PROFESSIONAL MASTER'S DISSERTATION

Evaluating Architectural Strategies for Personal Data Detection in Java Source Code Using Transformer Classifiers and Large Language Models

Fabiano Damasceno Sousa Falcão

Brasília, May 26th 2026

UNIVERSITY OF BRASILIA

Faculty of Technology
DEPARTMENT OF ELECTRICAL ENGINEERING

UNIVERSITY OF BRASILIA
Faculty of Technology

PROFESSIONAL MASTER'S DISSERTATION

Evaluating Architectural Strategies for Personal Data Detection in Java Source Code Using Transformer Classifiers and Large Language Models

Avaliação de Estratégias Arquiteturais para Detecção de Dados Pessoais em Código-Fonte Java Utilizando Classificadores Transformers e Grandes Modelos de Linguagem

Fabiano Damasceno Sousa Falcão

Advisor: Professor Edna Dias Canedo, Ph.D

PUBLICATION: PPEE.MP.114

Brasília, May 26th 2026

UNIVERSITY OF BRASILIA
Faculty of Technology

PROFESSIONAL MASTER'S DISSERTATION

Evaluating Architectural Strategies for Personal Data Detection in Java Source Code Using Transformer Classifiers and Large Language Models

Fabiano Damasceno Sousa Falcão

Professional Master's Dissertation submitted to the Department of Electrical Engineering as a partial requirement to obtain the degree of Master in Electrical Engineering

Examination Board

Professor Edna Dias Canedo, Ph.D, FT/UnB ______________________
Advisor

Professor Fábio Lúcio Lopes de Mendonça, Ph.D, FT/UnB ______________________
Internal Examiner

Professor Emmanuel Sávio Silva Freire, Ph.D, Instituto Federal do Ceará - IFCE/Morada Nova ______________________
External Examiner

Professor Georges Daniel Amvame Nze, Ph.D, FT/UnB ______________________
Internal Examiner

CATALOG SHEET

FALCÃO, FABIANO DAMASCENO SOUSA
Evaluating Architectural Strategies for Personal Data Detection in Java Source Code Using Transformer Classifiers and Large Language Models [Federal District] 2026.
xvi, 131 p., 210 x 297 mm (ENE/FT/UnB, Master, Electrical engineering, 2026).
Professional Master's Dissertation - University of Brasilia, Faculty of Technology.
Department of Electrical Engineering

1. Personal Data Detection	2. Java Source Code
3. Large Language Models	4. Privacy-Aware Software Engineering
I. ENE/FT/UnB	II. PPEE.MP.114

BIBLIOGRAPHIC REFERENCE

FALCÃO, F.D.S. (2026). *Evaluating Architectural Strategies for Personal Data Detection in Java Source Code Using Transformer Classifiers and Large Language Models*. Professional Master's Dissertation, Department of Electrical Engineering, University of Brasília, Brasília, DF, 131 p.

ASSIGNMENT OF RIGHTS

AUTHOR: Fabiano Damasceno Sousa Falcão
TITLE: Evaluating Architectural Strategies for Personal Data Detection in Java Source Code Using Transformer Classifiers and Large Language Models.
GRADE: Master in Electrical Engineering YEAR: 2026

Permission is granted to the University of Brasilia to reproduce copies of The Professional Master's Dissertation and loan or sell such copies for academic and scientific purposes only. The authors reserve other publication rights and no part of The Professional Master's Dissertation may be reproduced without the written permission of the authors.

Fabiano Damasceno Sousa Falcão
Dept. of Electrical Engineering (ENE) - FT
University of Brasilia (UnB)
Darcy Ribeiro Campus
CEP 70919-970 - Brasília - DF - Brazil

DEDICATION

First and foremost, I would like to dedicate this work to God, who granted me the strength, health, wisdom, and grace to accomplish this important milestone and to persevere throughout this long journey.

I dedicate this work to my beloved wife, Ana Carolina, whose unwavering love, support, encouragement, and companionship have been a constant source of strength and inspiration. I am deeply grateful for everything she has done for me throughout my academic journey and professional career.

I also dedicate this work to my daughters, Maria Regina and Maria Luiza, who have been a continual source of joy, motivation, and inspiration in my life.

This achievement belongs not only to me, but also to my family, who stood by my side throughout this journey.

ACKNOWLEDGMENTS

I would like to express my gratitude to God and to my family for their unwavering support throughout this journey.

I am especially grateful to my mother-in-law, Wania, for her immeasurable help in caring for my family during this long and demanding period. I also express my deepest gratitude to my mother, Alzenir, for always believing that this achievement would be possible, for instilling in me the values of education and perseverance, and above all for the countless sacrifices she made throughout my childhood and youth to provide me with a quality education.

I would like to express my sincere appreciation to my advisor, Ph.D. Edna Dias Canedo, for her invaluable guidance, academic rigor, dedication, and continuous support throughout this research. Her mentorship was fundamental not only to the completion of this dissertation, but also to the publication of the scientific articles related to this work. Without her guidance and encouragement, this achievement would not have been possible.

I am also grateful to my friend PhD Ricardo José Menezes Maia, who became an important research reference for me in the field of Artificial Intelligence and generously shared valuable insights and advice throughout this journey.

I especially thank my friends and colleagues in the IT Department of the Superior Electoral Court (TSE), where I worked for almost seventeen years. During that time, I experienced an exceptionally high level of professionalism in the planning and management of Brazil's computerized electoral process, particularly in matters related to information security. This experience profoundly shaped my professional mindset and greatly influenced my perspective on technology, management, governance, software engineering, and cybersecurity. I also extend my gratitude to my new friends and colleagues in the IT Department of the Chamber of Deputies, where each conversation with them provided a valuable exchange of academic and professional knowledge and experience.

Finally, I would like to thank the professors, administrative staff, and fellow students of the Professional Postgraduate Program in Electrical Engineering (PPEE) at the University of Brasília (UnB) for their excellence in teaching, academic support, collaboration, and institutional commitment, all of which were fundamental to this master's journey.

RESUMO

A detecção de informações de identificação pessoal (*Personally Identifiable Information* – PII) em código-fonte tornou-se um importante desafio na engenharia de software orientada à privacidade, particularmente diante do aumento das exigências regulatórias e da ampla adoção de grandes repositórios de software e de fluxos de desenvolvimento de software assistidos por Inteligência Artificial (IA). Avanços recentes em modelos baseados em *Transformers* e em Grandes Modelos de Linguagem (*Large Language Models* – LLMs) possibilitaram novas abordagens para a detecção de PII; entretanto, ainda existem evidências empíricas controladas limitadas sobre os trade-offs arquiteturais associados a diferentes estratégias de integração.

Esta dissertação investiga três estratégias arquiteturais para a detecção de PII em código-fonte Java: (i) um *pipeline* baseado exclusivamente em classificadores fundamentados em modelos *Transformers*; (ii) um *pipeline* híbrido classificador+LLM, no qual um LLM atua como juiz semântico sobre candidatos gerados pelos classificadores; e (iii) um *pipeline* de extração estruturada centrado em LLM, que combina extração de ponta a ponta com etapas determinísticas de validação e sanitização. Para suportar uma comparação controlada, o estudo adota um *framework* experimental unificado com pré-processamento compartilhado, configurações determinísticas de inferência, templates estruturados de prompt, correspondência em nível de valor (*value-level matching*) e procedimentos de avaliação que preservam os artefatos experimentais.

A avaliação empírica baseia-se em múltiplas execuções controladas de experimentos que utilizam um conjunto de dados sintético de código Java e artefatos integralmente arquivados, entre os quais se encontram saídas estruturadas, rastros de execução, configurações resolvidas e registros brutos das interações com LLMs. Os resultados mostram que as abordagens baseadas em classificadores geralmente alcançaram os maiores níveis de *recall*, mas apresentaram precisão substancialmente menor devido à ambiguidade lexical e aos efeitos de supergeração (*over-generation*). A arquitetura híbrida reduz parte da carga de falsos positivos, mas também remove sistematicamente verdadeiros positivos, o que produz um comportamento de filtragem orientado à precisão que prioriza a seletividade em detrimento da cobertura. Em contraste, o *pipeline* de extração estruturada centrado em LLM alcança um equilíbrio mais favorável entre precisão e *recall* quando a estabilidade da extração estruturada e a sanitização determinística são mantidas.

Os achados demonstram que a estratégia de integração arquitetural e as restrições impostas pelo *pipeline* influenciam substancialmente o comportamento da detecção de PII, frequentemente de forma mais significativa do que a escolha isolada de modelos individuais. Além disso, o estudo demonstra que experimentos de engenharia de software baseados em LLMs podem alcançar reprodutibilidade e auditabilidade por meio de configurações estruturadas, pós-processamento determinístico e práticas de avaliação centradas em artefatos.

Este trabalho contribui com evidências empíricas sobre estratégias de integração arquitetural

que envolvem classificadores baseados em *Transformers* e LLMs para análise de código-fonte orientada à privacidade e fornece um framework experimental reprodutível para a avaliação de pipelines de engenharia de software baseados em LLMs.

// ABSTRACT

The detection of personally identifiable information (PII) in source code has become an important challenge in privacy-aware software engineering, particularly under increasing regulatory requirements and the widespread adoption of large-scale software repositories and software development workflows assisted by Artificial Intelligence (AI). Recent advances in transformer-based models and Large Language Models (LLMs) have enabled new approaches to PII detection; however, there is still limited controlled empirical evidence regarding the architectural trade-offs associated with different integration strategies.

This dissertation investigates three architectural strategies for detecting PII in Java source code: (i) a classifier-only pipeline based on transformer models, (ii) a hybrid classifier+LLM pipeline in which an LLM acts as a semantic judge over classifier-generated candidates, and (iii) an LLM-centered structured-extraction pipeline combining end-to-end extraction with deterministic validation and sanitization stages. To support controlled comparison, the study adopts a unified experimental framework with shared preprocessing, deterministic inference settings, structured prompt templates, value-level matching, and artifact-preserving evaluation procedures.

The empirical evaluation is based on multiple controlled experiment runs using a synthetic Java dataset and fully archived artifacts, including structured outputs, execution traces, resolved configurations, and raw LLM interaction logs. The results show that classifier-based approaches generally achieved the highest recall levels but suffered from substantially lower precision due to lexical ambiguity and over-generation effects. The hybrid architecture reduces a portion of the false positive load but also systematically removes true positives, resulting in a precision-oriented filtering behavior that prioritizes selectivity over coverage. In contrast, the LLM-centered structured-extraction pipeline achieves a more favorable precision-recall balance when structured extraction stability and deterministic sanitization are maintained.

The findings demonstrate that architectural integration strategy and pipeline constraints substantially influence PII detection behavior, often more significantly than the isolated choice of individual models. In addition, the study shows that reproducible and auditable LLM-based software engineering experiments can be achieved through structured configurations, deterministic post-processing, and artifact-centered evaluation practices.

This work contributes empirical evidence on architectural integration strategies involving transformer classifiers and LLMs for privacy-oriented source code analysis and provides a reproducible experimental framework for evaluating LLM-based software engineering pipelines.

INDEX

1 INTRODUCTION **1**
1.1 RESEARCH PROBLEM 2
1.2 JUSTIFICATION 3
1.3 OBJECTIVES 4
1.3.1 GENERAL OBJECTIVE 4
1.3.2 SPECIFIC OBJECTIVE 5
1.4 EXPECTED RESULTS AND CONTRIBUTION 5
1.5 PUBLICATIONS OF THIS DISSERTATION 6
1.6 RESEARCH METHODOLOGY 7
1.7 DISSERTATION STRUCTURE 9

2 BACKGROUND AND RELATED WORK **10**
2.1 PERSONAL DATA AND PRIVACY IN SOFTWARE ENGINEERING 10
2.1.1 DEFINITIONS OF PERSONAL DATA UNDER DATA PROTECTION LAWS 11
2.1.2 PERSONAL DATA IN SOFTWARE ARTIFACTS AND SOURCE CODE 12
2.1.3 COMPLIANCE AND ACCOUNTABILITY IMPLICATIONS 13
2.2 PRIVACY-AWARE SOFTWARE ENGINEERING AND PRIVACY AS CODE 13
2.2.1 THE PRIVACY AS CODE PARADIGM 14
2.2.2 INTEGRATING PRIVACY-ENHANCING TECHNOLOGIES INTO THE SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC) 14
2.2.3 GDPR COMPLIANCE IN CONTINUOUS INTEGRATION (CI) AND CONTINUOUS COMPLIANCE (CC) 15
2.3 AUTOMATED DETECTION OF PERSONAL DATA IN SOURCE CODE 15
2.3.1 RULE-BASED AND PATTERN-BASED APPROACHES 16
2.3.2 MACHINE LEARNING AND TRANSFORMER-BASED CLASSIFIERS 16
2.4 LARGE LANGUAGE MODELS FOR PRIVACY AND CODE ANALYSIS 16
2.4.1 LLMS FOR PRIVACY AND COMPLIANCE TASKS 17
2.5 RELATED WORK 17
2.6 SUMMARY 19

3 RESEARCH METHODOLOGY **20**
3.1 EMPIRICAL EVALUATION IN SOFTWARE ENGINEERING 20
3.1.1 RESEARCH QUESTIONS AND EVALUATION GOALS 21
3.1.2 LLMS AS SUBSTITUTES FOR MANUAL ANNOTATION IN SOFTWARE ENGINEERING (SE) EVALUATIONS 21
3.1.3 CONTROLLED EXPERIMENTS IN SOFTWARE ENGINEERING 22

3.2 Experimental Design and Evaluation Protocol 22
3.2.1 Evaluated Models 24
3.2.2 Factors, Treatments, and Controls 26
3.2.3 Detection Infrastructure and Experimental Execution 27
3.3 Datasets and Ground Truth 27
3.3.1 Synthetic Dataset Generation and Validation Infrastructure 28
3.3.2 Dataset Characteristics and Distribution 30
3.4 Study Design and Objectives 33
3.4.1 Specific Objectives 33
3.4.2 Evaluation Pipelines 34
3.5 Scoring Procedure and Reporting 37
3.5.1 LLM-as-a-Judge as a Methodological Component 38
3.6 Evidence-Oriented Experimental Design 38
3.6.1 Evidence Bundle Specification 40
3.6.2 Verification-Oriented Controls 42
3.6.3 Traceability Across Pipeline Stages 43
3.6.4 Reproducibility vs. Verifiability 44
3.6.5 LLM as First-Class Experimental Component 45
3.7 Threats to Validity 45
3.8 Summary 48

4 Experimental Evaluation 50
4.1 Experimental Setup 50
4.1.1 Execution Environment 50
4.2 Overview of the Experimental Evaluation 52
4.3 Research Questions 55
4.4 Results by Pipeline 55
4.5 Pipeline-Level Error Patterns 64
4.6 Cross-Pipeline Comparative Analysis 64
4.7 Threats to Validity 67
4.8 Summary 68

5 Discussion 69
5.1 Architectural Interpretation of the Results 69
5.2 Analysis of the Research Questions 71
5.2.1 RQ1: Effectiveness of transformer-based classifiers 71
5.2.2 RQ2: Effectiveness of hybrid architectures 72
5.2.3 RQ3: Effectiveness of LLM-centered extraction 72
5.3 Implications for Privacy-Aware Software Engineering 73
5.3.1 Operational Trade-offs 74

5.3.2 Verifiability and Evidence-Oriented Experimental Design 75
5.4 Limitations and Open Challenges 75
5.5 Summary 76

6 Conclusion 77
6.1 Summary of Contributions 77
6.2 Key Findings 78
6.3 Implications for Research and Practice 79
6.4 Future Work 79
6.5 Final Remarks 81

References 82

Appendix 89

I Synthetic Dataset Generation Framework 90
I.1 Design Goals and Scope 90
I.2 High-Level Architecture 92
I.3 Input Configuration and Parameters 94
I.4 Label Taxonomy and Annotation Strategy 95
I.5 Synthetic Code Generation Process 96
I.6 Execution Artifacts and Directory Layout 97
I.7 Ground-Truth Output Format 98
I.8 Reproducibility and Versioning 99

II PII Detection Framework 101
II.1 Architectural Overview 101
II.2 Detection Pipeline 103
II.2.1 Stage 1: Configuration resolution and execution context 105
II.2.2 Stage 2: Dataset and JSONL loading 105
II.2.3 Stage 3: Detection execution 105
II.2.4 Stage 4: Deterministic post-processing and normalization 105
II.2.5 Stage 5: Evaluation and metrics 106
II.2.6 Stage 6: Artifact persistence and reporting 106
II.3 Supported Detection Strategies 106
II.3.1 Pipeline 1: Classifier-only (Transformer ensemble) 106
II.3.2 Pipeline 2: Hybrid (Classifier candidates + LLM-as-a-Judge) 106
II.3.3 Pipeline 3: LLM-only (Structured extraction) 107
II.4 Normalization and Output Schema 107
II.4.1 Label mapping and equivalences 108
II.4.2 Value normalization and validity filters 108

II.4.3 Strict JSON schemas and validation 108
II.4.4 GDPR-focused normalization 109
II.5 Execution Modes and Configurations 109
II.5.1 Command-line execution 109
II.5.2 Notebook-based execution 109
II.5.3 Configuration-driven experimentation 110
II.6 Artifact Retention and Logging 110
II.6.1 LLM interaction logging (Ollama) 110
II.6.2 Prompt templates (Jinja2) and template-only policy 114
II.6.3 Java parsing utilities 115
II.7 Configuration Resources in `shared/config` 115
II.7.1 `label_mapping.yml`: Canonical taxonomy and per-classifier label mapping 115
II.7.2 `label_equivalences.yml`: Classifier-specific direct maps and equivalence sets 115
II.7.3 `label_equivalences_p3.yml`: LLM-output normalization and alias collapse (P3 isolation) 116
II.7.4 `gdpr_labels.yml`: GDPR allowlist with descriptions and examples 116
II.7.5 `gdpr_labels_p3.yml`: GDPR allowlist + aliases + exclusions for LLM-only 116
II.7.6 `p2_hybrid_prefilter.yml`: Hybrid prefilter heuristics (routing and thresholds) 116
II.7.7 `p2_hybrid_sanitizer.yml`: Hybrid sanitization rules and telemetry 116
II.8 Structured Output Enforcement: Prompt Templates and JSON Schemas 117
II.8.1 Jinja2 Prompt Templates for LLM-Based Pipelines 117
II.8.2 Deterministic Response Sanitization 119
II.8.3 JSON Schemas for Structured Output Validation 119
II.8.4 Schema Validation and Failure Handling 119
II.9 Summary 120

III Experimental Configuration and Reproducibility Artifacts 121
III.1 Dataset Snapshot and Ground-Truth Schema 121
III.1.1 Record Structure 121
III.1.2 Annotation Fields and Label Semantics 122
III.1.3 Negative Samples and Matching Assumptions 122
III.1.4 Dataset Versioning and Integrity Checks 122
III.2 Evaluation Protocol Invariants and Matching Criteria 122

III.2.1 Join Key and Sample Identity 123
III.2.2 Entity Matching Key 123
III.2.3 Metrics Outputs 123
III.3 Experimental Configuration Files 123
III.3.1 Configuration Directory Organization 123
III.3.2 Top-Level Configuration Schema 124
III.3.3 Comparability and Control Policies 124
III.3.4 Path Resolution and Portability Assumptions 124
III.4 Models, Backends, and Identifiers 124
III.4.1 Classifier Models (Pipeline 1) 124
III.4.2 LLM Models and Ollama Tags (Pipelines 2–3) 125
III.5 Experimental Execution and Generated Artifacts 125
III.5.1 Execution Orchestration 126
III.5.2 Output Directory Layout 126
III.5.3 Pipeline-Specific Intermediate Artifacts 126
III.5.4 Detection Outputs 126
III.5.5 Metrics and Aggregated Results 127
III.5.6 Execution Logs and LLM Interaction Records 127
III.6 Prompt Templates, Determinism, and Logging Controls 127
III.6.1 Prompt Templates 127
III.6.2 Determinism and Decoding Settings 127
III.6.3 LLM Interaction Log Format 128
III.7 Execution Environment and Software Stack 128
III.7.1 Local Workstation 128
III.7.2 Cloud Environment 128
III.7.3 Software Stack 128
III.8 Reproduction Guidelines and Constraints 129
III.8.1 Required Artifacts 129
III.8.2 Permitted and Non-Permitted Modifications 129
III.8.3 Hardware and Environment Sensitivity 129
III.8.4 Threats to Reproducibility 130
III.9 PII Category Distribution in Dataset 130

List of Figures

3.1 Experimental architecture overview 23
3.2 Pipelines comparison 35
3.3 P2 decision flow 36
3.4 P3 structured extraction flow 37
3.5 Evidence-ready pipeline 39
3.6 Verification-oriented controls 42
3.7 Traceability pipeline 43

4.1 Pipeline precision, recall, and F1-score comparison 53
4.2 P2 candidate-routing behavior 58
4.3 Representative category-level F1-score comparison for P1 and P3 62
4.4 F1-score variability across archived P3 LLM-centered extraction configurations. ... 63
4.5 Empirical behavior across evaluated pipelines 65

5.1 Architectural trade-off spectrum 69
5.2 Pipeline positioning by LLM dependence and deterministic control 73

I.1 Operational decision procedure for GDPR-scoped category inclusion 92
I.2 Layered architecture of the dataset generator 93
I.3 Pipeline and dataflow of the dataset generator 93
I.4 Ground-truth JSONL schema and invariants 99

II.1 Layered architecture 102
II.2 Execution lifecycle 103
II.3 Pipeline comparison 104
II.4 Artifacts per run (P1) 111
II.5 Artifacts per run (P2) 112
II.6 Artifacts per run (P3) 113
II.7 Structured output enforcement 118

List of Tables

3.1 Transformer-based classifiers evaluated in Pipeline 1 24
3.2 Open-weight LLMs evaluated in Pipelines 2 and 3 25
3.3 Experimental factors, treatments, and controls 26
3.4 Summary statistics of the synthetic Java PII dataset used in the study........ 31
3.5 Dataset distribution overview 32
3.6 Pipelines comparison summary 36
3.7 Evidence bundle specification 41

4.1 Summary of experimental configurations across pipelines 51
4.2 Representative aggregate metrics across the evaluated pipeline regimes 53
4.3 Representative aggregate results across pipelines 54
4.4 Performance of individual classifiers and OR-ensemble aggregation in Pipeline 1 .. 56
4.5 Aggregate metrics across archived P2 hybrid judge executions 57
4.6 Aggregate metrics across archived P3 LLM-centered extraction executions 59
4.7 Representative best- and worst-performing categories observed in P1 and P3 61
4.8 Representative category-level F1-score patterns for P1 and P3 61
4.9 Pipeline trade-offs observed in the experimental evaluation 66
4.10 Summary comparison of the evaluated pipelines. 66

5.1 Architectural characteristics observed across the evaluated pipelines. 71

II.1 Repository map. 102
II.2 Pipeline features. 104
II.3 Common run-level artifacts (all pipelines). 114

III.1 Distribution of PII categories in the synthetic Java dataset. 131

LIST OF SYMBOLS

Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
AST	Abstract Syntax Tree
CCPA	California Consumer Privacy Act
CC	Continuous Compliance
CD	Continuous Delivery
CI	Continuous Integration
CLI	Command-Line Interface
CPRA	California Privacy Rights Act
FM	Foundation Model
FN	False Negative
FP	False Positive
GDPR	General Data Protection Regulation
GPU	Graphics Processing Unit
JSON	JavaScript Object Notation
JSONL	JSON Lines
LGPD	Brazilian General Data Protection Law
LLM	Large Language Model
LLM4SE	Large Language Model for Software Engineering
NDJSON	Newline-Delimited JSON
P1	Pipeline 1: classifier-only baseline
P2	Pipeline 2: hybrid classifier and LLM-as-a-judge pipeline
P3	Pipeline 3: LLM-centered structured extraction pipeline
PET	Privacy-Enhancing Technology
PII	Personally Identifiable Information
SDLC	Software Development Life Cycle
SE	Software Engineering
Regex	Regular-expression
TN	True Negative
TP	True Positive
VRAM	Video Random Access Memory
YAML	YAML Ain't Markup Language

1 INTRODUCTION

Privacy and personal data protection have become first-class concerns in software engineering because privacy requirements increasingly materialize in software artifacts, development practices, and operational workflows (1). Empirical work on privacy-relevant source code shows that privacy-related concerns can be embedded in code elements that developers must locate and reason about to support compliance and governance activities (2).

Regulatory frameworks such as the General Data Protection Regulation (GDPR) and the Brazilian General Data Protection Law (LGPD) establish legally binding obligations for the processing of personal data and define personal data broadly as information relating to an identified or identifiable natural person (3, 4). This broad legal scope implies that identifiers and personal attributes that appear in software artifacts, including source code, can be in scope for compliance controls and accountability mechanisms (3, 4).

Prior research characterizes privacy-relevant analysis in codebases as a non-trivial task that requires identifying code fragments tied to privacy-related behaviors and data handling (2). In parallel, literature on "privacy as code" argues that privacy concerns are increasingly expressed through concrete software constructs, which motivates privacy-aware analysis methods that are compatible with engineering pipelines (1).

Studies investigating Large Language Models (LLMs) for privacy-related code analysis report that LLMs can be used to identify personal data processing aspects in source code, supporting richer context-sensitive reasoning than purely surface-level approaches (5). However, empirical evidence on LLM-as-a-judge indicates that using LLMs as evaluators introduces reliability and stability considerations that depend on evaluation design choices and can influence observed outcomes (6, 7).

These findings motivate systematic and controlled comparisons of alternative integration strategies between classifiers and LLMs for entity-level detection of explicit Personally Identifiable Information (PII) values in source code (8, 9), while emphasizing the need for documented artifacts and traceable experimental protocols to support auditability and independent verification in compliance-sensitive settings.

These artifacts include the preservation of experimental configurations, prompt templates, deterministic processing rules, and model inputs/outputs, enabling post-hoc inspection, traceability, and evidence-centric verification of experimental outcomes even when exact model re-execution is infeasible, with implementation, storage, and logging procedures detailed in Chapter 3.

1.1 RESEARCH PROBLEM

The automated detection of explicit personal data entities and privacy-relevant textual artifacts in source code has been approached through multiple paradigms, but the literature still reports heterogeneous assumptions, artifacts, and evaluation settings that hinder direct comparison (10, 2).

Empirical research on privacy-relevant source code highlights that privacy relevance is often context-dependent, which challenges approaches that rely only on shallow syntactic cues (2). Recent work demonstrates the feasibility of using LLMs to identify personal data processing in source code, but it also frames the problem as one where practical trade-offs must be examined rather than assumed (5). At the same time, the broader body of research on LLM-as-a-judge shows that judge behavior can be sensitive and can introduce biases or instability, which is critical when judge outputs become decision points in hybrid pipelines (6, 7).

Studies discussing privacy risks in source code using LLMs further report both potential and problems, reinforcing the need for evaluations that explicitly quantify error trade-offs such as false positives and false negatives (10).

Controlled experimentation is a well-established method in software engineering for producing credible comparative evidence when multiple alternatives must be evaluated under consistent conditions (8, 9).

In the context of GDPR- and LGPD-aligned interpretations of personal data, the consequences of missed detections and spurious alarms strengthen the need for comparable, transparent, and reproducible evaluations across competing strategies (3, 4).

To enable a fair comparison under controlled conditions, this dissertation operationalizes three integration strategies: classifier-only, hybrid classifier plus LLM-as-a-judge, and LLM-only under the same dataset and identical evaluation criteria, so that observed differences can be interpreted as relative trade-offs rather than artifacts of heterogeneous experimental settings (8, 9).

Importantly, these pipelines are not architecturally equivalent systems. Classifier-only detection, hybrid selective filtering, and end-to-end LLM extraction operate under different decision spaces and constraints. Accordingly, the comparison is intended to characterize architectural trade-offs rather than to establish a strict head-to-head benchmark between interchangeable methods.

All evaluated pipelines operate over structured textual Java snippets and do not incorporate Abstract Syntax Tree (AST) representations, taint analysis, data-flow tracking, or broader program-level semantic reasoning.

In this dissertation, effectiveness and trade-offs are operationalized in terms of standard detection and information-retrieval metrics (e.g., precision, recall, and F1-score) computed under a

shared dataset, label set, and explicit matching criteria, and complemented by error profiles (false positives and false negatives). Entities are matched using value-level keys, while positional offsets are intentionally ignored, reflecting the practical objective of identifying which PII values are present rather than their exact spans in source code.

To support applicability to mainstream software engineering practice, the experimental scope focuses on Java source code, with dataset characteristics and evaluation details described in Chapter 3. In this context, this dissertation adopts an architectural perspective on PII detection in source code, focusing on how different integration strategies between classifiers and LLMs influence detection effectiveness and trade-offs under controlled experimental conditions.

This dissertation therefore addresses the following research problem:

How do different strategies for integrating transformer-based classifiers and LLMs compare, in terms of effectiveness and trade-offs, when applied to the detection of personal data in Java source code under controlled experimental conditions?

This research problem is addressed from an architectural perspective in which the focus extends beyond aggregate detection performance to the structural behavior of different integration strategies, including how classifier ensembling, selective LLM-based filtering, and end-to-end structured extraction reshape precision–recall trade-offs under controlled conditions.

This research problem is motivated not only by the need to improve personal-data detection effectiveness, but also by the lack of controlled empirical evidence regarding how alternative architectural integration strategies behave under reproducible and auditable experimental conditions.

1.2 JUSTIFICATION

Research on privacy-relevant source code indicates that identifying privacy-related code elements is a practical challenge that affects compliance-oriented activities and motivates automated support (2). The literature on privacy as code argues that privacy concerns are increasingly encoded in software artifacts and practices, which increases the relevance of engineering methods that can systematically analyze such artifacts (1).

Legal frameworks such as GDPR and LGPD establish obligations and accountability expectations for the processing of personal data, which raises the stakes of undetected exposures and the operational burden of inaccurate tooling (3, 4).

Work assessing privacy risk detection with LLMs explicitly reports both potential and problems, supporting the claim that naive adoption can yield costly error profiles and that empirical validation is necessary (10). Evidence that LLMs can be applied to identify personal data processing in source code motivates further study, but it also implies that the suitability of LLM-driven

strategies should be assessed comparatively against alternatives under controlled conditions (5). Empirical studies on LLM-as-a-judge show that judge-based evaluation can be powerful but also subject to bias and instability, which strengthens the need to evaluate hybrid designs that rely on judge decisions (6, 7).

In hybrid pipelines, this concern is particularly important because LLM-based judging operates as a selective filtering layer over a fixed candidate pool. Under this design, reductions in false positives may be systematically coupled with suppression of true positives, making recall degradation a governance-relevant trade-off in privacy-sensitive software engineering workflows.

Methodological guidance for controlled experiments in software engineering emphasizes that fair comparisons require controlled variables, consistent datasets, and transparent reporting, which is directly aligned with the goals of this dissertation (8, 9). Recent literature on synthetic data generation and synthetic coding datasets supports the use of carefully designed synthetic benchmarks to enable controlled evaluation while mitigating ethical and legal risks associated with real personal data (11, 12).

Accordingly, a controlled and reproducible comparison of classifier-only, hybrid, and LLM-only strategies is justified to clarify effectiveness and architectural trade-offs for privacy-oriented source code analysis in software engineering contexts (10, 2, 5).

From an operational perspective, error profiles directly translate into cost and risk. High false-positive rates increase manual triage effort, slow down remediation workflows, and can reduce practitioner trust in automated tooling, whereas false negatives leave privacy-relevant exposures undetected and therefore weaken compliance assurance activities (2, 10).

Under GDPR- and LGPD-aligned governance, undetected personal data exposures may trigger reporting duties, sanctions, and reputational impact, while excessive false alarms can impose recurring inspection and process costs (3, 4). For this reason, compliance-sensitive adoption of detection strategies requires empirical evidence that characterizes trade-offs and supports auditability through traceable experimental protocols and transparent reporting when tools are embedded in engineering pipelines (6, 7).

1.3 OBJECTIVES

1.3.1 General Objective

To systematically compare and analyze architectural strategies for integrating transformer-based classifiers and large language models for personal-data detection in Java source code, with emphasis on their effectiveness and architectural trade-offs.

1.3.2 Specific Objective

To achieve the general objective of this research work, the following specific objectives were defined:

- To characterize the effectiveness of a classifier-only approach for detecting personal data in Java source code, establishing a baseline in terms of precision, recall, and related evaluation metrics.
- To evaluate the impact of hybrid detection strategies in which LLMs operate as selective judges over a fixed candidate pool, with particular attention to false-positive reduction, true-positive suppression, and the resulting precision–recall trade-offs.
- To assess the viability of end-to-end LLM-centered extraction strategies that combine probabilistic generation with deterministic schema validation, sanitization, normalization, and output constraints relevant to privacy-aware software engineering contexts.
- To compare the different integration strategies under identical datasets, categories of personal data, and evaluation criteria, enabling a controlled and fair analysis of their relative strengths and limitations.
- To analyze the implications of the observed results for privacy-aware software engineering and governance-support workflows, discussing how different detection strategies may support or hinder organizational decision-making under data protection regulations.

1.4 EXPECTED RESULTS AND CONTRIBUTION

The expected results of this research work are:

- The generation of empirical evidence that characterizes the performance of different strategies for detecting personal data in Java source code, highlighting their respective strengths, weaknesses, and trade-offs under controlled experimental conditions.
- A systematic comparison of classifier-only, hybrid, and LLM-only approaches using the same datasets, categories of personal data, and evaluation criteria, enabling consistent interpretation of effectiveness across strategies.
- The identification of practical implications associated with false positives and false negatives in privacy-related code analysis, supporting a clearer understanding of the risks involved in automated detection for privacy-sensitive software engineering workflows.

- An assessment of robustness and feasibility aspects relevant to the adoption of advanced detection strategies in real-world software engineering environments, including considerations related to consistency, auditability, and operational constraints.

The expected contributions of this research work are:

- An architectural contribution that characterizes how different integration strategies between transformer-based classifiers and LLMs affect the effectiveness of PII detection in source code.
- An empirical contribution demonstrating that selective LLM-based filtering over fixed candidate pools may systematically redistribute errors by reducing false positives while suppressing a non-trivial share of true positives under value-level matching.
- A systematic comparison of classifier-only, hybrid, and LLM-only approaches under controlled conditions, enabling consistent interpretation of their strengths, limitations, and operational trade-offs.
- A methodological contribution in the form of an evidence-ready experimental framework that supports controlled comparison, artifact-level auditability, and post-hoc verification.
- A practical contribution to privacy-aware software engineering by providing evidence-based insights to support the adoption of detection strategies in compliance-sensitive environments.

1.5 PUBLICATIONS OF THIS DISSERTATION

The research developed in this dissertation is associated with the following scientific manuscripts derived from different dimensions of the investigation conducted in this work.

The first manuscript focuses on the comparative evaluation of architectural strategies for detecting PII in Java source code using transformer-based classifiers, LLMs, and hybrid integration approaches:

- F. D. S. Falcão and E. D. Canedo,"Detecting PII in Java Software Engineering Pipelines," manuscript under review in *Empirical Software Engineering* (EMSE), Springer.

This manuscript investigates architectural trade-offs among classifier-only, hybrid classifier plus LLM-as-a-judge, and LLM-centered extraction strategies under controlled experimental conditions.

The second manuscript focuses on methodological and verification-oriented aspects of configuration-driven LLM-based experimental pipelines:

- F. D. S. Falcão, S. Soares, S. Freire, and E. D. Canedo, "Evidence-Ready Experiments for Verifiable LLM-Based Analysis Pipelines: A Blueprint for Empirical Software Engineering," manuscript under review for the *Brazilian Symposium on Software Engineering* (SBES), CBSoft 2026.

This manuscript proposes an evidence-ready experimental blueprint centered on auditability, traceability, deterministic processing, and post-hoc verification of LLM-based empirical pipelines under practical constraints such as model drift and limited re-execution feasibility.

Together, these works complement the broader objectives of this dissertation by combining architectural evaluation of privacy-aware detection pipelines with methodological contributions related to evidence-centric verifiability and experimental rigor in empirical software engineering.

1.6 RESEARCH METHODOLOGY

This research adopts an empirical and experimental methodology grounded in established practices of software engineering research, combining systematic literature analysis with controlled experimentation. The methodological design is guided by classical and contemporary references on empirical studies in software engineering, which emphasize rigor, reproducibility, and clear alignment between research questions, experimental design, and evaluation criteria (13, 8, 9).

