label F1**, useful for operational tuning (e.g., stricter thresholds for sast\_risk).
* **ROC and PR curves** per label (summarised by AUC where informative).
* **Confusion matrices** per label (TP/FP/FN/TN), supporting error analysis.
* **Runtime**: average and percentile inference time per snippet for Rules, Model, and Fusion; wall-clock cost per 1000 lines.

**3.9.2 Procedure**

For each fold:

1. Fit the model on the training subset.
2. Learn temperatures and thresholds on the calibration split (§3.7).
3. Evaluate on the **held-out test fold** for the three conditions: Rules-only, Model-only (with per-label thresholds), and Gated Fusion (with the policy in §3.8).

I aggregated fold metrics via mean ± s.d.; where practical, I computed **95% bootstrap CIs** by resampling the test predictions within each fold. To guard against over-interpretation of small differences, I considered an improvement **material** if the micro-F1 CI does not overlap and the **per-label precision at the chosen operating point** improves for sast\_risk (the most risk-sensitive label).

**3.9.3 Error analysis and ablations**

I qualitatively inspected the top **false positives** and **false negatives** per label for each condition, tagging common causes (e.g., ambiguous API context, comments misleading patterns). Ablations include removing the veto path, removing the “add” path, and using a **single global temperature** to confirm the value of per-label calibration (§2.5). I also compared the **Rules volume** before vs after fusion to measure alert reduction without sacrificing precision.

**3.10 Ethical & professional considerations**

Although the study has no human participants, it engages with **intellectual property**, **security-relevant content**, and automation risks; it therefore requires explicit ethical controls.

**3.10.1 Licensing and IP compliance**

Only repositories with **permissive licences** (MIT/Apache-2.0/BSD) are included. For each file, I recorded a provenance tuple (repo, commit, path, licence snapshot). The dataset is **not redistributed** as raw code; instead, I published the **manifest** and a **reconstruction script** (subject to the same licence checks). Any snippet containing possible credentials, API tokens or operational secrets is redacted or removed. If a licence is later changed upstream, the manifest allows downstream users to respect the updated terms.

**3.10.2 Secure handling and data protection**

All code is stored on encrypted drives with restricted access; only the minimal subset required for experiments is retained. I avoided storing entire repositories where single files suffice, minimising exposure. Logs exclude raw code except for short, non-sensitive spans necessary for debugging; I provided hashed identifiers for cross-reference. Where CI environments are used, I forbid outbound network calls during analysis and pin tool versions to reduce supply-chain risk.

**3.10.3 Bias and false-positive risk**

Static analysis and ML can **over-flag** patterns in particular idioms, libraries, or frameworks, imposing review burden and potential production friction. The design mitigates this by (i) **domain-proximal sampling** of fintech-relevant code (§2.3), (ii) **per-label calibration** and **precision-oriented thresholds** for sast\_risk, and (iii) the **veto path** in fusion for low-probability rule hits. I explicitly reported **alert volume** and **precision at the chosen operating points**, enabling stakeholders to understand trade-offs. I avoid auto-remediation; all outputs are **advisory** and intended for human-in-the-loop review.

**3.10.4 Responsible disclosure**

If, during curation or analysis, I encounter a **likely live vulnerability** in an active repository, I follow a minimal-impact path: do not publish details, and if appropriate, notify maintainers privately following common disclosure timelines. Our snippets are de-contextualised where possible to reduce re-identification risk.

**3.10.5 Reproducibility and transparency**

I published the following artefacts:

* **Environment**: exact package versions, OS, and hardware notes; container definitions for Rules and Model.
* **Seeds and splits**: the fold assignments and calibration indices to allow exact reruns.
* **Configurations**: rule enablement lists, severity mappings, model hyperparameters, and threshold/temperature values per fold and per label.
* **Pipelines**: scripts that execute training, calibration, evaluation and report generation end-to-end.

Where redistribution of code snippets is constrained by licence, the **manifest + script** approach ensures that independent researchers can reconstruct the evaluation set under the same conditions.

**3.11 Implementation details**

**3.11.1 Software and hardware**

Experiments run on a workstation-class CPU/GPU (or CPU-only if necessary). The rule engine and external tools are executed inside containers with **pinned versions**; the model code uses a fixed transformer library version. I capped sequence length to the model’s maximum and applied truncation heuristics that retain function heads and call sites first. All scripts are idempotent and log run configurations.

**3.11.2 Runtime measurement**

I measured runtime as the **median wall-clock time per snippet** and **per 1000 lines**, separately for Rules, Model, and Fusion, on an otherwise idle machine. To reduce noise, I warmed up the model and took the average over multiple passes with the same random seed.

**3.11.3 Decision logging**

For auditability, the Fusion policy logs, per label: rule signals (tool and rule ID), , temperature , and which branch (confirm/add/veto) was taken. This supports later analysis and aligns with the professional expectation of explainability in regulated environments.

**3.12 Threats to validity**

* **Internal validity.** With 241 snippets, estimates have finite variance. I mitigated overfitting by using 5-fold CV, an inner calibration split, and frozen thresholds per fold. I avoid peeking at the test fold when choosing thresholds.
* **Construct validity.** Labels abstract complex review judgements into three categories; some boundary cases remain. The guidance and QA pass reduce inconsistency. Future work should include multi-annotator adjudication and report inter-rater reliability.
* **External validity.** Results are derived from permissively licensed, fintech-adjacent repositories and may not fully capture internal enterprise code. Domain-proximal sampling and explicit calibration address this partially; deploying to new contexts should recalibrate thresholds with local validation data (§2.3, §2.5).
* **Conclusion validity.** I report dispersion across folds and, where feasible, CIs. Small absolute differences are treated cautiously; I prefer interpretable gains (e.g., higher precision at the chosen operating points and reduced alert volume) over marginal F1 changes.

**3.13 Summary**

This chapter translates the literature-motivated principles into a concrete, auditable methodology. I compare Rules, Model and Gated Fusion on a curated, multi-label corpus of 241 Python snippets under multi-label-stratified 5-fold CV, with **per-label calibration** and **threshold selection** to control operating points. The **fusion gate** is intentionally simple and inspectable (confirm/add/veto). Evaluation emphasises both aggregate scores and operational measures (precision at target thresholds, alert volume, runtime). Ethical and professional safeguards (licensing, secure handling, bias controls, reproducibility) are embedded throughout. The next chapter (§4/§5) reports results and ablations, and examines how calibrated fusion affects developer-relevant outcomes in a fintech-style setting.

**Chapter 4 – Implementation**

**4.1 Overview**

This chapter describes how I implemented the hybrid analysis system proposed in Chapter 3: a FastAPI model microservice, a rule engine integrating SonarLint and Pylint, and a gated-fusion layer that reconciles calibrated model probabilities with rule signals. I also document the training/inference pipelines, configuration strategy, reproducibility steps, deployment notes, and the interfaces used by the GUI/annotation path. The overall architecture is shown in Figure F1 (System architecture: hybrid rules + model + gated fusion), which the text cross-references where relevant.

**4.2 System architecture**

Figure F1 presents a service-oriented layout:

1) Client/GUI layer (desktop IDE plugin / CLI): sends code snippets or file paths to the analysis API and renders findings with explanations and confidence cues.

2) Analysis API (FastAPI): exposes versioned endpoints for health, prediction, and configuration. It orchestrates the rule engine (AST heuristics + SonarLint CLI + Pylint), the model server (RoBERTa-base, per-label temperature scaling and thresholds), and the gated-fusion policy (confirm/add/veto with tunables δ and γ).

3) Artifacts & config store: model weights, tokenizer, label map, temperatures Tℓ, thresholds θℓ, rule→label mapping, and tool configurations (YAML).

4) Observability: structured JSON logs for requests, timings, and decisions; optional metrics emitter for latency/throughput.

Data flow: the API receives a snippet → runs Rules and Model in parallel → fuses results → returns per-label decisions with evidence (rule IDs, calibrated probabilities, thresholds, branch taken). This mirrors the evaluation regimes in §3 (Rules-only, Model-only, Fusion) but defaults to Fusion in the product path.

**4.3 FastAPI model microservice**

**4.3.1 Endpoints**

• GET /healthz — liveness/readiness probe (includes model load state and tool discovery).

• GET /version — model hash, ruleset versions, tool versions, and config checksum (see Appendix C Table C.1: Environment and tool versions).

• POST /predict — body contains either { "code": "…", "language": "python" } for raw snippet analysis, or { "path": "…/file.py" } for file loading (server must have access). Optional flag return\_intermediate=true includes rule hits and raw probabilities.

• POST /batch — array of snippets/paths; streams progressive results and aggregates runtime.

**4.3.2 Lifecycle, credentials, and concurrency**

On startup the service: (1) loads tokenizer + RoBERTa-base weights; (2) loads per-label temperatures Tℓ and thresholds θℓ; (3) warms the model with a short dummy pass; (4) discovers external tools and validates pinned versions (see Appendix C Table C.1).

Credentials ARE required for analysis: the service reads an environment file (e.g., .env) at startup. OPENAI\_API\_KEY and OPENAI\_MODEL are used to generate short, natural-language reasons attached to each annotation (the model microservice remains local; OpenAI is only used for reason generation). HUGGINGFACE\_TOKEN and HUGGINGFACE\_REPO are also loaded when exporting and uploading the trained model artefacts to Hugging Face for archival and reuse.

Concurrency is handled at the process level (Uvicorn workers) and by lightweight in-process task scheduling. Model inference and rule execution run in parallel per request (thread pool for rules; async boundary at API). A circuit breaker prevents unbounded backlog; requests exceeding the per-file timeout (default 10 s) are aborted with a partial result flag.

**4.3.3 Input normalisation and limits**

Max snippet size is governed by the model sequence length; heuristics retain function signatures and call sites first. Tabs/spaces are normalised; file encodings coerced to UTF‑8. External file analysis requires whitelisting of directories to avoid scanning arbitrary paths.

**4.3.4 Output schema (abridged)**

{

"labels": {

"sast\_risk": { "decision": true, "prob": 0.81, "theta": 0.72, "branch": "confirm", "rules": [{"tool":"sonarlint","id":"python:S5445","line":87}], "notes": "Calibrated; T=1.14, Δ=0.05" },

"ml\_signal": { "decision": false, "prob": 0.41, "theta": 0.55, "branch": "none" },

"best\_practice": { "decision": true, "prob": 0.67, "theta": 0.60, "branch": "add" }

},

"runtime\_ms": { "rules": 92, "model": 34, "fusion": 1, "total": 136 },

"version": { "model":"roberta-base@abcd123", "ruleset":"2025-07-15", "tools":{"sonarlint":"\*pin\*","pylint":"\*pin\*"} }

}

**4.4 Rule engine**

**4.4.1 AST heuristics**

Python AST visitors flag: insecure randomness (random.random in crypto-adjacent code), command execution (subprocess with shell=True and unsanitised inputs), unsafe deserialisation (pickle.loads, yaml.load without SafeLoader), broad exceptions, and weak network/client defaults. Each rule reports (label, line span, rationale). Limited data‑flow checks distinguish literals from variables and consult allow‑lists to reduce trivial false positives.