The study begins with a targeted review of the scientific literature on privacy-aware software engineering and automated detection of personal data in source code. This review encompasses works on privacy as code (1), privacy-relevant code identification (2), and the use of LLMs for privacy and compliance-related tasks (10, 5, 14). In addition, the legal and regulatory context is established through authoritative data protection frameworks, including the GDPR (3) and the LGPD (4), which inform the definition and interpretation of personal data categories considered in the experiments.

Based on limitations identified in prior work, including the lack of controlled comparisons across alternative model integration strategies, heterogeneous evaluation settings, and limited support for reproducible architectural evaluation, this research adopts an experimental approach as its primary methodological strategy. Controlled experimentation is employed to systematically compare three distinct detection paradigms: classifier-only approaches based on transformer models, hybrid approaches combining classifiers with LLMs acting as judges, and end-to-end approaches relying exclusively on LLMs. The choice of controlled experiments is motivated by their suitability for isolating variables, ensuring comparability, and supporting causal reasoning in software engineering research (15, 8).

The experimental setup relies on a synthetic but carefully constructed dataset of Java source

code annotated with personal data categories, designed to support reproducibility and verifiability. The use of synthetic data follows recent advances demonstrating its effectiveness for controlled evaluation of code-related tasks while mitigating ethical and legal risks associated with real personal data (11, 12). All detection strategies are evaluated under identical conditions, using the same datasets, label definitions, and matching criteria, thereby enabling fair and systematic comparison. Accordingly, the dataset is used as controlled experimental evidence for comparative architectural evaluation rather than as a benchmark for estimating industrial deployment performance.

Validation of the results is primarily quantitative, based on established information retrieval and classification metrics such as precision, recall, and F1-score, which are widely used in empirical software engineering studies on automated analysis tools. These metrics are complemented by qualitative analysis focused on error patterns, particularly false positives and false negatives, in order to assess practical implications for privacy compliance in software engineering contexts. Observed results are discussed in relation to findings reported in previous studies on privacy risk detection in source code and on the use of LLMs as evaluators or decision-support components (10, 6, 7).

Finally, the methodological design emphasizes transparency, traceability, and evidence-centric verifiability. Experimental artifacts, resolved configurations, prompt templates, deterministic processing rules, and raw interaction logs are preserved to support independent auditability, metric recomputation, and post-hoc verification under controlled conditions, aligning the study with best practices for empirical research in software engineering (13).

Key threats to validity in this study include (i) the use of a synthetic Java dataset, which may not reflect the full diversity and evolution dynamics of real-world repositories, and (ii) the inherent non-determinism of LLM-based components, which can introduce run-to-run variability in detection and judge decisions in hybrid filtering (6, 7).

These threats are mitigated by holding constant the dataset, label set aligned with GDPR/LGPD, matching criteria, and evaluation metrics across all strategies, while preserving resolved configurations, prompt templates, deterministic processing rules, and raw interaction logs to support auditability, traceability, and evidence-centric post-hoc verification, with conclusions centered on relative comparisons rather than absolute performance claims (8, 9).

Although these mechanisms improve auditability, traceability, and deterministic verification of reported outcomes, they should not be interpreted as guarantees of semantic correctness, real-world validity, or regulatory completeness.

While the methodological design emphasizes reproducibility, transparency, and artifact-based verification, its primary role is to support a controlled architectural comparison of detection strategies. The main contribution of this study lies in the analysis of how different integration paradigms between classifiers and LLMs affect detection outcomes, rather than in the experimen-

tal framework itself.

In this dissertation, experimental validity is approached not only through controlled execution and reproducibility practices, but also through preservation of inspectable evidence artifacts that enable post-hoc verification of reported outcomes under realistic LLM ecosystem constraints such as model drift and limited re-execution feasibility.

1.7 DISSERTATION STRUCTURE

This dissertation is organized into chapters that reflect the logical progression of the research problem, methodological choices, experimental investigation, and analysis of results.

Chapter 2 presents the theoretical and regulatory background that supports the study. It reviews fundamental concepts related to personal data protection, privacy-aware software engineering, and automated analysis of source code, as well as the main legal frameworks that inform the interpretation of personal data categories.

Chapter 3 describes the research methodology in detail. It outlines the experimental design, datasets, evaluation criteria, and validation procedures adopted in the study, providing sufficient information to ensure transparency and reproducibility.

Chapter 4 reports the experimental setup and results obtained from the evaluation of the different detection strategies. The chapter presents quantitative and qualitative analyses that enable systematic comparison across approaches under controlled conditions.

Chapter 5 discusses the results in light of the research objectives and existing literature. This chapter critically analyzes the findings, examines their implications for privacy-aware software engineering workflows, and addresses the limitations and threats to validity of the study.

Finally, Chapter 6 concludes the dissertation by summarizing the main contributions of the research, revisiting the key findings, and outlining directions for future work.

2 BACKGROUND AND RELATED WORK

This chapter provides the conceptual, regulatory, and technical background necessary to understand the research problem addressed in this dissertation (3, 4, 16).

It reviews foundational definitions of personal data, explains how privacy concerns manifest in software artifacts and source code, and synthesizes technical approaches for detecting personal data in code, including rule-based, classifier-based, and LLM-based strategies (2, 10, 5).

Privacy compliance in software engineering is treated as an auditable engineering concern because accountability obligations require evidence that can be inspected and traced to concrete artifacts and decisions (3, 4, 14).

A broader context for this dissertation is the rapidly expanding body of work on Large Language Models for Software Engineering, which has been consolidated through systematic evidence synthesis (17).

This synthesis motivates treating LLM-based code analysis as an empirical and operational discipline, where limitations and artifact retention must be made explicit rather than assumed from anecdotal success cases (17).

Complementarily, evidence from practitioners highlights that privacy engineering outcomes depend not only on technical methods but also on organizational structures, practitioner mindset, and how privacy work is operationalized across roles and processes (16).

2.1 PERSONAL DATA AND PRIVACY IN SOFTWARE ENGINEERING

Personal data protection has become a central concern in software engineering due to the increasing role of software systems in collecting, processing, and storing information related to identifiable individuals (3, 4).

Regulatory frameworks such as the GDPR and the LGPD establish legally binding definitions of personal data that affect how software artifacts must be designed, analyzed, and governed (3, 4).

These obligations extend beyond runtime data handling to development-time artifacts such as source code, configuration files, test assets, and documentation that may embed identifiers or personal attributes (1, 2).

In compliance-oriented software engineering, the operational question is not only whether

personal data is processed at runtime, but also whether personal data is introduced, copied, or persisted across engineering artifacts in ways that create accountability and audit burdens (3, 4, 14).

Empirical work on privacy-aware software engineering emphasizes that privacy requirements increasingly materialize as concrete software constructs and development practices, motivating analysis methods that can be integrated into engineering pipelines and support governance activities (1).

This motivates systematic approaches for detecting personal data in source code under regulatory interpretations that hinge on identifiability and contextual re-identification risk (3, 4, 18).

2.1.1 Definitions of Personal Data under Data Protection Laws

The GDPR defines personal data as any information relating to an identified or identifiable natural person and frames identifiability as potentially direct or indirect, including by reference to identifiers and contextual factors (3).

This broad scope implies that artifacts may become privacy-relevant when they contain values or references that can single out a person, either alone or in combination with additional information (3, 18).

Similarly, the LGPD defines personal data as information related to an identified or identifiable natural person and adopts a compatible concept of identifiability consistent with the GDPR's expansive approach (4, 18).

Under these definitions, the compliance boundary is not limited to databases or runtime logs because development artifacts may contain identifiers or quasi-identifiers, including in code constants, tests, and configuration files (3, 4, 1).

Although this dissertation focuses on GDPR- and LGPD-aligned interpretations, broader privacy regulation landscapes reinforce the relevance of software artifacts in compliance discussions (19, 3, 4).

Consumer privacy statutes such as the California Consumer Privacy Act (CCPA) similarly foreground personal information and organizational obligations, supporting the broader motivation for systematic detection and governance (19, 3, 4).

Accordingly, this study operationalizes personal data categories in a way that is compatible with GDPR/LGPD definitions while remaining interpretable for software engineering practitioners who must reason about privacy risk and accountability in codebases (3, 4, 2).

2.1.2 Personal Data in Software Artifacts and Source Code

Personal data may appear in software artifacts in multiple forms, including hard-coded constants, configuration parameters, logging statements, test fixtures, and example or mock data (2).

In code-review settings, an additional practical requirement is to surface where personal data is likely being *processed* rather than merely where it is lexically present (20).

A static-analysis approach can operationalize this need by defining personal-data sources and modeling personal-data processing via sinks that capture action types and relevant API usage (20).

The same work highlights that purely occurrence-based detection can generate substantial false positives, while focusing on processing-oriented flows can yield higher precision for reviewer-relevant findings (20).

Empirical studies on privacy-relevant code highlight that privacy-related concerns can be embedded in specific program locations and code elements that developers must locate and reason about during privacy reviews (2).

This supports the premise that detecting personal data in source code is a recurring engineering task with practical implications for inspection and governance (2, 1).

Beyond source code and test assets, operational logs are prominent carriers of personal data because they record identifiers and context-rich traces used for debugging and incident response (21).

In Kubernetes-based microservice architectures, log management is complicated by the short life cycle of pods and the decentralized generation of log streams, which can propagate PII across distributed components and log pipelines (21).

A Kubernetes-based design integrates a PII detector into the log pipeline via a Sidecar container so that each log line can be processed for identification and redaction in a modular manner while preserving isolation from the main application container (21).

A key challenge is that the syntactic presence of data-like tokens does not necessarily imply privacy-relevant processing because legal scope depends on semantic context and the role of the value in the system (3, 1).

Recent work applying LLMs to privacy and compliance tasks in code suggests that semantic reasoning can help disambiguate privacy relevance by leveraging contextual cues beyond surface patterns (10, 5).

This motivates distinguishing (i) token- or pattern-level detection of candidate values from (ii) context-sensitive reasoning required to assess privacy relevance under regulatory definitions of identifiability (3, 4, 1).

2.1.3 Compliance and Accountability Implications

Data protection regulations establish accountability as a core principle, requiring organizations to demonstrate compliance with legal obligations rather than merely asserting it (3, 4).

In software engineering, accountability implies that development artifacts, including source code, must be auditable with respect to how personal data is introduced, handled, and justified (3, 4, 1).

Accountability becomes more complex when engineering teams reuse third-party or open-source artifacts, because organizations may inherit constraints and privacy risks embedded in reusable components, patterns, or documentation (22, 3).

Prior work argues that compliance automation approaches often fail to provide justification and transparent decision logic, limiting their usefulness in accountability-driven contexts (14, 3).

From this perspective, LLMs are positioned as an enabling mechanism to generate compliance reports that incorporate rationale and traceability, which is closer to accountability-oriented needs than purely local label assignment (14).

Complementary empirical evidence argues that accountability-oriented compliance depends on sustaining traceability artifacts that connect legal provisions to engineering requirements and implementation-level decisions (23).

The study reports that visual and structured legal design representations can function as shared reference points that reduce ambiguity and enable reviewers to validate how regulatory interpretations were transformed into technical requirements (23).

A recurring requirement is traceability between organizational statements about personal data handling and concrete implementation evidence, since audits and governance activities depend on demonstrating how declared practices are realized in software artifacts (3, 24).

This motivates approaches that operationalize compliance support as a traceability problem between textual artifacts (e.g., privacy policies) and code-level operations identified via automated analysis (24).

2.2 PRIVACY-AWARE SOFTWARE ENGINEERING AND PRIVACY AS CODE

Privacy-aware software engineering studies privacy requirements and risks as first-class engineering concerns addressed through processes, artifacts, and tools across the software lifecycle (1).

Within this perspective, privacy is not only a matter of runtime behavior, but also a property

reflected in source code, configuration, and developer decisions that can be inspected, reviewed, and governed (1, 2).

Because privacy-related processing and identifiers may be sparse and distributed across a codebase, pipeline-compatible detection strategies are needed to support prioritization and structured inspection rather than exhaustive manual review (2, 3).

2.2.1 The Privacy as Code Paradigm

The *privacy as code* paradigm frames privacy concerns as increasingly expressed through concrete software constructs, such as code patterns, libraries, configurations, and workflows that implement privacy-relevant behaviors (1).

A rapid literature review reports that privacy-as-code research spans diverse approaches and objectives, but also highlights uneven empirical support and limited usability evidence across studies (1).

From a tooling perspective, privacy as code encourages integration of privacy checks into engineering pipelines (1).

However, because privacy relevance depends on contextual interpretation and legal definitions of identifiability, effective approaches must connect technical detections to compliance meaning and preserve evidence for accountability (3, 4, 14).

2.2.2 Integrating Privacy-Enhancing Technologies into the Software Development Life Cycle (SDLC)

Privacy-enhancing technologies (PETs) are frequently discussed as technical means to reduce privacy risks, but their practical adoption depends on how they are integrated into concrete software engineering activities rather than treated as isolated technical add-ons (25).

Empirical and design-oriented evidence indicates that PET integration requires aligning technical choices with process stages, responsibilities, and the surrounding organizational context (25).

Field evidence further indicates that privacy engineering adoption is strongly influenced by practitioner mindset and by how organizations conceptualize privacy responsibilities across roles (16).

2.2.3 GDPR Compliance in Continuous Integration (CI) and Continuous Compliance (CC)

A longitudinal design-science study in a startup practicing CI reports that GDPR compliance work in small organizations is constrained by time pressure and by the unsustainability of manual verification (26).

In that setting, the authors operationalize selected GDPR principles into privacy requirements that can be checked automatically over cloud infrastructure resources to support a CC workflow (26, 3).

The same study highlights that automation alone does not guarantee CC in practice because findings may fail to be translated into prioritized engineering work (26).

2.3 AUTOMATED DETECTION OF PERSONAL DATA IN SOURCE CODE

Automated detection methods for personal data in source code can be grouped into rule-based approaches, supervised learning approaches including transformer-based classifiers, and LLM-based or hybrid strategies that emphasize contextual reasoning (2, 10, 5).

Across these families, a key design tension is how to balance recall-oriented candidate generation against precision-oriented filtering (3, 14).

In privacy-aware code review, static-analysis-centered pipelines can be positioned as review-friendly candidate generators that localize personal-data *processing* patterns with minimal deployment friction (20).

One concrete instantiation uses Semgrep, a static-analysis tool that matches grammatical patterns on parsed programs represented as Abstract Syntax Trees (ASTs) rather than relying solely on string or regular-expression (Regex) matching, to detect clear-text personal data occurrences and within-file flows via taint-style matching between predefined sources and sinks (20).

Beyond code, operational detection architectures also target application logs, where the compliance objective extends from detection to automated redaction before storage or downstream processing (21).

A Kubernetes-native approach compares multiple techniques and reports trade-offs among effectiveness and runtime resource consumption under load (21).

Static-analysis frameworks have also been proposed to scan codebases for privacy-related operations and connect findings to compliance-relevant concerns through traceability between policy statements and implementation points (24, 3).

2.3.1 Rule-Based and Pattern-Based Approaches

Rule-based and pattern-based approaches typically use regular expressions, heuristics, and hand-crafted rules to detect values that match known identifier formats (2).

Their main strengths are transparency, low computational cost, and straightforward integration into engineering pipelines (1).

However, these approaches often struggle with context dependence and semantic ambiguity, which can yield high False-Positive (FP) rates (1, 2).

Conversely, False Negatives (FN) can arise when identifiers are indirect, obfuscated, or encoded in ways that do not match predefined patterns (3, 4).

2.3.2 Machine Learning and Transformer-Based Classifiers

Supervised learning methods, including transformer-based classifiers, aim to learn detection patterns from labeled examples and can generalize beyond hand-crafted rules (5).

Empirical evidence shows that transformer-based classifiers constitute a strong supervised baseline for sensitive data detection by leveraging contextual token representations learned from large-scale pretraining (27).

However, classifier performance depends on training data quality, label definitions, and representativeness, and models may fail under distribution shifts or novel coding conventions (10, 28).

2.4 LARGE LANGUAGE MODELS FOR PRIVACY AND CODE ANALYSIS

LLMs have been increasingly applied to code understanding tasks with the promise of capturing context and semantics beyond surface patterns (28, 29).

Recent evidence indicates that semantics-related behavior varies substantially across models, tasks, and transformation conditions, reinforcing that LLM-based code analysis components should be validated before being treated as reliable decision support (29).

LLM-based privacy analysis can be situated within the broader LLM for Software Engineering (LLM4SE) landscape, where recent evidence highlights substantial variability in model behavior across tasks, datasets, and evaluation configurations (17). Therefore, architectural alternatives involving transformer classifiers, LLM-based judges, and LLM-centered extraction pipelines require systematic empirical evaluation under controlled experimental conditions.

2.4.1 LLMs for Privacy and Compliance Tasks

Recent work demonstrates that LLMs can be used to identify personal data processing aspects in source code by framing the task with privacy taxonomies and prompting strategies (5).

Empirical evaluations also explore LLMs for detecting privacy risks in code and report both potential and problems (10).

Policy-oriented compliance support has also been explored, reinforcing that governance workflows span heterogeneous artifacts and require document-grounded interpretations rather than only code-centric analysis (30, 3).

Requirements-engineering research further argues that compliance automation must provide justification and traceability to satisfy accountability demands (14, 3).

2.5 RELATED WORK

Prior work establishes privacy-aware software engineering as a discipline where privacy obligations materialize into concrete engineering artifacts, practices, and toolchains that must be inspected and governed across the SDLC rather than treated as a purely legal concern (16, 1). Within this perspective, the *privacy as code* paradigm consolidates how privacy requirements and privacy-relevant decisions become embedded in code patterns, configurations, workflows, and supporting documentation, while also highlighting the uneven maturity of evaluation and usability evidence in the current research landscape (1). Complementary evidence indicates that privacy engineering outcomes depend on organizational structures and practitioner mindset, which motivates approaches that are auditable, pipeline-compatible, and aligned with real review practices rather than only producing isolated labels (16).

A central line of work targets the detection and localization of privacy-relevant code through static and hybrid analysis, emphasizing that developer-relevant findings often require identifying *processing* contexts rather than only lexical occurrences of data-like strings (2, 20). Processing-oriented approaches operationalize privacy-relevant behavior through sources and sinks and can reduce false positives compared to occurrence-only scanning, while preserving interpretability and deployability in engineering pipelines (20). Beyond source code, runtime artifacts such as application logs remain prominent carriers of personal data, motivating detection-and-redaction architectures that integrate into log pipelines (e.g., Kubernetes-native designs) and expose runtime trade-offs among accuracy and resource consumption (21). In parallel, learning-based approaches frame sensitive-data detection as a supervised classification task, where transformer-based models serve as strong baselines but remain sensitive to labeling definitions, representativeness, and distribution shifts across projects and coding conventions (27).

Recent work also explores large language models for privacy and compliance tasks in code,

arguing that LLMs may leverage contextual cues to disambiguate privacy relevance beyond surface patterns, while still requiring controlled protocols due to instability and hallucination risks (5, 30). Requirements- and compliance-oriented research further emphasizes that automation is insufficient when it cannot provide justification and traceability, motivating approaches that connect regulatory interpretations and textual policies to concrete implementation evidence that can be audited (14, 24).

Large-scale empirical evidence on LLM-assisted inspection of static-analysis warnings suggests that LLMs can be effective as a precision-oriented validation layer to reduce false alarms in high-volume settings, which supports hybrid designs that separate recall-oriented candidate generation from precision-oriented filtering (31). Finally, systematic evidence syntheses in LLM4SE reinforce that model behavior varies across tasks and conditions and that empirical claims should be grounded in explicit evaluation protocols, artifact retention, and transparent reporting (17, 29).

Although these studies provide important foundations for privacy-aware software engineering, personal-data detection, static analysis, transformer-based classification, and LLM-assisted code analysis, they do not yet provide a controlled architectural comparison of alternative integration strategies for detecting personal data in source code. In particular, prior work has not systematically compared classifier-only, classifier–LLM hybrid, and LLM-centered extraction pipelines under a shared dataset, common scoring criteria, deterministic post-processing controls, and preserved evidence artifacts for post-hoc verification. This gap motivates the comparative architectural evaluation conducted in this dissertation.

Despite the breadth of approaches surveyed above, the literature still exhibits substantial variability in how personal-data detection in source code is operationalized, including differences in label taxonomies, matching criteria, dataset composition, and reporting practices, which makes direct comparison across studies difficult and can obscure the practical trade-off between false positives and false negatives in compliance-oriented workflows (13, 17).

Accordingly, this dissertation positions the evaluation protocol itself as a first-class research artifact, requiring controlled inputs, explicit scoring rules, and artifact retention so that outcomes can be audited, replicated, and interpreted as engineering evidence rather than anecdotal model behavior (13, 32).

This motivation leads directly to the methodological choices detailed in Chapter 3, where the study defines a controlled comparative design to assess rule-based, classifier-based, and LLM-based detection strategies under consistent dataset snapshots and consistent evaluation criteria aligned with GDPR/LGPD-informed interpretations of identifiability (3, 4, 13). The resulting evaluation framework may also provide a reusable basis for systematic comparisons of alternative detection architectures across different programming languages, datasets, and software-engineering contexts.

2.6 SUMMARY

This chapter established the regulatory and conceptual basis for interpreting personal data in software artifacts under GDPR and LGPD aligned definitions, emphasizing identifiability and accountability (3, 4, 18).

It discussed how personal data can appear in source code and why distinguishing syntactic appearance from semantic relevance is crucial for legally meaningful analysis (1, 2).

It reviewed automated detection approaches ranging from rule-based methods to transformer classifiers and LLM-based strategies, highlighting that these approaches differ in deployability, context sensitivity, and error profiles (2, 27, 10).

It positioned privacy compliance as an auditable engineering concern connected to traceability, organizational practice, and SDLC integration of privacy mechanisms (14, 16, 25).

3 RESEARCH METHODOLOGY

This study adopts a controlled experimental design complemented by an evidence-oriented approach in which experimental claims are explicitly bound to inspectable run-level artifacts. Rather than relying solely on execution reproducibility, the methodology emphasizes post-hoc verifiability through preserved evidence bundles, enabling independent validation without requiring model re-execution. This combination enables both controlled comparison and independent post-hoc verification of results under evolving LLM ecosystem conditions.

3.1 EMPIRICAL EVALUATION IN SOFTWARE ENGINEERING

Empirical software engineering emphasizes systematic methods for generating credible evidence about techniques, tools, and processes, particularly when multiple alternatives must be compared under consistent conditions (13, 8).

Controlled experimentation is a common strategy for isolating variables and supporting interpretable comparisons, provided that datasets, protocols, and reporting practices are clearly documented (13, 15).

In addition to controlled experiments and benchmark-based comparisons, empirical SE studies increasingly use structured protocols to analyze large corpora where labelling effort is substantial and consistency across raters is a primary concern (33).

Recent work has demonstrated how foundation model (FM) jury-based protocols can incorporate human-labelled golden sets, iterative prompt refinement, and inter-rater agreement measures (Cohen's κ) as explicit quality-control mechanisms in qualitative software engineering studies (33). Complementary evidence suggests that human–LLM collaborative evaluation workflows can achieve levels of agreement comparable to human annotators in several software engineering assessment tasks (34).

In this dissertation, the experimental protocol is operationalized through two dedicated infrastructure artifacts: (i) a synthetic dataset generation framework that produces Java code fragments paired with ground-truth annotations (Appendix I), and (ii) a detection framework that executes all treatments under standardized input handling, output normalization, and artifact retention (Appendix II). The concrete configuration files, file formats, and archived evidence artifacts required to support protocol inspection and post-hoc verification are consolidated in Appendix III (13). These elements jointly define the empirical foundation of the study, ensuring that experimental claims are supported by both controlled execution and preserved evidence artifacts.

3.1.1 Research Questions and Evaluation Goals

This dissertation evaluates alternative strategies for detecting personal data in source code under GDPR/LGPD-aligned interpretations and compares their effectiveness under controlled inputs and consistent scoring rules (3, 4, 13).

The evaluation goal is to quantify and explain the trade-off between false positives and false negatives in compliance-sensitive workflows where manual triage cost and missed exposure risk are both operationally relevant (3, 13).

Accordingly, the empirical study is structured as a comparative evaluation of detection strategies, where the manipulated factor is the detection approach and the outputs are scored against a fixed ground truth under explicit matching criteria (8, 13).

To ensure that all treatments are compared under identical inputs and consistent execution conditions, the study uses a versioned dataset snapshot produced by the dataset generation framework (Appendix I) and executes all detection strategies through a unified detection framework (Appendix II), with the full set of experiment configurations and run parameters archived as verification artifacts (Appendix III) (8, 13).

3.1.2 LLMs as Substitutes for Manual Annotation in Software Engineering (SE) Evaluations

A recurring bottleneck in empirical software engineering is the cost and scalability of manual annotation, especially in studies that require human judgments over code-related artifacts (34).

Ahmed et al. investigate whether LLM queries can replace part of this manual effort by applying multiple LLMs to established annotation tasks drawn from prior datasets (34).

Their results indicate that, for some tasks, LLM-generated annotations can reach agreement levels comparable to human–human agreement, supporting mixed human–LLM designs rather than fully manual workflows (34).

The study proposes using model–model agreement and model confidence as practical gates to decide when LLMs can substitute humans more safely, framing substitution as selective and auditable rather than unconditional (34).

This evidence motivates treating LLM-based components, including LLM-as-a-judge mechanisms such as those used in Pipeline 2, as evaluation instruments that require calibration and explicit acceptance criteria (34).

This rationale is directly relevant to the present study, in which LLMs are not treated merely as predictive engines, but also as filtering and evaluation components whose behavior must be empirically characterized, particularly in hybrid configurations such as Pipeline 2.

Evaluations of LLM-based compliance tools further emphasize that assessments should consider output stability and faithfulness because generative models may hallucinate plausible but incorrect content (30).

3.1.3 Controlled Experiments in Software Engineering

Controlled experiments support comparative evaluation by holding key variables constant while manipulating the factor(s) of interest, enabling more credible inference about observed differences (13, 15).

Surveys of controlled experiments in software engineering emphasize that rigor depends on clear hypotheses, transparent protocols, and careful reporting of threats to validity (8).

Recent evidence further indicates that threats-to-validity sections may still lack depth and consistency, reinforcing the need to report threats as an explicit analysis grounded in concrete design and measurement choices (32).

Because the experiments in this dissertation focus on controlled model comparisons rather than human-subject studies, the design emphasizes protocol transparency, artifact preservation, and separation of experimental factors from incidental sources of variance, with protocol details and retained artifacts provided as explicit appendices to support independent verification (Appendix III) (8, 13).

3.2 EXPERIMENTAL DESIGN AND EVALUATION PROTOCOL

This dissertation adopts a controlled, comparative evaluation design to assess alternative detection strategies under consistent inputs, consistent labeling rules, and explicit reporting of design choices (13, 15).

The experimental factor of interest is the detection strategy under study, while primary control variables include dataset version, ground-truth labeling procedure, and the evaluation criteria used for matching and scoring (8, 13).

The design is motivated by the many-variable problem in software engineering, which makes it essential to isolate manipulated factors and minimize incidental variance that could confound interpretation (35).

In this context, the protocol defines the experimental logic, while the supporting frameworks implement its execution and produce the artifacts required for verification (13, 32). The concrete protocol definition and its operational artifacts are documented in Appendix III.

In summary, the experimental protocol defines the comparative logic of the study, including

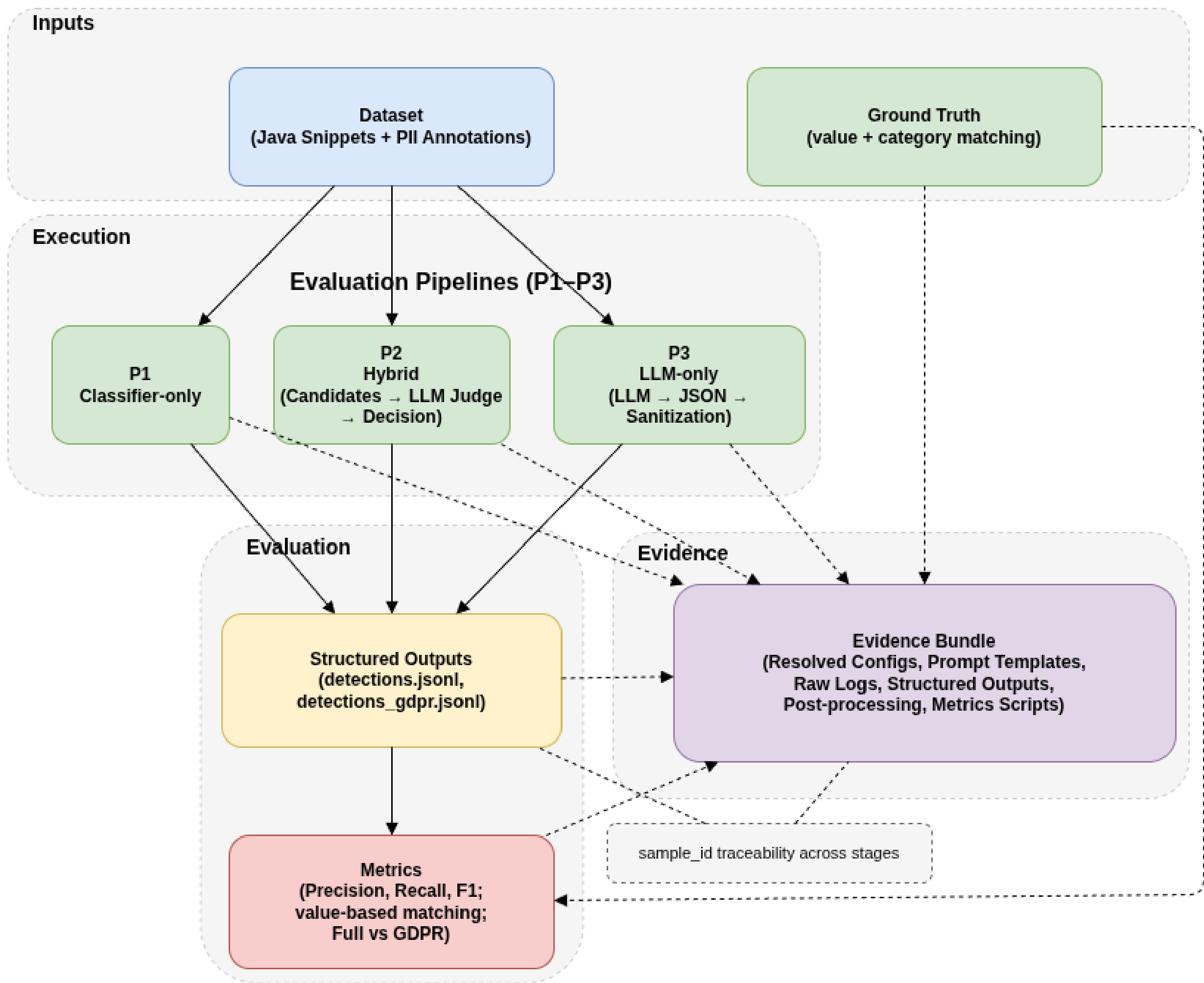

Figure 3.1: Overview of the experimental architecture. A shared dataset is processed by three evaluation pipelines (P1–P3), producing structured outputs that are evaluated against a ground truth using value-level matching. All stages generate artifacts that are preserved as an evidence bundle, enabling traceability, auditability, and post-hoc verification.

inputs, treatments, and evaluation criteria. The evidence-ready layer introduced later in this chapter complements this protocol by specifying how each experimental claim is explicitly bound to persisted artifacts, enabling post-hoc verification beyond mere reproducibility.

To provide a high-level overview of the experimental design and its evidence-oriented structure, Figure 3.1 illustrates the end-to-end architecture of the study. The figure highlights the relationship between inputs, evaluation pipelines, structured outputs, evaluation metrics, and the evidence bundle that supports post-hoc verification.

As shown in Figure 3.1, the experimental design separates execution concerns from evaluation and evidence generation, ensuring that all results are derived from structured outputs and independently verifiable artifacts.

Table 3.1: Transformer-based classifiers evaluated in Pipeline 1

Model	Primary role
StarPII	Source-code PII detection
AB-AI/PII	General PII detection
Piiranha-v1	Privacy entity detection
DeBERTa (H2O)	PII classification
DeBERTa (lakshyakh93)	PII classification
DistilBERT Multilingual	Multilingual PII detection

Hugging Face repository identifiers used in the experiments: bigcode/starpii; ab-ai/pii_model; iiiorg/piiranha-v1-detect-personal-information; h2oai/deberta_finetuned_pii; lakshyakh93/deberta_finetuned_pii; yonigo/distilbert-base-multilingual-cased-pii.

3.2.1 Evaluated Models

The experimental evaluation employed two categories of machine-learning artifacts: transformer-based classifiers used for candidate generation and open-weight LLMs used in the hybrid and LLM-centered pipelines.

Model selection prioritized publicly available models that had previously been applied to PII detection, source-code analysis, instruction-following, or related software-engineering tasks. This selection was intended to provide diversity in architectural characteristics while maintaining a practical and reproducible experimental setup.

Pipeline 1 relies on an ensemble of transformer-based classifiers that operate independently over the same Java snippet. These classifiers were selected to represent different training strategies and model families previously used for privacy-related entity detection. The evaluated classifiers are summarized in Table 3.1.

Together, these classifiers form the candidate-generation stage of Pipeline 1. The evaluated models were StarPII (36, 37), AB-AI/PII (38), Piiranha v1 (39), DeBERTa PII–H2O (40), DeBERTa PII–lakshyakh93 (41), and DistilBERT Multilingual (42). Their predictions are subsequently aggregated through the OR-ensemble strategy described later in this chapter, enabling broad candidate coverage prior to any contextual validation.

Pipelines 2 and 3 employ open-weight LLMs as their primary reasoning components. To reduce dependence on a single model family and enable architectural comparison across different reasoning capabilities, the evaluation included both code-specialized and general-purpose models. Table 3.2 summarizes the LLMs evaluated in the study.

The code-specialized models included Qwen3-Coder (43), CodeGemma (44), CodeLlama (45), Codestral (46), and Devstral-Small-2 (47).

The general-purpose or reasoning-oriented models included Qwen3 (43), DeepSeek-R1 (48), Phi-4 (49), Gemma 3 (50), GPT-OSS (51), and Mistral (52).

Table 3.2: Open-weight LLMs evaluated in Pipelines 2 and 3

Model	Category	Usage
CodeGemma	Code-specialized	P2 and P3
CodeLlama	Code-specialized	P2 and P3
Codestral	Code-specialized	P2 and P3
Devstral-Small-2	Code-specialized	P2 and P3
Qwen3-Coder	Code-specialized	P2 and P3
DeepSeek-R1	General-purpose / reasoning-oriented	P2 and P3
Gemma 3	General-purpose / reasoning-oriented	P2 and P3
GPT-OSS	General-purpose / reasoning-oriented	P2 and P3
Mistral	General-purpose / reasoning-oriented	P2 and P3
Phi-4	General-purpose / reasoning-oriented	P2 and P3
Qwen3	General-purpose / reasoning-oriented	P2 and P3

The corresponding Ollama model identifiers used in the experiments were:

Code-specialized models:

- Qwen3-Coder (`qwen3-coder:30b-a3b-q4_K_M`)
- CodeGemma (`codegemma:7b-instruct`)
- CodeLlama (`codellama:13b-instruct-q4_K_M`)
- Codestral (`codestral:22b-v0.1-q4_K_M`)
- Devstral-Small-2 (`devstral-small-2:24b-instruct-2512-q4_K_M`)

General-purpose / reasoning-oriented models:

- Qwen3 (`qwen3:14b-q4_K_M`)
- Qwen3 (`qwen3:30b-a3b-instruct-2507-q4_K_M`)
- DeepSeek-R1 (`deepseek-r1:14b`)
- Phi-4 (`phi4:14b-q4_K_M`)
- Gemma 3 (`gemma3:12b-it-q4_K_M`)
- GPT-OSS (`gpt-oss:20b`)

Table 3.3: Experimental factors, treatments, and controls used in the comparative evaluation design.

Element	**Definition**	**Instantiated in this study**
Factor	Experimental variable intentionally manipulated	Detection strategy, operationalized as three alternative pipelines (P1–P3).
Treatments	Concrete realizations of the experimental factor	Pipeline-specific configurations executed under the same protocol, including classifier-only, hybrid, and LLM-only strategies.
Controls	Variables held constant to reduce confounding effects	Dataset snapshot, label definitions, matching criteria, scoring rules, input formatting, and archived execution policies.
Outputs	Observable results produced by each treatment	Structured predictions and derived evaluation metrics, including precision, recall, and F1 under value-level matching.
Artifacts	Persisted evidence supporting verification	Configuration files, prompt templates, raw interaction logs, structured outputs, and metric reports archived in the reproducibility package.

- Mistral (`mistral:7b-instruct`)

The same pool of LLMs was used across the hybrid (P2) and LLM-centered (P3) architectures whenever applicable, allowing the analysis to focus on the impact of architectural design choices rather than on the behavior of a single model family.

3.2.2 Factors, Treatments, and Controls

Each treatment corresponds to one detection strategy configured for the same dataset snapshot and executed under controlled conditions, with treatment configurations captured as versioned experiment specifications (Appendix III) (8, 13).

Table 3.3 summarizes the main elements of the experimental design, distinguishing the manipulated factor, its instantiated treatments, the control variables held constant across executions, and the resulting outputs and artifacts.

As shown in Table 3.3, the comparative design isolates the detection strategy as the main experimental factor while preserving a common set of controls across all treatments.

Control variables include consistent label definitions, identical scoring rules, and consistent

input formatting to ensure that observed differences are attributable to the detection approach rather than protocol drift, and these control policies are recorded as part of the reproducibility package (Appendix III) (13, 15).

Because compute budget can constrain sample size, repetitions, and tuning choices in judge-centric settings, the protocol records budget-related choices as part of the experimental design, including run parameters retained with the reproducibility and verification artifacts (Appendix III) (53).

3.2.3 Detection Infrastructure and Experimental Execution

All treatments are executed using a dedicated detection framework that standardizes input preparation, model invocation, output normalization, and artifact retention across detection strategies. This design ensures that execution differences do not confound the comparative evaluation. Appendix II provides the framework design, execution modes, and output schema used in the experiments, while Appendix III specifies the configuration files and the standardized run layout used to archive artifacts (13).

This framework instantiates each detection approach as a treatment configured against the same dataset snapshot, ensuring that observed differences are attributable to the detection strategy rather than to execution drift or protocol inconsistencies, and the exact treatment definitions are captured as reproducibility artifacts (Appendix III) (8, 13).

The execution pipeline records the full experimental context required for protocol reconstruction and post-hoc verification, including dataset identifiers, run parameters, and the produced structured outputs that are subsequently consumed by the scoring procedure, and the concrete retained artifacts are enumerated in Appendix III (13, 15).

To support auditability and post-hoc error analysis, the framework archives run artifacts in a structured layout, enabling independent verification of intermediate outputs. This also facilitates systematic inspection of false positives and false negatives under the declared matching criteria, and the retained outputs and their formats are documented in Appendix III (32, 13).

3.3 DATASETS AND GROUND TRUTH

The evaluation uses datasets designed for controlled experimentation over privacy-relevant artifacts, where the use of synthetic data mitigates legal and ethical risks associated with handling real personal data while enabling repeatable labeling and verification (11, 3).

Ground truth is treated as a first-class artifact that defines the target construct for detection, and its labeling procedure is documented to support interpretability of performance claims (13, 8).

Synthetic dataset construction is motivated by evidence that LLMs can generate labeled examples that support downstream learning and evaluation, while still requiring explicit discussion of representativeness limits and utility validation (54, 11).

When relevant, privacy-risk considerations for synthetic artifacts are acknowledged as a methodological constraint because synthetic outputs can still leak sensitive patterns and remain subject to regulatory interpretation in certain contexts (55, 3).

3.3.1 Synthetic Dataset Generation and Validation Infrastructure

To support controlled experimentation while reducing legal and ethical risks, the study uses a dedicated dataset generation framework that produces Java code fragments paired with ground-truth annotations for personal-data entities under the adopted label taxonomy. Appendix I documents the generator design goals, configuration parameters, and the ground-truth schema consumed by the scoring procedure, while Appendix III documents the dataset snapshot identifiers and the archived generator configuration required to reproduce the dataset used in the experiments (3, 11).

The `bigcode/bigcode-pii-dataset` is a large multilingual corpus annotated for PII in source code (56). However, the synthetic dataset used in this study was not directly derived from BigCode samples; instead, the BigCode annotation structure and label taxonomy were used as conceptual references for defining the dataset schema and annotation organization adopted in this study.

The framework operationalizes synthetic-data construction as a controlled and configurable procedure with explicit configuration of entity categories, sampling constraints, and annotation outputs, enabling consistent dataset snapshots to be reused across treatments, and the exact generation configuration is retained as part of the reproducibility artifacts (Appendix III) (13).

For auditability, each dataset snapshot is versioned and archived alongside the experiment artifacts, including the dataset identifier, generation configuration parameters, label definitions, and the produced ground-truth file used by the scoring procedure, and the concrete retained files are enumerated in Appendix III (13, 8).

Because synthetic artifacts may still encode sensitive-like patterns and remain subject to privacy interpretation, the dataset is treated as an experimental artifact whose construction assumptions are reported as part of the methodological constraints of the study (55, 3).

The generation workflow additionally incorporated deterministic validation procedures intended to reduce inconsistencies commonly associated with LLM-assisted synthetic artifact generation. After snippet generation, the framework verified whether each expected synthetic entity value appeared verbatim in the produced Java code fragment before annotation construction. Generated outputs that failed to preserve the expected entity values or produced malformed anno-

tation structures were not retained as valid annotated entities. This procedure reduced annotation ambiguity and supported deterministic value-level evaluation across all experimental treatments.

To improve annotation consistency and dataset integrity, the framework also performed structural validation of the generated dataset artifacts before export. Validation procedures included schema-conformance checks for the JavaScript Object Notation Lines (JSONL) records, verification of annotation offsets relative to the generated snippet text, consistency checks between entity categories and annotated values, and inspection of malformed or incomplete outputs produced during LLM-assisted generation. The dataset generation workflow additionally retained generation warnings and execution metadata as part of the archived reproducibility artifacts. Archived artifacts additionally retained raw LLM interaction logs, execution warnings, and Graphics Processing Unit (GPU) telemetry generated during dataset construction to support auditability and reproducibility analysis.

Validation procedures included schema-conformance checks for the JSONL records, verification of annotation offsets against the generated source code, consistency checks between entity categories and annotated values, and detection of malformed outputs. Because the dataset was generated through a controlled synthetic process, the expected entity values and categories were known *a priori*. Consequently, the corresponding ground-truth annotations could be deterministically verified before inclusion in the final dataset. Validation procedures included verification that generated entity values appeared verbatim in the associated Java snippets, annotation-offset consistency checks, and category-consistency verification. Records failing any validation step were discarded.

The final experimental dataset used in the study was assembled from archived generation executions performed under controlled configuration parameters, including fixed positive/negative sampling ratios, predefined entity-category constraints, and explicit generation settings for the adopted LLM provider and model configuration. The resulting dataset contains 2,200 Java snippets, including both PII-containing and non-PII examples, together with 3,596 annotated entity fragments distributed across 42 personal-data categories. This composition strategy was intended to support controlled architectural comparison under known and reproducible ground-truth conditions. The final dataset artifact was assembled from two archived generation executions whose concatenated outputs exactly reproduce the retained dataset snapshot used in the experiments.

Because the annotation process relied on deterministic matching between generated entity values and their occurrences in the produced Java snippets, the resulting ground truth remained explicitly observable and reproducible at generation time. Annotation offsets were calculated directly from the generated snippet content, enabling deterministic reconstruction of the annotated spans used during evaluation. This design supported the adopted value-level matching protocol and reduced dependence on probabilistic post-hoc annotation procedures.

From a methodological perspective, the synthetic dataset was treated as a controlled experimental artifact rather than a direct representation of industrial software repositories. The primary

objective of the dataset construction process was to provide reproducible experimental conditions suitable for comparative architectural evaluation of PII-detection pipelines under shared inputs, shared label taxonomy, and shared evaluation procedures.

Nevertheless, the study acknowledges that synthetic Java snippets may not fully capture the structural diversity, contextual ambiguity, and development practices observed in real-world industrial software systems. In addition, archived generation artifacts revealed some category-specific generation and post-processing limitations, representing important constraints regarding external validity.

3.3.2 Dataset Characteristics and Distribution

Due to licensing restrictions on the `bigcode/bigcode-pii-dataset` that prohibit redistribution or publication of derived datasets (56), this study does not use or redistribute samples from that corpus. Instead, an independent fully synthetic dataset tailored to the evaluation of PII detection in Java source code was developed for the controlled experimental evaluation (3, 13).

Following design principles widely adopted in empirical software engineering for controlled comparisons, the dataset was constructed with 2,200 Java snippets stored as JSONL records containing a deterministic identifier, the source code, and fine-grained PII fragment annotations (9, 15, 13, 8).

The dataset includes 3,596 annotated PII entities spanning 42 categories, with approximately 68% of snippets containing at least one entity and the remaining 32% serving as negative examples, supporting entity-level evaluation under controlled coverage constraints (13, 8).

Synthetic code datasets have increasingly been adopted as controlled post-training and evaluation testbeds for code-focused LLMs, and recent work supports their methodological viability for controlled comparative evaluation under observable conditions (12, 11).

Controlled comparisons between real and synthetic datasets further indicate that automatically generated programs can exhibit substantial structural diversity and support rigorous empirical analyses even when they differ from domain-specific production idioms (57).

Taken together, this body of evidence supports the claim that well-constructed synthetic code datasets constitute a legitimate foundation for controlled experimentation, and in this study they justify the use of synthetic Java snippets as a controlled and evidence-oriented evaluation testbed for benchmarking PII detection approaches at the entity level (11, 8).

In this dissertation, the synthetic dataset is therefore interpreted as a controlled evaluation testbed designed to support architectural comparison, measurement consistency, and consistent entity-level evaluation, rather than as a proxy for organically evolved industrial Java repositories.

The study focuses on explicit value identification under controlled conditions rather than on

Table 3.4: Summary statistics of the synthetic Java PII dataset used in the study.

Characteristic	**Value**
Total snippets (N)	2,200
Snippets with at least one PII entity	1,497 (68.0%)
Snippets without PII (negative examples)	703 (32.0%)
Total PII entities annotated	3,596
Number of distinct PII categories	42
Entities per snippet (min/median/mean/max)	1 / 2 / 2.40 / 12
Snippet length in lines (min/median/mean/max)	4 / 10 / 11.4 / 40
Snippet length in characters (min/median/mean/max)	92 / 326 / 395 / 1,705
Minimum entities per category	20
Maximum entities per category	156
Annotation method	Deterministic value-level annotation with preserved offsets metadata
PII source	Faker-based synthetic provider
Code generation model	Open-weight code LLM (archived run metadata)

full program semantics, implicit data-flow reasoning, or comprehensive privacy analysis of industrial software systems.

Table 3.4 summarizes the key characteristics of the dataset, and the full per-category distribution and snapshot integrity artifacts are documented in Appendix III (13).

This design reflects an intentional trade-off between experimental control and ecological realism, prioritizing internal validity and comparability while explicitly acknowledging limitations in representativeness.

Table 3.5 complements the dataset statistics by summarizing the distribution of PII presence and category coverage. The dataset presents a balanced mix of positive and negative samples, along with controlled category coverage, supporting reliable entity-level evaluation while maintaining sufficient variability for comparative analysis.

The use of controlled synthetic artifacts also reduces hidden environmental variability that could otherwise confound architectural comparison across pipelines, thereby strengthening interpretability of observed precision–recall trade-offs under the declared protocol.

Table 3.5: Overview of dataset distribution across PII presence and category coverage.

Aspect	**Value**
Snippets with PII	1,497 (68.0%)
Snippets without PII	703 (32.0%)
Total PII entities	3,596
Number of categories	42
Category balance	Minimum 20 entities per category
Category skew	Maximum 156 entities in a single category
Entity density	Mean of 2.40 entities per snippet
Negative-to-positive ratio	0.47 (negative / positive samples)

3.3.2.1 Dataset Limitations

Although synthetic datasets offer strong experimental control and ensure precise entity-level annotation, they introduce limitations that must be considered when interpreting results (11).

First, while the generated snippets resemble compact Java methods, they do not capture the stylistic, architectural, and dependency diversity of industrial Java codebases, including framework-driven design patterns or organically evolved naming practices (8, 11).

Second, synthetic PII values do not fully reflect the noisy, malformed, or partially redacted data often present in production systems, which can affect both detection difficulty and error profiles (11, 55).

Third, template-guided generation can reduce stylistic variance and increase regularity, potentially simplifying contextual inference for LLM-based detectors compared with heterogeneous real-world corpora (8, 10).

Nonetheless, synthetic data provides methodological advantages that are essential for this controlled comparison, including deterministically generated annotations, broad category coverage, and complete transparency regarding provenance and generation parameters, and these properties enable attribution of observed differences to the detection strategy rather than to uncontrolled data variation (13, 8).

This controlled setting is therefore particularly suitable for isolating the effects of architectural design choices across the evaluated pipelines, which is essential for interpreting the observed differences as architectural signals rather than as artifacts of uncontrolled dataset variability or incidental distributional effects.

Accordingly, the reported findings should be interpreted as controlled comparative architectural signals rather than as direct estimates of industrial deployment performance.

Future work should extend this evaluation to industrial repositories and multilingual corpora to strengthen external validity and assess robustness under more heterogeneous conditions (8, 5).

3.4 STUDY DESIGN AND OBJECTIVES

To investigate the effectiveness and engineering trade-offs of PII detection strategies for Java source code, we define a set of specific objectives that structure a comparative and diagnostic evaluation of GenAI-enabled analysis pipelines (8, 13).

Rather than presupposing the superiority of any approach, these objectives are designed to empirically characterize three architectural choices commonly discussed in recent software engineering literature: (i) transformer-based classifier detection, (ii) hybrid pipelines in which an open-weight LLM filters a fixed pool of classifier-generated candidates, and (iii) LLM-centered extraction pipelines with deterministic post-processing (34, 58).

The objectives are intentionally stated to expose practical trade-offs, judge behaviors, and failure modes that are directly relevant to privacy-sensitive software engineering settings, where excessive false positives inflate audit workload while false negatives increase regulatory and compliance risk (3, 4, 5).

3.4.1 Specific Objectives

To achieve the general objective of this study, the following specific objectives were defined:

- To characterize the effectiveness of a classifier-only approach for detecting personal data in Java source code, establishing a baseline in terms of precision, recall, and related evaluation metrics (8, 13).
- To evaluate the impact of hybrid detection strategies in which LLMs act as judges over classifier outputs, with particular attention to changes in false positives, false negatives, and overall detection trade-offs (6, 7).
- To assess the viability of end-to-end detection strategies based primarily on LLMs coupled with deterministic post-processing, considering effectiveness, robustness, and practical constraints relevant to software engineering contexts (58, 30).
- To compare the different integration strategies under identical datasets, categories of personal data, and evaluation criteria, enabling a controlled and fair analysis of their relative strengths and limitations (8, 13).

- To analyze the implications of the observed results for privacy compliance in software engineering, discussing how different detection strategies may support or hinder organizational decision-making under data protection regulations (3, 4, 5).

In the hybrid pipeline (P2), we additionally quantify judge behavior using acceptance rate and true-positive rejection rate to explain shifts in the precision–recall trade-off introduced by LLM-based filtering, and the concrete computation of these indicators is defined by the archived artifacts (Appendix III) (6, 7).

These objectives are operationalized through three evaluation pipelines (P1–P3) executed under identical datasets, matching criteria, and metrics, and complemented by error-profile analyses (false positives and false negatives) and judge-behavior indicators for the hybrid pipeline (13, 8).

3.4.2 Evaluation Pipelines

The three evaluated pipelines are depicted in Figure 3.2 as parallel execution paths that process the same dataset under identical evaluation conditions.

We evaluate three detection strategies that embody distinct assumptions about how PII can be identified in source code and how contextual information should be incorporated into automated entity-level PII detection workflows (5, 10).

All pipelines operate at the entity level and are evaluated on the same dataset, label set, matching criterion, and metrics, ensuring strict comparability across configurations, and produce outputs conforming to a common schema, enabling direct comparison across pipelines (8, 13). The concrete evaluation invariants are documented in Appendix III.

All evaluated pipelines operate over structured textual Java snippets and do not incorporate abstract syntax tree (AST) representations, taint analysis, data-flow tracking, inter-procedural reasoning, or broader program-level semantic analysis. Accordingly, the study focuses on explicit entity-level identification under controlled conditions rather than on holistic privacy auditing of industrial software systems.

Figure 3.2 presents an architectural comparison of the three evaluated PII detection pipelines. Each pipeline represents a distinct strategy for integrating classification, contextual reasoning, and structured extraction, while preserving a common output schema to ensure strict comparability across approaches.

Table 3.6 summarizes the main architectural characteristics of the three evaluated pipelines, highlighting their architectural roles, the involvement of LLM components, and the design assumptions associated with each detection strategy.

As shown in Table 3.6, the pipelines differ in their architectural structure, the role assigned to LLM components, and the design assumptions that motivate each detection strategy.

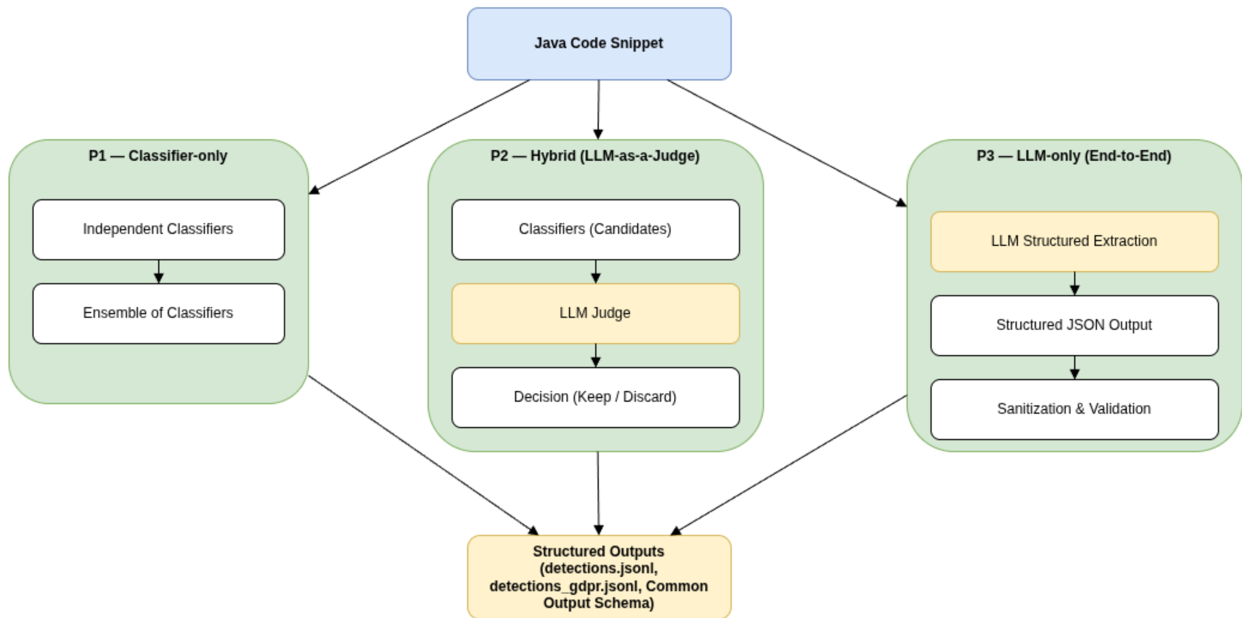

Figure 3.2: Architectural comparison of PII detection pipelines (P1–P3). P1 uses an ensemble of independent classifiers, P2 introduces an LLM-based judge to filter classifier-generated candidates, and P3 performs LLM-centered structured extraction coupled with deterministic validation, sanitization, normalization, and filtering stages. All pipelines produce standardized outputs under a common schema, enabling controlled and comparable evaluation.

Pipeline 1: Classifier-only (P1). This pipeline applies multiple transformer-based PII classifiers independently to each Java snippet and aggregates predictions using an OR-ensemble, aiming to maximize candidate coverage through a high-sensitivity detection strategy that mirrors common practice in PII scanning tools (13, 8).

Pipeline 2: Hybrid (P1 + LLM-as-a-judge) (P2). The hybrid pipeline treats the P1 OR-ensemble output as a fixed high-sensitivity candidate set and applies an open-weight LLM as a contextual judge to each candidate, issuing an explicit *keep* or *drop* decision, as illustrated in Figure 3.3 (6, 7).

As illustrated in Figure 3.3, the hybrid pipeline introduces an additional decision stage in which each candidate entity is explicitly validated by the LLM. While this filtering step is intended to reduce false positives through contextual reasoning, it also creates a failure point where valid entities may be incorrectly discarded, leading to false negatives.

Because the hybrid stage operates exclusively over the fixed and deduplicated candidate pool generated by Pipeline 1, entities absent from the upstream ensemble cannot be recovered by the judge stage. Accordingly, all hybrid outcomes are entirely determined by keep/drop decisions applied to the preserved candidate set.

Because LLM-as-a-judge may exhibit systematic biases and sensitivity to evaluation framing, this decision stage (Figure 3.3) must be treated as an explicit component whose behavior can

Table 3.6: Comparative summary of the evaluated PII detection pipelines.

Pipeline	Type	Role of the LLM	Architectural Motivation	Potential Limitation
P1	Classifier-only	None	Broad candidate coverage through ensemble detection	Propagation of classifier-generated false positives
P2	Hybrid	Judge / decision component	Context-aware validation of candidate entities	Dependence on judge acceptance decisions
P3	LLM-centered extraction	Structured extractor	Direct structured extraction with deterministic controls	Dependence on LLM output quality and schema compliance

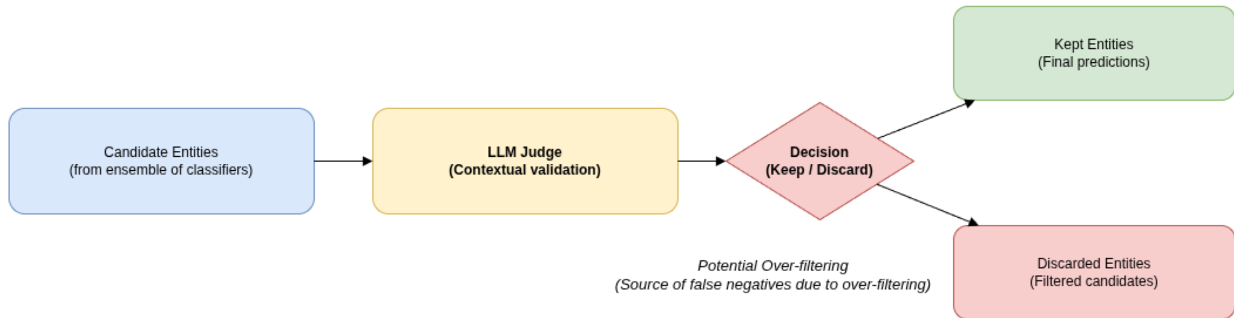

Figure 3.3: Decision flow of the hybrid pipeline (P2). Classifier-generated candidates are evaluated by an LLM acting as a judge, which issues a *keep* or *discard* decision. This filtering step reduces false positives but may introduce false negatives due to selective filtering decisions.

influence the precision–recall trade-off (6, 7).

Pipeline 3: LLM-only (P3). This pipeline replaces classifier-based candidate generation with an LLM-centered structured extraction process in which an open-weight LLM produces structured outputs that are subsequently subjected to explicit validation and deterministic sanitization, normalization, and filtering stages prior to scoring. This design ensures compatibility with the standardized output schema used for evaluation while constraining probabilistic generation through deterministic post-processing controls (58, 30).

As illustrated in Figure 3.4, the pipeline enforces a structured extraction process in which the LLM generates JavaScript Object Notation (JSON) outputs that are subsequently validated and transformed through deterministic post-processing steps, including normalization, filtering, and rule-based constraints. This design ensures that outputs are schema-compliant, inspectable, and suitable for consistent value-level evaluation.

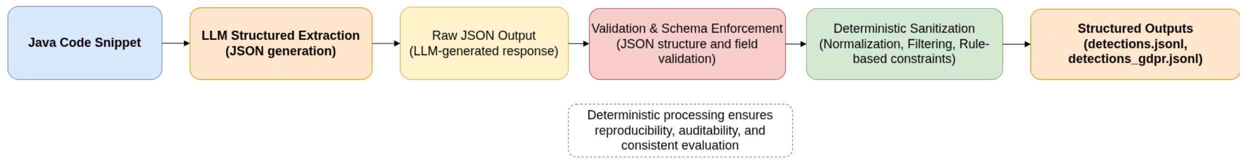

Figure 3.4: Detailed flow of the LLM-only pipeline (P3). A Java code snippet is processed by an LLM to produce structured JSON output, which is subsequently validated and subjected to deterministic post-processing steps, including normalization, filtering, and rule-based constraints. This design ensures that final outputs conform to a standardized schema and enables deterministic and auditable evaluation.

Together, these pipelines enable a controlled comparison of (i) transformer-based detection without contextual reasoning, (ii) hybrid designs in which an LLM performs candidate validation as an explicit decision component, and (iii) LLM-centered extraction coupled with deterministic post-processing, isolating the contributions of candidate generation, LLM-based filtering, and end-to-end extraction to overall detection effectiveness (13, 8).

As illustrated in Figure 3.2, the pipelines differ primarily in how detection responsibilities are distributed between classifiers and LLM components, while maintaining a shared output representation for evaluation.

3.5 SCORING PROCEDURE AND REPORTING

All evaluation metrics are computed from structured outputs and ground-truth data, as illustrated in Figure 3.5, ensuring that metric derivation can be independently recomputed from persisted artifacts.

Effectiveness is quantified using precision, recall, and F1-score, which are standard metrics for detection and retrieval tasks and capture complementary aspects of performance (10, 5).

The protocol interprets these metrics through the operational lens of compliance-sensitive trade-offs, where false positives increase triage cost and false negatives represent missed exposures under broad definitions of personal data (3, 4, 2).

As recommended by empirical evaluation guidance, the dissertation reports aggregate results and supports post-hoc analysis through artifact retention and transparent criteria, enabling independent verification and deeper error analysis (13, 32).

To ensure strict comparability across approaches, all pipelines are evaluated under identical post-processing and scoring rules, and the concrete matching key, join assumptions, and evaluation invariants are documented as archived evidence artifacts in Appendix III (8, 13).

In this dissertation, the evaluation protocol adopts a *value-level matching* criterion, in which predicted entities are matched against ground-truth annotations using normalized entity values and categories rather than character offsets or token spans. This design choice reflects the op-

erational perspective of privacy-oriented auditing workflows, where the primary concern is the correct identification of sensitive information values rather than exact positional reconstruction within the source code. Accordingly, entity positions are treated as auxiliary metadata and are not used as matching keys during metric computation.

3.5.1 LLM-as-a-Judge as a Methodological Component

When LLMs are employed as judges or annotators, the judging procedure is treated as part of the experimental design because judge performance and stability can vary across benchmarks and prompt realizations (6, 7).

Accordingly, the protocol requires that judge configurations be documented alongside datasets and models, including prompt structure, output schema, and aggregation strategy, and these elements are archived in the reproducibility package (6, 13).

Infrastructure-oriented guidance further emphasizes that judge-based experiments require explicit protocols controlling what evidence is shown, how the task is operationalized, and how artifacts are archived for repeatability (58).

Because nondeterminism can influence judge-based outcomes, the protocol records execution conditions and retains raw interaction traces to support stability-aware interpretation and independent diagnosis (30, 7).

Recent work further shows that judge-centric evaluations must account for judge reliability, ranking consistency, and task-dependent disagreement patterns, reinforcing the need to treat LLM judging as an empirical object of study rather than as an oracle (59, 60, 61), particularly in hybrid pipelines where judge behavior directly affects precision–recall trade-offs. This perspective is directly reflected in the present study, where judge behavior is explicitly measured and analyzed as part of the experimental outcomes, constituting a methodological element of the evaluation rather than a fixed assessment oracle.

In privacy-oriented evaluations, alignment between human assessments and LLM-based judgments may be imperfect and sensitive to framing, so auditing practices should retain judge evidence and preserve decision traceability to support post-hoc analysis of disagreements (62).

3.6 EVIDENCE-ORIENTED EXPERIMENTAL DESIGN

As illustrated in Figure 3.1, the evidence-ready layer spans all stages of the pipeline, capturing configurations, prompts, logs, outputs, and evaluation artifacts to support post-hoc verification.

In this dissertation, an *evidence-ready* experiment refers to an experimental design in which all reported claims are explicitly associated with preserved, inspectable, and linkable artifacts that

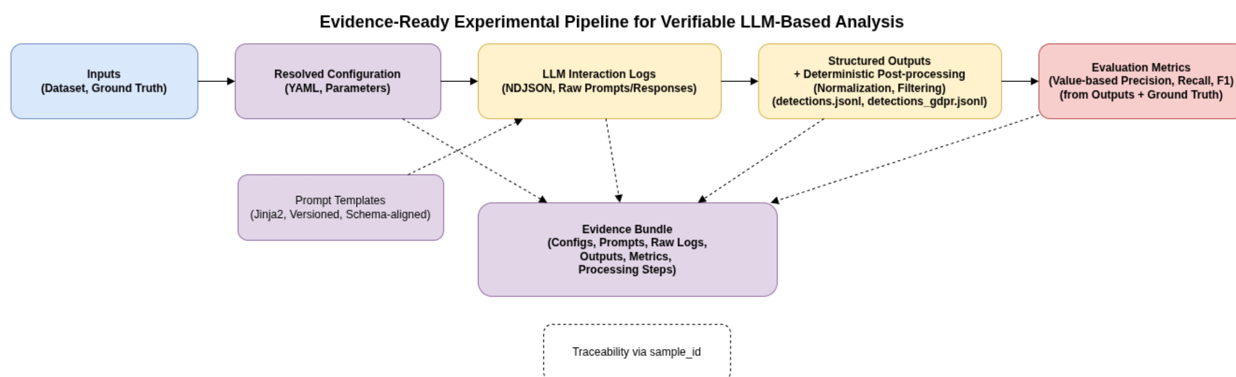

Figure 3.5: Evidence-ready experimental pipeline for verifiable LLM-based analysis. Each stage produces structured and inspectable artifacts, including configuration snapshots, versioned prompt templates, raw LLM interaction logs, structured outputs with deterministic post-processing (normalization and filtering), and value-level evaluation metrics derived from outputs and ground truth. All artifacts are preserved in a unified evidence bundle, enabling traceability, evidence-oriented verification support, and post-hoc verification without requiring model re-execution.

enable post-hoc verification independently of mandatory model re-execution.

Figure 3.5 presents the evidence-ready experimental pipeline adopted in this study. The design explicitly represents how all execution stages generate structured and inspectable artifacts, including configuration snapshots, prompt templates, raw interaction logs, structured outputs, deterministic post-processing steps, and evaluation metrics. These artifacts are consolidated into a unified evidence bundle, enabling traceability, independent verification, and metric recomputation without requiring model re-execution.

As shown in Figure 3.5, the evidence-ready pipeline explicitly separates execution, artifact generation, and evaluation concerns, ensuring that all reported results can be traced back to persisted and inspectable evidence.

Building on the controlled comparative protocol defined in the previous sections, the study extends the methodology with an *evidence-ready experimental design*, in which all experimental claims are explicitly grounded in inspectable and persisted artifacts. Rather than relying solely on execution reproducibility, the methodology emphasizes **post-hoc verifiability**, enabling independent validation of results without requiring re-execution of the underlying models.

Whereas the experimental protocol defines the comparative structure of the study (inputs, treatments, and evaluation criteria), the evidence-ready layer defines the verification contract that binds reported claims to persisted run-level artifacts.

This design choice is particularly relevant in LLM-based pipelines, where model availability, version drift, and infrastructure constraints may prevent exact reproducibility over time. By preserving structured evidence at each stage of the pipeline, the experiment ensures that reported results can be independently audited, inspected, and recomputed from the archived artifacts.

To operationalize this approach, the experimental design incorporates (i) structured evidence

bundles, (ii) verification-oriented controls, and (iii) traceability mechanisms across all pipeline stages.

This perspective aligns with recent discussions in empirical software engineering that emphasize the need for verifiable experimental pipelines, in which claims are supported by inspectable evidence rather than solely by executable artifacts or reported summaries.

This design is aligned with recent proposals for verifiable LLM-based analysis pipelines, which emphasize evidence-centric evaluation and artifact-based validation as key requirements for scientific rigor in configuration-driven experimental settings (58).

In this perspective, experimental evidence is treated as a first-class research artifact, and reported results are interpreted as derivations over preserved evidence rather than as transient outputs of a specific execution environment.

3.6.1 Evidence Bundle Specification

Each experimental execution produces a structured *evidence bundle* that captures all artifacts required to support post-hoc verification.

Table 3.7 summarizes the structure of the evidence bundle, detailing the main artifact types, their formats, and their role in supporting traceability, auditability, and post-hoc verification.

The evidence bundle includes:

- **Input artifacts:** immutable dataset samples and identifiers;
- **Configuration snapshot:** fully resolved YAML configuration capturing all parameters used during execution;
- **Prompt templates:** versioned Jinja2 templates defining both system and user instructions;
- **LLM interaction logs:** raw NDJSON records containing timestamps, model identifiers, parameters, full prompts, and unprocessed responses or errors;
- **Structured outputs:** predicted entities and normalized outputs generated by each pipeline, including deterministic post-processing steps (normalization and filtering) applied prior to evaluation;
- **Evaluation artifacts:** metric reports and intermediate matching structures used to compute precision, recall, and F1.

Together, these artifacts enable independent inspection, validation, and recomputation of experimental outcomes without requiring access to the original execution environment.

Table 3.7: Evidence bundle specification. Each component represents a class of artifacts preserved during execution to support traceability, auditability, and post-hoc verification.

Component	Format	Description
Dataset (Inputs)	JSONL	Java code snippets with PII annotations and stable identifiers used as input for all pipelines.
Ground Truth	JSONL	Annotated reference data defining entity values and categories used for value-level matching during evaluation.
Configuration (Config)	YAML	Fully resolved experiment configuration capturing parameters, model settings, and execution options.
Prompt Templates	Jinja2	Versioned templates defining system and user prompts, aligned with the expected output schema.
LLM Interaction Logs	Newline-Delimited JSON (NDJSON)	Raw logs of all LLM interactions, including timestamps, prompts, parameters, and unprocessed responses or errors.
Structured Outputs	JSONL	Predicted entities generated by each pipeline, including deterministic post-processing (normalization and filtering).
Evaluation Metrics	Reports (JSON)	Computed precision, recall, and F1 scores derived from outputs and ground truth using value-level matching.

In methodological terms, this evidence bundle is expected to satisfy three core properties: completeness, in the sense that all artifacts required for verification are preserved; linkability, in the sense that artifacts can be joined across stages through stable identifiers; and determinism, in the sense that post-processing and metric computation can be replayed over the preserved outputs.

These properties collectively define a minimal verification boundary, within which all reported results can be independently inspected, audited, inspected, and independently verified from the preserved artifacts without access to the original runtime environment.

In this dissertation, this verification boundary is concretely instantiated by the archived configuration snapshots, versioned prompt templates, NDJSON interaction logs, structured JSONL outputs, and metric artifacts documented in Appendix III.