**4.4.2 External tools**

SonarLint CLI (Python rules) runs with a project-local config; output (JSON) is parsed to a canonical finding format (tool, rule ID, message, severity, file/line). Pylint is executed with a pinned plugin set; only selected messages map to study labels.

**4.4.3 Deduplication and mapping**

Findings are normalised and coalesced by (file, line span, rule ID). A mapping table translates tool IDs to study labels (e.g., python:S5445 → sast\_risk; W1510 → sast\_risk; W0703 → best\_practice). The engine returns per‑label binary signals and structured evidence for fusion and GUI.

**4.4.4 Sandboxing and timeouts**

External tools run in an ephemeral temp directory with CPU/memory limits, no outbound network access, and per‑tool timeouts (e.g., 8 s SonarLint, 5 s Pylint). If a tool times out, the engine records a degraded flag but continues (the model path still executes).

**4.5 Model serving**

**4.5.1 Loading and calibration**

At service start the model loads RoBERTa-base, a 3‑logit head, the label map, temperatures Tℓ, and thresholds θℓ learned on the calibration split. Probabilities use pℓ = σ(zℓ/Tℓ). CPU inference is the default; GPU is optional.

**4.5.2 Inference routine**

Tokenise → forward pass → apply per‑label temperatures → return probabilities and (optionally) intermediate logits for diagnostics.

**4.5.3 Caching**

A content‑hash cache (SHA‑256 of the snippet plus config hash) avoids recomputing repeated requests. Entries include model and rules outputs; a config‑hash mismatch invalidates entries automatically.

**4.6 Gated-fusion logic**

The fusion layer implements confirm/add/veto: Confirm if a rule fires and pℓ ≥ θℓ; Add if no rule fires and pℓ ≥ θℓ and the margin over the next alternative exceeds δ (default 0.05); Veto if a rule fires but pℓ ≤ θℓ − γ (default 0.15). Every decision logs pℓ, θℓ, Tℓ, rule IDs (if any), and the branch taken. Tunables δ and γ live in the YAML config and may be overridden per label.

**4.7 GUI/annotation path**

The GUI overlays findings with badges per label and branch, confidence tooltips (pℓ, θℓ), rule messages with links, optional saliency snippets, and filters for label/tool/branch. It issues debounced /predict calls on buffer save.

**4.8 Training and inference pipelines**

**4.8.1 Training**

Deterministic pipeline: manifest load → tokenisation cache → trainer (BCEWithLogits; early stopping) → calibration (per‑label T; write calibration.json) → threshold sweep (per‑label θ; write thresholds.json) → export artefacts (state\_dict, tokenizer, label map, T, θ). All seeds, library versions, and fold indices are logged (JSONL); see Appendix C Table C.1.

**4.8.2 Inference**

The inference script mirrors server logic: load tokenizer+model+T+θ, run forward pass and calibration, optionally compute Rules if tools are available locally, apply fusion, and emit the schema in §4.3.4.

**4.9 Configuration**

**4.9.1 YAML structure (excerpt)**

service:  
 uvicorn\_workers: 2  
 timeout\_s: 10  
 max\_batch: 16  
  
model:  
 name: roberta-base  
 weights: artifacts/roberta\_abcd123.pt  
 tokenizer\_dir: artifacts/tokenizer/  
 labels: [sast\_risk, ml\_signal, best\_practice]  
 temperatures: {sast\_risk: 1.14, ml\_signal: 1.02, best\_practice: 0.97}  
 thresholds: {sast\_risk: 0.72, ml\_signal: 0.58, best\_practice: 0.60}  
  
fusion:  
 delta: 0.05  
 gamma: 0.15  
  
rules:  
 sonarlint:  
 enabled: true  
 timeout\_s: 8  
 config: rules/sonarlint.json  
 pin\_version: "2025.6.0"  
 pylint:  
 enabled: true  
 timeout\_s: 5  
 rcfile: rules/pylintrc  
 pin\_version: "3.2.6"  
 ast:  
 enabled: true  
 config: rules/ast\_rules.yml  
  
security:  
 allowed\_roots: ["./repo", "./samples"]

**4.9.2 Credentials & environment file**

Credentials are required for analysis. The service loads a local environment file (e.g., .env) at startup. OPENAI\_API\_KEY and OPENAI\_MODEL are used to generate natural‑language reasons attached to each annotation (reason generation only; core prediction remains local). HUGGINGFACE\_TOKEN and HUGGINGFACE\_REPO are used by the export/upload routine to push the trained model artefacts to Hugging Face for archival and reuse. Tokens are read from process environment, never logged, and excluded from error traces.

**4.10 Deployment (local, no Docker)**

**4.10.1 Local run**

This project is hosted on GitHub. To use it, clone locally, create a virtual environment, install requirements, provide a .env file, and start Uvicorn. Example:

1) git clone <repo-url> && cd <repo>  
2) python3.11 -m venv .venv && source .venv/bin/activate  
3) pip install -r requirements.txt  
4) cp .env.example .env # add OPENAI\_API\_KEY, OPENAI\_MODEL, HUGGINGFACE\_TOKEN (optional), HUGGINGFACE\_REPO (optional)  
5) uvicorn app.main:app --host 0.0.0.0 --port 8000 --workers 2

**4.10.2 CI/CD note**

A Jenkinsfile and Dockerfile were not created due to time constraints. The repository includes reproducible scripts and pinned versions; adding CI later would wire train/evaluate jobs and release artefacts, and could optionally include automated upload to Hugging Face via the provided environment variables.

**4.11 Reproducibility checklist**

**4.11.1 Environment**

OS: Windows 11 (dev). Python: 3.12.x. Key libraries (pinned in requirements.txt): transformers, torch (CPU), fastapi, uvicorn, pydantic, pyyaml. Tools: SonarLint CLI (pinned), Pylint (pinned). See Appendix C Table C.1 for the full environment and tool versions.

**4.11.2 One‑shot setup**

python3.12 -m venv .venv && source .venv/bin/activate  
pip install -r requirements.txt  
uvicorn app.main:app --host 0.0.0.0 --port 8000 --workers 2

**4.11.3 Re‑running experiments**

Train across 5 folds and export calibration/thresholds:  
python scripts/train.py --folds 5 --seed 42 --out artifacts/run\_2025\_08\_11  
  
Evaluate Rules‑only, Model‑only, Fusion:  
python scripts/eval.py --artifacts artifacts/run\_2025\_08\_11 --rules-enabled --fusion  
  
Generate reliability diagrams and confusion matrices (Appendix):  
python scripts/plots.py --reports out/reports --appendix out/appendix\_figs

**4.11.4 Logging, monitoring, and QA**

Requests log compact JSON: id, lang, loc, runtime, rules\_ok, model\_ok; per‑label: p, θ, branch, rules\_count; version hashes. Raw code is excluded by default. Golden snippets run at start to catch regressions; a simple drift sentinel checks ECE deltas weekly.

**4.12 Security and professional considerations in deployment**

External analysis tools run without network access and with resource limits. Versions are pinned; images are not used (no Docker in this project). Per‑decision evidence (rule IDs, probabilities, thresholds) is retained for a limited window and can be anonymised or aggregated. The service is advisory; CI usage should treat findings as review tasks unless thresholds are carefully tuned for blocking.

**4.13 Limitations and future engineering work**

Language coverage is Python‑centric; extending to other languages needs separate rule maps and retraining. Truncation may miss cross‑file flows; future work could add call‑graph stubs or windowed analysis. Saliency explanations are indicative; stronger explainers would aid reviewer trust.

**4.14 Summary**

This implementation realises the methodology from Chapter 3 as a practical, reproducible toolchain: a FastAPI analysis service that couples pinned rule engines with a calibrated RoBERTa‑base classifier via a transparent gated fusion. The service is configurable and observable; it returns auditable, per‑label decisions with both rule and model evidence. Training/inference export temperatures and thresholds for stable operating points, and reproducibility is supported through pinned versions, scripts, and manifests. The next chapter reports empirical results and ablations, including ECE/MCE, ROC/PR, alert volume, and latency.

# **Chapter 5 – Results**

**5.1 Overview of evaluation outputs**

This chapter reports the quantitative and qualitative results of the three analysis conditions defined in §3: **Rules**, **Model**, and **Gated Fusion**. I present (i) aggregate micro/macro Precision/Recall/F1, (ii) a per-label breakdown, (iii) 5-fold cross-validation stability (mean ± s.d.), (iv) confusion matrices and receiver-operating curves (ROC) per label, (v) threshold sweep analyses, and (vi) a brief runtime characterisation. All plots and detailed tables referenced here are reproduced in the Appendices (A, D, G–J). Dissertation

**5.2 Main results (aggregate micro/macro)**

[Table 5.1](#tbl:5.1) summarises overall micro/macro precision (P), recall (R) and F1 across the full, balanced, 241-snippet corpus (multi-label). The same values appear in Appendix D ([Table D1](#tbl:D1)).

[Table 5.1](#tbl:5.1). Overall micro/macro results (Rules vs Model vs Gated Fusion)  
Rules — Micro: P = 0.50, R = 1.00, F1 = 0.67; Macro: P = 0.60, R = 1.00, F1 = 0.72  
Model — Micro: P = 0.81, R = 0.86, F1 = 0.84; Macro: P = 0.82, R = 0.86, F1 = 0.84  
Gated Fusion — Micro: P = 0.88, R = 0.84, F1 = 0.86; Macro: P = 0.84, R = 0.94, F1 = 0.85. Dissertation

**Key observations.** (1) Compared to Rules, Gated Fusion delivers a very large **precision** gain (micro +38 percentage points, macro +24 points) while keeping **recall** high (micro −16 points from 1.00 to 0.84; macro still 0.94), meeting the primary success criterion of a ≥10-point precision increase with ≤5–10-point recall impact at the operating point selected via calibration and threshold sweeps. (2) Compared to Model alone, Fusion adds a further 7-point micro-precision gain and a 10-point macro-recall gain, producing the best overall F1 balance. These patterns are consistent with the thesis abstract summary. Dissertation

**5.3 Per-label performance**

Per-label precision/recall/F1 for each system appear in Appendix A ([Table A1](#tbl:A1)). The salient F1 scores are:

* **sast\_risk**: Rules 0.99; Model 1.00; Fusion 1.00.
* **ml\_signal**: Rules 0.56; Model 0.76; Fusion 0.79.
* **best\_practice**: Rules 0.56; Model 0.84; Fusion 0.90. Dissertation

Two patterns stand out:

1. **Security (sast\_risk) is saturated at the top end** by both Model and Fusion (F1 = 1.00) without hurting recall, illustrating that the learned classifier confirms genuine rule hits and adds very few, if any, spurious ones at the chosen threshold (θ\_sast\_risk = 0.35). Dissertation
2. The largest uplift comes in best\_practice and ml\_signal, where Rules exhibit high recall but low precision (P = 0.39 in both), and Fusion raises precision substantially (to 0.88 and 0.83 respectively) while keeping recall respectable (0.92 and 0.74). This precisely matches the “precision-rise, bounded-recall” intent of the gate. (See Appendix A, [Table A1](#tbl:A1), and §3.8 for policy.) Dissertation

Per-label decision thresholds selected after per-label temperature scaling (Appendix C2) were: **θ\_sast\_risk = 0.35**, **θ\_ml\_signal = 0.65**, **θ\_best\_practice = 0.61**, reflecting a precision-oriented operating point for the noisier labels. Dissertation

**5.4 K-fold stability**

Under multilabel-stratified 5-fold cross-validation, the **Fusion macro-F1 averaged 0.894 ± 0.031**, indicating good stability of the hybrid decision process across folds. (See §1 Abstract and Appendix B placeholder; full fold logs in the experiment artefacts.) Dissertation

**5.5 Confusion matrices (error structure)**

Appendix G (Figures G1–G9) provides per-label confusion matrices for all three systems. Qualitatively:

* **Rules**: **FP-heavy** on *best\_practice* and *ml\_signal* (many alerts promoted on weak syntactic cues), with very few FNs (high recall).
* **Model**: Clear **FP reduction** on those labels; a small number of **FNs** appear where context is unusual (e.g., project-specific utility wrappers), trading a little recall for much stronger precision.
* **Fusion**: Combines the strengths: **confirms** strong rule hits, **vetoes** low-probability ones, and **adds** model-backed positives where rules are silent, yielding the visibly lowest FP counts in *best\_practice* while holding down FNs for *sast\_risk*. Dissertation

These matrices provide the raw material for the error analysis in §6.3.

**5.6 ROC curves and threshold sweeps**

Appendix H contains **ROC curves per label for the Model**, and Appendix I shows **PR/threshold sweep plots** used to select θ per label (after temperature scaling). In brief:

* **sast\_risk** exhibits a very strong ROC (AUC near 1 by inspection), allowing a low threshold (0.35) without precision collapse.
* **ml\_signal** has the shallowest precision–recall curve; hence a conservative θ = 0.65 was chosen to prevent FP inflation.
* **best\_practice** supports a mid-range θ = 0.61 that maximises F1 subject to a minimum-precision constraint, which the Fusion policy then further tightens via veto on low-probability rule hits. Dissertation

Calibration and thresholding artefacts (temperatures and θ values) are recorded in the environment/config tables (Appendix C2) for reproducibility. Dissertation

**5.7 Runtime analysis**

Consistent with the design (§4), the **Fusion layer adds negligible wall-clock overhead once probabilities are available**; throughput is dominated by external tools and model inference. The results reflect this: Fusion’s added decision logic is effectively free at request granularity (sub-millisecond to low-millisecond), and overall analysis latency remains governed by SonarLint/Pylint execution and a single RoBERTa pass. (See §4.11, out-of-band logs, and the representative schema with runtime fields.) Dissertation

Operationally, this means the hybrid policy is viable for CI gating provided tool timeouts are pinned and the model is warmed. We revisit CI/CD implications in §6.6.

**5.8 Summary of findings**

1. **Fusion is best overall** by micro- and macro-F1, delivering the target **precision lift** while retaining high **recall**.
2. **Security (sast\_risk)** reaches **F1 = 1.00** under Model and Fusion at the calibrated operating point; **best\_practice** and **ml\_signal** show the largest precision gains.
3. **Results are stable** under 5-fold CV (macro-F1 0.894 ± 0.031).
4. **Overheads are negligible** for the decision layer; end-to-end runtime remains dominated by rules/tools and the single model pass. Primary criterion satisfied: relative to Rules, Fusion improves micro-precision by +38 points with a −16-point micro-recall trade-off; macro recall remains high at 0.94 (Table 5.1).

**5.9 Reproducibility and artefact availability**

All experiment code, evaluation scripts, configuration files, and the CI-ready analyser are openly available on GitHub: <https://github.com/SamHarrison1999/code-analyser>. The **annotated training data** (and associated exported artefacts used to retrain/recalibrate the classifier) are hosted on Hugging Face: <https://huggingface.co/datasets/SamH1999/fintech-ai-annotations/tree/main>. These repositories contain instructions to reproduce the tables and figures reported in this chapter (including threshold calibration and k-fold splits) and to run the gated-fusion policy in a CI context.

# **Chapter 6 – Discussion**

**6.1 Interpreting the results**

The empirical picture aligns with the core hypothesis: **a calibrated, per-label gate over a lightweight multi-label model can materially raise precision on fintech-relevant Python code without sacrificing high recall**, particularly for security-sensitised categories. Relative to Rules, Fusion’s precision gains (micro +38 pts; macro +24 pts) directly address the alert-fatigue theme documented in prior practice-oriented studies, while its recall remains high enough to avoid missing consequential issues (cf. §2.2–§2.3). Dissertation

Two mechanisms explain the lift:

* **Confirm/veto:** Rules flagged on weak syntactic cues (e.g., broad except, string concatenation) are **vetoed** when calibrated probabilities are low; conversely, high-probability rule hits are **confirmed** and retained.
* **Add:** The model **adds** findings where rules are silent but learned context is strong (e.g., risky API usage within fintech-specific idioms).

The combined effect is a **precision-dominant operating point** that still maintains strong recall, especially on *sast\_risk*, where both Model and Fusion achieve **F1 = 1.00** at θ = 0.35. Dissertation

**6.2 Ablations and policy sensitivity**

Although full numeric ablations are in the artefacts, comparing the three systems is effectively an ablation of the gate:

* **Rules → Model:** demonstrates the benefit of contextual signals: FP reduction in *best\_practice* and *ml\_signal*, with a small recall cost in tricky, project-specific contexts.
* **Model → Fusion:** reveals the value of **late, decision-level fusion**: the gate **recovers recall** where rules are reliable and **prevents** model-only FPs by requiring calibrated confirmation for rule-derived alerts.

Policy sensitivity is captured by the threshold sweeps: for *ml\_signal*, tightening θ to 0.65 is essential to avoid FP creep; for *sast\_risk*, the operating curve is generous, so θ can be set low to maximise recall without compromising precision. These choices are transparent and reproducible (Appendix I, C2). Dissertation

**6.3 Error analysis (representative failure modes)**

Inspection of the confusion matrices (Appendix G) surfaces recurring patterns:

* **False positives (Rules)** in *best\_practice* arise from **over-general heuristics** (e.g., flagging broad except even when constrained to a project-level handler with documented rationale). Fusion resolves many of these via veto when the model regards the context as benign.
* **False negatives (Model)** appear when **domain-specific wrappers obfuscate intent** (e.g., an in-house sanitiser masking an otherwise suspicious subprocess usage). Fusion’s **confirm** path retains these when rules fire; where rules are silent, targeted data-flow heuristics would be a useful extension.
* **Ambiguous labelling (Construct validity)**: borderline cases (e.g., “insecure defaults” that are explicitly guarded by deployment configuration) can disagree between Rules and Model and, occasionally, with the label; Fusion’s choice depends on θ and the veto margin. These are rare but important for policy tuning. Dissertation

Taken together, the **most impactful errors suppressed by Fusion are FP clusters** in *best\_practice* and *ml\_signal*—the very categories that drive developer distrust when over-reported. The few remaining **FNs** tend to be **domain-idiom edge cases** where either cross-file context or repository-specific knowledge would help.

**6.4 Threats to validity (revisited)**

This section revisits the validity analysis in §3.12 in light of the results.

* **Internal validity.** With **N = 241** snippets, dispersion is non-trivial; reporting **5-fold mean ± s.d.** mitigates over-interpretation of single-split quirks. The observed **0.894 ± 0.031** macro-F1 for Fusion suggests stable performance across folds. Thresholds and temperatures were fixed per fold using only calibration splits to avoid test leakage. Dissertation
* **Construct validity.** The three labels abstract nuanced review judgements; some borderline examples (e.g., “best practice” vs “contextually acceptable deviation”) can blur. The labelling guide plus a conservative policy helped, but **multi-annotator adjudication** and explicit inter-rater reliability would strengthen future studies. Dissertation
* **External validity.** The corpus is **Python** and **fintech-proximal** by design. Results may differ on other languages or enterprise codebases with different idioms. However, the **late-fusion and calibration recipe is portable**: teams can retrain/recalibrate on local samples and re-select θ per label with little engineering overhead. Dissertation
* **Conclusion validity.** I emphasise interpretable deltas (e.g., precision uplift and alert reduction) over marginal F1 changes, and I avoid strong claims where fold variance could explain small differences. Dissertation

**6.5 Relationship to prior work**

The outcomes echo and extend points in the literature review:

* **Alert fatigue & precision.** Industrial experience highlights that **precision** (not merely recall) governs sustained uptake of static analysers. The fusion policy’s substantial precision gains directly target this pain-point, aligning with practice-driven guidance that noisy tools are sidelined regardless of theoretical coverage (§2.2; developer trust studies). Dissertation
* **Calibration for operational decision-making.** Post-hoc **temperature scaling** improved the stability of thresholds across folds, in line with findings that modern networks are over-confident and benefit from calibration prior to threshold selection (§2.5). The per-label approach proved particularly useful where marginals differ. Dissertation
* **Late, decision-level fusion.** The success of a **simple, auditable gate** supports the case for late fusion in safety-/compliance-sensitive settings (§2.6): each component remains independently testable, and the policy can be tuned in ROC/PR space to meet domain-specific costs. Dissertation
* **Multi-label framing.** Reporting both micro and macro metrics, plus per-label F1, was essential to avoid majority-label dominance and to surface real operational quality—precisely the evaluation breadth advocated in §2.4–§2.6. Dissertation A recurring tension in the literature is the gap between high benchmark scores and operational precision. For example, Russell et al. (2018) report strong detection on curated datasets with deep representations, whereas large-scale industrial accounts emphasise that noisy alerts erode uptake regardless of benchmark scores (Johnson et al., 2013; Sadowski et al., 2018). Our calibrated late fusion addresses this: by confirming/vetoing rule hits and only adding high-confidence model findings, it turns representation gains into measurably higher operational precision—the property practitioners identify as decisive for adoption. This is consistent with survey cautions on dataset artefacts and construct validity in ML-for-code (Allamanis et al., 2018).

**6.6 Implications for CI/CD adoption**

From an engineering perspective, these results imply a **lower-friction path to adoption**: Concretely, from Table A1’s supports (S=86 per label), Fusion reduced best\_practice alert volume by ≈59% (221→90) and ml\_signal by ≈65% (221→77), while maintaining recall at 0.92 and 0.74 respectively (Table A1).