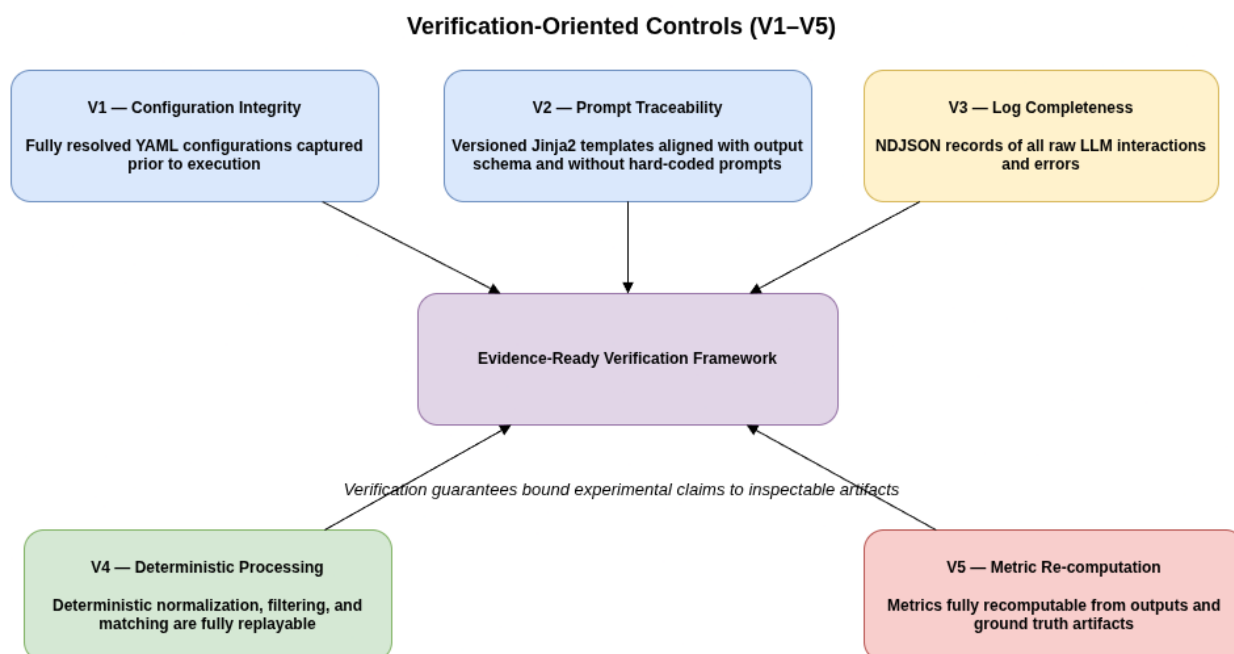

Figure 3.6: Verification-oriented controls (V1–V5) in the evidence-ready framework. Each control binds a distinct aspect of the experiment to inspectable artifacts, covering configuration integrity, prompt traceability, log completeness, deterministic processing, and metric recomputation. Together, these controls define a verification contract that enables independent inspection and validation of experimental results without requiring model re-execution.

3.6.2 Verification-Oriented Controls

The verification hooks (V1–V5) correspond to the stages illustrated in Figure 3.5 and are summarized as a unified verification framework in Figure 3.6, which define how configuration integrity, prompt traceability, log completeness, deterministic processing, and metric recomputation are operationalized through persisted artifacts.

Figure 3.6 summarizes the verification-oriented controls (V1–V5) that operationalize the evidence-ready design. Each control binds a specific aspect of the experimental pipeline to inspectable artifacts, collectively defining a coherent verification framework that ensures traceability, auditability, and independent validation of experimental claims.

To ensure that experimental results are verifiable, the methodology incorporates a set of five *verification hooks* (V1–V5) that operationalize the evidence-ready protocol as a set of explicit verification guarantees by binding each reported metric and experimental claim to concrete, inspectable evidence. Collectively, these controls define a *verification contract* for the experimental pipeline, specifying the minimum set of evidence artifacts and deterministic guarantees required to independently inspect, audit, and recompute the reported experimental outcomes.

- **V1 – Configuration integrity:** All executions are governed by externally defined YAML configurations, which are resolved and persisted prior to execution. This guarantees that all

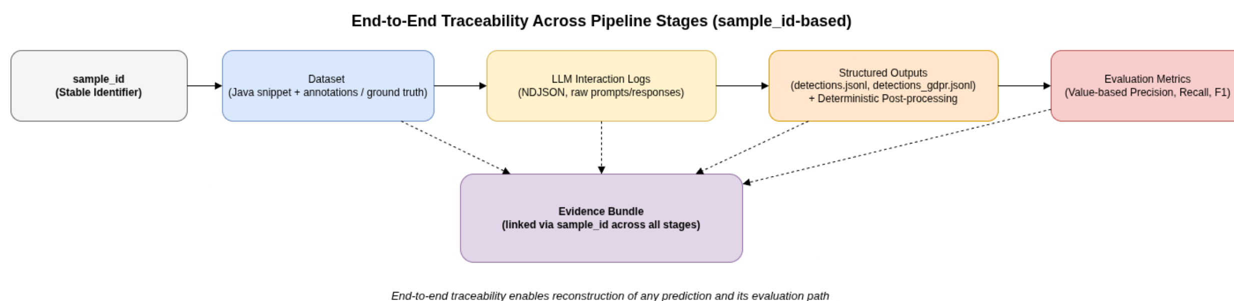

Figure 3.7: End-to-end traceability across pipeline stages using a stable sample identifier. The `sample_id` is propagated across dataset entries, LLM interaction logs, structured outputs, and evaluation metrics, enabling inspection of the full processing path of each prediction and supporting artifact-level auditability.

parameters influencing the experiment are explicitly recorded.

- **V2 – Prompt traceability:** All prompt definitions are implemented exclusively through versioned Jinja2 templates. No prompt fragments are hard-coded in the source code, ensuring that all LLM interactions are fully inspectable and traceable from artifacts.
- **V3 – Log completeness:** Every interaction with a language model, including health checks and batch processing steps, is recorded as a raw NDJSON entry containing timestamps, model identifiers, parameters, full prompts, and raw responses or errors.
- **V4 – Deterministic post-processing:** All normalization, filtering, and matching procedures are implemented as deterministic transformations. This ensures that outputs can be recomputed exactly from logged model responses.
- **V5 – Metric recomputation:** All evaluation metrics are derived from persisted prediction outputs and ground-truth data, enabling independent recomputation and validation of reported results.

Together, as illustrated in Figure 3.6, these controls ensure that each stage of the experimental pipeline can be independently verified through artifact inspection. In practical terms, V1–V5 define the minimum verification contract required, as visually summarized in Figure 3.6, to inspect configuration integrity, prompt traceability, execution evidence, deterministic processing, and metric derivation without mandatory model re-execution.

3.6.3 Traceability Across Pipeline Stages

As illustrated in Figures 3.1, 3.5, 3.6, and 3.7, traceability is maintained through stable identifiers propagated across all pipeline stages.

Figure 3.7 provides a concrete view of this traceability mechanism, showing how a stable

identifier (`sample_id`) links all artifacts across the pipeline, from dataset inputs and LLM interaction logs to structured outputs and evaluation metrics.

All artifacts generated during the experiment are linked through stable identifiers propagated across pipeline stages. In particular, each sample is associated with a unique identifier that is preserved across:

- input dataset entries;
- LLM interaction logs;
- predicted outputs;
- evaluation artifacts.

This traceability mechanism, based on stable join keys propagated across stages, enables inspection of the full processing path of any prediction, making it possible to trace results back to their originating inputs, intermediate LLM interactions, and post-processing transformations. As a result, both correct predictions and errors can be systematically analyzed at the artifact level.

3.6.4 Reproducibility vs. Verifiability

Traditional experimental designs emphasize reproducibility, assuming that results can be validated by re-executing the experiment under the same conditions. However, in LLM-based systems, this assumption is often fragile due to model updates, non-deterministic inference behavior, and restricted access to proprietary models.

This study adopts a complementary perspective by emphasizing *verifiability through artifacts*. Instead of requiring re-execution, the experimental design enables independent validation through inspection of persisted evidence, deterministic re-execution of post-processing steps, and recomputation of evaluation metrics.

This distinction is particularly important for ensuring long-term scientific validity in LLM-based software engineering research.

Importantly, verifiability through preserved evidence should not be interpreted as a guarantee of semantic correctness, regulatory completeness, or real-world deployment validity. Instead, the evidence-ready methodology ensures that reported outcomes remain inspectable, auditable, and derivable from preserved experimental artifacts under controlled conditions.

In particular, the methodology recognizes that exact re-execution may become infeasible over time due to model deprecation, infrastructure changes, provider-side updates, or nondeterministic inference behavior. Under such conditions, preserved evidence artifacts become the primary mechanism for sustaining long-term inspectability and scientific auditability.

This methodological distinction is central to the present dissertation and extends traditional reproducibility-oriented reporting by treating preserved run evidence as the primary basis for post-hoc verification of experimental claims.

Accordingly, the evidence-ready layer presented in this chapter should be understood not as an auxiliary reporting mechanism, but as a methodological contribution of the dissertation that operationalizes how experimental claims are made independently auditable in configuration-driven LLM pipelines.

Under this perspective, reproducibility and verifiability are treated as complementary but distinct methodological properties, where artifact preservation becomes the primary mechanism for sustaining long-term scientific inspectability in evolving LLM ecosystems.

3.6.5 LLM as First-Class Experimental Component

In this study, LLMs are treated as first-class experimental components. Their behavior is explicitly parameterized, logged, and controlled through external configuration and template-based prompting.

All model interactions are captured as raw artifacts, ensuring that their behavior can be audited independently of the execution environment. This approach contrasts with traditional black-box usage of LLMs and enables systematic analysis of model decisions within the experimental pipeline.

3.7 THREATS TO VALIDITY

Threats to validity are treated as structured methodological elements that link concrete design choices to plausible risks for inference (32).

Evidence suggests that superficial checklist-style reporting weakens auditability and interpretability, motivating threat analyses that explicitly connect protocol decisions to risks (32).

Accordingly, the dissertation reports threats in relation to constructs, measurements, and generalization scope, rather than treating the section as a conventional formality (32, 13).

As with any empirical study, our findings are subject to threats to validity.

We discuss these threats following the standard categories in empirical software engineering: internal, construct, external, and conclusion validity (13).

Internal validity concerns whether the observed differences among pipelines can be attributed to the intended experimental factors rather than to uncontrolled variation.

A primary threat arises from prompt design and configuration choices in the LLM-based

pipelines, because LLM behavior in privacy-related code analysis can be sensitive to instruction phrasing, contextual framing, and input formatting (10).

To mitigate this threat, we fixed prompt templates across all runs, disabled stochastic decoding where supported, preserved complete execution evidence artifacts, and applied deterministic post-processing, while archiving complete run artifacts to support auditability (13, 8).

A related threat is that LLM-as-a-judge configurations may exhibit systematic biases (e.g., position or verbosity effects) and task-dependent limitations that distort evaluation outcomes.

Although we mitigated this risk through fixed prompts, deterministic settings where supported, and complete raw logging, residual judge bias may still contribute to recall degradation in the hybrid pipeline (7, 6).

We therefore interpret judge-induced recall loss as an empirical observation under the declared protocol rather than as a definitive property of LLM judging in general (7, 6).

A second internal validity threat concerns the construction of the hybrid pipeline, where classifier outputs are aggregated into a candidate pool subsequently judged by an LLM.

Variations in confidence thresholds, normalization rules, aggregation logic, or deduplication criteria could alter this pool and thereby change the apparent effectiveness of judging (8, 13).

We addressed this threat by holding the candidate pool and evaluation invariants constant as archived artifacts, enabling recomputation of judge outcomes directly from retained evidence (13, 32).

A third threat relates to implementation and runtime effects, including differences in backends, quantization, or hardware, which may influence latency, output stability, or instruction-following behavior (30).

We mitigated this risk through configuration versioning, consistent preprocessing and scoring logic, and complete archiving of run artifacts (13, 15).

Construct validity concerns whether the operationalization of the task adequately captures the intended phenomenon.

Our evaluation operationalizes privacy auditing as entity-level PII extraction using normalized values as the matching key, aligning with practical auditing scenarios that prioritize identification of sensitive literals over exact character spans (3, 13).

Additionally, construct validity is strengthened by the deterministic validation procedures applied during dataset generation, including schema-conformance checks, annotation-offset verification, category-consistency validation, and exclusion of malformed records prior to inclusion in the final ground-truth dataset.

However, this construct does not capture broader notions of privacy impact, such as personal data flows, implicit identifiers derived from program state, or higher-level processing purposes

(5).

Additionally, because the dataset is organized as isolated code snippets, the evaluation does not capture broader inter-procedural context, cross-file relationships, runtime propagation, or architectural privacy flows that may influence privacy interpretation in industrial systems.

A related construct validity threat concerns label coverage and equivalence, because alternative taxonomies or jurisdiction-specific interpretations can alter which categories are considered compliance relevant.

To mitigate ambiguity, label definitions, mappings, and scoring rules are made explicit and archived as verification artifacts (3, 4, 13).

External validity concerns the extent to which the results generalize beyond the study setting.

The primary limitation is the use of a synthetic dataset of Java snippets, which enables controlled annotation and broad category coverage but does not fully capture the stylistic and architectural diversity of industrial systems (11, 8).

Template-guided generation may also reduce stylistic variance and increase regularity, potentially making structured prompting and end-to-end extraction easier than on organically evolved code (8, 10).

Nevertheless, the controlled setting is appropriate for the comparative objective of this study, because all pipelines are evaluated on identical samples using the same matching criteria and metrics, making the observed trade-offs informative as architectural signals (8, 13).

A second limitation concerns language scope, because the study focuses on Java and results may not transfer directly to languages with different idioms or syntax such as Python, JavaScript, or C# (10, 5).

Finally, rapid model evolution poses an ongoing external validity threat, and the study mitigates this risk through exact model identification and complete archiving of configurations and run artifacts for auditability and post-hoc verification (13, 32).

Conclusion validity concerns whether the conclusions are supported by the data and whether the analysis is sufficiently robust.

We report micro-averaged precision, recall, and F_1-score under deterministic value-level matching criteria, which is appropriate for summarizing overall extraction effectiveness but can obscure per-label variance (13, 8).

To support deeper inspection, the reproducibility package includes per-label breakdowns and retained artifacts enabling recomputation and error analysis beyond aggregate values (32, 13).

Another threat arises from variability in LLM outputs even under constrained decoding, and the study mitigates this risk through deterministic settings where supported and by retaining raw interaction traces for independent diagnosis (30).

Furthermore, some inference backends and hosted execution environments may not provide strict guarantees of determinism even when stochastic decoding parameters are disabled, which reinforces the importance of artifact preservation and raw interaction logging for independent verification.

3.8 SUMMARY

This chapter presented the methodological foundations for controlled, comparative evaluation of privacy-relevant code analysis techniques, emphasizing protocol transparency, preserved evidence artifacts, and interpretable metrics (13, 15).

It operationalized the experimental protocol through two supporting infrastructure artifacts: a synthetic dataset generation framework that produces versioned dataset snapshots and ground truth (Appendix I), and a unified detection framework that standardizes treatment execution and archives run artifacts for auditability (Appendix II) (13).

The concrete configuration files, file schemas, evaluation invariants, and retained archived evidence artifacts required to verify the reported runs are consolidated in Appendix III (32, 13).

It positioned LLMs as scalable annotators and, when applicable, as judges within empirical SE pipelines, while stressing that such uses require explicit acceptance criteria, documented configurations, and stability-aware reporting (34, 6).

It motivated the use of synthetic datasets as an experimental enabler under privacy constraints, while acknowledging utility and external-validity limitations that must be reported to preserve scientific rigor (11, 54, 8).

Finally, it established that threats to validity are treated as structured methodological artifacts tied to concrete design choices, supporting auditable and independently verifiable conclusions (32, 13).

More specifically, the chapter established a methodological separation between the comparative protocol that structures the experiment and the evidence-ready verification layer that binds reported claims to archived artifacts.

The next chapter (Chapter 4) reports the experimental setup and the quantitative results obtained from the evaluated detection strategies under the protocol defined in this chapter, using the archived artifacts described in Appendix III (13, 32).

In particular, the chapter establishes that experimental claims are not supported solely by executable pipelines, but by a verification-oriented evidence structure that enables independent inspection, traceability, and metric recomputation from archived artifacts, thereby enabling experimental claims to remain inspectable, auditable, and scientifically verifiable even under evolving

LLM ecosystems where exact re-execution may no longer be feasible.

4 EXPERIMENTAL EVALUATION

This chapter reports the experimental results obtained from the archived executions of the three detection strategies introduced in Chapter 3: (i) a classifier-only baseline (P1), (ii) a hybrid pipeline combining classifier candidates with an LLM-as-a-judge stage (P2), and (iii) an LLM-centered structured extraction pipeline with deterministic post-processing (P3). The chapter presents the results extracted from the archived experiment artifacts, highlights recurrent error patterns, and discusses the main trade-offs observed across the evaluated strategies.

An important methodological note concerns the architectural heterogeneity of the archived evaluation regimes. The result artifacts differ across pipelines in aggregation, post-processing, and reporting structure. P1 exposes mapped aggregate metrics produced by the classifier ensemble, P2 stores hybrid outputs generated from constrained candidate routing and LLM-based filtering, and P3 reports value-normalized metrics after deterministic sanitization and normalization. Accordingly, the comparisons presented in this chapter should be interpreted as comparative architectural signals derived from the archived artifacts rather than as a strictly harmonized benchmark under a single unified evaluation abstraction.

To avoid misinterpretation, the numerical results reported throughout this chapter are not used to establish absolute performance rankings across pipelines. Instead, they are interpreted as architecture-dependent signals observed under distinct evaluation regimes. In particular, Pipeline 3 results reflect value-normalized predictions after deterministic sanitization, whereas Pipeline 2 results reflect aggregated classifier judgments under constrained candidate routing. Therefore, any cross-pipeline comparison should be understood as a comparative architectural analysis of integration strategies rather than a strictly controlled benchmark comparison.

4.1 EXPERIMENTAL SETUP

4.1.1 Execution Environment

All experiments were executed using a combination of a local workstation and cloud-based computational resources in order to support reproducibility while reflecting realistic hardware constraints commonly faced by academic research environments.

Local workstation. A desktop machine equipped with an NVIDIA RTX 4060 GPU with 8 GB of VRAM, an Intel i7-class processor, and 32 GB of RAM was used for the development, debugging, and validation of the experimental framework. This environment also supported the execution of the transformer-based classifiers of Pipeline 1 using the HuggingFace inference

stack, as well as baseline executions of Pipeline 2 using the CodeLlama (`codellama:13b-instruct-q4_K_M`) model deployed locally through Ollama.

Cloud environment. To support the execution of larger open-weight language models, additional experiments were conducted using Google Colab Pro environments equipped with NVIDIA T4 and L4 GPUs, providing between 15 GB and 24 GB of VRAM. All cloud-based executions preserved the same datasets, configuration files, prompt templates, and evaluation procedures used in the local environment.

Software stack. All pipelines were implemented in Python 3.10 with dependency management handled through Poetry. Transformer-based classifiers were executed using the HuggingFace *Transformers* and *Accelerate* libraries, while open-weight language models were executed through Ollama version 0.3.x. Prompt definitions were implemented exclusively through Jinja2 templates, and all configurations, prompts, and experiment parameters are included in the archived research artifacts.

Reproducibility controls. Every interaction with a language model was recorded as raw NDJSON logs containing timestamps, model identifiers, prompts, parameters, and raw responses or errors. All pipelines were executed using a shared YAML-based configuration structure to ensure consistent preprocessing, normalization, matching criteria, and metric computation across runs.

To summarize the main experimental configurations used across the three pipelines, Table 4.1 presents an overview of the models, execution environments, and key parameters involved in each setup. While the experimental setup has been described in detail, a consolidated view is provided next to facilitate reproducibility.

Table 4.1: Summary of experimental configurations across pipelines

Pipeline	**Models / Approach**	**Execution Environment**	**Key Parameters**
P1 (Classifier-Only)	Ensemble of 6 transformer-based classifiers	Local workstation (RTX 4060, 8GB VRAM)	HuggingFace Transformers; OR aggregation
P2 (Hybrid)	Classifier ensemble + LLM-as-a-judge (CodeLlama, Mistral, Qwen3, etc.)	Local (Ollama) + Colab Pro (T4/L4)	Candidate routing; prefiltering; LLM judging
P3 (LLM-Centered Extraction)	Open-weight LLMs (CodeLlama, CodeGemma, Devstral, etc.)	Colab Pro (T4/L4) + local validation	Structured JSON output; deterministic sanitization; value normalization

This consolidated view highlights the increasing computational and architectural complexity from P1 to P3, as well as the growing reliance on LLM-based components and structured output

processing mechanisms.

The transformer-based classifiers and open-weight LLMs evaluated in this chapter were previously introduced in Section 3.2.1, including their architectural categories, experimental roles, and corresponding model identifiers.

4.2 OVERVIEW OF THE EXPERIMENTAL EVALUATION

The experiment configuration artifacts include one baseline configuration for Pipeline 1, thirteen hybrid configurations for Pipeline 2, and twelve archived LLM-centered extraction configurations for Pipeline 3. However, the archived execution results are not complete for all configured runs.

The result bundle contains one complete execution for P1, twelve complete executions and one incomplete execution for P2, and twelve execution directories for P3. Among the P3 executions, eleven expose the full extended metric bundle used throughout the detailed analysis, while one execution provides only a reduced metric artifact set. The latter still exposes aggregate metrics but lacks the complete extended evaluation bundle used for deeper per-label analysis. Accordingly, this chapter analyzes the complete and directly usable archived result artifacts.

For P1, the available execution corresponds to an OR-ensemble of six transformer-based classifiers.

For P2, the complete archived judge runs include CodeGemma, CodeLlama, Codestral, DeepSeek, Devstral-small-2, Gemma3, GPT-OSS, Mistral, Phi-4, Qwen3, Qwen3-Coder, and Qwen3-IT. One additional P2 execution (StarCoder2) is present in the artifact bundle but marked as incomplete and therefore excluded from the comparative analysis because the model did not consistently produce a complete set of valid judge outputs for all evaluation samples, preventing generation of a fully comparable result set.

For P3, the archived executions include CodeGemma, CodeLlama, Codestral, DeepSeek, Devstral-small-2, Gemma3, GPT-OSS, Mistral, Phi-4, Qwen3, Qwen3-Coder, and Qwen3-IT, although one of these runs exposes only a reduced metric artifact.

To provide a consolidated view of the overall performance across the three evaluated pipelines, Table 4.2 summarizes their aggregate precision, recall, and F1-score.

As shown in Table 4.2, the three pipelines exhibit distinct performance profiles. The classifier-only baseline (P1) achieves the highest recall, reflecting its role as a candidate-generation strategy, but at the cost of lower precision.

Figure 4.1 visually highlights the distinct behavioral profiles of the evaluated pipelines. The classifier-only baseline (P1) emphasizes recall-oriented candidate generation, the hybrid pipeline

Table 4.2: Representative aggregate metrics across the evaluated pipeline regimes

Pipeline	**Precision**	**Recall**	**F1-score**
P1 (Classifier-Only)	0.4339	0.7143	0.5398
P2 (Hybrid)	0.5183	0.2428	0.3193
P3 (LLM-Centered Extraction)	0.8232	0.6377	0.7186

Metrics are reported under the native archived evaluation regime of each pipeline and should therefore be interpreted as architectural signals rather than as directly harmonized benchmark scores.

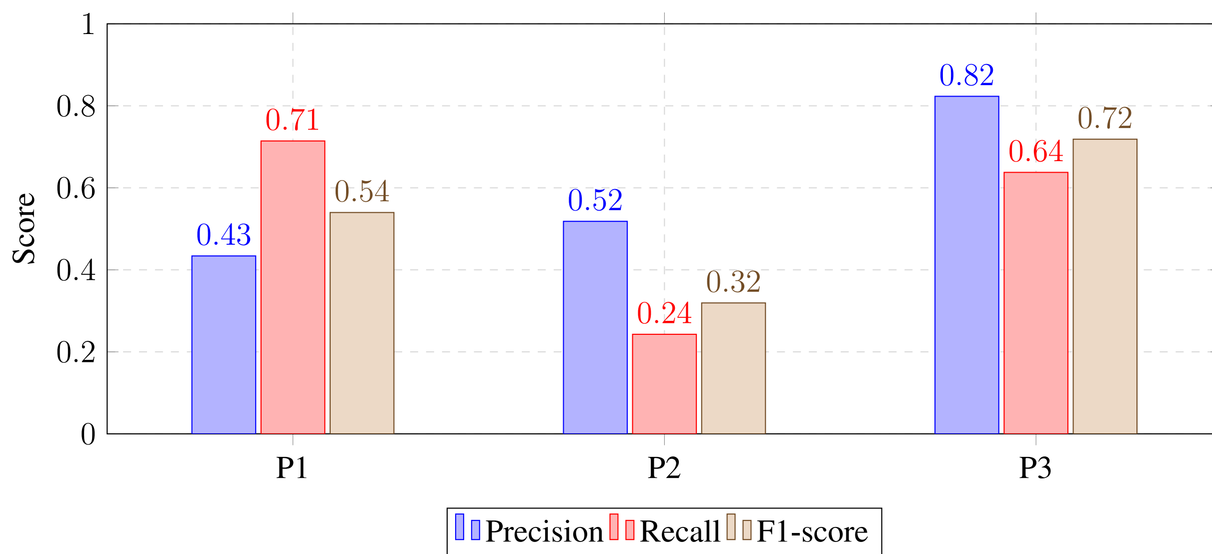

Figure 4.1: Comparative precision, recall, and F1-score profiles across the evaluated pipeline regimes.

(P2) exhibits a substantial recall collapse despite modest precision gains, and the LLM-centered extraction pipeline (P3) achieves the strongest observed precision–recall balance under the archived evaluation regimes.

The hybrid pipeline (P2) improves precision compared to P1 but introduces a substantial reduction in recall, indicating that the LLM-based filtering stage behaves conservatively and discards a significant number of true positives.

In contrast, the LLM-centered extraction pipeline (P3) achieves the strongest observed precision–recall balance under its archived evaluation regime, resulting in the highest F1-score. This suggests that, under the respective evaluation regimes of each pipeline, direct structured extraction combined with deterministic post-processing provides a more favorable precision–recall trade-off under the archived experimental conditions analyzed in this study.

Following the overall comparison, Table 4.3 presents a representative subset of aggregate results used to summarize the main cross-pipeline patterns observed in the archived executions. The table is intentionally restricted to the baseline, the aggregate hybrid summary, and a subset of representative LLM-centered extraction runs so as to foreground the dominant empirical patterns without overloading the narrative with per-model detail.

Table 4.3: Representative subset of aggregate results from the archived experiment executions. The table highlights selected runs used to illustrate the dominant cross-pipeline performance patterns.

Pipeline / Model	Precision	Recall	F1
P1 Ensemble OR	0.4339	0.7143	0.5398
Representative P2 run (Mistral)	0.5162	0.2442	0.3198
P2 Mean of valid runs	0.5183	0.2428	0.3193
P3 CodeGemma	0.7965	0.5016	0.6155
P3 CodeLlama	0.8200	0.6124	0.7012
P3 Codestral	0.7841	0.5580	0.6520
P3 DeepSeek	0.8066	0.5889	0.6808
P3 Devstral-small-2	**0.8232**	**0.6377**	**0.7186**

A complete set of results, including all evaluated models and execution runs, is available in the archived experiment artifacts. The subset presented here is intended to illustrate dominant patterns rather than to provide exhaustive coverage of all configurations.

These observations are consistent with the detailed results presented in Table 4.3, where the classifier-only baseline achieves the highest recall among the evaluated approaches, while the hybrid pipeline significantly reduces recall without improving overall effectiveness. In contrast, the strongest LLM-centered extraction configurations achieve higher precision and stronger F1 values, suggesting a more balanced detection behavior.

4.3 RESEARCH QUESTIONS

The experimental evaluation presented in this chapter aims to answer the research questions introduced in Chapter 1. In particular, the analysis focuses on the following questions:

- **RQ1:** How effective are transformer-based classifiers in detecting PII within Java source code?
- **RQ2:** Does a hybrid architecture combining classifiers with an LLM-as-a-judge stage improve detection performance?
- **RQ3:** Can an LLM-centered structured extraction pipeline with deterministic post-processing provide a stronger precision–recall balance than classifier-centered approaches for PII extraction in source code?

RQ1 is addressed through the analysis of the classifier-only baseline (Pipeline 1). RQ2 is evaluated through the hybrid architecture (Pipeline 2). RQ3 is investigated using the LLM-centered structured extraction pipeline (Pipeline 3).

4.4 RESULTS BY PIPELINE

The results are presented according to the research questions defined in Chapter 1. Because each research question is evaluated through a specific pipeline configuration, the analysis is organized by research question while preserving the corresponding pipeline structure.

RQ1 – Effectiveness of Transformer-Based Classifiers (Pipeline 1)

The P1 OR-ensemble, obtained through the union-based aggregation of six transformer-based classifiers, achieved a precision of 0.4339, recall of 0.7143, and F1-score of 0.5398. This result is consistent with the architectural role of the baseline: the union of six classifier outputs produces a broad candidate set that captures many true positives but also admits a large number of false positives.

These results address RQ1 by establishing the empirical effectiveness and limitations of transformer-based classifier ensembles as a baseline strategy for detecting PII in Java source code.

Individual classifiers versus OR aggregation

As shown in Table 4.4, the individual classifier results reveal substantial behavioral heterogeneity across the evaluated transformer-based models. Some classifiers prioritize precision at

Table 4.4: Performance of individual classifiers and OR-ensemble aggregation in Pipeline 1

Classifier	Precision	Recall	F1-score
AB-AIPII Model	0.4636	0.3894	0.4233
H2OAI DeBERTa Fine-Tuned PII	0.4513	0.3318	0.3824
Lakshyakh93 DeBERTa Fine-Tuned PII	0.4426	0.3399	0.3845
PIIRanha-v1	**0.7089**	0.2454	0.3646
DistilBERT Multilingual	0.5553	0.1587	0.2469
BigCodeStarPII	0.6780	0.0976	0.1707
P1 OR-ensemble	0.4339	**0.7143**	**0.5398**

the expense of recall, whereas others exhibit comparatively more balanced but still limited coverage.

The results reported in Table 4.4 show that the OR-based ensemble aggregation substantially amplifies recall by combining heterogeneous detection behaviors across classifiers, increasing recall from values below 0.40 in all individual models to 0.7143 in the aggregated ensemble. However, this aggregation strategy also propagates classifier-specific false positives, reducing overall precision and producing the false-positive inflation observed in the final ensemble output.

These results reinforce the interpretation of Pipeline 1 as a recall-oriented candidate-generation architecture rather than a precision-oriented extraction pipeline. In practice, the OR ensemble maximizes detection coverage while substantially increasing downstream validation burden due to the large number of false positives admitted into the candidate pool.

The per-label results reinforce this interpretation. The baseline performed particularly well on highly structured categories such as `MAC`, `BITCOINADDRESS`, `SECONDARYADDRESS`, `IPV6`, `KEY`, `PIN`, and `ZIPCODE`. In contrast, it performed poorly on categories such as `PHONEIMEI`, `COUNTY`, `NEARBYGPSCOORDINATE`, `ACCOUNTNAME`, `MASKEDNUMBER`, `ACCOUNTNUMBER`, and `JOBTITLE`. These weaker categories are typically more context-sensitive or more easily confused with program text and contextual identifiers.

From an operational perspective, the baseline constitutes a recall-oriented candidate-generation strategy. Its main limitation lies in the operational burden associated with false positives, which may increase manual triage effort, reduce reviewer trust, and make the baseline less attractive as a standalone privacy-oriented detection workflow.

From a privacy-governance perspective, this behavior implies increased manual validation effort and a higher likelihood of alert fatigue, which may reduce reviewer trust in automated detection systems.

RQ2 – Impact of the Hybrid LLM-as-a-Judge Architecture (Pipeline 2)

The archived hybrid runs produced highly concentrated aggregate metrics across the twelve complete judge executions. The mean result across the valid runs was approximately 0.5183 pre-

cision, 0.2428 recall, and 0.3193 F1. Although minor metric variations are observable across models, the archived hybrid executions remain concentrated within an extremely narrow performance range, with multiple judge models converging to nearly identical aggregate results.

These results provide the primary empirical evidence for RQ2, evaluating whether hybrid architectures combining classifier outputs with LLM-based candidate judgment improve overall detection effectiveness.

Table 4.5: Aggregate metrics across archived P2 hybrid judge executions

Judge Model	**Precision**	**Recall**	**F1-score**
CodeGemma	0.5161	0.2442	0.3198
CodeLlama	0.5161	0.2442	0.3198
Codestral	0.5240	0.2390	0.3177
DeepSeek	0.5161	0.2442	0.3198
Devstral-small-2	0.5161	0.2442	0.3198
Gemma3	0.5161	0.2442	0.3198
GPT-OSS	0.5200	0.2411	0.3183
Mistral	0.5162	0.2442	**0.3198**
Phi-4	0.5161	0.2442	0.3198
Qwen3	0.5201	0.2413	0.3184
Qwen3-Coder	0.5188	0.2428	0.3193
Qwen3-IT	0.5244	0.2406	0.3190
Mean of valid runs	0.5183	0.2428	0.3193

As shown in Table 4.5, the archived hybrid runs exhibit remarkably low variability across different judge models. Despite differences in model families and parameter scales, the aggregate metrics remain concentrated around nearly identical precision, recall, and F1-score ranges.

The metric concentration observed in Table 4.5 suggests that the dominant limitation of the archived hybrid pipeline is primarily architectural rather than model-specific. In particular, the constrained candidate-routing strategy and upstream filtering policies appear to impose a strong recall ceiling that cannot be substantially mitigated by replacing the downstream LLM judge.

A key empirical finding is that the LLM-as-a-judge stage did not improve the overall effectiveness of the archived hybrid configuration. Instead, the judge acted primarily as a conservative selective filtering layer that removed some false positives but also discarded a non-negligible number of true positives, producing a systematic reduction in recall. Artifact-level traces provide additional insight into this behavior.

Figure 4.2 illustrates the constrained routing behavior observed in the archived hybrid architecture. Most candidates are automatically retained upstream, while only a relatively small fraction is routed to the LLM judge. This distribution indicates that the downstream judge has limited influence over the overall pipeline behavior, reinforcing the interpretation that the observed recall degradation emerges primarily from upstream filtering and candidate-generation constraints rather than from the semantic capabilities of the language model itself.

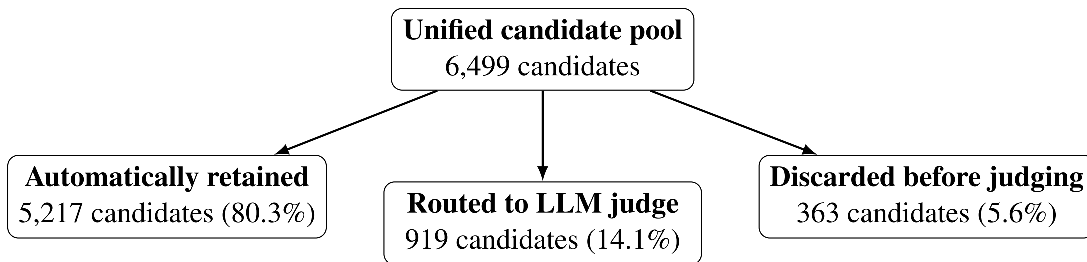

Figure 4.2: Representative candidate-routing behavior observed in the archived P2 hybrid pipeline execution.

In a representative archived execution (e.g., CodeGemma-based judging), the unified candidate pool contained 6,499 candidates. Of these, 5,217 (80.3%) were automatically retained by the prefilter, 919 (14.1%) were routed to the LLM judge, and 363 (5.6%) were discarded before the judging stage. The final predicted output decreased only modestly from 12,221 entries in `predicted_before.jsonl` to 12,078 entries in `predicted_after.jsonl`, suggesting that the hybrid behavior was driven more strongly by upstream filtering policies than by context-sensitive candidate filtering performed by the judge.

In practical terms, this means that the archived hybrid design did not deliver its intended architectural promise of reducing false positives while preserving enough true positives to improve end-to-end effectiveness.

This behavior is further supported by artifact-level evidence, including the limited reduction between `predicted_before.jsonl` and `predicted_after.jsonl`, which indicates that the LLM judge operates under constrained influence within the pipeline.

These routing proportions indicate that the LLM judge operates under constrained influence within the hybrid architecture. Since the majority of candidates are either automatically retained or discarded before reaching the judge, the LLM stage has limited opportunity to alter the overall system behavior. As a result, the observed performance appears to be dominated by upstream filtering policies rather than by downstream context-sensitive filtering performed by the judge.

Because the hybrid stage operates exclusively over the fixed and deduplicated candidate pool generated upstream, entities absent from the classifier ensemble cannot be recovered by the judge stage. Accordingly, the observed hybrid behavior is structurally constrained by the recall ceiling imposed by candidate generation.

This low variability across judge models further reinforces that the dominant limitation of the archived hybrid configuration appears to be architectural rather than model-specific.

In compliance-sensitive scenarios, this recall reduction may translate into missed detections of sensitive data, posing potential risks in regulatory contexts such as GDPR and LGPD.

From an architectural perspective, the primary contribution of Pipeline 2 is not its aggregate effectiveness, but the empirical evidence it provides regarding the interaction between candidate

generation and semantic verification. The archived results suggest a persistent architectural tension between precision-oriented filtering and recall preservation. Because the judge operates only over candidates supplied by the upstream classifier ensemble, its ability to improve detection coverage is structurally constrained. At the same time, conservative acceptance behavior tends to reject both false positives and true positives, producing limited precision gains at the cost of substantial recall loss. The remarkably small performance variation observed across judge models further indicates that the dominant limitation of the archived hybrid configuration is architectural rather than model-specific.

RQ3 – Effectiveness of LLM-Centered Structured Extraction (Pipeline 3)

The LLM-centered extraction pipeline, which combines structured LLM generation with deterministic sanitization and normalization, produced the strongest results among the archived experiments under its evaluation regime. The strongest archived result under the evaluated structured extraction regime was obtained by Devstral-small-2, with 0.8232 precision, 0.6377 recall, and 0.7186 F1. CodeLlama also performed strongly, reaching 0.8200 precision, 0.6124 recall, and 0.7012 F1.

These findings provide the central empirical evidence for RQ3, indicating that LLM-centered structured extraction pipelines with deterministic post-processing can provide stronger precision–recall balance than classifier-centered approaches under the archived experimental conditions analyzed in this study.

Table 4.6: Aggregate metrics across archived P3 LLM-centered extraction executions

Model	Precision	Recall	F1-score
Devstral-small-2	**0.8232**	**0.6377**	**0.7186**
CodeLlama	0.8200	0.6124	0.7012
GPT-OSS	0.8166	0.6010	0.6922
DeepSeek	0.8066	0.5889	0.6808
Codestral	0.7841	0.5580	0.6520
CodeGemma	0.7965	0.5016	0.6155
Gemma3	0.7762	0.4885	0.5990
Qwen3-Coder	0.7668	0.4529	0.5700
Mistral	0.7542	0.3604	0.4890
Qwen3-IT	0.7444	0.3443	0.4740
Phi-4	0.6330	0.2852	0.3950
Qwen3	0.1910	0.0057	0.0107

As shown in Table 4.6, the P3 configurations exhibit substantial variability across models. While several models achieve strong precision–recall balance, others exhibit severe degradation, including near-collapse behavior.

The variability observed in Table 4.6 suggests that the effectiveness of LLM-centered structured extraction depends strongly on model capability, structured-output robustness, and compat-

ibility with deterministic sanitization and normalization stages. In particular, the near-collapse observed in the Qwen3 execution indicates that failures in structured output generation can propagate through the sanitization pipeline and substantially degrade end-to-end extraction effectiveness.

The deterministic sanitization pipeline applied after LLM extraction includes structured-output validation, category verification, value normalization, and rule-based filtering of malformed or unsupported entities. These stages enforce schema compliance and evaluation consistency, but malformed or non-compliant LLM outputs may be rejected during sanitization, directly affecting recall and end-to-end extraction effectiveness.

Artifact-level evidence further reinforces this interpretation. The archived Qwen3 execution produced 2,186 sanitizer-related errors and filtering events, indicating substantial instability in structured output generation. This behavior contrasts sharply with the stronger-performing models, which maintained substantially more stable structured extraction behavior under the same deterministic post-processing pipeline.

Compared with the classifier-only baseline, the LLM-centered extraction strategy substantially improves precision while maintaining competitive recall. Compared with the hybrid strategy, it exhibits substantially stronger recall and F1 in the archived result artifacts analyzed in this chapter. This result suggests that direct structured extraction coupled with deterministic sanitization may be a more effective architectural combination than the two-stage validation logic implemented in the hybrid pipeline within the controlled experimental conditions analyzed in this study.

Category-Level Performance Analysis

Aggregate metrics alone obscure substantial variation across PII categories. To address this limitation, the archived per-label artifacts were inspected to identify representative categories with strong and weak performance patterns. This category-level analysis complements the aggregate comparison by showing that the evaluated architectures differ not only in overall precision and recall, but also in how they handle structured identifiers, context-sensitive values, and categories with weaker lexical anchors.

Table 4.7 highlights that aggregate metrics conceal substantial category-level variation. Categories exhibiting strong lexical structure or highly regular formatting patterns tend to achieve the highest F1-scores, whereas categories requiring contextual interpretation or semantic disambiguation remain substantially more challenging across architectures.

Because the archived dataset contains a large number of PII categories, Table 4.8 presents only representative examples selected to illustrate distinct architectural behaviors.The selected labels include categories with consistently strong performance (`BITCOINADDRESS` and `IPV6`), cate-

Table 4.7: Representative best- and worst-performing categories observed in P1 and P3

Pipeline	Representative best-performing categories	Representative worst-performing categories
P1	SECONDARYADDRESS, MAC, BITCOINADDRESS, IPV6, ZIPCODE	ACCOUNTNAME, COUNTY, PHONEIMEI, NEARBYGPSCOORDINATE, MASKEDNUMBER
P3	USERAGENT, IPV6, VEHICLEVIN, TAXNUM, PHONEIMEI	IP, NEARBYGPSCOORDINATE, PASSWORD, DATE, PHONENUMBER

gories exhibiting substantial differences between architectures (PHONEIMEI), and representative challenging categories (DATE, IP, and PHONENUMBER). The complete category-level results are available in the archived experiment artifacts.

Table 4.8: Representative category-level F1-score patterns for P1 and P3

Label	P1 F1	P3 F1
BITCOINADDRESS	1.0000	0.9487
IPV6	0.9669	0.9565
PHONEIMEI	0.0000	0.9346
DATE	0.4688	0.2188
IP	0.4370	0.0000
PHONENUMBER	0.3277	0.2209

The category-level results show that aggregate scores hide important differences in detection difficulty. Highly structured identifiers, such as BITCOINADDRESS and IPV6, are comparatively easier because they expose stable lexical or formatting patterns. In contrast, categories such as DATE, IP, and PHONENUMBER are more sensitive to context and normalization, resulting in lower and more variable F1-scores.

The contrast between P1 and P3 also reveals architecture-dependent behavior. P1 benefits from broad classifier aggregation and performs well on categories with strong lexical signatures, but it fails on some categories such as PHONEIMEI. P3, by contrast, substantially improves PHONEIMEI, suggesting that structured extraction combined with deterministic post-processing can recover categories that are poorly handled by classifier aggregation.

Pipeline 2 should be interpreted differently. Unlike P1 and P3, the archived P2 artifacts do not expose directly harmonized per-label F1 metrics. Consequently, P2 is incorporated into the category-level analysis through its architectural behavior rather than through quantitative category-level rankings. The archived routing and filtering artifacts indicate that the hybrid pipeline inherits the category coverage limitations of the classifier ensemble while introducing an additional recall-reducing filtering stage.

This limitation originates from the architectural design of the archived hybrid pipeline rather

than from the underlying dataset. While P1 and P3 persist category-level prediction artifacts that can be directly mapped to individual labels during evaluation, the archived P2 implementation was designed primarily to assess the aggregate impact of the LLM-as-a-judge stage on the candidate pool generated by the classifier ensemble. Consequently, category-level performance statistics were not persisted in a form directly comparable to those produced by P1 and P3.

Because the hybrid stage operates over a fixed classifier-generated candidate pool, it cannot recover categories missed upstream. Its conservative filtering behavior further amplifies recall loss, especially for categories whose evidence is ambiguous or context-dependent. Thus, even without directly harmonized per-label F1 artifacts, the archived routing and aggregate results indicate that P2 tends to preserve the limitations of P1 while adding an additional recall-reducing decision layer.

PII category	**P1 F1**	**P3 F1**
`BITCOINADDRESS`	1.0000	0.9487
`IPV6`	0.9669	0.9565
`PHONEIMEI`	0.0000	0.9346
`DATE`	0.4688	0.2188
`IP`	0.4370	0.0000
`PHONENUMBER`	0.3277	0.2209

Figure 4.3: Heatmap-style comparison of representative category-level F1-scores for P1 and P3. Darker blue cells indicate higher F1-score values. P2 is discussed qualitatively based on archived routing and filtering artifacts because directly harmonized per-label F1 metrics are not available under the same reporting regime.

Figure 4.3 provides a visual summary of the category-level variation observed in representative labels. Structured categories such as `BITCOINADDRESS` and `IPV6` remain strong across both pipelines, whereas context-sensitive labels such as `DATE`, `IP`, and `PHONENUMBER` show weaker and more variable behavior. The sharp contrast for `PHONEIMEI` highlights a case in which the LLM-centered extraction pipeline substantially improves over the classifier-only baseline.

For Pipeline 2, the archived artifacts do not expose directly harmonized per-label F1 scores under the same reporting structure used for P1 and P3. Therefore, P2 is included in the category-level interpretation through its observed architectural behavior: the hybrid configuration systematically reduced recall by applying conservative filtering over classifier-generated candidates. This means that categories already weakly covered by the upstream classifier ensemble could not be recovered by the judge stage, while categories requiring contextual interpretation were especially vulnerable to true-positive rejection.

The category-level comparison highlights substantial variation in pipeline behavior across different types of PII. Highly structured categories such as `BITCOINADDRESS` and `IPV6` achieve

strong performance across both P1 and P3, reflecting the presence of clear lexical patterns.

In contrast, context-sensitive categories such as `DATE`, `IP`, and `PHONENUMBER` remain challenging, particularly for the LLM-centered extraction pipeline, where performance degradation suggests sensitivity to contextual interpretation.

An important observation is the dramatic improvement of P3 over P1 in categories such as `PHONEIMEI`, where the classifier baseline fails completely while the LLM-based approach achieves high F1-score.

It is important to emphasize that these results should not be attributed solely to the language models. The observed performance emerges from the interaction between the LLM outputs and the deterministic post-processing pipeline, including schema enforcement, value normalization, and category-level filtering. This combination plays a central role in controlling false positives and ensuring evaluation consistency.

However, performance varies significantly across models. While several configurations achieve strong precision–recall balance, other models exhibit substantial degradation, including cases of near-collapse in recall. This variability indicates that the effectiveness of LLM-centered pipelines is strongly dependent on model capability and adherence to structured output requirements.

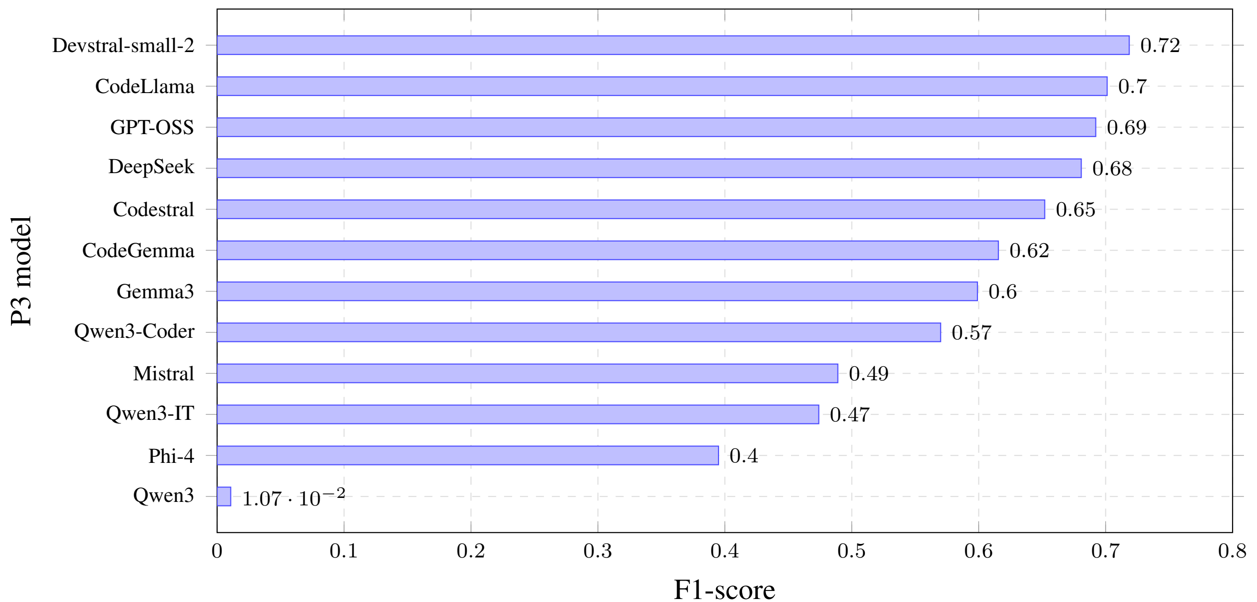

Figure 4.4: F1-score variability across archived P3 LLM-centered extraction configurations.

Figure 4.4 highlights the substantial performance variability across the archived P3 configurations. While several models achieve strong precision–recall balance, others exhibit severe degradation, including near-collapse behavior. This variability reinforces that the effectiveness of LLM-centered extraction depends on model capability, structured-output robustness, and compatibility with deterministic sanitization and normalization stages.

In this sense, the deterministic sanitization layer should be understood as an integral architec-

tural component of the pipeline rather than as a post-processing detail.

4.5 PIPELINE-LEVEL ERROR PATTERNS

The archived artifacts reveal distinct error profiles for the three strategies.

For P1, the dominant pattern is false-positive inflation. This occurs particularly in categories where lexical cues are abundant but semantically ambiguous, such as account-like identifiers, masked values, and name-related spans.

For P2, the dominant pattern is conservative selective filtering. The hybrid stage suppresses both false positives and true positives, resulting in a substantial reduction in recall.

For P3, the dominant pattern is selective extraction robustness under controlled conditions. The best models achieve strong overall precision and recall but still fail on categories that exhibit high contextual variability, such as dates, generic IP-related mentions, and some location-like expressions.

Taken together, these error profiles suggest that the three pipelines fail in qualitatively different ways: P1 tends to over-detect, P2 tends toward conservative selective filtering, and P3 tends to remain selective while missing a narrower subset of categories with higher contextual variability.

These error patterns are directly supported by the archived execution artifacts. For example, candidate routing statistics, intermediate prediction files (e.g., `predicted_before.jsonl` and `predicted_after.jsonl`), and sanitizer logs provide traceable evidence of how errors are introduced, propagated, or filtered across pipeline stages. This artifact-level visibility enables post-hoc verification of the observed behaviors without requiring re-execution of the models.

These error patterns are not incidental but emerge from the architectural design choices of each pipeline. In P1, the union-based aggregation amplifies lexical signals without downstream semantic filtering, leading to over-detection. In P2, early filtering and constrained judge routing limit the system's ability to recover true positives, resulting in systematic recall loss. In P3, structured generation combined with deterministic filtering promotes precision, but remaining errors concentrate on context-sensitive categories that require deeper contextual understanding.

4.6 CROSS-PIPELINE COMPARATIVE ANALYSIS

Figure 4.5 summarizes the dominant empirical behavior observed across the evaluated architectures. The classifier-only baseline emphasizes broad candidate generation and high recall at the cost of false-positive inflation. The hybrid architecture applies additional filtering but introduces substantial recall degradation. In contrast, the LLM-centered extraction pipeline combines

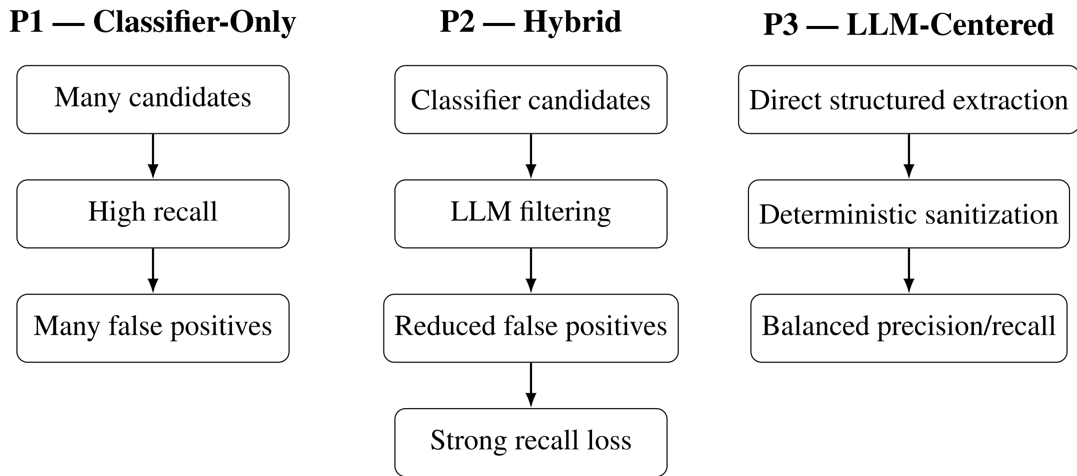

Figure 4.5: Conceptual summary of the dominant empirical behavior observed across the evaluated pipeline architectures.

structured extraction with deterministic sanitization to achieve a more balanced precision–recall profile under the archived evaluation regimes.

From the perspective of foundation model integration strategies, the three pipelines represent distinct architectural paradigms: (i) classifiers without LLM involvement (P1), (ii) LLM-as-a-judge layered on top of classifier candidates (P2), and (iii) LLM-centered structured extraction (P3). The comparison presented in this section should therefore be interpreted as an analysis of integration strategies rather than isolated model performance.

The archived P2 runs exhibit very low variability but remain concentrated around a low F1 range, whereas P3 shows higher model-dependent variability and a substantially higher best observed F1-score.

The archived evidence reveals three distinct architectural and empirical profiles across the evaluated pipelines. First, the classifier-only baseline behaves as a recall-oriented candidate-generation strategy, achieving the highest recall among the evaluated pipelines but with limited precision. Second, the hybrid pipeline consistently increases filtering aggressiveness but suffers a marked recall reduction, leading to lower F1 than the baseline. Third, the LLM-centered extraction pipeline exhibits the strongest observed precision–recall balance among the archived evaluation regimes analyzed in this study, combining high precision with substantially stronger recall than the hybrid setup.

The classifier-only baseline (P1) operates as a recall-oriented candidate-generation architecture. It is particularly suitable for scenarios where sensitivity is prioritized.

The archived hybrid strategy (P2) did not provide clear end-to-end effectiveness improvements over the classifier ensemble baseline under the evaluated experimental conditions. Although precision may increase slightly, the reduction in recall consistently leads to lower F1 values.

The LLM-centered extraction strategy (P3) exhibits the strongest overall balance among the evaluated approaches under the archived reporting conditions, combining high precision with substantially higher recall than the hybrid pipeline.

To synthesize the main trade-offs observed across the evaluated pipelines, Table 4.9 summarizes their key strengths, weaknesses, and operational characteristics.

Table 4.9: Pipeline trade-offs observed in the experimental evaluation

Pipeline	Strength	Weakness	Cost	Complexity
P1 (Classifier-Only)	High recall	Low precision	Low	Low
P2 (Hybrid)	Reduced false positives	Low recall	Medium	High
P3 (LLM-Centered Extraction)	Best F1 balance	LLM computational cost	High	Medium

The comparison highlights that each pipeline exhibits a distinct operational profile. The classifier-only baseline favors recall, making it suitable for candidate generation but prone to false positives. The hybrid approach attempts to reduce false positives but introduces a significant recall penalty due to aggressive filtering.

In contrast, the LLM-centered extraction pipeline provides the strongest observed precision–recall balance under its archived evaluation regime, although at the cost of increased computational requirements and dependency on model capabilities.

Table 4.10 summarizes the dominant behavioral profile observed for each pipeline, highlighting their typical precision–recall trade-offs and their overall operational roles within the evaluated architectures.

Table 4.10: Summary comparison of the evaluated pipelines.

Pipeline	Precision Profile	Recall Profile	Overall Behavior
P1 Classifier Ensemble	Low–Moderate	High	Candidate generator
P2 Hybrid LLM Judge	Moderate	Low	Conservative selective filtering behavior
P3 LLM-Centered Extraction	High	Moderate–High	Balanced extraction pipeline

As shown in Table 4.10, the three pipelines exhibit complementary strengths and weaknesses, reinforcing the interpretation that their architectural trade-offs differ fundamentally in terms of recall orientation, filtering behavior, and extraction robustness. This synthesized view also reinforces the answers to RQ1, RQ2, and RQ3 by showing that the three strategies differ not only in aggregate effectiveness but also in the type of trade-off they impose on downstream review and auditing workflows.

Overall, the results indicate that the effectiveness of foundation models in this domain depends less on their isolated capability and more on how they are integrated into the detection pipeline.

These findings suggest that the design of PII detection systems should prioritize architectural alignment between model capabilities and pipeline structure, rather than relying solely on improvements in individual model performance.

Accordingly, the reported findings should be interpreted as controlled comparative architectural signals derived from the archived experimental artifacts rather than as direct estimates of industrial deployment performance or generalized superiority of any individual model architecture.

4.7 THREATS TO VALIDITY

As with any empirical study, several factors may influence the interpretation of the results presented in this chapter.

Internal validity. The pipelines evolved independently during the research process and therefore intentionally preserve architecture-specific evaluation characteristics. For example, P3 reports value-only metrics after deterministic sanitization, whereas P2 aggregates classifier judgments across candidate streams. These differences introduce non-trivial variations in evaluation regimes across pipelines. In particular, the distinction between value-only metrics (P3) and candidate-level aggregation (P2) reflects structurally different evaluation perspectives, which must be considered when interpreting cross-pipeline comparisons.

Construct validity. The evaluation relies primarily on precision, recall, and F1-score metrics. Although widely used in information extraction research, these metrics do not fully capture operational costs associated with false positives or missed detections.

External validity. The dataset consists of synthetic Java source code fragments with annotated PII instances. Results may vary when applied to other programming languages, repositories, or industrial environments. In addition, the distribution of PII categories present in the dataset may influence the relative performance of different detection strategies.

The synthetic dataset used in this study was intentionally designed as a controlled experimental artifact to support reproducible architectural comparison under shared inputs, deterministic annotation verification, and consistent evaluation criteria. Although this design strengthens internal validity and comparability, it may not fully represent the structural diversity, contextual ambiguity, noisy data representations, and development practices observed in industrial software repositories.

Consequently, the reported results should be interpreted primarily as controlled comparative architectural signals rather than as direct estimates of real-world deployment performance or generalized superiority of any individual pipeline.

Conclusion validity. Although multiple models and configurations were evaluated, additional

experiments and more harmonized cross-pipeline evaluation protocols may refine the conclusions presented here.

4.8 SUMMARY

This chapter analyzed the archived experimental results of three strategies for detecting personal data in Java source code. The classifier-only baseline produced the highest recall but suffered from limited precision. The hybrid strategy consistently reduced recall and did not improve overall effectiveness. The LLM-centered structured extraction pipeline with deterministic post-processing produced the strongest observed precision–recall balance under the archived evaluation regimes analyzed in this chapter.

From the perspective of the research questions, the evidence indicates that classifier ensembles provide an effective high-recall baseline, the current hybrid judge-based architecture does not improve end-to-end detection effectiveness, and LLM-centered structured extraction with deterministic post-processing offers the strongest observed precision–recall balance among the evaluated approaches.

These findings support the central claim of this study that the effectiveness of foundation models in entity-level PII detection in source code depends primarily on how they are integrated into the detection pipeline rather than on isolated model capability evaluated independently of architectural integration and deterministic control layers.

These results reinforce that architectural design choices play a dominant role in determining detection effectiveness, often outweighing isolated model improvements in LLM-based software analysis pipelines.

Although the archived evidence artifacts support traceability, auditability, and independent recomputation of reported metrics, these properties should not be interpreted as guarantees of semantic correctness, regulatory completeness, or real-world deployment validity.

5 DISCUSSION

This chapter interprets the empirical findings presented in Chapter 4 and discusses their implications from an architectural, methodological, and practical perspective. Rather than reiterating numerical results, the analysis focuses on explaining the observed behavior of each pipeline, identifying the structural factors that drive their performance, and reflecting on their implications for privacy-aware software engineering workflows.

5.1 ARCHITECTURAL INTERPRETATION OF THE RESULTS

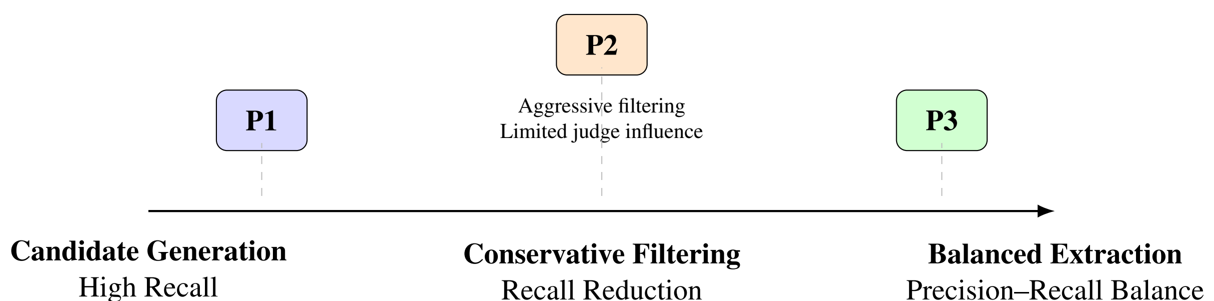

Figure 5.1: Conceptual architectural trade-off spectrum observed across the evaluated pipelines.

Figure 5.1 summarizes the dominant architectural trade-offs observed across the evaluated pipelines. Pipeline 1 emphasizes broad candidate generation and high recall at the cost of false-positive inflation. Pipeline 2 introduces additional filtering mechanisms but exhibits conservative selective filtering behavior and substantial recall reduction. Pipeline 3 adopts a generation-centered extraction strategy coupled with deterministic sanitization, producing the strongest observed precision–recall balance under the archived evaluation regimes analyzed in this study.

The experimental results reveal that the three evaluated pipelines exhibit distinct and stable architectural behaviors. These differences are not merely quantitative but reflect fundamentally different strategies for detecting personal data in source code.

Pipeline 1 (classifier-only baseline) behaves as a recall-oriented candidate-generation mechanism. By aggregating predictions from multiple transformer-based classifiers using an OR strategy, it maximizes coverage of potential PII instances. This explains its high recall and relatively low precision: the architecture prioritizes sensitivity over selectivity and systematically admits a large number of false positives.

Pipeline 2 (hybrid classifier with downstream LLM-based candidate filtering) introduces a second-stage validation mechanism intended to filter false positives through downstream context-

sensitive filtering decisions. However, the archived executions show that this architecture consistently exhibits conservative selective filtering behavior. The downstream LLM-based filtering stage removes not only false positives but also a substantial portion of true positives, leading to a significant reduction in recall and lower overall effectiveness.

Importantly, the results show that this behavior is highly stable across different judge models. Despite evaluating multiple LLMs, the performance variation remains minimal, suggesting that the observed limitation is primarily architectural rather than model-specific. Furthermore, artifact-level evidence indicates that the majority of decisions are determined upstream by candidate generation and prefiltering, with the judge affecting only a limited subset of instances.

Pipeline 3 (LLM-centered structured extraction) adopts a structurally different extraction strategy, directly generating structured predictions from raw source code. This removes the dependency on upstream candidate generation and enables joint processing of lexical patterns and local contextual cues within the evaluated snippets. The results show that this strategy achieves the strongest observed precision–recall balance under the archived evaluation regimes analyzed in this study among the evaluated approaches.

However, unlike Pipeline 2, Pipeline 3 exhibits substantial model-dependent variability. While several models (e.g., Devstral-small-2, CodeLlama, GPT-OSS, and DeepSeek) achieve strong and consistent performance, others show significantly weaker or unstable behavior. These results indicate that, under the LLM-centered extraction regime, model selection becomes a critical factor in determining overall effectiveness.

This stronger observed precision–recall balance should be interpreted with caution. The strong dependence on model selection and configuration indicates that the observed performance is not uniformly guaranteed. In particular, the presence of unstable or near-collapse behaviors in some models highlights the sensitivity of LLM-centered extraction pipelines to prompt design, output constraints, and inference conditions.

These findings suggest that architectural design plays a decisive role in detection performance, but its interaction with model capabilities varies across paradigms.

These findings support a central claim of this study: the effectiveness of foundation models in entity-level PII detection in source code is primarily determined by how they are integrated into the detection pipeline, rather than by isolated model capability evaluated independently of architectural integration and deterministic control layers. Architectural design choices, including candidate generation, decision routing, and post-processing constraints, play a dominant role in shaping system behavior.

These three pipelines can be understood as instances of broader architectural patterns for integrating machine learning and language models: (i) detection-first architectures based on independent classifiers, (ii) downstream filtering-layer architectures that apply downstream LLM-assisted contextual filtering as a post-processing stage, and (iii) generation-centric architectures that rely

on LLM-centered structured extraction coupled with deterministic post-processing layers. This categorization provides a conceptual framework for reasoning about future system designs beyond the specific implementations evaluated in this study.

These interpretations are directly grounded in the artifact-level evidence presented in Chapter 4, including candidate routing behavior, model variability, and post-processing effects observed across pipeline executions.

Table 5.1: Architectural characteristics observed across the evaluated pipelines.

Characteristic	**P1**	**P2**	**P3**
Candidate generation	High	Medium	None
Dependence on upstream filtering	Low	High	None
LLM dependence	None	Moderate	High
Recall orientation	High	Low	Medium–High
Precision orientation	Low	Medium	High
Model variability	Low	Low	High
Deterministic post-processing	Minimal	Moderate	Strong
Sensitivity to output structure	Low	Low	High
Architectural complexity	Low	High	Medium
Dominant empirical behavior	Over-detection	Conservative filtering	Balanced extraction

Table 5.1 synthesizes the dominant architectural properties observed across the evaluated pipelines. The comparison highlights that the three strategies differ not only in aggregate effectiveness but also in their dependency on upstream filtering, sensitivity to model variability, deterministic control mechanisms, and dominant operational behavior. These differences reinforce that architectural integration strategy plays a central role in shaping the behavior of LLM-based privacy detection systems.

5.2 ANALYSIS OF THE RESEARCH QUESTIONS

This section revisits the research questions introduced in Chapter 1 and interprets the empirical findings in light of each question.

5.2.1 RQ1: Effectiveness of transformer-based classifiers

The results demonstrate that transformer-based classifiers are effective in detecting structured and lexically distinctive categories of personal data in Java source code. Categories characterized by strong regular patterns or well-defined formats are consistently detected with high recall.

However, the classifier ensemble exhibits limited precision due to the inherent ambiguity of many categories. Variables, constants, and identifiers in source code often resemble personal data patterns, leading to systematic false positives. This reflects a known limitation of token-level classification models, which rely heavily on local lexical cues and lack robust global contextual processing capabilities.

The empirical results further reinforce that classifier ensembles are particularly sensitive to over-detection in ambiguous categories such as account-like identifiers, generic numeric patterns, and context-dependent labels.

Therefore, transformer-based classifiers are best interpreted as high-recall candidate generators rather than complete solutions for precise PII detection.

5.2.2 RQ2: Effectiveness of hybrid architectures

The hybrid architecture was designed to combine the strengths of classifiers (high recall) and downstream LLM-assisted candidate filtering. However, the empirical results do not support the hypothesis that this combination improves overall detection effectiveness in its current form.

The dominant behavior observed across all archived executions is consistent conservative selective filtering. The downstream LLM-based filtering stage applies a conservative filtering strategy that removes both false positives and true positives, leading to a systematic collapse in recall.

A key empirical observation is the low variability of results across different judge models. This suggests that the performance limitation is not driven by the capabilities of individual LLMs, but rather by the structural design of the pipeline. In particular, the limited exposure of candidates to the judge and the strong influence of upstream filtering mechanisms constrain the potential impact of downstream semantic filtering.

These findings indicate that the hybrid architecture, as implemented in the archived experiments, does not effectively leverage downstream LLM-assisted contextual filtering to improve detection outcomes. Alternative designs—such as confidence-aware filtering, soft decision fusion, or deeper integration between classifier outputs and LLM-driven contextual extraction may be required to realize the intended benefits of hybrid systems.

5.2.3 RQ3: Effectiveness of LLM-centered extraction

The LLM-centered structured extraction pipeline achieves the strongest observed precision–recall balance under the archived evaluation regimes analyzed in this study. This result supports the hypothesis that direct structured extraction can effectively leverage the contextual processing capabilities of LLMs.

Unlike classifier-based approaches, the LLM-centered extraction pipeline is not constrained

by predefined candidate spans. Instead, it performs direct structured extraction over the input prior to deterministic post-processing, allowing joint processing of lexical patterns, structured textual cues, and local contextual information.

However, the results also reveal important variability across models. While some models consistently achieve strong performance, others exhibit degraded recall or unstable behavior. In extreme cases, certain configurations produce near-zero recall, indicating failure to generalize under the given prompt and inference setup.

Additionally, some categories remain challenging across models, particularly those with high contextual ambiguity, such as dates, generic IP mentions, and loosely structured identifiers.

Overall, the findings indicate that LLM-centered extraction is a promising architectural direction for privacy-aware code analysis particularly when combined with deterministic sanitization and normalization layers that mitigate the inherent variability of LLM outputs, but its effectiveness depends strongly on model selection and configuration.

5.3 IMPLICATIONS FOR PRIVACY-AWARE SOFTWARE ENGINEERING

The observed results have direct implications for practical software engineering workflows, especially in contexts involving privacy compliance and code auditing.

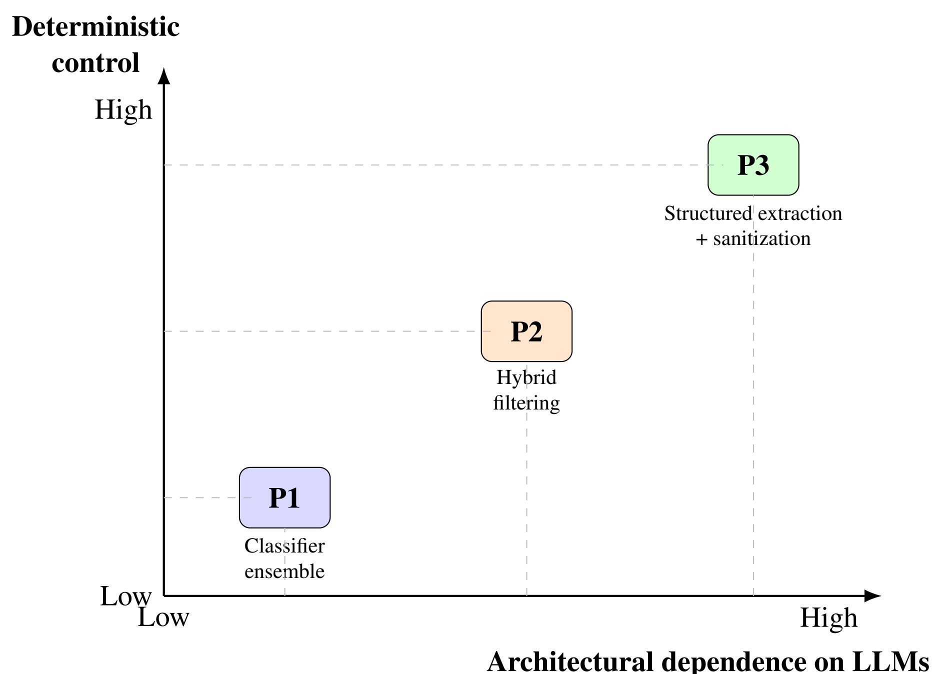

Figure 5.2: Conceptual positioning of the evaluated pipelines according to architectural dependence on LLMs and deterministic control mechanisms.

Figure 5.2 illustrates the conceptual positioning of the evaluated pipelines with respect to architectural dependence on LLMs and deterministic control mechanisms. Pipeline 1 relies minimally on LLM-driven behavior and deterministic post-processing, operating primarily as a classifier ensemble. Pipeline 2 introduces moderate LLM dependence through downstream filtering, but still remains strongly constrained by upstream candidate-generation mechanisms. Pipeline 3 combines high dependence on LLM-centered extraction with strong deterministic sanitization and normalization layers, reflecting a generation-centric architecture coupled with explicit control mechanisms.

Pipeline 1 is well suited for high-sensitivity scenarios, such as initial risk discovery or exploratory analysis, where missing potential PII instances is undesirable. However, its high false-positive rate may significantly increase manual review effort and reduce usability in production environments.

Pipeline 2, in its current form, appears less suitable for operational use. Its tendency to suppress true positives introduces unacceptable risks in compliance-sensitive contexts, where missed detections may have legal or regulatory consequences.

Pipeline 3 offers the most favorable precision–recall balance under the archived evaluation regimes analyzed in this study. Its combination of high precision and competitive recall makes it a promising candidate for controlled privacy-oriented entity-level detection workflows. Its structured output and compatibility with deterministic post-processing pipelines further support its integration into static analysis tools, secure code review processes, and privacy-governance support workflows.