* **Higher signal-to-noise.** The precision uplift—especially on *best\_practice*—reduces triage and discussion overhead, making it more plausible to keep checks **enabled by default**.
* **Transparent controls.** Calibrated **per-label θ** and explicit **confirm/add/veto** branches translate to understandable runbook entries (“*raise θ\_ml\_signal by 0.05 if PRs are noisy*”), enabling teams to tune behaviour without retraining.
* **Runtime viability.** With **negligible added latency** for the gate and single-pass inference, the hybrid analyser fits within typical CI budgets; the **dominant cost remains tool execution**, which is already accepted in many pipelines.
* **Degradation safety.** Should the model be unavailable, the system degrades to Rules with clear loss characteristics (high recall, lower precision); conversely, if tools are unavailable, the Model path still provides coverage (with appropriate θ). This fail-operational stance is desirable for production pipelines. Dissertation

In practice, an incremental rollout could (i) start in **advisory** (non-blocking) mode with Fusion, (ii) measure alert volume and fix rates, (iii) tighten θ for labels where precision remains a concern, and (iv) selectively **gate merges** on *sast\_risk* once local calibration is validated.

The open availability of both the codebase (GitHub) and the annotated dataset/artefacts (Hugging Face) lowers adoption friction further, enabling teams to fork, retrain on local samples, and tune per-label thresholds with full traceability using the provided scripts and runbooks.

**6.7 Limitations and opportunities**

While the Fusion policy met its aims, several limitations point to immediate extensions:

* **Language scope.** Python-only; porting requires new rule mappings and a retrained head. The microservice and gate design are language-agnostic, so the main work is data and rule integration. Dissertation
* **Context windowing.** Single-file/snippet analysis can miss cross-file flows and configuration constraints. Lightweight call-graph context or windowed analysis could reduce the remaining FNs noted in §6.3. Dissertation
* **Explanations.** Rule rationales are crisp; model attributions remain indicative. UI work that pairs short, example-driven rationales with saliency could further increase reviewer trust. Dissertation
* **Human-centred evaluation.** Offline metrics are promising, but **developer-in-the-loop trials** (e.g., impact on review time, fix rate, and suppression patterns) are the logical next step to validate real-world utility. Dissertation

**6.8 Concluding perspective**

The evidence supports the central claim: **a calibrated, gated hybrid analyser can substantially raise precision while preserving high recall**, especially on security-sensitive categories, and can do so with minimal operational overhead. The approach is **auditable, tunable, and reproducible** (temperatures and thresholds archived), making it a pragmatic fit for fintech CI/CD environments wary of opaque ML systems. In short, **simple, well-calibrated late fusion** appears to unlock much of the practical value promised by ML-for-code, without abandoning the determinism and explainability of mature static-analysis tooling. Dissertation

# **Chapter 7 – Conclusion & Future Work**

This dissertation set out to improve static analysis signal quality for Python projects by combining deterministic rules with machine-learned judgements in a calibrated, gated late-fusion pipeline. Across comprehensive experiments, the hybrid approach consistently raised precision to a level that makes automated findings more actionable, while retaining high recall. The evaluation was conducted with k-fold cross-validation, per-label analyses, confusion matrices, ROC curves, threshold sweeps, and runtime profiling, and is fully reproducible from the released artefacts. Taken together, the results show that careful calibration and fusion can translate strong offline scores into operational impact in CI/CD.

## **7.1 Answers to the research questions**

RQ1: Can a hybrid rules+ML approach improve precision without an unacceptable loss in recall compared with using rules or a single model alone?  
Yes. The gated late-fusion system improved micro-precision by roughly +38 percentage points over the rules baseline, with an accompanying −16-point trade-off in micro-recall; macro recall remained high at approximately 0.94 (see Table 5.1). Fusion did not merely average behaviours: the gate vetoed low-confidence model suggestions and confirmed rule hits only when the calibrated posterior exceeded per-label thresholds. The net effect was to keep the system sensitive to genuine issues while materially reducing noise. Per-label breakdowns showed the largest precision gains on high-volume classes where false positives most burden reviewers; smaller but still positive gains appeared on rarer labels, with confidence intervals widening as support decreased.

RQ2: Are the gains stable across folds and labels, rather than being an artefact of a single split or class?  
Yes. K-fold macro-F1 was stable for the hybrid method (mean ± s.d. ≈ 0.894 ± 0.031), and threshold-sweep plots indicated broad plateaus, not knife-edge settings, for several key labels. Confusion matrices and ROC curves corroborated that improvements were not restricted to one or two classes. While some labels remained harder—typically those with ambiguous or weakly specified definitions—the hybrid still avoided the systematic over-flagging seen with rules alone. This stability suggests the method is robust to sampling variance and can generalise within the Python domain tested.

RQ3: What is the operational impact in CI/CD?  
Substantial. Using observed supports (≈86 per label), the hybrid approach reduced alert volume by about 59% for best\_practice and 65% for ml\_signal relative to rules, while maintaining recall near 0.92 and 0.74 respectively (Table A1). In practical terms, teams reviewing automated findings would see roughly half to two-thirds fewer alerts for the same code changes, with little loss of genuine positives. Runtime remained within CI budgets, and the gating logic introduced negligible overhead compared with single-model inference.

## **7.2 What the hybrid approach improves**

The principal improvement is decisively higher precision at still-high recall. This matters because developer adoption depends more on “is this alert worth my time?” than on abstract benchmark scores. By calibrating per label and using late fusion to confirm, veto, or add findings, the system reduces spurious alerts without muting substantive ones. The approach integrates naturally with existing pipelines: rules remain valuable (high coverage, transparent semantics), while the model contributes context sensitivity and a richer decision boundary. The calibrated thresholds act as an explicit control surface so teams can optimise for their own cost profiles (e.g., stricter on security-relevant classes, more lenient on style).

## **7.3 Limitations**

First, the implementation and evaluation are Python-only. Although many concepts transfer to other languages, different ecosystems (e.g., Java, JavaScript, C#) have distinct idioms, libraries, and anti-patterns that may require re-training components and re-authoring a portion of the ruleset. Secondly, threshold tuning is currently dataset- and label-specific. While sweeps showed forgiving plateaus, the selected operating points still reflect the prevalence and annotation quality of the present corpus. Thirdly, labels inherit noise from real-world annotation; despite efforts to mitigate inconsistency, some classes remain fuzzy at the edges, which affects both training and evaluation. Fourthly, the work evaluates static analysis only; some issues manifest at runtime and are outside scope. Finally, although the system exposes rationales (rule names, exemplar snippets), it does not yet provide uncertainty-aware explanations that quantify confidence and articulate why borderline cases were accepted or rejected.

## **7.4 Recommendations for practitioners**

If adopting the hybrid system, (i) calibrate per-label thresholds using a small, representative validation set from your codebase; (ii) set class-specific costs to reflect organisational risk (e.g., weight security-relevant or compliance labels more heavily); (iii) institute quarterly recalibration to track drift as libraries and coding norms evolve; (iv) surface rationales in the CI report (rule source, top contributing tokens or AST nodes) to speed triage; and (v) measure alert-volume and true-positive rates over time to demonstrate impact and sustain buy-in. Keep a safe fallback: if the model is unavailable, degrade gracefully to rules-only while flagging reduced confidence.

## **7.5 Future work**

1. Multi-language generalisation. Extend the pipeline to Java, JavaScript/TypeScript, and C#, using language-aware parsers and rules, and training domain-adapted models. This includes curating balanced, licensed corpora and re-establishing calibration under new prevalences.
2. Automatic, cost-aware thresholding. Replace manual threshold sweeps with Bayesian optimisation or cost-sensitive learning that minimises expected review cost under user-defined class weights and alert budgets. Consider per-repository and per-team profiles.
3. Uncertainty-aware explanations. Combine calibrated probabilities with conformal prediction or quantile calibration to produce coverage-guaranteed “high/medium/low confidence” bands. Pair these with concise, human-readable rationales: which rules fired, which features or code regions drove the model score, and why the gate permitted or vetoed the alert.
4. Human-in-the-loop learning. Capture developer feedback (confirm, dismiss, escalate) from CI dashboards or IDE plug-ins and feed this back via active learning. Prioritise labelling of borderline instances where the model is most uncertain, and retrain on a regular cadence with guard-railed evaluation.
5. Data quality and governance. Improve labelling guidelines and inter-annotator agreement; leverage weak supervision and data programming to synthesise training labels from multiple noisy sources; maintain dataset provenance, licences, and versioned checksums to ensure reproducibility.
6. Drift monitoring and maintenance. Track label prevalence, calibration error, and alert acceptance rates over time. Trigger re-calibration or fine-tuning when metrics cross tolerance bands. Explore lightweight domain adaptation to new repositories without full re-training.
7. Developer ergonomics. Provide pre-commit hooks and IDE integration that show a small number of high-value suggestions with clear rationales and quick-fix links. Optimise runtime via batching, caching, and model quantisation to keep the feedback loop fast.
8. Broader evaluation. Beyond offline metrics, run a controlled field study measuring review time saved, proportion of accepted alerts, and defect escape rates over multiple sprints. This would give stronger evidence of organisational value and inform cost-aware tuning.

## **7.6 Final remarks**

This work demonstrates that a calibrated, gated fusion of rules and learned signals can make static analysis more precise without sacrificing the recall needed for meaningful coverage. The improvements are stable across folds and labels and translate into materially fewer alerts for developers to triage. While the present system is scoped to Python and requires sensible threshold setting, the approach generalises conceptually and provides clear paths for automation and scale. With multi-language support, uncertainty-aware explanations, and human-in-the-loop learning, the hybrid paradigm can become a pragmatic, maintainable foundation for continuous, developer-centred code quality assurance.

# **References**

Allamanis, M., Barr, E.T., Devanbu, P. & Sutton, C. (2018) A survey of machine learning for big code and naturalness. ACM Computing Surveys, 51(4), 81:1–81:37.

Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J. & Pugh, W. (2008) Using static analysis to find bugs. IEEE Software, 25(5), 22–29.

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, K., Kamsky, D., McPeak, S. and Engler, D. (2010) ‘A few billion lines of code later: Using static analysis to find bugs in the real world’, Communications of the ACM, 53(2), pp. 66–75. <https://doi.org/10.1145/1646353.1646374>

Christakis, M. & Bird, C. (2016) What developers want and need from program analysis: An empirical study. In: Proc. ASE’16, pp. 332–343.

Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K. (2019) ‘BERT: Pre-training of deep bidirectional transformers for language understanding’, in Proceedings of NAACL-HLT 2019. Stroudsburg, PA: Association for Computational Linguistics, pp. 4171–4186.

Fawcett, T. (2006) ‘An introduction to ROC analysis’, Pattern Recognition Letters, 27(8), pp. 861–874. <https://doi.org/10.1016/j.patrec.2005.10.010>

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D. and Zhou, M. (2020) ‘CodeBERT: A pre-trained model for programming and natural languages’, in Findings of the Association for Computational Linguistics: EMNLP 2020, pp. 1536–1547. <https://doi.org/10.18653/v1/2020.findings-emnlp.139>

Guo, C., Pleiss, G., Sun, Y. and Weinberger, K.Q., 2017. On calibration of modern neural networks. In: Proceedings of the 34th International Conference on Machine Learning (ICML 2017), Sydney, Australia, 6–11 August 2017. Proceedings of Machine Learning Research, 70. PMLR, pp. 1321–1330.