These results suggest that LLM-centered architectures combined with deterministic post-processing represent a promising architectural direction for future privacy-aware code analysis systems, provided that artifact preservation, deterministic post-processing, and post-hoc verification mechanisms are carefully controlled.

These implications suggest that operational privacy-aware code analysis systems may benefit from combining architectural elements, such as using classifier ensembles for candidate discovery and LLM-based extraction for validation or refinement, rather than adopting a single pipeline strategy in isolation.

5.3.1 Operational Trade-offs

The choice between detection strategies involves trade-offs between precision, recall, interpretability, and operational cost.

High-recall approaches reduce the risk of missed detections but increase the burden of manual triage. High-precision approaches reduce noise but may fail to capture edge cases. Hybrid approaches aim to balance these factors but, if not carefully designed, may introduce unintended

behaviors such as excessively conservative filtering.

In practice, the optimal strategy depends on the application context. Risk discovery tasks may favor recall-oriented approaches, while compliance verification and auditing workflows may require more balanced or precision-oriented strategies.

5.3.2 Verifiability and Evidence-Oriented Experimental Design

An important contribution of this study lies in its emphasis on artifact-oriented verifiability and evidence-ready experimental design. The use of structured experiment configurations, archived evidence artifacts, deterministic post-processing traces, and detailed interaction logging enables systematic and transparent comparison across architectures.

This study demonstrates that LLM-based experimental pipelines can achieve a level of auditability and post-hoc inspectability comparable to traditional machine learning workflows when supported by structured artifacts, configuration tracking, and interaction logging.

The findings also highlight the importance of consistent evaluation protocols. Differences in metric definitions and aggregation strategies can significantly affect cross-pipeline comparisons. As such, future work should investigate more harmonized evaluation frameworks for PII detection in source code.

The artifact-driven approach adopted in this study demonstrates that LLM-based experiments can be conducted with a level of structured auditability and methodological rigor comparable to traditional machine learning experimentation.

5.4 LIMITATIONS AND OPEN CHALLENGES

Despite the strengths of the experimental design, several limitations remain.

First, the evaluation is conducted on a synthetic dataset. Although carefully designed, synthetic data may not fully capture the complexity and variability of real-world codebases.

Second, the evaluation metrics focus on precision, recall, and F1-score. While appropriate for comparative analysis, these metrics do not fully capture operational costs, such as the impact of false positives on developer productivity or the risk associated with missed detections.

Third, the hybrid pipeline design represents only one possible realization of classifier-LLM integration. More sophisticated architectures may yield different results.

Fourth, the performance of LLM-based approaches shows sensitivity to model selection and configuration. The variability observed across models indicates that architectural conclusions should be interpreted jointly with model-level considerations.

These limitations highlight several open challenges for future research in privacy-aware code analysis.

In addition, the use of synthetic data may underestimate the level of noise, inconsistency, and contextual ambiguity present in real-world repositories, potentially affecting the external validity of the observed performance differences.

These limitations suggest that future research should focus not only on improving model performance but also on designing robust and adaptive integration strategies that remain stable under realistic conditions.

Accordingly, the findings should be interpreted as controlled comparative architectural signals rather than as direct estimates of industrial deployment performance or generalized superiority of any individual pipeline.

5.5 SUMMARY

This chapter interpreted the experimental results from an architectural and practical perspective. The analysis showed that classifier-based approaches are effective as high-recall candidate generators, hybrid architectures may exhibit structurally conservative selective filtering behavior, and LLM-centered structured extraction pipelines with deterministic post-processing provide the strongest observed precision–recall balance under the archived evaluation regimes analyzed in this study, albeit with sensitivity to model selection.

The findings emphasize the importance of architectural design, model choice, artifact preservation, and post-hoc verifiability in LLM-based software engineering research. These findings reinforce the central role of architectural design in LLM-based software engineering systems and highlight that effective integration strategies are more critical than isolated model performance for achieving auditable and controlled entity-level PII detection.

6 CONCLUSION

This dissertation investigated the effectiveness of different architectural strategies for detecting personally identifiable information (PII) in Java source code, building upon the architectural analysis and empirical findings discussed in Chapters 4 and 5.

The study was motivated by the increasing importance of privacy-aware software engineering and the lack of systematic empirical evidence comparing traditional machine learning approaches, hybrid architectures, and emerging LLM-centric paradigms in this domain.

6.1 SUMMARY OF CONTRIBUTIONS

This work makes several contributions to the field of software engineering and privacy-aware code analysis.

First, it introduces a controlled experimental framework for evaluating PII detection strategies in source code. The framework enables controlled comparison across heterogeneous architectures by standardizing datasets, configuration mechanisms, prompt templates, and evaluation protocols.

Second, the study provides a large-scale empirical evaluation of three distinct architectural paradigms: (i) classifier-only, (ii) hybrid classifier with downstream LLM-based candidate filtering, and (iii) LLM-centered structured extraction with deterministic post-processing. The evaluation is supported by fully archived experiment configurations and results, ensuring auditability, artifact preservation, and post-hoc verifiability.

Third, the results demonstrate that architectural design choices have a decisive impact on detection performance. In particular:

- Classifier-based approaches achieve high recall but suffer from systematic false positives.
- Hybrid architectures, as implemented in this study, exhibit structurally conservative selective filtering behavior that reduces overall effectiveness.
- LLM-centered structured extraction pipelines with deterministic post-processing achieve the strongest observed precision–recall balance under the archived evaluation regimes analyzed in this study, but their performance is sensitive to model selection.

Fourth, this work advances the concept of evidence-ready experimentation in LLM-based software engineering. By combining structured configurations, artifact archiving, deterministic inference settings, and full logging of model interactions, the study demonstrates how to achieve

artifact-oriented verifiability and post-hoc inspection in a domain often characterized by variability and opacity.

Collectively, these contributions advance the understanding of how foundation models can be operationally integrated into software engineering workflows, moving beyond model-centric evaluations toward architecture-aware analysis.

To the best of our knowledge, this study is among the first to systematically compare classifier-based, hybrid, and LLM-centric architectures for PII detection in source code under an artifact-driven experimental framework.

6.2 KEY FINDINGS

The empirical results lead to several key findings.

First, transformer-based classifiers are effective as high-recall candidate generators but are insufficient as standalone solutions for precise PII detection due to their reliance on local lexical features.

Second, the hybrid classifier + LLM architecture does not automatically translate into improved performance. In the evaluated design, the downstream LLM-based filtering stage consistently removes true positives along with false positives, indicating that naïve integration of downstream LLM-assisted candidate filtering may degrade performance rather than improve it.

Third, LLM-centered structured extraction with deterministic post-processing emerges as the most promising architectural direction among the evaluated approaches. Its ability to perform direct structured extraction prior to deterministic post-processing enables more balanced detection behavior, leveraging structured extraction over local contextual cues combined with deterministic post-processing controls.

However, the results also show that LLM-based approaches are not universally robust. Performance varies significantly across models, highlighting the importance of model selection, prompt design, and inference configuration.

Finally, the findings confirm that artifact-oriented verifiability and post-hoc inspection are achievable in LLM-based experiments when proper controls are in place, including deterministic settings, structured logging, and artifact-driven evaluation.

Importantly, artifact-oriented verifiability and post-hoc inspectability should not be interpreted as guarantees of semantic correctness, regulatory completeness, or real-world deployment validity. Instead, they ensure that reported outcomes remain independently inspectable, auditable, and derivable from preserved experimental evidence under the declared protocol.

Taken together, these findings support the central claim of this dissertation: the effectiveness

of foundation models in entity-level PII detection in source code is determined primarily by how they are integrated into the detection pipeline, rather than by isolated model capability evaluated independently of architectural integration and deterministic control layers.

6.3 IMPLICATIONS FOR RESEARCH AND PRACTICE

From a research perspective, this work highlights the need to move beyond model-centric evaluations toward architecture-aware experimentation. The results show that how models are integrated into a pipeline can be more important than which model is used.

The results further indicate that architectural integration affects the degree to which model-specific characteristics influence system behavior. In the hybrid architecture (P2), upstream candidate-generation and filtering mechanisms constrained the impact of individual LLM differences, resulting in low variability across judge models. In contrast, the LLM-centered architecture (P3) exposed a much stronger dependence on model-specific capabilities, leading to substantially higher variability across evaluated models.

Accordingly, the findings reported in this dissertation should be interpreted as controlled comparative architectural signals derived from archived experimental evidence rather than as generalized superiority claims for any specific model or pipeline.

The study also reinforces the importance of artifact preservation and post-hoc verifiability in LLM-based research. The proposed artifact-driven approach provides a blueprint for designing experiments that can be independently verified and extended.

From a practical perspective, the findings provide guidance for designing privacy-aware software engineering tools.

Classifier-based approaches may be useful for high-sensitivity screening, while LLM-centered structured extraction architectures with deterministic post-processing offer a more balanced alternative for automated analysis workflows. Hybrid architectures, although conceptually appealing, require more careful design to avoid unintended degradation in performance.

These insights are particularly relevant for organizations seeking to adopt AI-based tools for compliance, secure code review, and privacy-oriented entity-level detection in software systems.

6.4 FUTURE WORK

This work opens several avenues for future research.

The complementary characteristics observed across the evaluated architectures suggest an ad-

ditional research direction. In particular, the high-recall candidate generation behavior observed in Pipeline 1 and the stronger precision–recall balance achieved by Pipeline 3 indicate that composite architectures may be worthy of investigation. Future work could therefore explore designs that combine classifier-based candidate generation with LLM-centered structured extraction and deterministic post-processing. Such architectures may provide an alternative strategy for balancing detection coverage and precision while mitigating some of the limitations observed in the evaluated pipelines.

First, more advanced hybrid architectures should be explored. Instead of binary filtering, future designs could incorporate confidence-aware fusion, soft scoring mechanisms, or iterative interaction between classifiers and LLMs. More broadly, future research should investigate additional architectural compositions beyond the three paradigms evaluated in this dissertation, including multi-stage pipelines, adaptive routing strategies, ensemble-based LLM orchestration, and architectures that combine deterministic program analysis with foundation-model reasoning.

Second, evaluation on real-world software repositories is necessary to validate the generalizability of the findings. While synthetic datasets provide control, they may not fully capture the diversity and complexity of production software. Future studies should also investigate the applicability of the proposed architectures to additional programming languages and multilingual datasets, as well as to other software-engineering artifacts containing personal data, such as requirements documents, technical documentation, issue-tracking records, and other textual artifacts.

Third, further investigation is needed into the sensitivity of LLM-based pipelines to prompt design, model size, quantization, and inference settings. Understanding these factors is essential for deploying auditable and controlled systems.

Fourth, future work could incorporate additional evaluation dimensions, including computational cost, latency, and human-in-the-loop validation, to better reflect operational constraints.

Finally, the integration of PII detection with broader software engineering toolchains—such as static analysis frameworks, CI/CD pipelines, and developer platforms—represents an important direction for practical impact.

Overall, future research should prioritize the co-design of models and architectures, focusing on integration strategies that balance flexibility, robustness, auditability, and post-hoc verifiability in real-world software engineering environments.

Future work should also investigate alternative ensemble strategies for the classifier-only pipeline, including majority voting, weighted voting, consensus-based aggregation, and confidence-aware candidate selection, since the present study adopted an OR-based aggregation strategy to emphasize recall. In addition, repeated executions and statistical analyses should be conducted to estimate variability, confidence intervals, and robustness across stochastic LLM-based components. Finally, the synthetic dataset generation framework should be further documented,

released, and extended as a configurable research artifact, enabling other researchers to adapt category distributions, templates, and generation parameters for related privacy-aware software engineering studies.

6.5 FINAL REMARKS

The comparative evaluation of classifier-based, hybrid filtering-based, and LLM-centered structured extraction pipelines shows that architectural choices fundamentally shape system behavior, affecting precision, recall, robustness, and usability.

The results indicate that LLM-centric approaches, when combined with deterministic processing and rigorous experimental controls, offer a promising architectural direction for entity-level PII detection in source code under controlled conditions.

At the same time, the study highlights the importance of artifact-oriented verifiability, careful design, and empirical validation in advancing the use of LLMs in software engineering.

This dissertation demonstrates that detecting personal data in source code is fundamentally an architectural problem rather than solely a modeling challenge. The results show that the effectiveness of foundation models depends critically on how they are embedded within structured pipelines that constrain, validate, and refine their outputs. By combining empirical evidence with an evidence-ready experimental framework, this work provides both practical guidance and a conceptual foundation for the design of auditable and controlled PII detection workflows for software engineering environments.

Ultimately, this work contributes to bridging the gap between experimental LLM research and the design of auditable and operational LLM-based software engineering workflows.

REFERENCES

1 FERREYRA, N. E. D.; KHELIFI, S.; ARACHCHILAGE, N. A. G.; SCANDARIATO, R. The good, the bad, and the (un)usable: a rapid literature review on privacy as code. In: *18th IEEE/ACM International Conference on Cooperative and Human Aspects of Software Engineering, CHASE@ICSE 2025, Ottawa, ON, Canada, April 27-28, 2025*. IEEE, 2025. p. 173–178. Disponível em: <https://doi.org/10.1109/CHASE66643.2025.00028>.

2 TANG, F.; ØSTVOLD, B. M. Finding privacy-relevant source code. In: *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024 - Companion, Rovaniemi, Finland, March 12, 2024*. IEEE, 2024. p. 111–118. Disponível em: <https://doi.org/10.1109/SANER-C62648.2024.00020>.

3 The European Parliament and The Council. *General Data Protection Regulation (GDPR): EU Data Protection Rules*. 2018. Disponível em: <https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679>.

4 MACEDO, P. N. Brazilian general data protection law (lgpd). *Nartional Congress, accessed in October 18, 2019*, 2018. Disponível em: <https://www.pnm.adv.br/wp-content/uploads/2018/08/Brazilian-General-Data-Protection-Law.pdf>.

5 KUNZ, I.; KAO, C.; KOWATSCH, D.; HILLER, J.; SCHÜTTE, J.; PROKHORENKOV, D.; BÖTTINGER, K. Using llms to identify personal data processing in source code. In: *2025 IEEE Security and Privacy, SP 2025 - Workshops, San Francisco, CA, USA, May 15, 2025*. IEEE, 2025. p. 137–144. Disponível em: <https://doi.org/10.1109/SPW67851.2025.00018>.

6 WANG, R.; GUO, J.; GAO, C.; FAN, G.; CHONG, C. Y.; XIA, X. Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering. *Proc. ACM Softw. Eng.*, v. 2, n. ISSTA, p. 1955–1977, 2025. Disponível em: <https://doi.org/10.1145/3728963>.

7 ZHENG, L.; CHIANG, W.; SHENG, Y.; ZHUANG, S.; WU, Z.; ZHUANG, Y.; LIN, Z.; LI, Z.; LI, D.; XING, E. P.; ZHANG, H.; GONZALEZ, J. E.; STOICA, I. Judging llm-as-a-judge with mt-bench and chatbot arena. In: OH, A.; NAUMANN, T.; GLOBERSON, A.; SAENKO, K.; HARDT, M.; LEVINE, S. (Ed.). *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*. [s.n.], 2023. Disponível em: <http://papers.nips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-Datasets_and_Benchmarks.html>.

8 SJØBERG, D. I. K.; HANNAY, J. E.; HANSEN, O.; KAMPENES, V. B.; KARAHASANOVIC, A.; LIBORG, N.; REKDAL, A. C. A survey of controlled experiments in software engineering. *IEEE Trans. Software Eng.*, v. 31, n. 9, p. 733–753, 2005. Disponível em: <https://doi.org/10.1109/TSE.2005.97>.

9 KO, A. J.; LATOZA, T. D.; BURNETT, M. M. A practical guide to controlled experiments of software engineering tools with human participants. *Empir. Softw. Eng.*, v. 20, n. 1, p. 110–141, 2015. Disponível em: <https://doi.org/10.1007/s10664-013-9279-3>.

10 GURJAR, A.; MA, X.; CHAORA, A.; KUMAR, V.; MURALIDHARAN, A.; CAMP, L. J. Potential and problems in detecting privacy risks in source code using large language models. In: *Proceedings of the Cyber Supply Chain Risk Management for Critical Systems (CySCRM 24), Richland, WA, United States (2024)*. PNNL, 2024. p. 22–29. Disponível em: <https://www.pnnl.gov/sites/default/files/media/file/Potential%20and%20problems%20in%20LLMs%20Gurjar.pdf>.

11 NADAS, M.; DIOSAN, L.; TOMESCU, A. Synthetic data generation using large language models: Advances in text and code. *IEEE Access*, v. 13, p. 134615–134633, 2025. Disponível em: <https://doi.org/10.1109/ACCESS.2025.3589503>.

12 XU, Z.; LIU, Y.; YIN, Y.; ZHOU, M.; POOVENDRAN, R. Kodcode: A diverse, challenging, and verifiable synthetic dataset for coding. In: CHE, W.; NABENDE, J.; SHUTOVA, E.; PILEHVAR, M. T. (Ed.). *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*. Association for Computational Linguistics, 2025. p. 6980–7008. Disponível em: <https://aclanthology.org/2025.findings-acl.365/>.

13 WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. *Experimentation in Software Engineering*. Springer, 2012. ISBN 978-3-642-29043-5. Disponível em: <https://doi.org/10.1007/978-3-642-29044-2>.

14 HASSANI, S.; SABETZADEH, M.; AMYOT, D.; LIAO, J. Rethinking legal compliance automation: Opportunities with large language models. In: LIEBEL, G.; HADAR, I.; SPOLETINI, P. (Ed.). *32nd IEEE International Requirements Engineering Conference, RE 2024, Reykjavik, Iceland, June 24-28, 2024*. IEEE, 2024. p. 432–440. Disponível em: <https://doi.org/10.1109/RE59067.2024.00051>.

15 WOHLIN, C.; HÖST, M. Special section: Controlled experiments in software engineering. *Inf. Softw. Technol.*, v. 43, n. 15, p. 921–924, 2001. Disponível em: <https://doi.org/10.1016/S0950-5849(01)00200-2>.

16 IWAYA, L. H.; BABAR, M. A.; RASHID, A. Privacy engineering in the wild: Understanding the practitioners' mindset, organizational aspects, and current practices. *IEEE Trans. Software Eng.*, v. 49, n. 9, p. 4324–4348, 2023. Disponível em: <https://doi.org/10.1109/TSE.2023.3290237>.

17 HOU, X.; ZHAO, Y.; LIU, Y.; YANG, Z.; WANG, K.; LI, L.; LUO, X.; LO, D.; GRUNDY, J.; WANG, H. Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, v. 33, n. 8, p. 220:1–220:79, 2024. Disponível em: <https://doi.org/10.1145/3695988>.

18 ROCHA, L. D.; CANEDO, E. D. Optimizing compliance: Comparative study of data laws and privacy frameworks. *Journal of Internet Services and Applications*, v. 16, n. 1, p. 431–452, Jul. 2025. Disponível em: <https://journals-sol.sbc.org.br/index.php/jisa/article/view/5247>.

19 State of California Department of Justice. *California Consumer Privacy Act*. 2018. Disponível em: <https://cppa.ca.gov/regulations/pdf/cppa_act.pdf>.

20 TANG, F.; ØSTVOLD, B. M.; BRUNTINK, M. Identifying personal data processing for code review. In: MORI, P.; LENZINI, G.; FURNELL, S. (Ed.). *Proceedings of the*

9th International Conference on Information Systems Security and Privacy, ICISSP 2023, Lisbon, Portugal, February 22-24, 2023. SciTePress, 2023. p. 568–575. Disponível em: <https://doi.org/10.5220/0011725700003405>.

21 ALMASAH, A. S.; ALYASERI, S. A kubernetes-based ai framework for scalable pii detection and redaction in application logs. In: *2025 IEEE Region 10 Symposium (TENSYMP)*. [S.l.: s.n.], 2025. p. 1–7.

22 GUBER, J.; REINHARTZ-BERGER, I. Privacy-compliant software reuse in early development phases: A systematic literature review. *Inf. Softw. Technol.*, v. 167, p. 107351, 2024. Disponível em: <https://doi.org/10.1016/j.infsof.2023.107351>.

23 SICLARI, M.; LANNIER, S.; VOORDECKERS, O.; TOSZA, S.; ABUALHAIJA, S.; CECI, M.; SANNIER, N.; BIANCULLI, D. Beyond silos: Bridging the gap between law and software engineering – challenges, successes, and lesson drawing. *Internet Policy Review*, v. 14, 11 2025.

24 ZHAO, Y.; YI, G.; LIU, F.; HUI, Z.; ZHAO, J. A framework for scanning privacy information based on static analysis. In: *22nd IEEE International Conference on Software Quality, Reliability and Security, QRS 2022, Guangzhou, China, December 5-9, 2022*. IEEE, 2022. p. 1135–1145. Disponível em: <https://doi.org/10.1109/QRS57517.2022.00116>.

25 KLYMENKO, A.; MEISENBACHER, S.; FAVARO, L.; MATTHES, F. Supporting the integration of privacy-enhancing technologies into the software development life cycle. *SN Comput. Sci.*, v. 6, n. 6, p. 592, 2025. Disponível em: <https://doi.org/10.1007/s42979-025-04127-6>.

26 LI, Z. S.; WERNER, C. M.; ERNST, N. A.; DAMIAN, D. E. GDPR compliance in the context of continuous integration. *CoRR*, abs/2002.06830, 2020. Disponível em: <https://arxiv.org/abs/2002.06830>.

27 PETROLINI, M.; CAGNONI, S.; MORDONINI, M. Automatic detection of sensitive data using transformer- based classifiers. *Future Internet*, v. 14, n. 8, p. 228, 2022. Disponível em: <https://doi.org/10.3390/fi14080228>.

28 YANG, Z.; SUN, Z.; ZHUO, T. Y.; DEVANBU, P. T.; LO, D. Robustness, security, privacy, explainability, efficiency, and usability of large language models for code. *CoRR*, abs/2403.07506, 2024. Disponível em: <https://doi.org/10.48550/arXiv.2403.07506>.

29 NGUYEN, T.-T.; VU, T. T.; VO, H. D.; NGUYEN, S. An empirical study on capability of large language models in understanding code semantics. *Information and Software Technology*, v. 185, p. 107780, 2025. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584925001193>.

30 WOODRING, J.; PEREZ, K.; ALI-GOMBE, A. I. Enhancing privacy policy comprehension through privacify: A user-centric approach using advanced language models. *Comput. Secur.*, v. 145, p. 103997, 2024. Disponível em: <https://doi.org/10.1016/j.cose.2024.103997>.

31 WEN, C.; CAI, Y.; ZHANG, B.; SU, J.; XU, Z.; LIU, D.; QIN, S.; MING, Z.; TIAN, C. Automatically inspecting thousands of static bug warnings with large language model: How far are we? *ACM Trans. Knowl. Discov. Data*, v. 18, n. 7, p. 168, 2024. Disponível em: <https://doi.org/10.1145/3653718>.

32 LAGO, P.; RUNESON, P.; SONG, Q.; VERDECCHIA, R. Threats to validity in software engineering - hypocritical paper section or essential analysis? In: FRANCH, X.; DANEVA, M.; MARTÍNEZ-FERNÁNDEZ, S.; QUARANTA, L. (Ed.). *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2024, Barcelona, Spain, October 24-25, 2024*. ACM, 2024. p. 314–324. Disponível em: <https://doi.org/10.1145/3674805.3686691>.

33 LI, H.; BEZEMER, C.; HASSAN, A. E. Software engineering and foundation models: Insights from industry blogs using a jury of foundation models. In: *47th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2025, Ottawa, ON, Canada, April 27 - May 3, 2025*. IEEE, 2025. p. 307–318. Disponível em: <https://doi.org/10.1109/ICSE-SEIP66354.2025.00033>.

34 AHMED, T.; DEVANBU, P. T.; TREUDE, C.; PRADEL, M. Can llms replace manual annotation of software engineering artifacts? In: *22nd IEEE/ACM International Conference on Mining Software Repositories, MSR@ICSE 2025, Ottawa, ON, Canada, April 28-29, 2025*. IEEE, 2025. p. 526–538. Disponível em: <https://doi.org/10.1109/MSR66628.2025.00086>.

35 SCHMID, K. If you want better empirical research, value your theory: On the importance of strong theories for progress in empirical software engineering research. In: *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021. (EASE '21), p. 359–364. ISBN 9781450390538. Disponível em: <https://doi.org/10.1145/3463274.3463360>.

36 ALLAL, L. B.; LI, R.; KOCETKOV, D.; MOU, C.; AKIKI, C.; FERRANDIS, C. M.; MUENNIGHOFF, N.; MISHRA, M.; GU, A.; DEY, M.; UMAPATHI, L. K.; ANDERSON, C. J.; ZI, Y.; LAMY-POIRIER, J.; SCHOELKOPF, H.; TROSHIN, S.; ABULKHANOV, D.; ROMERO, M.; LAPPERT, M.; TONI, F. D.; RÍO, B. G. del; LIU, Q.; BOSE, S.; BHATTACHARYYA, U.; ZHUO, T. Y.; YU, I.; VILLEGAS, P.; ZOCCA, M.; MANGRULKAR, S.; LANSKY, D.; NGUYEN, H.; CONTRACTOR, D.; VILLA, L.; LI, J.; BAHDANAU, D.; JERNITE, Y.; HUGHES, S.; FRIED, D.; GUHA, A.; VRIES, H. de; WERRA, L. von. Santacoder: don't reach for the stars! *CoRR*, abs/2301.03988, 2023. Disponível em: <https://doi.org/10.48550/arXiv.2301.03988>.

37 Hugging Face. *bigcode/starpii · Hugging Face*. 2023. Disponível em: <https://huggingface.co/bigcode/starpii>.

38 Hugging Face. *ab-ai/pii_model · Hugging Face*. 2024. Disponível em: <https://huggingface.co/ab-ai/pii_model>.

39 Hugging Face. *iiiorg/piiranha-v1-detect-personal-information · Hugging Face*. 2025. Disponível em: <https://huggingface.co/iiiorg/piiranha-v1-detect-personal-information>.

40 Hugging Face. *h2oai/deberta_finetuned_pii · Hugging Face*. 2024. Disponível em: <https://huggingface.co/h2oai/deberta_finetuned_pii>.

41 Hugging Face. *lakshyakh93/deberta_finetuned_pii · Hugging Face*. 2023. Disponível em: <https://huggingface.co/lakshyakh93/deberta_finetuned_pii>.

42 Hugging Face. *yonigo/distilbert-base-multilingual-cased-pii · Hugging Face*. 2024. Disponível em: <https://huggingface.co/yonigo/distilbert-base-multilingual-cased-pii>.

43 YANG, A.; LI, A.; YANG, B.; ZHANG, B.; HUI, B.; ZHENG, B.; YU, B.; GAO, C.; HUANG, C.; LV, C.; ZHENG, C.; LIU, D.; ZHOU, F.; HUANG, F.; HU, F.; GE, H.; WEI, H.; LIN, H.; TANG, J.; YANG, J.; TU, J.; ZHANG, J.; YANG, J.; YANG, J.; ZHOU, J.; LIN, J.; DANG, K.; BAO, K.; YANG, K.; YU, L.; DENG, L.; LI, M.; XUE, M.; LI, M.; ZHANG, P.; WANG, P.; ZHU, Q.; MEN, R.; GAO, R.; LIU, S.; LUO, S.; LI, T.; TANG, T.; YIN, W.; REN, X.; WANG, X.; ZHANG, X.; REN, X.; FAN, Y.; SU, Y.; ZHANG, Y.; ZHANG, Y.; WAN, Y.; LIU, Y.; WANG, Z.; CUI, Z.; ZHANG, Z.; ZHOU, Z.; QIU, Z. Qwen3 technical report. *CoRR*, abs/2505.09388, 2025. Disponível em: <https://doi.org/10.48550/arXiv.2505.09388>.

44 ZHAO, H.; HUI, J.; HOWLAND, J.; NGUYEN, N.; ZUO, S.; HU, A.; CHOQUETTE-CHOO, C. A.; SHEN, J.; KELLEY, J.; BANSAL, K.; VILNIS, L.; WIRTH, M.; MICHEL, P.; CHOY, P.; JOSHI, P.; KUMAR, R.; HASHMI, S.; AGRAWAL, S.; GONG, Z.; FINE, J.; WARKENTIN, T.; HARTMAN, A. J.; NI, B.; KOREVEC, K.; SCHAEFER, K.; HUFFMAN, S. Codegemma: Open code models based on gemma. *CoRR*, abs/2406.11409, 2024. Disponível em: <https://doi.org/10.48550/arXiv.2406.11409>.

45 ROZIÈRE, B.; GEHRING, J.; GLOECKLE, F.; SOOTLA, S.; GAT, I.; TAN, X. E.; ADI, Y.; LIU, J.; REMEZ, T.; RAPIN, J.; KOZHEVNIKOV, A.; EVTIMOV, I.; BITTON, J.; BHATT, M.; CANTON-FERRER, C.; GRATTAFIORI, A.; XIONG, W.; DÉFOSSEZ, A.; COPET, J.; AZHAR, F.; TOUVRON, H.; MARTIN, L.; USUNIER, N.; SCIALOM, T.; SYNNAEVE, G. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023. Disponível em: <https://doi.org/10.48550/arXiv.2308.12950>.

46 Mistral AI. *Codestral | Mistral AI*. 2025. Disponível em: <https://mistral.ai/news/codestral/>.

47 Mistral AI; All Hands AI. Devstral: Fine-tuning language models for coding agent applications. *CoRR*, abs/2509.25193, 2025. Disponível em: <https://doi.org/10.48550/arXiv.2509.25193>.

48 GUO, D.; YANG, D.; ZHANG, H.; SONG, J.; WANG, P.; ZHU, Q.; XU, R.; ZHANG, R.; MA, S.; BI, X.; ZHANG, X.; YU, X.; WU, Y.; WU, Z. F.; GOU, Z.; SHAO, Z.; LI, Z.; GAO, Z.; LIU, A.; XUE, B.; WANG, B.; WU, B.; FENG, B.; LU, C.; ZHAO, C.; DENG, C.; RUAN, C.; DAI, D.; CHEN, D.; JI, D.; LI, E.; LIN, F.; DAI, F.; LUO, F.; HAO, G.; CHEN, G.; LI, G.; ZHANG, H.; XU, H.; DING, H.; GAO, H.; QU, H.; LI, H.; GUO, J.; LI, J.; CHEN, J.; YUAN, J.; TU, J.; QIU, J.; LI, J.; CAI, J. L.; NI, J.; LIANG, J.; CHEN, J.; DONG, K.; HU, K.; YOU, K.; GAO, K.; GUAN, K.; HUANG, K.; YU, K.; WANG, L.; ZHANG, L.; ZHAO, L.; WANG, L.; ZHANG, L.; XU, L.; XIA, L.; ZHANG, M.; ZHANG, M.; TANG, M.; ZHOU, M.; LI, M.; WANG, M.; LI, M.; TIAN, N.; HUANG, P.; ZHANG, P.; WANG, Q.; CHEN, Q.; DU, Q.; GE, R.; ZHANG, R.; PAN, R.; WANG, R.; CHEN, R. J.; JIN, R. L.; CHEN, R.; LU, S.; ZHOU, S.; CHEN, S.; YE, S.; WANG, S.; YU, S.; ZHOU, S.; PAN, S.; LI, S. S.; ZHOU, S.; WU, S.; YUN, T.; PEI, T.; SUN, T.; WANG, T.; ZENG, W.; LIU, W.; LIANG, W.; GAO, W.; YU, W.; ZHANG, W.; XIAO, W. L.; AN, W.; LIU, X.; WANG, X.; CHEN, X.; NIE, X.; CHENG, X.; LIU, X.; XIE, X.; LIU, X.; YANG, X.; LI, X.; SU, X.; LIN, X.; LI, X. Q.; JIN, X.; SHEN, X.; CHEN, X.; SUN, X.; WANG, X.; SONG, X.; ZHOU, X.; WANG, X.; SHAN, X.; LI, Y. K.; WANG, Y. Q.; WEI, Y. X.; ZHANG, Y.; XU, Y.; LI, Y.; ZHAO, Y.; SUN, Y.; WANG, Y.; YU, Y.; ZHANG, Y.; SHI, Y.; XIONG, Y.; HE, Y.; PIAO, Y.; WANG, Y.; TAN, Y.; MA, Y.; LIU, Y.; GUO, Y.; OU, Y.; WANG, Y.; GONG, Y.; ZOU, Y.; HE, Y.; XIONG, Y.; LUO, Y.; YOU, Y.; LIU, Y.; ZHOU, Y.; ZHU, Y. X.; HUANG, Y.; LI, Y.; ZHENG, Y.; ZHU, Y.; MA, Y.; TANG,

Y.; ZHA, Y.; YAN, Y.; REN, Z. Z.; REN, Z.; SHA, Z.; FU, Z.; XU, Z.; XIE, Z.; ZHANG, Z.; HAO, Z.; MA, Z.; YAN, Z.; WU, Z.; GU, Z.; ZHU, Z.; LIU, Z.; LI, Z.; XIE, Z.; SONG, Z.; PAN, Z.; HUANG, Z.; XU, Z.; ZHANG, Z.; ZHANG, Z. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nat.*, v. 645, n. 8081, p. 633–638, 2025. Disponível em: <https://doi.org/10.1038/s41586-025-09422-z>.

49 ABDIN, M. I.; ANEJA, J.; BEHL, H. S.; BUBECK, S.; ELDAN, R.; GUNASEKAR, S.; HARRISON, M.; HEWETT, R. J.; JAVAHERIPI, M.; KAUFFMANN, P.; LEE, J. R.; LEE, Y. T.; LI, Y.; LIU, W.; MENDES, C. C. T.; NGUYEN, A.; PRICE, E.; ROSA, G. de; SAARIKIVI, O.; SALIM, A.; SHAH, S.; WANG, X.; WARD, R.; WU, Y.; YU, D.; ZHANG, C.; ZHANG, Y. Phi-4 technical report. *CoRR*, abs/2412.08905, 2024. Disponível em: <https://doi.org/10.48550/arXiv.2412.08905>.

50 Gemma Team. Gemma 3 technical report. *CoRR*, abs/2503.19786, 2025. Disponível em: <https://doi.org/10.48550/arXiv.2503.19786>.

51 OPENAI. gpt-oss-120b & gpt-oss-20b model card. *CoRR*, abs/2508.10925, 2025. Disponível em: <https://doi.org/10.48550/arXiv.2508.10925>.

52 JIANG, A. Q.; SABLAYROLLES, A.; MENSCH, A.; BAMFORD, C.; CHAPLOT, D. S.; CASAS, D. de L.; BRESSAND, F.; LENGYEL, G.; LAMPLE, G.; SAULNIER, L.; LAVAUD, L. R.; LACHAUX, M.; STOCK, P.; SCAO, T. L.; LAVRIL, T.; WANG, T.; LACROIX, T.; SAYED, W. E. Mistral 7b. *CoRR*, abs/2310.06825, 2023. Disponível em: <https://doi.org/10.48550/arXiv.2310.06825>.

53 SALINAS, D.; SWELAM, O.; HUTTER, F. Tuning LLM judge design decisions for 1/1000 of the cost. In: *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*. OpenReview.net, 2025. Disponível em: <https://openreview.net/forum?id=cve4NOiyVp>.

54 DAO, A.; TERANISHI, H.; MATSUMOTO, Y.; BOUDIN, F.; AIZAWA, A. Overcoming data scarcity in named entity recognition: Synthetic data generation with large language models. In: DEMNER-FUSHMAN, D.; ANANIADOU, S.; MIWA, M.; TSUJII, J. (Ed.). *Proceedings of the 24th Workshop on Biomedical Language Processing*. Viena, Austria: Association for Computational Linguistics, 2025. p. 328–340. ISBN 979-8-89176-275-6. Disponível em: <https://aclanthology.org/2025.bionlp-1.28/>.

55 LIU, Y.; ACHARYA, U. R.; TAN, J. H. Preserving privacy in healthcare: A systematic review of deep learning approaches for synthetic data generation. *Comput. Methods Programs Biomed.*, v. 260, p. 108571, 2025. Disponível em: <https://doi.org/10.1016/j.cmpb.2024.108571>.

56 YANG, Z.; LO, D. Hotfixing Large Language Models for Code. *CoRR*, arXiv, abs/2408.05727, 2024. Disponível em: <https://doi.org/10.48550/arXiv.2408.05727>.

57 MITCHELL, J.; VAIDYAN, V. M.; WANG, Y. On the effectiveness of automatic code generation for synthetic dataset creation. *IEEE Access*, v. 13, p. 142990–142997, 2025. Disponível em: <https://doi.org/10.1109/ACCESS.2025.3587104>.

58 DIETZ, L.; ZENDEL, O.; BAILEY, P.; CLARKE, C. L. A.; COTTERILL, E.; DALTON, J.; HASIBI, F.; SANDERSON, M.; CRASWELL, N. Principles and guidelines for the use of LLM

judges. In: ZAMANI, H.; DIETZ, L.; PIWOWARSKI, B.; BRUCH, S. (Ed.). *Proceedings of the 2025 International ACM SIGIR Conference on Innovative Concepts and Theories in Information Retrieval, ICTIR 2025, Padua, Italy, 18 July 2025*. ACM, 2025. p. 218–229. Disponível em: <https://doi.org/10.1145/3731120.3744588>.

59 D'SOUZA, J.; GIGLOU, H. B.; MÜNCH, Q. Yescieval: Robust llm-as-a-judge for scientific question answering. In: CHE, W.; NABENDE, J.; SHUTOVA, E.; PILEHVAR, M. T. (Ed.). *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*. Association for Computational Linguistics, 2025. p. 13749–13783. Disponível em: <https://aclanthology.org/2025.acl-long.675/>.

60 GERA, A.; BONI, O.; PERLITZ, Y.; BAR-HAIM, R.; EDEN, L.; YEHUDAI, A. Justrank: Benchmarking LLM judges for system ranking. In: CHE, W.; NABENDE, J.; SHUTOVA, E.; PILEHVAR, M. T. (Ed.). *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*. Association for Computational Linguistics, 2025. p. 682–712. Disponível em: <https://aclanthology.org/2025.acl-long.34/>.

61 BAVARESCO, A.; BERNARDI, R.; BERTOLAZZI, L.; ELLIOTT, D.; FERNÁNDEZ, R.; GATT, A.; GHALEB, E.; GIULIANELLI, M.; HANNA, M.; KOLLER, A.; MARTINS, A. F. T.; MONDORF, P.; NEPLENBROEK, V.; PEZZELLE, S.; PLANK, B.; SCHLANGEN, D.; SUGLIA, A.; SURIKUCHI, A. K.; TAKMAZ, E.; TESTONI, A. Llms instead of human judges? A large scale empirical study across 20 NLP evaluation tasks. In: CHE, W.; NABENDE, J.; SHUTOVA, E.; PILEHVAR, M. T. (Ed.). *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*. Association for Computational Linguistics, 2025. p. 238–255. Disponível em: <https://doi.org/10.18653/v1/2025.acl-short.20>.

62 MEISENBACHER, S.; KLYMENKO, A.; MATTHES, F. Llm-as-a-judge for privacy evaluation? exploring the alignment of human and llm perceptions of privacy in textual data. In: . New York, NY, USA: Association for Computing Machinery, 2025. (HAIPS '25), p. 126–138. ISBN 9798400719059. Disponível em: <https://doi.org/10.1145/3733816.3760760>.

63 FARAGLIA, D.; Other Contributors. *Faker*. 2025. Disponível em: <https://github.com/joke2k/faker>.

Appendix

I. SYNTHETIC DATASET GENERATION FRAMEWORK

This appendix documents the Python-based framework used to generate the synthetic Java code dataset with annotated PII entities. The generator is implemented as a small CLI application that orchestrates (i) Java snippet generation, (ii) personal-data value synthesis and injection, and (iii) JSONL serialization of ground-truth records for downstream evaluation.

I.1 DESIGN GOALS AND SCOPE

The dataset generator was designed with the following goals.

First, it aims to produce a large number of small, self-contained Java code fragments that resemble plausible code-like development artifacts, including common constructs such as variable declarations, constants, method calls, object instantiation, and logging statements.

Second, it supports controlled inclusion of positive and negative samples. In this context, a *positive sample* is a snippet that contains at least one annotated personal-data fragment, while a *negative sample* contains no annotated fragments. The CLI exposes a parameter controlling the target fraction of snippets that should contain at least one annotated fragment, enabling balanced or skewed datasets depending on experimental needs.

Third, it produces an auditable ground-truth file suitable for automated evaluation. For each snippet, the generator records the snippet text and a structured list of fragments including (i) the label/category, (ii) the verbatim inserted value, and (iii) character offsets locating the value within the snippet text.

Finally, the generator was engineered for reproducibility. Each execution emits run metadata capturing the effective CLI parameters and the model identifier used for snippet generation (when enabled). When the LLM backend is active, the execution also records interaction traces (prompts and raw model outputs), enabling post-hoc inspection. Together, these outputs constitute reproducibility artifacts that support auditability of both the dataset snapshot and the conditions under which it was generated.

The scope of the generator is limited to producing synthetic data for controlled experimentation. It does not represent real production codebases or real individuals, and it does not claim ecological validity beyond providing plausible code-like contexts under controlled conditions.

Operational definition of "PII" as GDPR-scoped personal data. Importantly, the generator's notion of "PII" is operationalized as a *GDPR-scoped personal data taxonomy* rather than an open-ended set of contextual attributes. Concretely, the framework implements an explicit allowlist of supported categories treated as *personal data* under the GDPR definition (Art. 4(1)) and its interpretative recitals on direct or indirect identifiability (Recitals 26–30). (3)

Inclusion criterion (personal data). A category is included when its values allow the direct or indirect identification of a natural person, including identifiers and factors such as online identifiers, location data, and official/financial/credit identifiers. (3) Examples represented in the implemented taxonomy include: names and contact identifiers (e.g., `FIRSTNAME`, `LASTNAME`, `EMAIL`, `PHONENUMBER`); online and device identifiers (e.g., `IPV4`/`IPV6`, `MAC`, `USERAGENT`); location-related attributes (e.g., `STREET`, `CITY`, `ZIPCODE`, `NEARBYGPSCOORDINATE`); and official/financial/credential-like identifiers (e.g., bank-account-like identifiers, card numbers, card verification values, vehicle identifiers). (3)

Exclusion criterion (not personal data in isolation). Conversely, generic or contextual attributes, including attributes referring to *non-natural entities* (e.g., company-only descriptors), are excluded when they do not establish a direct or indirect link to a natural person *by themselves*. (3) Under GDPR, such attributes would only become personal data if combined with other identifiers or identity factors (e.g., "age + name + city"), but in isolation they are treated as non-personal data for the purposes of this dataset snapshot. (3)

Operational decision procedure (include vs. exclude). To operationalize the inclusion and exclusion criteria above in a consistent and auditable way, the generator applies an explicit operational decision procedure for each candidate category, aligned with GDPR Art. 4(1) and Recitals 26–30. (3)

The procedure answers two questions. First, it checks whether a candidate attribute relates to an identified or identifiable natural person under GDPR Art. 4(1). (3) Second, if the answer is positive, it checks whether the attribute acts as a direct or indirect identifier or identity factor (including online identifiers and location-related factors) as clarified by Recitals 26–30. (3)

If an attribute does not enable identification *in isolation*, it is excluded from the taxonomy snapshot and would only become personal data when combined with other identifiers or identity factors (e.g., "age + name + city"). (3) In the implementation, this decision procedure is instantiated as a GDPR-scoped allowlist used during value synthesis and annotation, and the concrete allowlist snapshot is recorded as part of the reproducibility artifacts. (3)

Figure I.1 summarizes this include/exclude decision logic.

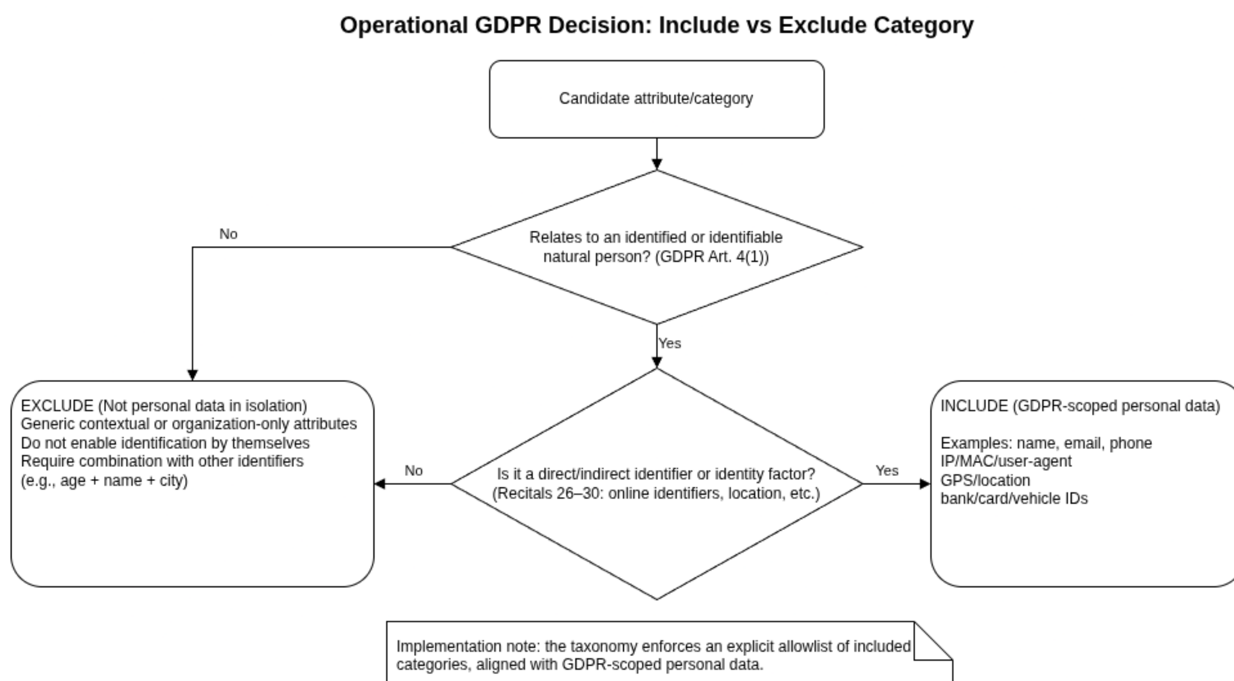

Figure I.1: Operational decision procedure for including or excluding candidate categories in the GDPR-scoped personal data taxonomy. (3)

I.2 HIGH-LEVEL ARCHITECTURE

At a high level, the generator follows a layered architecture that separates run orchestration concerns from domain logic and external dependencies.

The application layer orchestrates execution and records run-level metadata. Domain services implement snippet generation, value synthesis, and annotation logic. Infrastructure components provide external capabilities, such as fake-data providers, local open-weight LLM adapters, logging, and persistent storage of reproducibility artifacts.

Figure I.2 summarizes the main layers, responsibilities, and interactions among the framework components.

Figure I.3 complements this structural view by presenting the pipeline-oriented perspective of the same framework.

In this view, solid arrows represent execution and data flow across components, while dashed arrows indicate GDPR-scoped constraints on category selection and annotation. Each execution produces auditable reproducibility artifacts, including the JSONL dataset, run configuration metadata, summary statistics, and optional LLM interaction logs.

The generator is structured as a small application composed of cohesive components with explicit responsibilities.

- **CLI entrypoint**: parses run parameters (e.g., `total`, `pii-rate`, `domain`, `locale`,

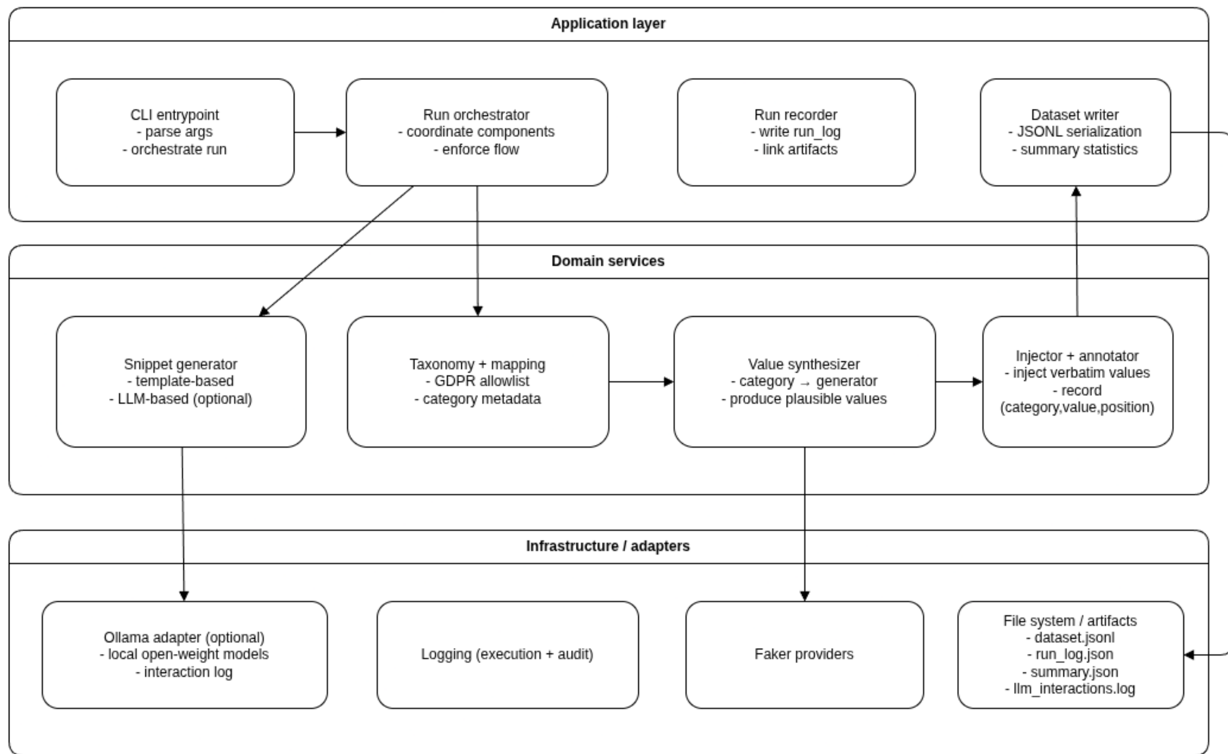

Figure I.2: High-level layered architecture of the synthetic dataset generation framework.

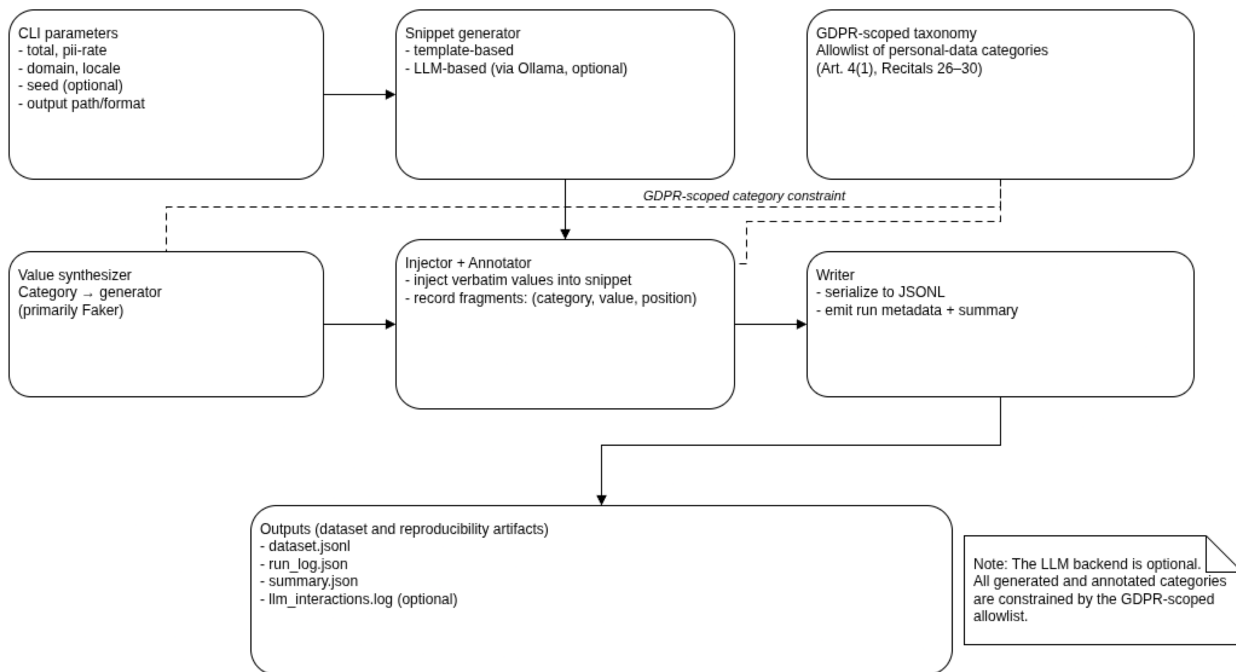

Figure I.3: High-level pipeline and dataflow of the synthetic dataset generation framework.

output path and format) and orchestrates execution.

- **Snippet generator**: produces Java snippet skeletons using either (i) template-based construction or (ii) an optional LLM backend.
- **Category taxonomy and mapping**: enforces a GDPR-scoped allowlist of categories and provides auxiliary metadata used in snippet construction (e.g., Java types and attribute names).
- **Value synthesizer**: maps each allowed category to a concrete value generator (primarily via Faker) to produce syntactically plausible values for insertion. (63)
- **Injector and annotator**: injects synthesized values into snippet text and records verbatim values and spans.
- **Writer and run recorder**: serializes records to JSONL and emits run-level metadata and summaries as reproducibility artifacts.
- **Ollama adapter (optional)**: issues local requests to open-weight models and persists interaction traces (prompts and raw responses) when LLM generation is enabled.

For clarity on how annotations are represented in the exported dataset, Figure I.4 summarizes the ground-truth JSONL record schema and the integrity invariants that enable exact span verification and downstream evaluation.

I.3 INPUT CONFIGURATION AND PARAMETERS

The framework provides a command-line interface (CLI) for dataset generation, exposed through a dedicated entrypoint. At a high level, a run is parameterized by:

- **Dataset size**: total number of snippets to generate (`total`).
- **Target positive rate**: fraction of snippets that must contain at least one annotated fragment (`pii-rate`).
- **Domain/mode**: a mode controlling the mixture of snippet templates (`domain`).
- **Locale**: locale for fake-data generation (`locale`).
- **Output directory and output format**: location for generated reproducibility artifacts and serialization format (JSONL as the primary format).
- **Optional seed**: a seed controlling pseudo-random decisions when explicitly provided.

Example CLI invocation. A representative invocation uses the CLI to request a JSONL dataset snapshot with a given `total`:

```
poetry run <cli-entrypoint> generate --total 1000 --format jsonl
```

The complete set of parameters used for a dataset snapshot is recorded in the corresponding run log.

LLM-based snippet generation backend. The generator supports LLM-based snippet generation through an Ollama-backed adapter, using only open-weight models hosted locally. When enabled, the run log records the selected model identifier and the location of the interaction log. Default decoding/runtime values are defined by the adapter implementation and are applied when not overridden via CLI parameters.

I.4 LABEL TAXONOMY AND ANNOTATION STRATEGY

The generator relies on a label taxonomy where each personal-data fragment is assigned a category label (e.g., `EMAIL`, `FIRSTNAME`, `IPV4`, `DATEOFBIRTH`).

GDPR-scoped category allowlist and mapping metadata. The framework implements an explicit mapping from uppercase category keys to auxiliary metadata used in snippet construction, including (i) Java field types, (ii) human-readable labels, and (iii) Java-style camelCase attribute names. The supported categories correspond to attributes treated as personal data under the GDPR definition of identifiability (Art. 4(1); Recitals 26–30). (3) Categories that do not qualify as personal data *in isolation* are intentionally excluded from generation and annotation, so that the dataset snapshot remains aligned with the GDPR-scoped taxonomy used throughout the experiments. (3)

Value synthesis. For supported categories, the generator maps each label to a concrete value generator, primarily via the Faker library. (63) This ensures that inserted values are syntactically plausible for each category (e.g., emails, names, addresses, identifiers, network addresses).

Annotation as verbatim insertion with span tracking. Annotation is defined as *value insertion with exact span tracking*. For each snippet, the generator selects one or more categories from the GDPR-scoped allowlist, produces one or more concrete values, and inserts each value into the snippet text such that it appears verbatim in the final Java fragment. After insertion, the framework records the character span locating the inserted value within the final snippet text.

The ground-truth schema encodes offsets as a `position` field containing a pair of integers `[start, end)` representing a half-open interval over the snippet `text`. This representation is directly verifiable by substring extraction and supports consistent downstream matching.

Integrity invariants. Each generated fragment must satisfy the following invariants:

- **Verbatim span correctness:** for every fragment, `text[position[0]:position[1]]` equals `value`.
- **Span validity:** `0` $\leq$ `position[0]` $\leq$ `position[1]` $\leq$ `len(text)`.
- **Category validity:** `category` belongs to the GDPR-scoped allowlist used by the generator.

These invariants are critical for auditability and for downstream evaluation based on exact matching of `(category, value)` pairs.

I.5 SYNTHETIC CODE GENERATION PROCESS

The dataset generation pipeline can be summarized as follows.

Step 1: Determine target counts for positive/negative snippets

Given the requested total number of snippets and the target positive rate, the generator computes the number of snippets that must contain at least one annotated fragment and the number of snippets that must contain none. The run then proceeds to generate each group accordingly.

Step 2: Generate snippet skeletons (Java code context)

For each snippet, the framework constructs a Java code fragment that provides a plausible context where personal data may naturally appear. When the LLM backend is enabled, the generator issues requests to a locally served open-weight model via the Ollama adapter, and preserves prompts and raw outputs as reproducibility artifacts for auditability.

The framework applies cleaning and normalization steps to the raw generated snippet text to remove formatting artifacts (e.g., code fences) and to enforce a clean Java-only output. This ensures that offsets are computed over the final snippet string used in the JSONL record.

Step 3: Generate personal-data values and inject into the snippet

For snippets that must contain personal data, the generator selects one or more categories from the GDPR-scoped allowlist, samples concrete values for each category via the fake-data provider, and injects these values into appropriate positions in the snippet text. The insertion strategy ensures that injected values remain contiguous spans and appear verbatim, enabling exact matching between the `value` field and the corresponding substring in `text`.

For snippets that must contain no personal data, no injection occurs and the annotation list remains empty.

Step 4: Create snippet records and compute stable identifiers

Each generated snippet is represented as a structured record containing its `text`, metadata fields, and a list of fragments. Each record carries a stable identifier `id` computed deterministically as the SHA-256 hash of the snippet `text` (hex-encoded). This deterministic identifier supports stable joins between dataset records and downstream outputs (e.g., detector predictions), even when files are reordered.

Step 5: Serialize dataset and write reproducibility artifacts

Finally, the writer serializes all records to a JSONL file. Each execution also emits a machine-readable run log capturing the effective parameters (dataset size, target positive rate, locale, domain, format, output directory) and the model identifier used by the Ollama adapter. In addition, the run directory includes an interaction log preserving per-call prompts and raw model responses to support auditability and reproducibility.

I.6 EXECUTION ARTIFACTS AND DIRECTORY LAYOUT

Each execution produces a self-contained run directory that groups reproducibility artifacts. A typical layout is as follows:

```
<run_dir>/
  dataset.jsonl
  run_log.json
  summary.json
  llm_interactions.log (present only when LLM backend is enabled)
  gpu_stats.csv (optional GPU telemetry artifact)
```

Run log. The `run_log.json` records the effective CLI parameters (including defaults applied by the application), timestamps, and the resolved output paths. When LLM generation is enabled, it also records the selected model identifier and references to the interaction log.

Interaction log. When LLM generation is enabled, `llm_interactions.log` stores one JSON object per model call (NDJSON), preserving the raw prompt and raw response. This artifact supports auditability of prompt adherence and diagnosis of malformed generations.

Summary. The `summary.json` contains descriptive statistics for the produced snapshot, including total snippets, number of positive/negative snippets, total annotated fragments, and per-category fragment counts. These statistics are used for consistency checks (e.g., total fragments equals the sum of per-category counts) and for descriptive reporting in the dissertation appendices.

I.7 GROUND-TRUTH OUTPUT FORMAT

The primary dataset artifact is a JSONL file where each line is a standalone JSON object representing one snippet. Each record follows a consistent schema:

- **`id`**: a deterministic identifier computed as the SHA-256 hash of `text` (hex-encoded).
- **`language`**: the programming language identifier (e.g., `java`).
- **`type`**: an implementation-defined field indicating the snippet origin/mode (e.g., LLM-generated), not the presence/absence of personal data.
- **`text`**: the full Java code fragment as a single string.
- **`fragments`**: a list of zero or more fragments. Each fragment contains:
  - **`category`**: the personal-data label/category (from the GDPR-scoped taxonomy).
  - **`value`**: the verbatim inserted value.
  - **`position`**: a pair `[start, end)` identifying the character span of `value` within `text`.

Negative samples are represented by `fragments: []`. This representation allows downstream evaluation to treat absence of personal data as a first-class case without special casing, while preserving auditability through explicit empty annotations.

Figure I.4 summarizes the ground-truth record schema and the integrity invariants that enable exact span verification and reproducible, automated evaluation.

Ground-truth schema and integrity invariants. Each record stores the Java snippet text and a list of zero or more annotated fragments. Fragments capture the GDPR-scoped category key, the verbatim inserted value, and its character span as a half-open interval `[start,end)`. The integrity invariants ensure verbatim span correctness, span validity, and category validity, supporting auditability and reproducible downstream evaluation.

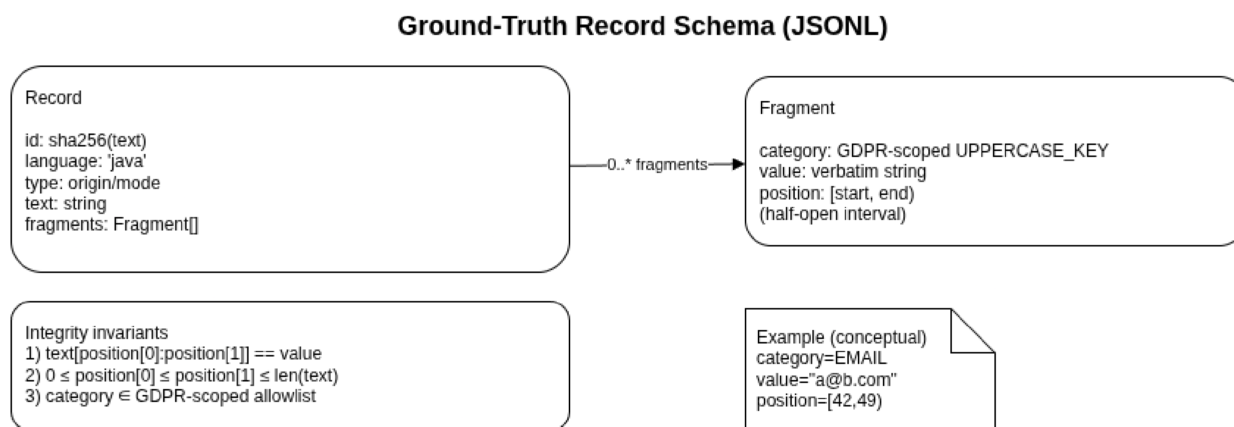

Figure I.4: Ground-truth JSONL schema and integrity invariants used by the synthetic dataset generator.

I.8 REPRODUCIBILITY AND VERSIONING

The generator supports reproducibility through explicit control of configuration and by recording execution artifacts.

Deterministic identifiers and stable joins

Each snippet record contains an `id` computed deterministically from the snippet `text`. This enables stable joins across artifacts produced in later stages of the experimental pipeline, independent of record ordering.

Recorded run configuration and defaults

Each execution directory includes a machine-readable run log that records the dataset parameters (`total`, `pii-rate`, `locale`, `domain`, `format`, output directory), as well as timing information and the selected model identifier when the LLM backend is enabled. The run log records the *effective configuration*, including default values applied when parameters are omitted.

LLM interaction records

When the LLM backend is enabled, the run artifacts include an interaction log that preserves prompts and raw model outputs per generation call. This log supports post-hoc diagnosis of malformed generations, prompt adherence issues, and reproducibility audits.

Archived Reproducibility Artifacts

Archived experimental packages retained: (i) the JSONL dataset file, (ii) the run log capturing effective CLI parameters and model identifier, (iii) a dependency snapshot (e.g., from project metadata), and (iv) the interaction log when applicable. Together, these reproducibility artifacts enable independent verification of both the dataset schema and the conditions under which the dataset was generated.

II. PII DETECTION FRAMEWORK

This appendix documents the Python-based framework used to detect PII in Java code fragments in the experimental pipelines evaluated in this dissertation. The framework was designed with explicit goals of modularity, deterministic post-processing, strict structured outputs, and artifact traceability, enabling reproducible experimentation across heterogeneous detection backends.

The implementation is organized as a package-based application (`src/`) with a command-line interface, pipeline runners, shared services (normalization, evaluation, reporting), infrastructure adapters (Hugging Face and Ollama), and configuration resources (YAML) that fully define an experimental execution. This appendix focuses on the concrete architecture and the operational flow of the framework, complementing the high-level experimental description in Chapter 3 and the reproducibility artifact documentation in Appendix III.

II.1 ARCHITECTURAL OVERVIEW

The framework follows a layered architecture separating orchestration, domain models, infrastructure adapters, and shared services. At a practical level, the codebase is organized into the following main packages:

- **CLI orchestration** (`src/cli/`): Typer-based commands for dataset validation, experiment execution (P1/P2/P3), and result inspection and comparison.
- **Experiment services** (`src/services/`): pipeline runners and a shared execution skeleton, including dataset utilities, evaluation services, metrics computation, and report generation.
- **Infrastructure integrations** (`src/infrastructure/`): Hugging Face classifier wrappers, LLM adapters (Ollama), JSONL loaders, and Java parsing utilities.
- **Shared configuration and utilities** (`src/shared/`): YAML-driven label mapping/equivalences, GDPR filtering, logging configuration, and cross-cutting utilities.
- **Resources** (`src/resources/`): strict JSON schemas and Jinja2 prompt templates used by LLM-driven pipelines.

Design principles. Two design principles are enforced across the codebase. First, the three pipelines share the same execution skeleton (dataset loading, normalization, evaluation, artifact

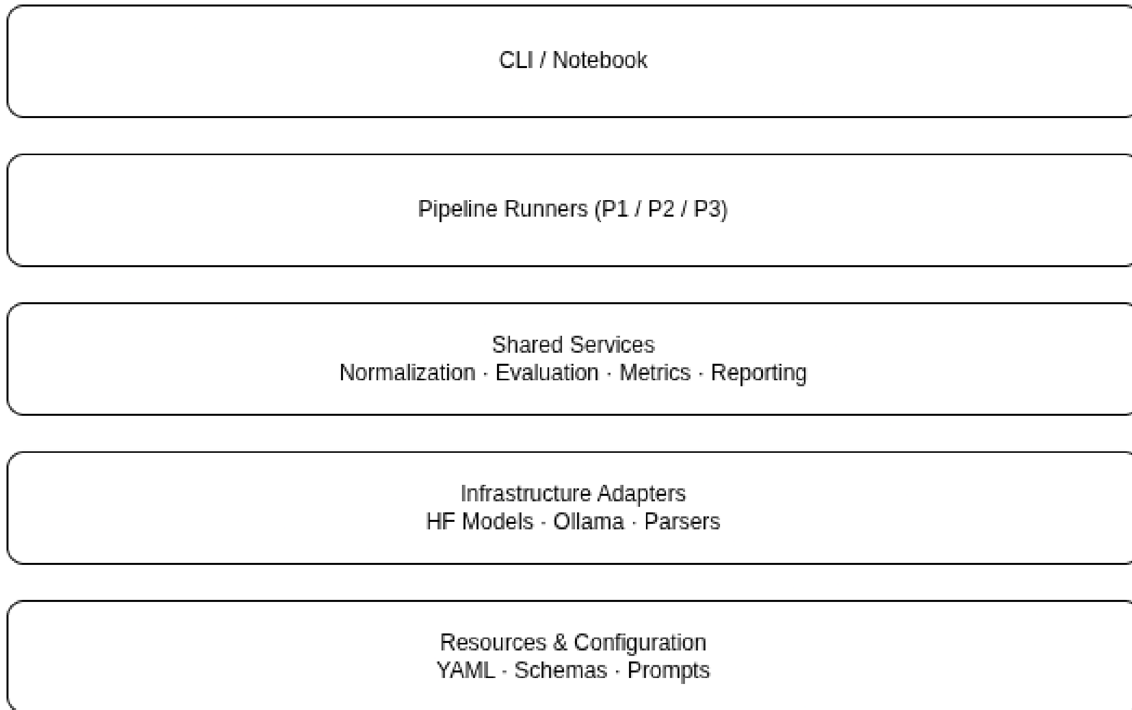

Figure II.1: Layered architecture of the PII detection framework, highlighting execution entrypoints (CLI/notebook), pipeline runners (P1/P2/P3), shared services (normalization, evaluation, metrics, reporting), infrastructure adapters (HF/Ollama/parsers), and configuration resources (YAML/schemas/prompts).

Table II.1: Repository map.

Package/Path	**Responsibility**	**Used by**
`src/cli/`	CLI entrypoints and commands (dataset validation, experiment execution, inspection, comparison).	P1–P3
`src/services/`	Pipeline runners and shared execution skeleton (loading, normalization, evaluation, metrics, reports).	P1–P3
`src/infrastructure/`	Detection backends and IO adapters (HF token classifiers, Ollama adapter/client, JSONL loaders, parsers).	P1–P3
`src/shared/`	Cross-cutting utilities and configuration-driven behavior (label mapping/equivalences, GDPR filters, logging).	P1–P3
`src/resources/`	Strict JSON schemas and Jinja2 prompt templates for LLM-driven pipelines.	P2–P3

persistence), so differences among pipelines arise from detection strategy rather than auxiliary logic. Second, any LLM interaction is constrained by strict structured-output contracts, validated via schema and deterministic post-processing, and logged verbatim for auditability.

II.2 DETECTION PIPELINE

Each run is driven by a pipeline runner selected by configuration and executed under a standardized lifecycle. This lifecycle is implemented consistently across P1, P2, and P3, ensuring that comparisons are attributable to detection strategy rather than to inconsistent execution procedures.

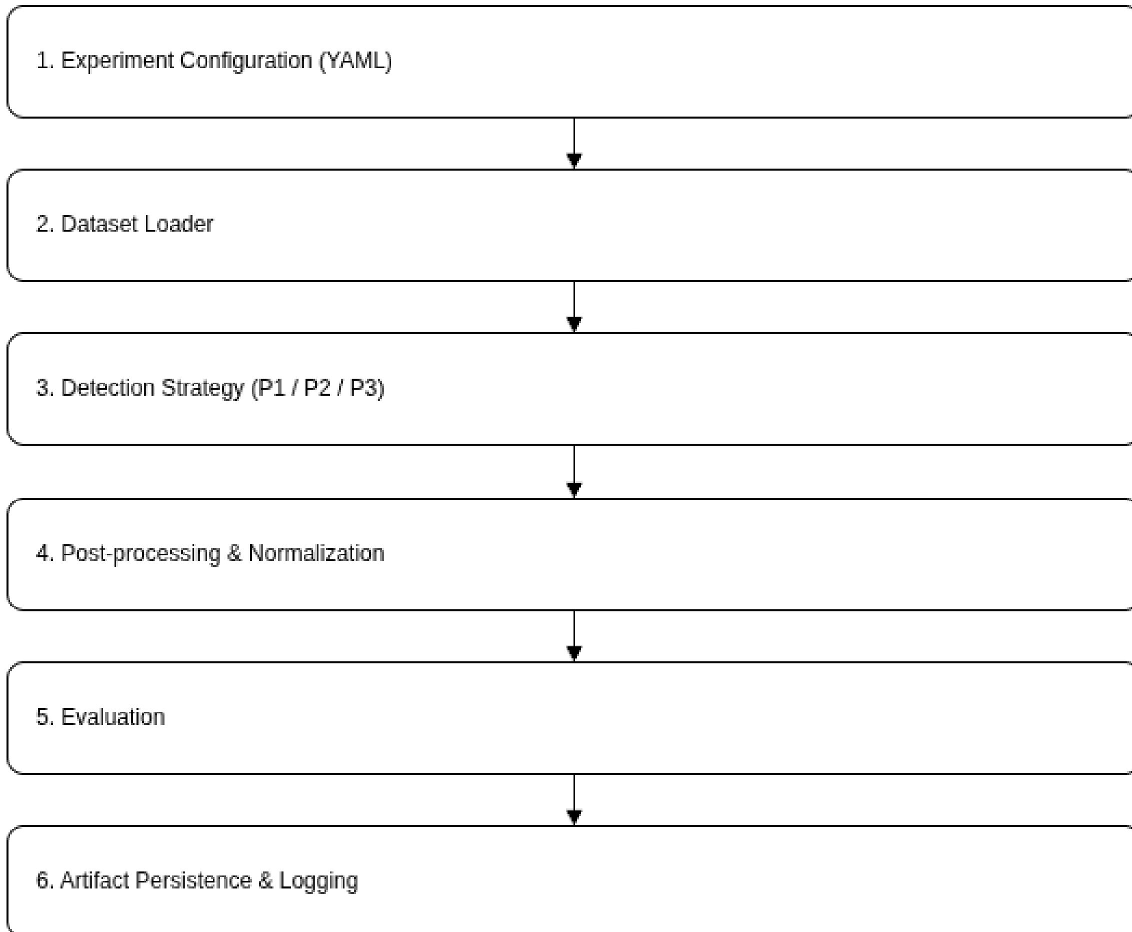

Figure II.2: Standardized execution lifecycle across detection pipelines. Runs are configured via YAML experiment files, load JSONL datasets, execute a pipeline-specific detection strategy (P1/P2/P3), apply deterministic post-processing and normalization, compute evaluation metrics, and persist artifacts and logs for traceability and reproducibility.