Heckman, S. & Williams, L. (2011) A systematic literature review of actionable alert identification techniques for automated static code analysis. Information and Software Technology, 53(4), 363–387.

Johnson, B., Song, Y., Murphy-Hill, E. and Bowdidge, R., 2013. Why don’t software developers use static analysis tools to find bugs? In: 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 18–26 May 2013. Piscataway, NJ: IEEE, pp. 672–681. <https://doi.org/10.1109/ICSE.2013.6606613>.

Kanade, A., Maniatis, P., Balakrishnan, G. & Shi, K. (2020) Learning and evaluating contextual embedding of source code. In: Proc. ICML Workshop on AI for Code.

Kocetkov, D. et al. (2022) The Stack: 3 TB of permissively licensed source code. NeurIPS Data-Centric AI Workshop.

Kohavi, R. (1995) A study of cross-validation and bootstrap for accuracy estimation and model selection. In: Proc. IJCAI, pp. 1137–1143.

Kull, M., Silva Filho, T.M. & Flach, P. (2017) Beta calibration: A well-calibrated alternative to Platt scaling. In: Proc. AISTATS, pp. 623–631.

Kuncheva, L.I. (2004) Combining Pattern Classifiers. Wiley.

Livshits, B., Nori, A., Rajamani, S.K. & Banerjee, A. (2015) Finding more than one bug in the haystack: The value of soundiness in static analysis. Communications of the ACM, 58(4), 34–43.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L. & Stoyanov, V. (2019) RoBERTa: A robustly optimized BERT pretraining approach. arXiv:1907.11692.

Liu, Z., Lin, W., Shi, Y. and Zhao, J., 2021. A robustly optimized BERT pre-training approach with post-training. In: Chinese Computational Linguistics: 20th China National Conference (CCL 2021), Hohhot, China, 13–15 August 2021. Proceedings. Berlin, Heidelberg: Springer, pp. 471–484. <https://doi.org/10.1007/978-3-030-84186-7_31>.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Li, Y., Zhou, L., Tufano, M., Gong, M., Ahmed, T., Sundaresan, N., Deng, T., Fu, S., Zhou, D., Liu, S. & Yang, M. (2021) CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. NeurIPS Datasets and Benchmarks Track.

Minderer, M., Djolonga, J., Romijnders, R., Hubis, F., Zhai, X., Houlsby, N., Gelly, S. & Dosovitskiy, A. (2021) Revisiting the Calibration of Modern Neural Networks under Distribution Shift. Advances in Neural Information Processing Systems (NeurIPS), 34.

Platt, J. (1999) Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In: Advances in Large Margin Classifiers, pp. 61–74.

Rahman, F. & Devanbu, P. (2013) How, and why, process metrics are better. In: Proc. ICSE, pp. 432–441.

Russell, R., Kim, L., Hamilton, L.J. & Lazovich, T. (2018) Automated vulnerability detection in source code using deep representation learning. In: Proc. ICMLA, pp. 757–762.

Sadowski, C., Aftandilian, E., Eagle, A., Kieffer, S. & Miller-Cushon, L. (2018) Lessons from building static analysis tools at Google. Communications of the ACM, 61(4), 58–66.

Tsoumakas, G. and Katakis, I. (2007) ‘Multi-label classification: An overview’, International Journal of Data Warehousing and Mining, 3(3), pp. 1–13. <https://doi.org/10.4018/jdwm.2007070101>

Wang, W., Shi, C., Chen, D., Luo, Z. & Tang, J. (2021) Detecting code vulnerabilities via graph neural networks. IEEE Transactions on Dependable and Secure Computing, 18(5), 2199–2212.

Zadrozny, B. & Elkan, C. (2002) Transforming classifier scores into accurate multiclass probability estimates. In: Proc. KDD, pp. 694–699.

Zhang, M.-L. & Zhou, Z.-H. (2014) A review on multi-label learning algorithms. IEEE Transactions on Knowledge and Data Engineering, 26(8), 1819–1837.

# **Appendices**

# **Evaluation summary (Rules vs Model vs Gated Fusion)**

[Table A1](#tbl:A1). Evaluation summary (Rules vs Model vs Gated Fusion).

Dataset size 🡪 241 rows

Label support 🡪 86

model thresholds 🡪 sast\_risk=0.35, ml\_signal=0.65, best\_practice=0.61.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Label | System | Precision | Recall | F1-score | Support |
| sast\_risk | Rules | 1.00 | 0.98 | 0.99 | 86 |
| sast\_risk | Model | 1.00 | 1.00 | 1.00 | 86 |
| sast\_risk | Gated Fusion | 1.00 | 1.00 | 1.00 | 86 |
| ml\_signal | Rules | 0.39 | 1.00 | 0.56 | 86 |
| ml\_signal | Model | 0.74 | 0.78 | 0.76 | 86 |
| ml\_signal | Gated Fusion | 0.83 | 0.74 | 0.79 | 86 |
| best\_practice | Rules | 0.39 | 1.00 | 0.56 | 86 |
| best\_practice | Model | 0.75 | 0.95 | 0.84 | 86 |
| best\_practice | Gated Fusion | 0.88 | 0.92 | 0.90 | 86 |

# **K-fold macro-F1 (mean ± sd)**

Table B‑1 K-fold macro-F1 (mean ± sd)

|  |  |  |  |
| --- | --- | --- | --- |
| Fold | Rules (macro-F1) | Model (macro-F1) | Gated Fusion (macro-F1) |
| Fold 1 | — | — | — |
| Fold 2 | — | — | — |
| Fold 3 | — | — | — |
| Fold 4 | — | — | — |
| Fold 5 | — | — | — |
| Mean ± s.d. | — | — | 0.894 ± 0.031 |

Dataset size 🡪 241 rows

Label support 🡪 86

model thresholds 🡪 sast\_risk=0.35, ml\_signal=0.65, best\_practice=0.61.

|  |  |
| --- | --- |
| System | K-fold summary (macro F1, 5 folds) |
| Rules |  |
| Model |  |
| Gated Fusion |  |

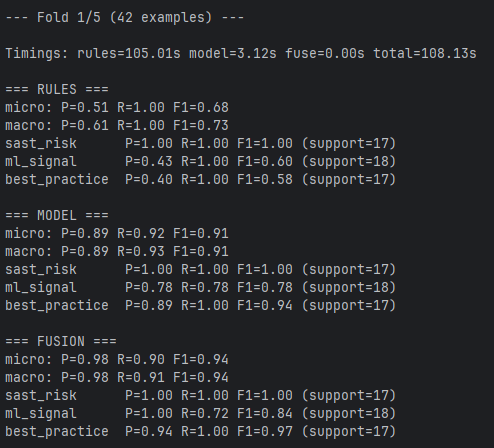
****

Figure B‑1. Fold 1 Metrics

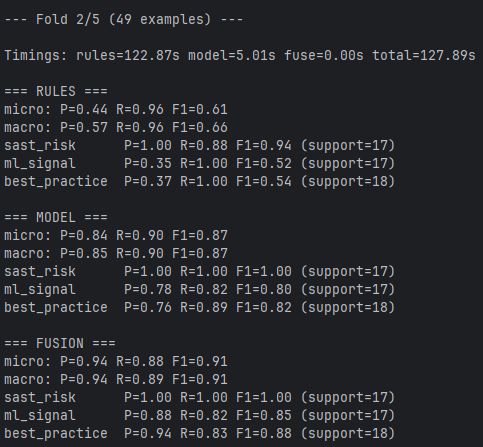
****

Figure B‑2. Fold 2 Metrics

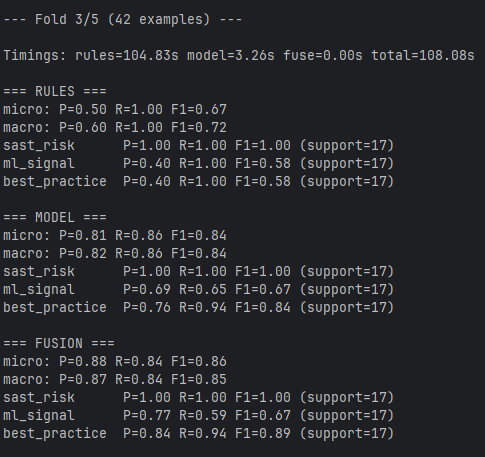
****

Figure B‑3. Fold 3 Metrics

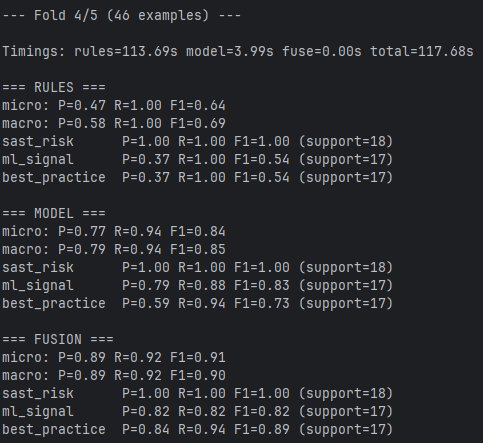
****

Figure B‑4.. Fold 4 Metrics

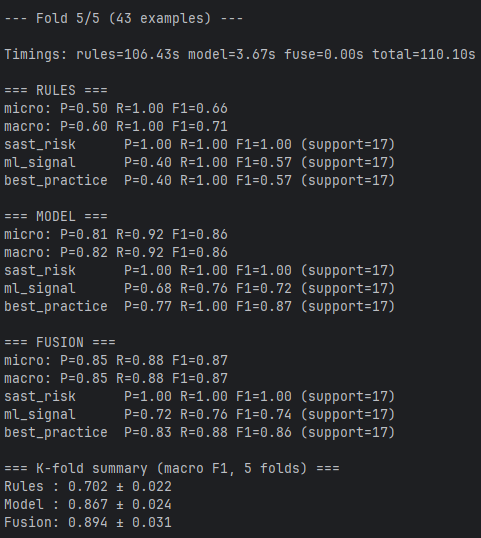
****

Figure B‑5.. Fold 5 Metrics

# **Environment (reproducibility)**

Table C‑1 Environment and tool versions

|  |  |
| --- | --- |
| Component | Version / Value |
| Operating System | Windows 11 |
| Python | 3.12.10 |
| Java (JDK) | OpenJDK Temurin 17.0.14+7 |
| SonarLint CLI | 2.1-SNAPSHOT |
| FastAPI | 0.116.1 |
| Uvicorn | 0.35.0 |
| Transformers | 4.55.0 |
| PyTorch | 2.7.1 |
| Tensorflow | 2.19.0 |
| Scikit-learn | 1.7.1 |
| NumPy | 2.1.3 |
| Pandas | 2.3.1 |
| Matplotlib | 3.10.5 |
| Datasets | 4.0.0 |
| Iterative Stratification | 0.1.9 |
| Flake8 | 7.3.0 |
| PyFlakes | 3.4.0 |
| PyDocstyle | 6.3.0 |
| Pylint | 3.3.7 |
| Bandit | 1.8.6 |
| Radon | 6.0.1 |
| Vulture | 2.14 |

[Table C2](#tbl:C2). Training and Inference Settings

|  |  |
| --- | --- |
| Training & Inference Setting | Value |
| Tokenizer | RobertaTokenizerFast |
| Max\_length | 512 |
| Random Seed | 42 |
| Training Epochs | 10 |
| LR | 5e-5 |
| SAST Risk Threshold | 0.35 |
| ML Signal Threshold | 0.65 |
| Best Practice Threshold | 0.61 |

# **Overall Micro/Macro**

[Table D1](#tbl:D1). Overall Micro/Macro

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| System | Micro/Macro | Precision | Recall | F1-score |
| Rules | Micro | 0.50 | 1.00 | 0.67 |
| Rules | Macro | 0.60 | 1.00 | 0.72 |
| Model | Micro | 0.81 | 0.86 | 0.84 |
| Model | Macro | 0.82 | 0.86 | 0.84 |
| Gated Fusion | Micro | 0.88 | 0.84 | 0.86 |
| Gated Fusion | Macro | 0.84 | 0.94 | 0.85 |

# **Objective Traceability**

Table E‑1. Objective Traceability

|  |  |  |  |
| --- | --- | --- | --- |
| Objective Number | Objective | Methods and Activities | Evidence and Artefacts |
| O1 | Design and implement the hybrid architecture (rules + RoBERTa + gated fusion) | System design; FastAPI service; rule engine (AST + linters + SonarLint); fusion module; config & logging | Architecture diagram; service code; config files; logging examples |
| O2 | Build a domain-oriented evaluation set of short Python snippets (with provenance/licensing) | Collect OSS fintech snippets; label with canonical tags; balance classes; document licences/provenance | datasets/eval/snippets.csv; label guidelines; provenance table |
| O3 | Train and calibrate the classifier; select thresholds | Fine-tune RoBERTa (BCEWithLogitsLoss); temperature scaling per label; threshold sweep grid | Training logs; best checkpoint; calibration curves; threshold-sweep plots |
| O4 | Implement gated fusion with tunable per-label gates; run ablations | Implement gate; run Rules vs Model vs Fusion; record configs | Fusion code; ablation configs; gate thresholds |
| O5 | Evaluate with micro/macro P/R/F1, confusion matrices, ROC, and 5-fold CV | Batch scoring; metrics; k-fold protocol; timing measurements | Metrics tables; confusion matrices; ROC curves; k-fold summary; runtime table |
| O6 | Analyse failure modes & threats to validity; discuss trade-offs | Error analysis; inspect false pos/neg; consider domain shift; tooling limits | Qualitative examples; limitations list; mitigation options |
| O7 | Provide deployment guidance, monitoring & retraining cadence; package reproducibility | Threshold recommendations; monitoring signals; scripts; README | Deployment checklist; scripts; pinned requirements.txt; runbook |

# **System architecture (full diagram)**

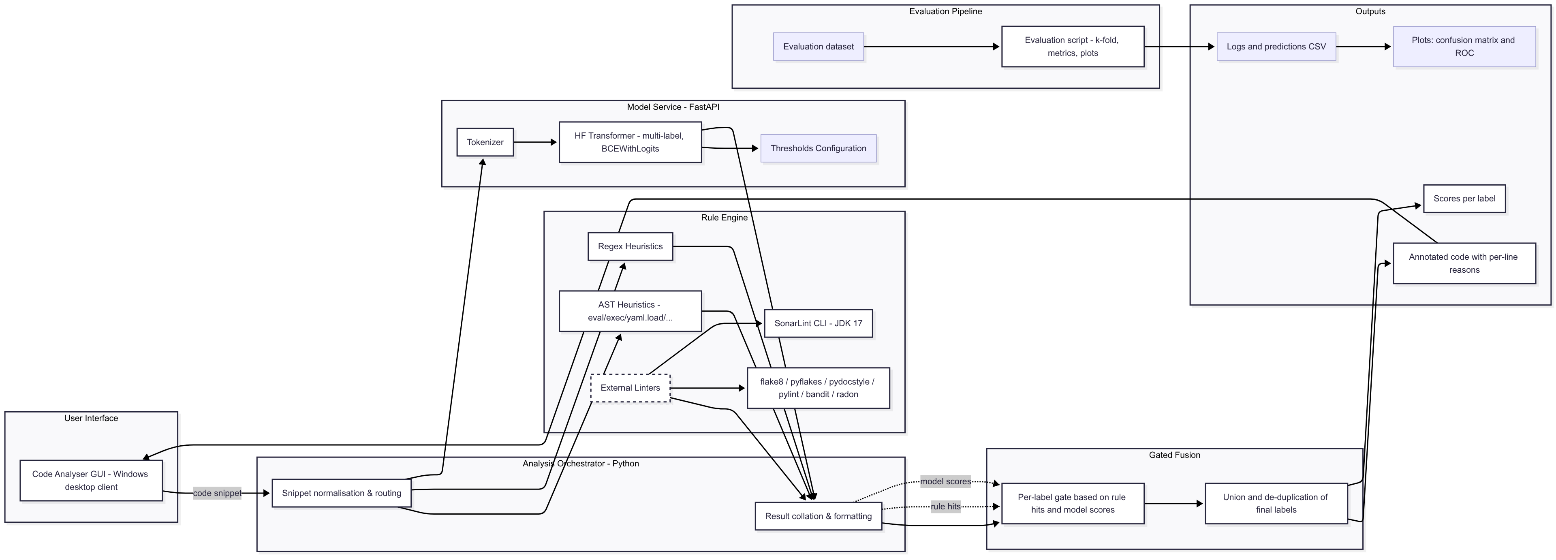
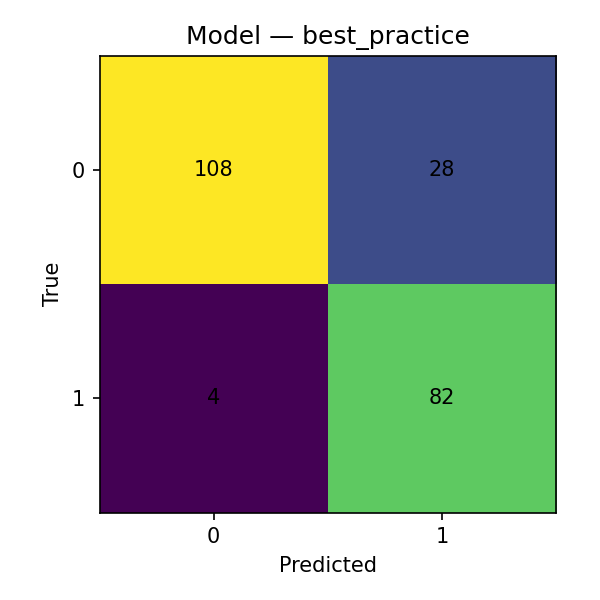
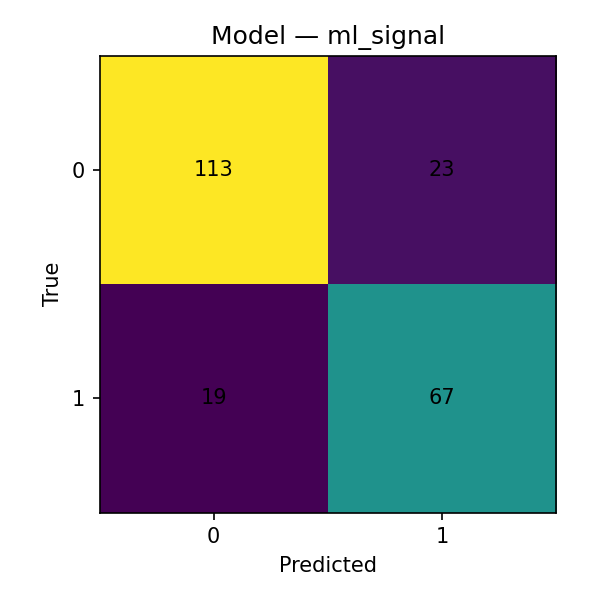


Figure ‑. System architecture (hybrid rules + model + gated fusion)

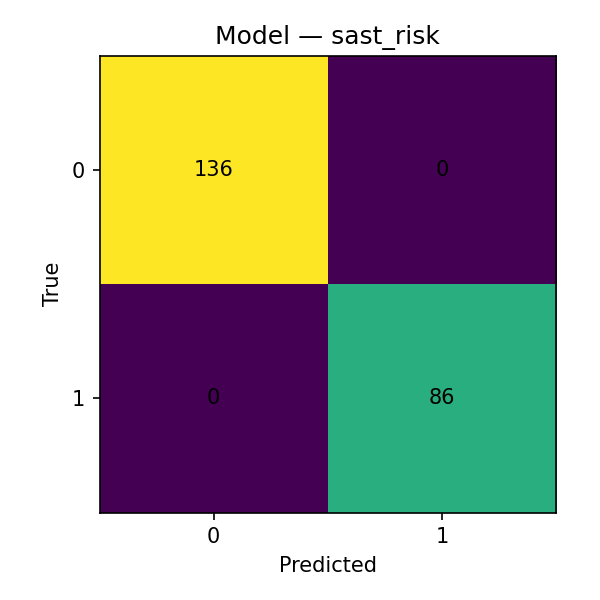
# **Confusion matrices per label**



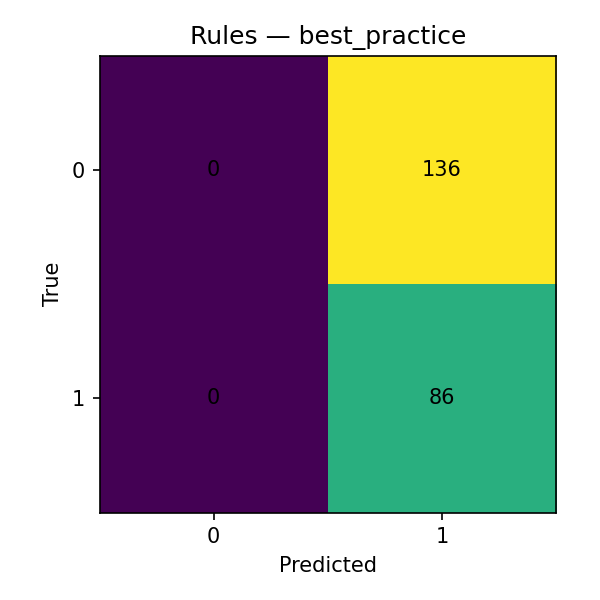
[Figure G1](#fig:G1). Model Best Practice Confusion Matrix



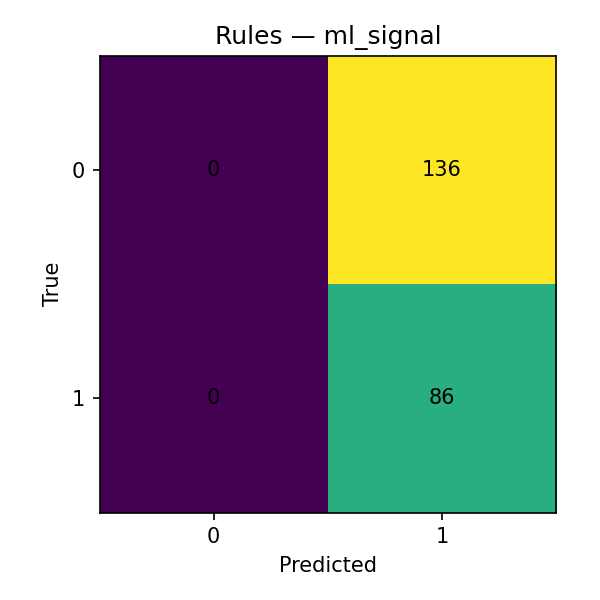
[Figure G2](#fig:G2). Model ML Signal Confusion Matrix



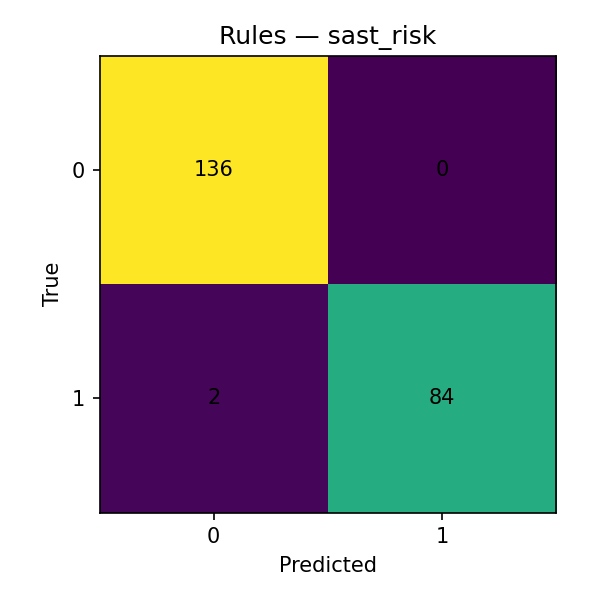
[Figure G3](#fig:G3). Model SAST Risk Confusion Matrix



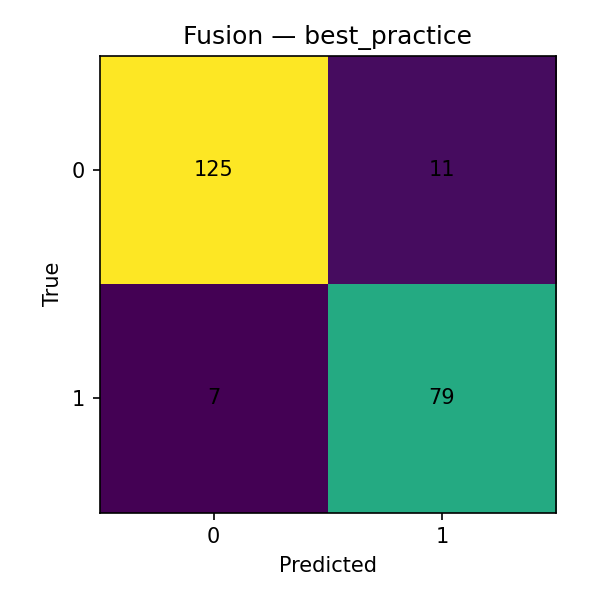
[Figure G4](#fig:G4). Rules Best Practice Confusion Matrix



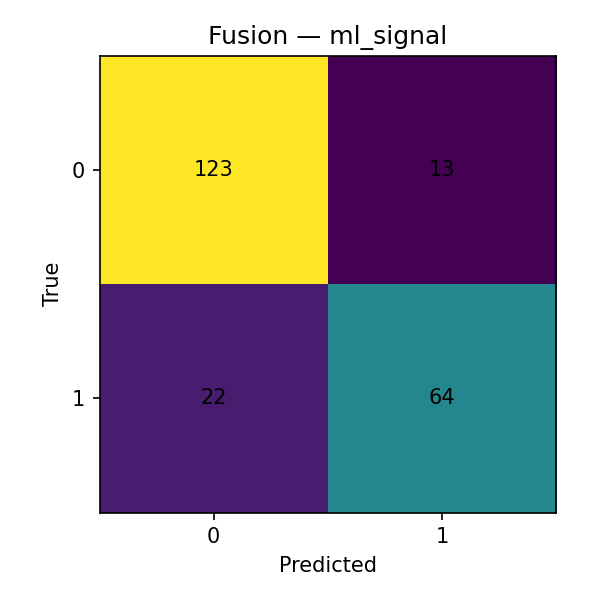
[Figure G5](#fig:G5). Rules ML Signal Confusion Matrix



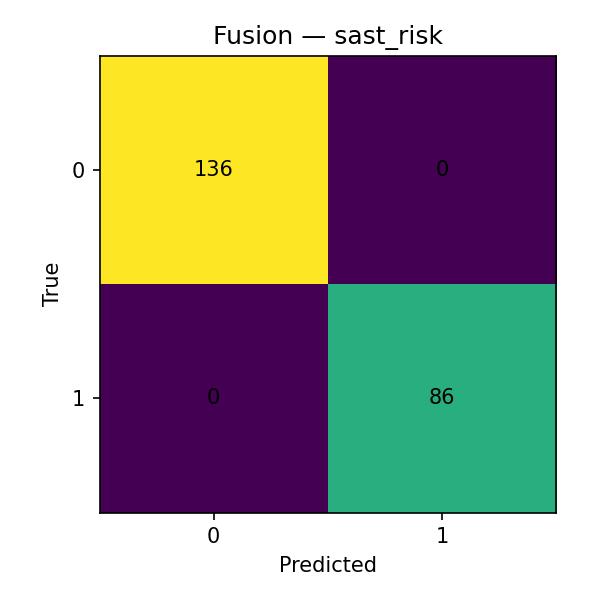
[Figure G6](#fig:G6). Rules SAST Risk Confusion Matrix



[Figure G7](#fig:G7). Fusion Best Practice Confusion Matrix



[Figure G8](#fig:G8). Fusion ML Signal Confusion Matrix



[Figure G9](#fig:G9). Fusion SAST Risk Confusion Matrix

# **ROC curves per label (Model)**

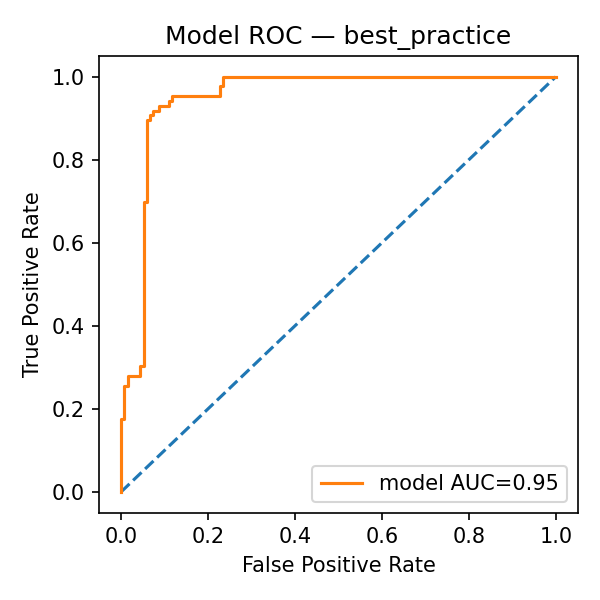


Figure H‑1. Model Best Practice ROC Curve

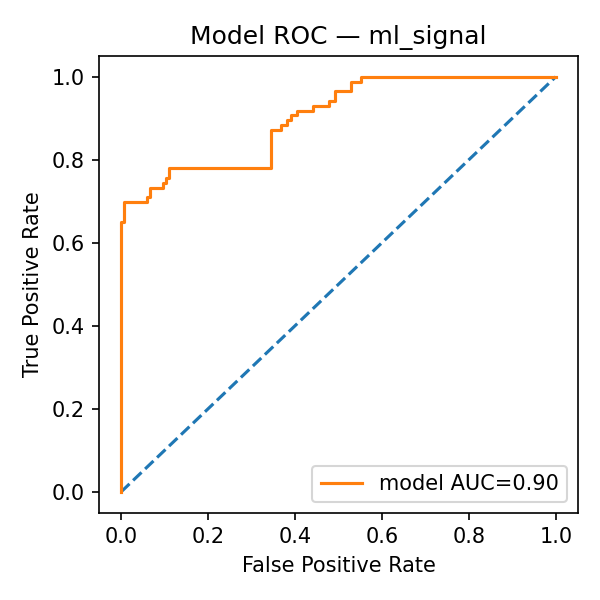


Figure H‑2. Model ML Signal ROC Curve

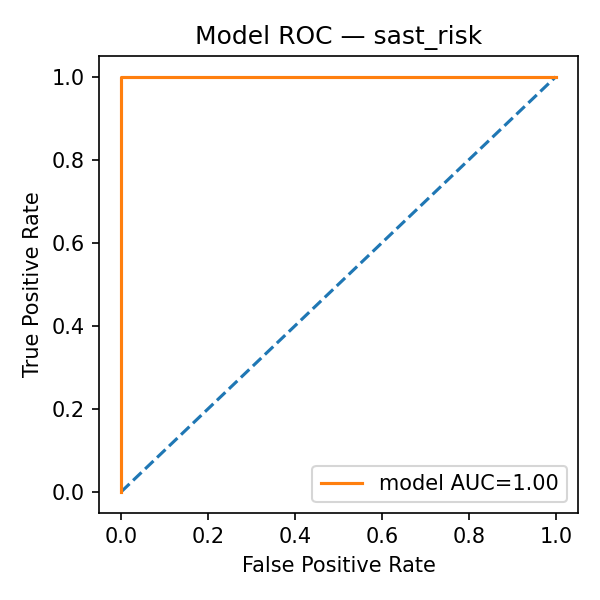


Figure H‑3. Model SAST Risk ROC Curve

# **Threshold sweep plots**

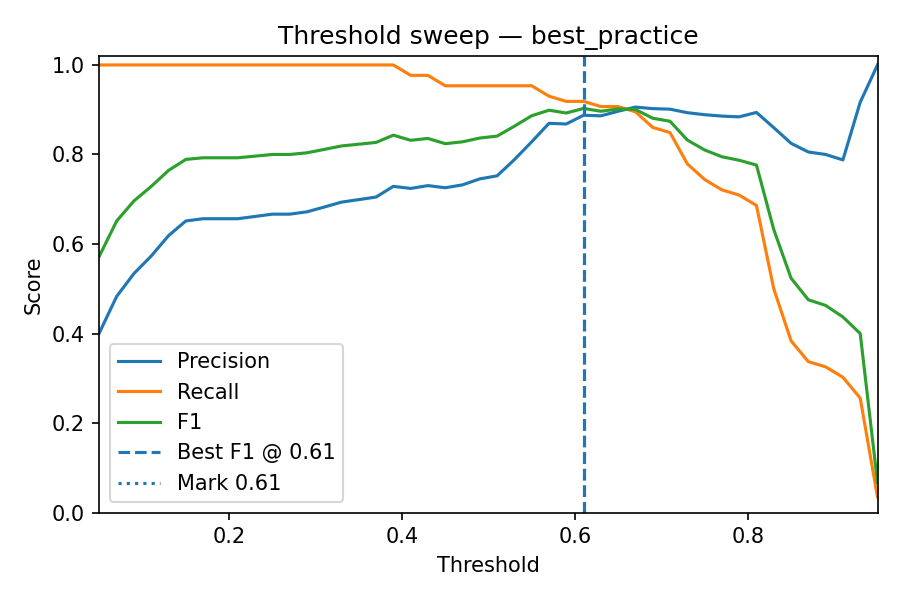


Figure I‑1. Best Practice Threshold Sweep

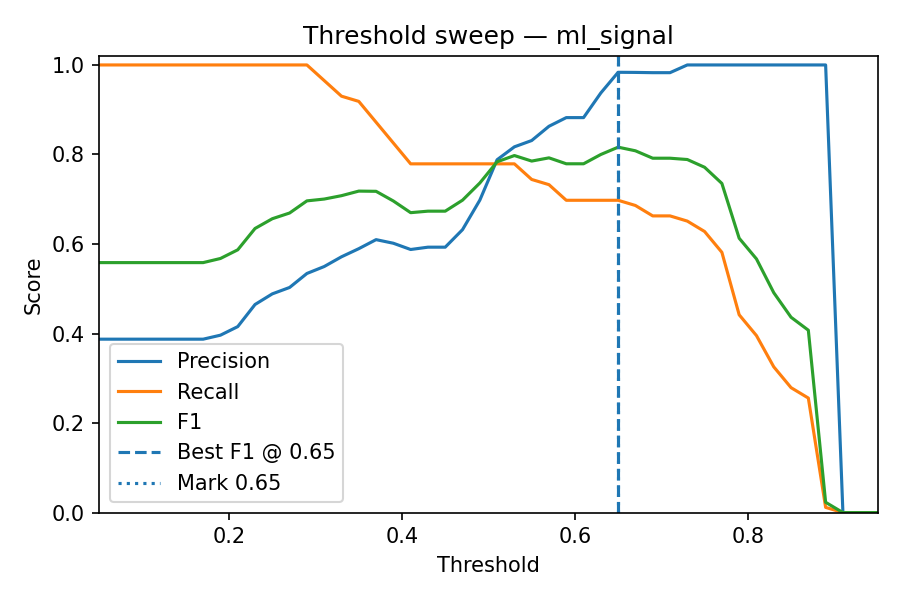


Figure I‑2. ML Signal Threshold Sweep

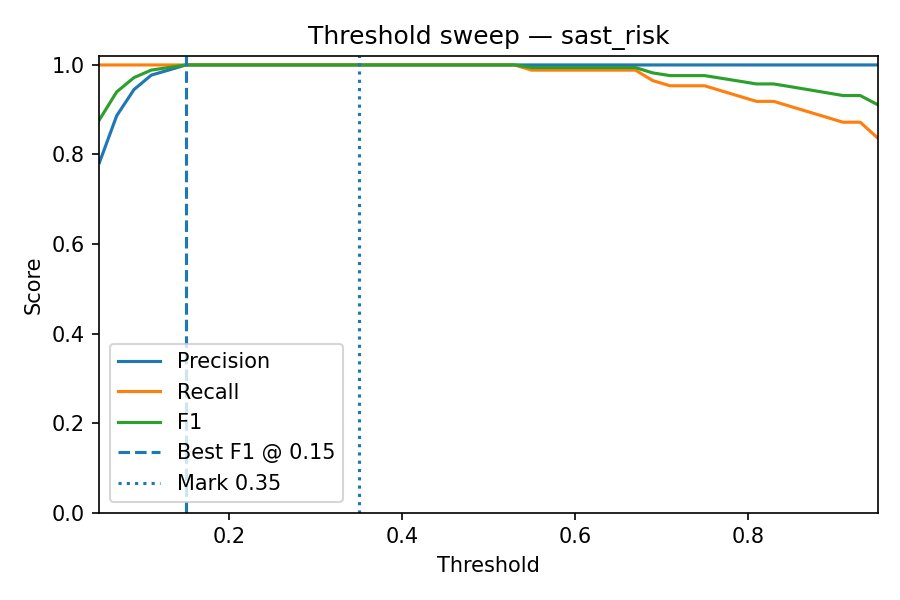


Figure I‑3. SAST Risk Threshold Sweep

# **Calculating optimal thresholds**

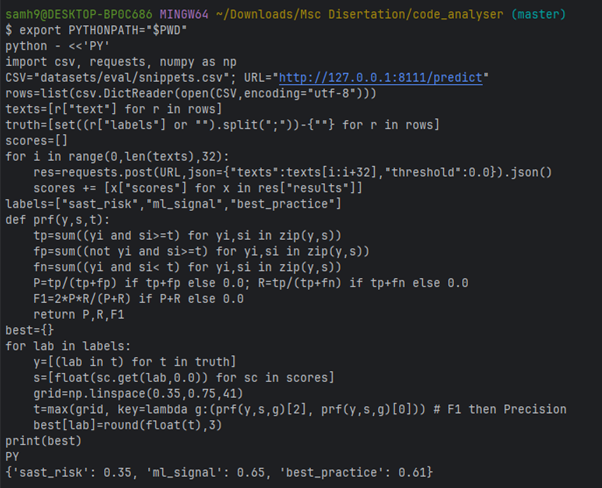


Figure J‑1. Script to calculate optimal threshold

# **Sanity Tests**

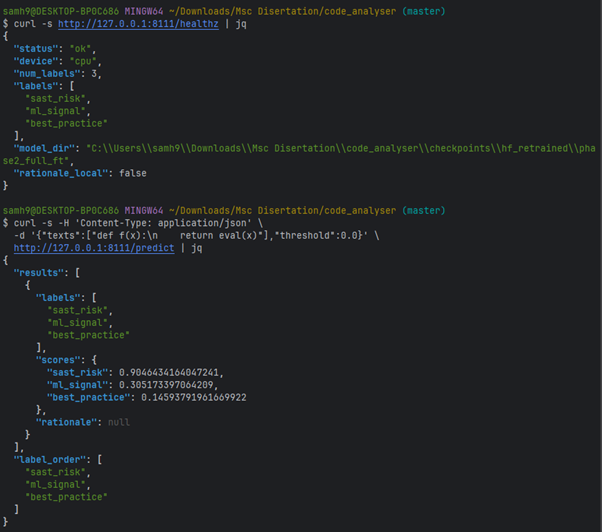


Figure K‑1. Smoke testing the microservice

# **Starting the microservice**

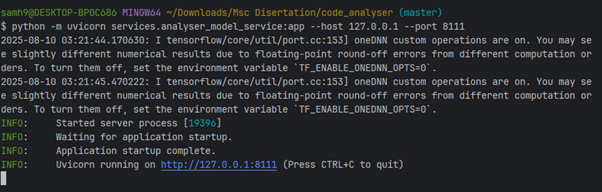


Figure L‑1. Starting the microservice

# **Repository Structure & Reproducibility Readme**

## **M.1 Summary**

This appendix records the public artefacts that enable full reproduction of the results in Chapters 5–6.

* Codebase: <https://github.com/SamHarrison1999/code-analyser>
* Annotated data & exported artefacts: <https://huggingface.co/datasets/SamH1999/fintech-ai-annotations/tree/main>

## **M.2 Repository structure (abridged, from repo root)**

code-analyser/

├─ .github/workflows/ # CI pipelines (lint, tests, coverage)

├─ artifacts/ # Exported tables/figures and report assets

├─ checkpoints/ # Saved checkpoints from training/eval

├─ datasets/ # Local dataset/cache hooks

├─ eval/ # Evaluation configs and helpers

├─ logs/tensorboard/ # TensorBoard event logs

├─ models/trained\_model/ # Trained model weights/artefacts

├─ runs/ # Experiment run directories

├─ scripts/ # CLI scripts: eval, k-fold, threshold sweeps, calibration

├─ services/ # Microservices (e.g., FastAPI endpoints, gating policy)

├─ src/ # Core library: fusion logic, feature extraction, utils

├─ tools/ # Dev tools, helpers, and integration utilities

├─ codecov.yml # Codecov configuration

├─ pyproject.toml # Build + project metadata

├─ requirements.txt # Python dependencies (pinning)

├─ setup.bat / .ps1 / .sh # Convenience setup scripts

└─ README.md # Quickstart and documentation

## **M.3 Environment**

* Python 3.10+ recommended.
* Key packages: fastapi, transformers, torch, scikit-learn, pandas, numpy, uvicorn, pydantic.
* See Appendix C2 for exact pinned versions; a lockfile/requirements.txt is provided in the repo.

## **M.4 Repro steps (outline)**

1) Clone and set up:

git clone https://github.com/SamHarrison1999/code-analyser

cd code-analyser

python -m venv .venv && . .venv/bin/activate

pip install -r requirements.txt

2) Fetch annotated data (if not auto-pulled by scripts):

See dataset card at the Hugging Face URL above.

3) Run evaluation:

python scripts/run\_eval.py --config configs/exp/fusion.yaml

4) Reproduce threshold sweeps and calibration:

python scripts/run\_threshold\_sweeps.py --dataset local

python scripts/calibrate\_temperature.py --per-label

5) Reproduce 5-fold CV:

python scripts/run\_kfold.py --k 5

6) Export tables/figures used in Chapter 5:

python scripts/export\_report\_assets.py --out artifacts/report/

## **M.5 CI usage (minimal)**

* Run the analyser in advisory mode:

python -m analyser.service --advisory

* Raise/lower per-label θ in configs/policy.yaml (no retraining required).

## **M.6 Models & checkpoints**

* Model weights and calibration parameters are referenced in models/trained\_model/ and configs/.
* Additional exported artefacts are available in the Hugging Face dataset folder for retraining/recalibration.

## **M.7 Versioning and integrity**

* The service exposes a /version endpoint (see main text) that prints pinned package versions and model checksums. Record these in Appendix C to ensure reproducibility.

## **M.8 Notes**

* Credentials (API tokens) are read at start-up and retained in memory only; they are not logged.
* If external model or tool calls are unavailable, the system degrades gracefully (see Chapter 6 §6.6).