To clarify the conceptual differences among the three detection strategies, Figure II.3 contrasts the functional role of each component in the pipeline flow. Pipeline 1 relies exclusively on transformer-based classifiers to directly emit detections. Pipeline 2 reuses classifier outputs as a candidate set and introduces an LLM judgment stage to filter candidates and reduce false positives. Pipeline 3 delegates detection to the LLM and then applies deterministic sanitization and normalization to enforce a strict structured output suitable for evaluation.

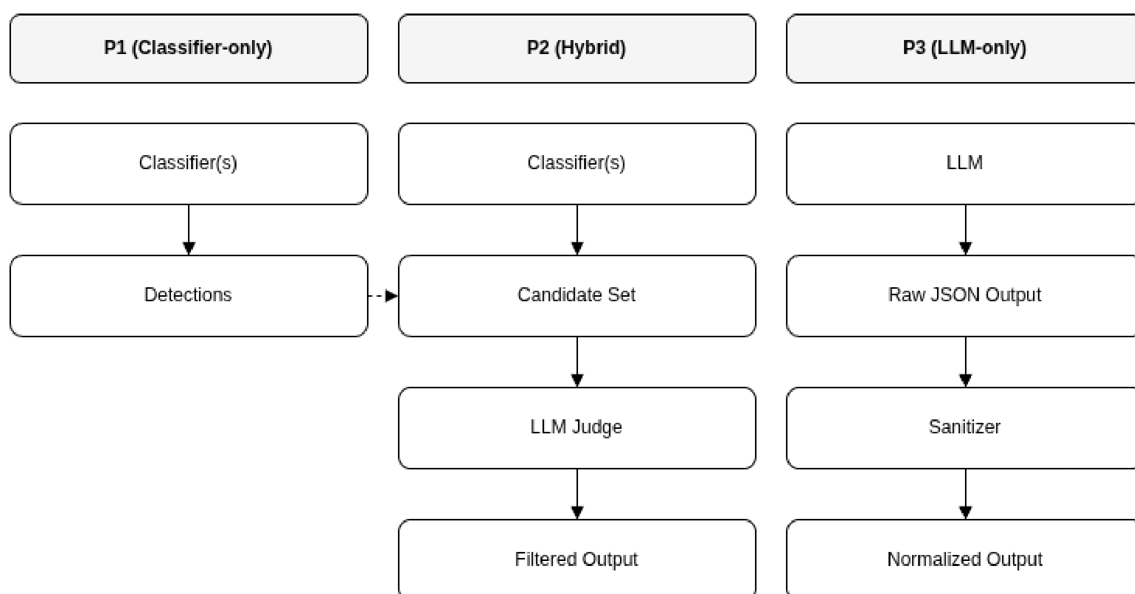

Figure II.3: Conceptual comparison of the three detection pipelines. Pipeline 1 (classifier-only) produces detections directly from transformer classifiers. Pipeline 2 (hybrid) constructs a candidate set from classifier outputs and applies an LLM-as-a-judge stage to filter candidates, yielding a filtered output. Pipeline 3 (LLM-only) performs end-to-end structured extraction, producing a raw JSON output that is sanitized and normalized into a final structured output.

Table II.2: Pipeline features.

Capability	**P1**	**P2**	**P3**
Primary detection backend	HF token classifiers (ensemble)	Classifier candidates + LLM judge	LLM structured extraction
Uses an LLM	No	Yes (judge)	Yes (detector)
Structured JSON contract enforced	No (token outputs)	Yes (judge decision JSON)	Yes (detection JSON)
JSON schema validation	No	Yes	Yes
Deterministic sanitization of LLM output	N/A	Yes	Yes
Deterministic label/value normalization	Yes	Yes	Yes
Optional GDPR-filtered outputs	Yes	Yes	Yes
Primary optimization goal	High recall baseline	Reduce false positives	End-to-end extraction

II.2.1 Stage 1: Configuration resolution and execution context

Runs are initiated either via the CLI (`src/cli/app.py` and the `src/cli/commands/experiment_evaluate_p*_*.py` entrypoints) or via a notebook runner (distributed as a separate artifact and documented in Appendix III). In both cases, execution is configuration-driven: YAML experiment files are parsed and resolved into a concrete runtime configuration. The resolved configuration is persisted alongside results to ensure that defaults, derived parameters, and runtime paths are fully traceable.

II.2.2 Stage 2: Dataset and JSONL loading

The framework loads dataset records from JSONL using dedicated loaders (e.g., JSONL loaders under `src/infrastructure/loaders/`). Each record contains a unique sample identifier and a Java code fragment treated as the unit of analysis. Dataset validation utilities are provided under `src/services/dataset/` to ensure structural consistency prior to running compute-intensive pipelines.

II.2.3 Stage 3: Detection execution

The detection stage depends on pipeline strategy:

- **P1 (Classifier-only)** executes one or more transformer-based token classifiers and aggregates detections using an OR-ensemble strategy.
- **P2 (Hybrid)** consumes P1 detections and applies configurable prefiltering, candidate construction, and LLM-as-a-judge decisions to reduce false positives.
- **P3 (LLM-only)** prompts the LLM to produce a structured JSON detection output, which is sanitized, schema-validated, and normalized.

II.2.4 Stage 4: Deterministic post-processing and normalization

All pipelines converge into a shared post-processing layer that harmonizes labels and values before evaluation. This includes label equivalence mapping, deterministic value normalization, confidence filtering, optional GDPR filtering, and rule-based validity filters. For LLM-driven pipelines, post-processing includes additional safeguards to reject non-verbatim values (i.e., values not present verbatim in the input Java code fragment) and to filter code-like or regex-like artifacts that commonly appear as spurious outputs.

II.2.5 Stage 5: Evaluation and metrics

The evaluation layer computes metrics by matching predicted entities against the ground truth using normalized category and normalized value (i.e., offsets are not required for matching). Evaluation services are implemented under `src/services/evaluation/` and `src/services/metrics/`, supporting aggregated metrics as well as per-label and extended diagnostic reports.

II.2.6 Stage 6: Artifact persistence and reporting

Each run produces a deterministic directory with serialized JSONL outputs, resolved configuration snapshots, execution logs, and evaluation reports. Reporting utilities in `src/services/reports/` generate structured summaries that support downstream interpretation in the experimental evaluation chapter.

II.3 SUPPORTED DETECTION STRATEGIES

II.3.1 Pipeline 1: Classifier-only (Transformer ensemble)

Transformer-based detection is implemented via Hugging Face token classification pipelines wrapped by a common base abstraction (`src/infrastructure/classifiers/base_hf_token_classifier.py`). Concrete wrappers exist for multiple supported models (e.g., `src/infrastructure/classifiers/starpii_wrapper.py`, `src/infrastructure/classifiers/deberta_wrapper.py`, `src/infrastructure/classifiers/ab_ai_wrapper.py`, `src/infrastructure/classifiers/piiranha_wrapper.py`, `src/infrastructure/classifiers/yonigo_distilbert_wrapper.py`, `src/infrastructure/classifiers/h2o_deberta_wrapper.py`). A classifier factory (`src/infrastructure/classifiers/factory.py`) standardizes model instantiation and enables pipeline configuration to select which classifiers are executed.

The OR-ensemble merge policy is implemented at the pipeline level, ensuring that candidate entities are retained when detected by any configured classifier. This prioritizes recall and establishes a high-sensitivity baseline for the hybrid pipeline.

II.3.2 Pipeline 2: Hybrid (Classifier candidates + LLM-as-a-Judge)

The hybrid pipeline extends the classifier-only strategy by validating candidate entities through a large language model acting as a judge. Conceptually, this pipeline comprises four functional phases, implemented by the runner and supporting services:

- **Candidate preparation:** the pipeline reads the classifier ensemble outputs and constructs candidate entity objects, preserving provenance information (e.g., origin classifier) to enable traceability.
- **Configurable prefiltering:** candidates that match clearly non-PII shapes (e.g., boolean-like tokens, placeholders, or code-like fragments) may be dropped before invoking the LLM, reducing cost and latency.
- **LLM judgment:** remaining candidates are submitted to the judge prompt, which returns a strictly structured JSON decision object indicating whether each candidate should be kept or rejected, optionally with normalized labels and confidence.
- **Deterministic sanitization and normalization:** judged outputs are validated, normalized, and serialized into the hybrid detection outputs used for evaluation.

This design ensures that improvements in precision are attributable to explicit judgment and filtering decisions, while preserving a complete audit trail from classifier candidates to judged outputs.

II.3.3 Pipeline 3: LLM-only (Structured extraction)

The LLM-only pipeline delegates detection to the LLM, enforcing structured extraction through strict prompt contracts and schema validation. Given a Java code fragment, the model is instructed to return a JSON object describing detected entities. The pipeline then applies deterministic sanitization to extract a valid JSON payload, validates it against an explicit schema, and performs the same label/value normalization applied to other pipelines.

This pipeline explicitly addresses common failure modes of LLM outputs (extra prose, markdown fences, truncated JSON, label drift, and hallucinated values) by combining sanitization, schema validation, and rule-based filtering before evaluation.

II.4 NORMALIZATION AND OUTPUT SCHEMA

Normalization is a first-class concern of the framework. It guarantees that outputs from heterogeneous sources (token classifiers and LLMs) are comparable under a unified taxonomy and a deterministic value representation.

II.4.1 Label mapping and equivalences

Canonical label mapping and equivalence handling are YAML-driven. The framework includes dedicated mapping and equivalence resources under `src/shared/config/`, including:

- `src/shared/config/label_mapping.yml`: canonical taxonomy definitions and mapping rules across models.
- `src/shared/config/label_equivalences.yml`: synonym/equivalence sets for harmonization, including classifier-specific direct maps and equivalence groups.
- `src/shared/config/label_equivalences_p3.yml`: pipeline-specific equivalences to isolate LLM-output normalization from classifier-based pipelines.

Label normalization services apply these rules deterministically, preventing label fragmentation during evaluation and ensuring that equivalent concepts are consistently mapped to a canonical taxonomy.

II.4.2 Value normalization and validity filters

Value normalization is implemented as a deterministic transformation step to standardize values used for matching and evaluation. It supports canonicalization of values such as email addresses, IP addresses, phone-like tokens, and dates. For LLM workflows, value normalization is reinforced by strict validity filters applied during response sanitization and post-processing, including:

- rejection of hallucinated values not present verbatim in the input Java code fragment;
- rejection of code-like and regex-like artifacts that are frequent false positives in generated outputs;
- confidence-based filtering with configurable thresholds.

II.4.3 Strict JSON schemas and validation

Structured outputs are constrained by explicit JSON schemas distributed with the framework:

- `src/resources/schema/bigcode_pii_schema.json`;
- `src/resources/schema/judged_detections.schema.json`.

Schema enforcement is performed immediately after sanitization and before normalization and persistence, ensuring that only syntactically valid and structurally compliant JSON objects are accepted.

II.4.4 GDPR-focused normalization

For GDPR-focused evaluation variants, the framework provides category allowlists and post-processing utilities:

- src/shared/config/gdpr_labels.yml and src/shared/config/gdpr_labels_p3.yml;
- GDPR filtering services that restrict detections to an allowlisted set while preserving sample identifiers.

This design supports parallel evaluation on full and GDPR-filtered variants without duplicating core pipeline logic.

II.5 EXECUTION MODES AND CONFIGURATIONS

The framework supports two equivalent execution modes.

II.5.1 Command-line execution

CLI entrypoints in src/cli/commands/ provide reproducible, batch-oriented execution of each pipeline: src/cli/commands/experiment_evaluate_p1_classifier.py, src/cli/commands/experiment_evaluate_p2_hybrid.py, and src/cli/commands/experiment_evaluate_p3_llm.py. Auxiliary commands enable dataset validation, statistics, result inspection, and run comparison (src/cli/commands/results_inspect.py, src/cli/commands/results_stats.py, src/cli/commands/results_compare.py, and src/cli/commands/experiment_repeat.py).

II.5.2 Notebook-based execution

A notebook runner (distributed as a separate artifact) demonstrates execution in cloud environments. It orchestrates dependency installation, model availability checks (including gated Hugging Face models), optional Ollama setup, and execution through the same YAML configuration files used by the CLI. This ensures that results are execution-mode invariant, assuming identical configurations and model versions.

II.5.3 Configuration-driven experimentation

All pipeline parameters are defined via YAML configurations, including dataset paths, selected pipeline, model identifiers, thresholds, logging settings, and output directories. The resolved configuration is persisted with each run, enabling exact re-execution and auditability.

II.6 ARTIFACT RETENTION AND LOGGING

The framework retains a complete set of artifacts for each run, enabling traceability from configuration to detections and metrics. Artifact structure is deterministic and includes:

- **Resolved configuration snapshots** (e.g., resolved YAML/JSON) persisted alongside outputs.
- **Detection outputs** as JSONL, including full and GDPR-filtered variants when enabled.
- **Evaluation reports** in JSON, including aggregate and per-label breakdowns.
- **Execution logs** capturing runtime details and failures without discarding partial results.

Table II.3 summarizes artifacts that are typically present across runs, while pipeline-specific artifacts differ substantially. Pipeline 1 materializes per-classifier outputs prior to aggregation, Pipeline 2 persists intermediate hybrid decision artifacts, and Pipeline 3 stores sanitization telemetry. In addition to core JSONL outputs and metrics, runs may also include auxiliary subdirectories (e.g., reports and pipeline-specific diagnostics) that support auditing and post-hoc analysis. Figures II.4–II.6 detail representative directory structures for each pipeline.

II.6.1 LLM interaction logging (Ollama)

LLM integration is implemented under `src/infrastructure/llm/`. The framework provides an Ollama client and adapter (`src/infrastructure/llm/ollama_llm_client.py` and `src/infrastructure/llm/ollama_adapter.py`) and a dedicated interaction logger (`src/infrastructure/llm/llm_interaction_logger.py`).

All LLM calls (including healthchecks/preflight interactions and per-sample generations) are recorded in structured NDJSON logs with timestamps, model identifiers, prompt contents, parameters, and raw responses. This log is produced only for pipelines that invoke an LLM (Pipeline 2 and Pipeline 3). This enables post-hoc auditing of LLM behavior and supports independent verification of structured-output constraints and downstream decisions.

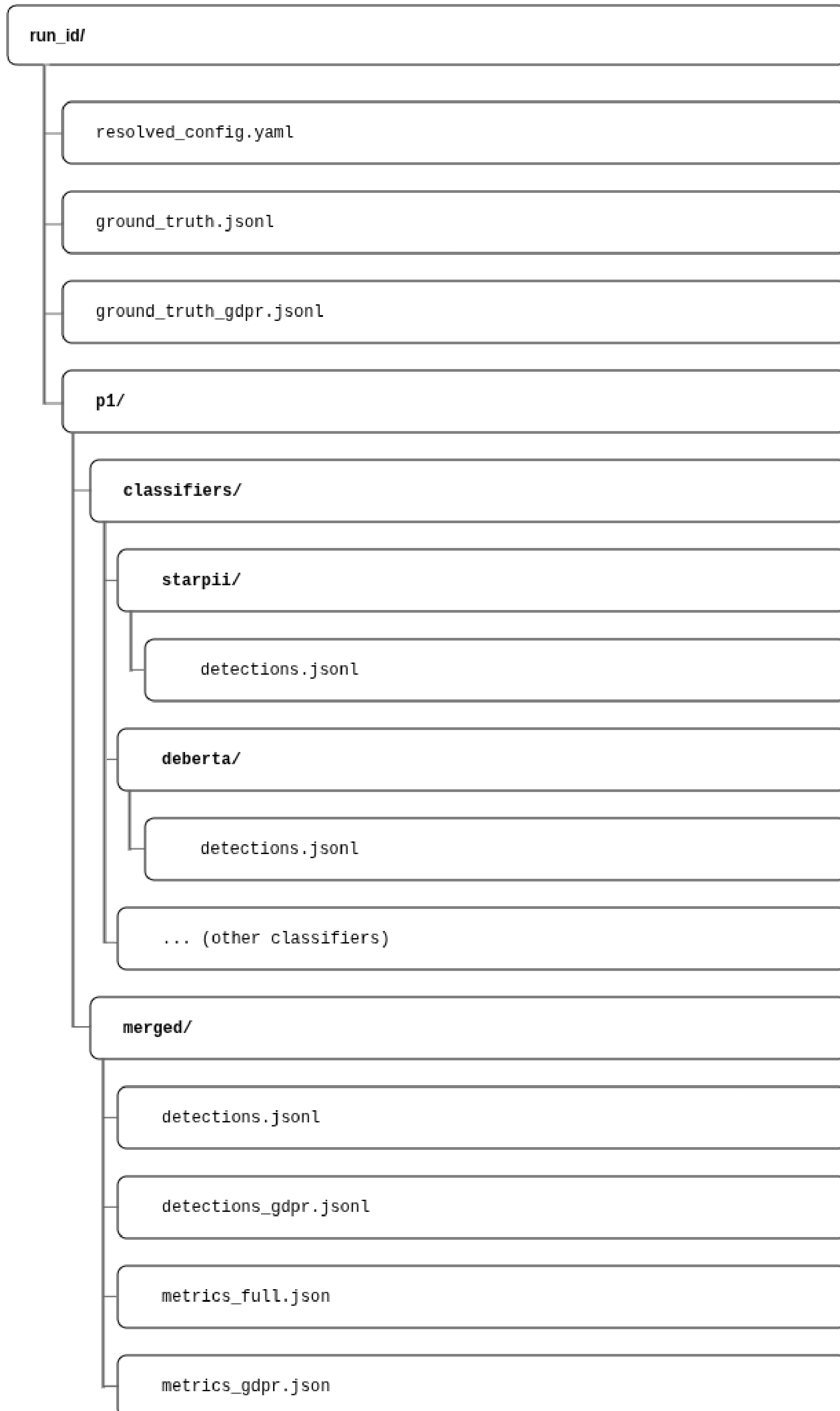

Figure II.4: Example directory structure of artifacts retained for a Pipeline 1 (classifier-only) run, including per-classifier outputs and merged detections and metrics.

Figure II.5: Example directory structure of artifacts retained for a Pipeline 2 (hybrid) run, including baseline classifier outputs, candidate sets, judged outputs, and hybrid metrics.

Figure II.6: Example directory structure of artifacts retained for a Pipeline 3 (LLM-only) run, including LLM interaction logs, normalized detections, and sanitization telemetry.

Table II.3: Common run-level artifacts (all pipelines).

Artifact	Format	Purpose
detections.jsonl	JSONL	Full predicted entities for each sample (pipeline output before GDPR-only restriction).
detections_gdpr.jsonl	JSONL	GDPR-normalized predicted entities (allowlist filtering applied, same sample identifiers).
metrics_full.json	JSON	Aggregated evaluation metrics for the full variant (ground truth vs detections.jsonl).
metrics_gdpr.json	JSON	Aggregated evaluation metrics for the GDPR-focused variant (filtered ground truth vs detections_gdpr.jsonl).
resolved_config.(yml | json)	YAML/JSON	Resolved configuration snapshot enabling exact re-execution and auditability.
llm_interactions.log	NDJSON	Verbatim log of all LLM interactions (healthchecks and per-sample generations/judgments), when applicable (P2–P3).

II.6.2 Prompt templates (Jinja2) and template-only policy

LLM prompts are generated exclusively from Jinja2 templates stored under src/resources/prompts/:

- src/resources/prompts/judge_prompt.j2: used by the hybrid pipeline judge.
- src/resources/prompts/llm_only_prompt.j2: used by the LLM-only pipeline.
- src/resources/prompts/p3_llm_full_taxonomy.j2: an extended taxonomy prompt variant for LLM-only configurations.

Prompt rendering is centralized (src/shared/prompts/prompt_loader.py) and uses strict template rendering rules to prevent missing variables and accidental prompt drift. Each template explicitly defines strict structured-output constraints (JSON-only), required fields, and decision rules. This design isolates prompt policy changes to templates, avoiding hard-coded prompt fragments in execution code.

II.6.3 Java parsing utilities

The framework includes Java parsing utilities under `src/infrastructure/parsers/java_code_parser.py` and a CLI command for analysis (`src/cli/commands/java_analyze.py`). These utilities support inspection and diagnostic workflows and contribute to consistent handling of Java code fragments during pre-processing.

Overall, the artifact retention and logging approach provides a transparent audit trail from inputs (dataset and configuration) to outputs (detections and metrics), supporting reproducible empirical evaluation and rigorous reporting.

II.7 CONFIGURATION RESOURCES IN `SHARED/CONFIG`

In addition to the pipeline runners and infrastructure adapters, the framework relies on YAML configuration resources located under `src/shared/config/`. These files encode the canonical label taxonomy, cross-model label mappings, GDPR-specific allowlists, pipeline-specific equivalences, and hybrid-pipeline heuristics used to reduce false positives. Because these resources directly affect detection outputs and evaluation comparability, they are treated as first-class artifacts in the framework design.

II.7.1 `label_mapping.yml`: Canonical taxonomy and per-classifier label mapping

The file `src/shared/config/label_mapping.yml` defines the canonical label space used across pipelines and provides deterministic mappings from model-specific labels to canonical categories.

- **Canonical taxonomy (`canonical_labels`).** The taxonomy enumerates canonical categories and associates each with normalized synonyms, enabling harmonization of heterogeneous label vocabularies into a single evaluation space.
- **Per-classifier overrides (`per_classifier_overrides`).** For each supported classifier backend, explicit mappings convert native label identifiers into canonical categories, preventing taxonomy drift across backends.

II.7.2 `label_equivalences.yml`: Classifier-specific direct maps and equivalence sets

The file `src/shared/config/label_equivalences.yml` encodes classifier-specific label normalization behavior beyond simple renaming. It contains versioned entries and classifier-

specific sections that define deterministic direct mappings and equivalence groups used during normalization and evaluation.

A key design goal is to prevent label fragmentation and to keep matching consistent even when backends emit BIO-style labels or coarse-grained categories that require canonical harmonization.

II.7.3 `label_equivalences_p3.yml`: LLM-output normalization and alias collapse (P3 isolation)

The file `src/shared/config/label_equivalences_p3.yml` provides a broad alias-to-canonical mapping tailored to the LLM-only pipeline, collapsing natural-language variants and common label spellings into canonical categories. This resource is pipeline-specific to ensure that LLM normalization changes do not affect P1 and P2 behavior.

II.7.4 `gdpr_labels.yml`: GDPR allowlist with descriptions and examples

The file `src/shared/config/gdpr_labels.yml` defines the GDPR-relevant category allowlist used for filtered variants of detections and ground truth. It provides structured documentation for each allowed category, including short descriptions and illustrative examples, enabling transparent interpretation of the regulatory scope enforced by filtering.

II.7.5 `gdpr_labels_p3.yml`: GDPR allowlist + aliases + exclusions for LLM-only

The file `src/shared/config/gdpr_labels_p3.yml` is a GDPR configuration tailored to the LLM-only pipeline. It includes an allowlist, alias mappings for LLM-produced label variants, and explicit exclusions used to prevent out-of-scope labels from propagating into evaluation.

II.7.6 `p2_hybrid_prefilter.yml`: Hybrid prefilter heuristics (routing and thresholds)

The file `src/shared/config/p2_hybrid_prefilter.yml` defines heuristics that reduce unnecessary LLM calls and drop obvious false positives before invoking the judge. It includes shape-based filtering (e.g., boolean-like tokens) and confidence-based routing rules, supporting global thresholds and targeted overrides for specific labels and backends.

II.7.7 `p2_hybrid_sanitizer.yml`: Hybrid sanitization rules and telemetry

The file `src/shared/config/p2_hybrid_sanitizer.yml` configures deterministic sanitization and auditing behavior for hybrid outputs. It enables rule-based removal of common

non-PII artifacts (e.g., placeholders, code-like tokens, mask-like patterns) and supports sanitizer telemetry for auditability, enabling post-hoc analysis of dropped candidates and their reasons.

II.8 STRUCTURED OUTPUT ENFORCEMENT: PROMPT TEMPLATES AND JSON SCHEMAS

A central design requirement of the framework is the enforcement of strictly structured outputs for all large language model (LLM) interactions. Unlike free-form text generation, all LLM-based pipelines are constrained to produce machine-readable JSON objects that conform to explicit schemas. This requirement is critical to ensure deterministic post-processing, robust evaluation, and auditability of model behavior.

To achieve this, the framework combines three complementary mechanisms: (i) Jinja2-based prompt templates with explicit output contracts, (ii) deterministic response sanitization, and (iii) JSON schema validation.

Figure II.7 summarizes the standardized enforcement flow adopted by the LLM-based pipelines. In this design, prompt policy is encoded in a Jinja2 template, the model output is treated as a raw response by default, and acceptance into the evaluation path requires deterministic sanitization followed by JSON schema validation. Only then are entities normalized, ensuring a consistent and comparable representation across pipelines.

II.8.1 Jinja2 Prompt Templates for LLM-Based Pipelines

All prompts submitted to large language models are generated exclusively from Jinja2 templates located under `src/resources/prompts/`. The framework enforces a strict *template-only* policy: no prompt fragments are hard-coded in Python code, and changes to prompt wording or output requirements are performed by editing the corresponding template file.

The templates define both the task instructions and the structured-output contract, including required fields, allowed values, and formatting constraints. Prompt rendering is centralized and validated to prevent missing variables and silent prompt drift.

Hybrid pipeline judge template. The hybrid pipeline relies on `src/resources/prompts/judge_prompt.j2` to validate candidate entities produced by the classifier ensemble. The template instructs the LLM to evaluate each candidate entity in the context of the original Java code fragment and to return a strictly structured JSON object encoding the judgment.

The template explicitly constrains the model to:

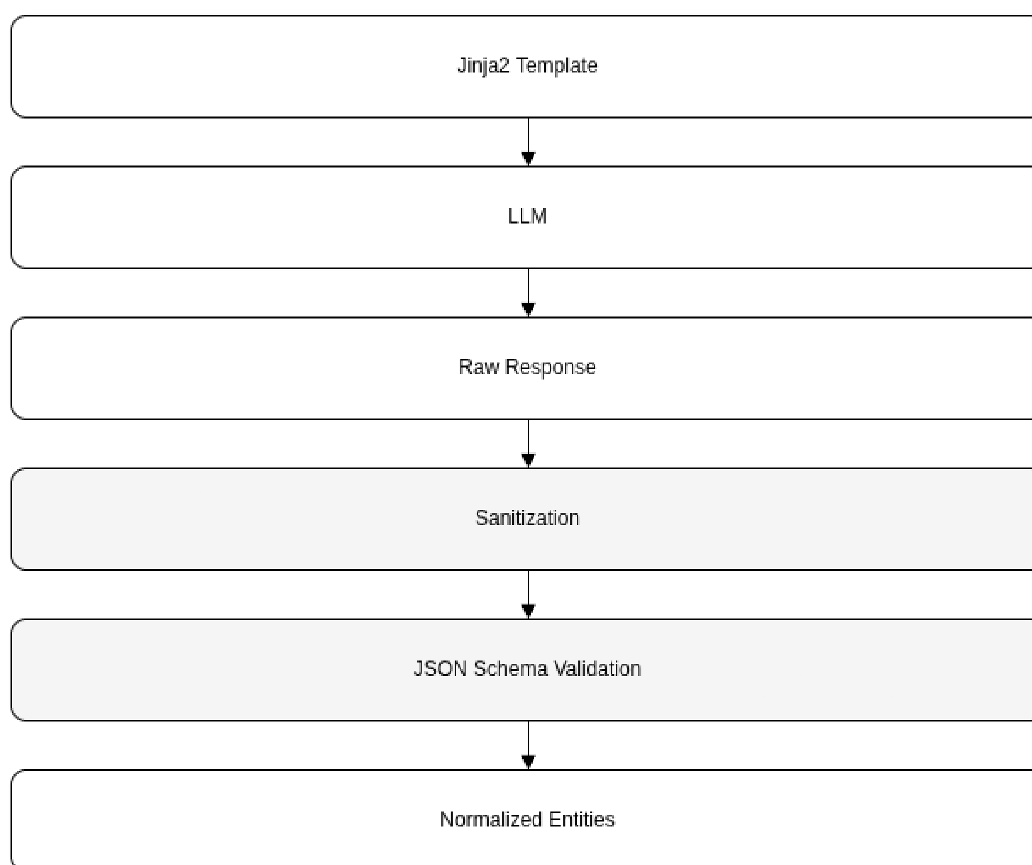

Figure II.7: Structured output enforcement flow for LLM-based pipelines. Prompts are rendered from Jinja2 templates, the LLM produces a raw response, and the framework applies deterministic sanitization and JSON schema validation before normalizing entities for downstream evaluation and reporting.

- produce a single JSON object with no additional prose or markup;
- use predefined decision fields (e.g., keep or reject);
- assign canonical category labels only;
- avoid introducing new entity values not present verbatim in the Java code fragment.

LLM-only detection templates. The LLM-only pipeline uses templates designed for end-to-end structured extraction. The primary template `src/resources/prompts/llm_only_prompt.j2` instructs the model to identify all PII instances in a given Java code fragment and to return them as JSON entities. An extended variant, `src/resources/prompts/p3_llm_full_taxonomy.j2`, provides a more detailed taxonomy for configurations requiring broader label coverage.

Both templates enforce a JSON-only output policy and define mandatory fields such as entity

value, category, and confidence score. They are designed to reduce common LLM failure modes by prohibiting markdown fences and discouraging speculative or inferred values not present in the input.

II.8.2 Deterministic Response Sanitization

Despite strict prompt constraints, LLM outputs may contain formatting noise or partial violations of the expected contract. The framework applies deterministic response sanitization to remove extraneous text, extract the first syntactically balanced JSON object, and normalize structural elements prior to validation.

This step ensures that minor formatting deviations do not compromise execution while preserving strict enforcement of structural correctness. Sanitization decisions are logged explicitly, enabling inspection of raw versus sanitized responses during auditing.

II.8.3 JSON Schemas for Structured Output Validation

The final enforcement layer is based on explicit JSON schemas distributed with the framework under `src/resources/schema/`. These schemas define allowed structure, required fields, data types, and constraints for structured outputs produced by LLM-based pipelines.

Detection schema. The schema `src/resources/schema/bigcode_pii_schema.json` defines the structure of detection outputs, including required fields and the representation of entity objects (values, categories, and confidence). It is applied to outputs generated by the LLM-only pipeline, ensuring uniform representation prior to normalization and evaluation.

Judged detection schema. The schema `src/resources/schema/judged_detections.schema.json` defines the structured output expected from the hybrid pipeline judge. It encodes the decision model used to accept or reject candidate entities and constrains allowed fields and values, preventing malformed or incomplete judgments from propagating into evaluation.

II.8.4 Schema Validation and Failure Handling

Schema validation is performed immediately after sanitization and before normalization and persistence. Outputs that fail schema validation are rejected or flagged explicitly, depending on pipeline configuration. This design prevents invalid outputs from contaminating detection results or evaluation metrics.

Together, template-driven prompting, deterministic sanitization, and schema-based validation provide strict structured-output guarantees. These guarantees are essential to enable reproducible

experimentation, reliable evaluation, and transparent auditing of LLM behavior in privacy analysis tasks.

II.9 SUMMARY

This appendix detailed the detection framework architecture, the standardized execution lifecycle shared across pipelines, the three supported detection strategies, and the configuration-driven normalization and evaluation mechanisms. Particular emphasis was placed on strict structured-output enforcement for LLM-based pipelines through Jinja2 templates, deterministic sanitization, and JSON schema validation. Finally, the artifact retention and logging strategy ensures traceability from configuration and inputs to detections and metrics, supporting reproducible empirical evaluation and independent verification.

III. EXPERIMENTAL CONFIGURATION AND REPRODUCIBILITY ARTIFACTS

This appendix consolidates the executable artifacts and configuration elements required to reproduce the experimental results reported in this dissertation.

It complements the methodological description in Chapter 3 by documenting dataset snapshots, configuration schemas, execution artifacts, directory layouts, and environment constraints used across all experimental treatments.

The purpose of this appendix is not to restate the experimental design rationale, but to provide an auditable reference that enables replication and independent verification through explicit, versioned reproducibility artifacts.

III.1 DATASET SNAPSHOT AND GROUND-TRUTH SCHEMA

All experiments use a fixed snapshot of a synthetic dataset composed of Java code fragments annotated with personal-data fragments under a GDPR-scoped taxonomy, implemented as an explicit allowlist of included categories. (3)

The dataset is distributed as a JSONL file, where each record corresponds to a single Java code fragment and its associated list of annotations (`fragments`).

To ensure auditability and repeatability, the snapshot is accompanied by reproducibility artifacts that capture (i) the dataset file checksum, (ii) the effective generation parameters, and (iii) run-level metadata describing the generation context (e.g., locale, positive-rate targets, and optional LLM backend identifiers).

The operational inclusion/exclusion decision procedure that underpins the GDPR-scoped allowlist is documented in Appendix I (Figure I.1). (3)

The concrete allowlist snapshot used for a dataset release is recorded as part of the run directory artifacts to support auditability.

III.1.1 Record Structure

Each JSONL line encodes a single record with a deterministic identifier `id`, the language tag (`java`), an implementation-defined `type` field indicating the origin/mode of the snippet, the full snippet `text`, and a list of zero or more `fragments`.

The schema and its integrity invariants are summarized in Appendix I (Figure I.4).

III.1.2 Annotation Fields and Label Semantics

Each fragment includes `category`, `value` (verbatim inserted string), and `position` as a half-open interval `[start,end)` over `text`.

Categories are restricted to the GDPR-scoped allowlist described in Appendix I, and the taxonomy design goal is to remain explicit, auditable, and stable across experimental treatments. (3)

III.1.3 Negative Samples and Matching Assumptions

Negative samples are represented as `fragments: []`, preserving auditability by explicitly encoding the absence of annotated personal data rather than relying on implicit conventions.

Downstream evaluation assumes that (i) each record can be joined deterministically via `id`, and (ii) fragment-level integrity invariants hold (e.g., `text[position[0]:position[1]] == value`) as documented in Appendix I.

These assumptions enable exact verification of span correctness and consistent automated evaluation.

III.1.4 Dataset Versioning and Integrity Checks

To enable reproducible reuse of dataset snapshots, each released snapshot is treated as immutable.

Integrity is ensured through file-level checksums and by recording the effective generation configuration.

When LLM-based snippet generation is enabled during dataset construction, the generation run also archives raw interaction traces (prompts and raw responses) as reproducibility artifacts, enabling auditability of the exact model inputs that produced the exported snippets.

III.2 EVALUATION PROTOCOL INVARIANTS AND MATCHING CRITERIA

This section records the concrete evaluation invariants that must be preserved to reproduce the reported scores, independent of the detection strategy.

III.2.1 Join Key and Sample Identity

All prediction files and derived artifacts are keyed by the dataset record identifier `id`.

When intermediate artifacts use `sample_id`, it is treated as equivalent to `id`, and joins are performed deterministically by this identifier.

III.2.2 Entity Matching Key

All pipelines are evaluated at the entity level using a value-centric matching rule.

Entities are matched using the tuple $(label, value_{norm})$, where `value_norm` denotes a deterministic normalization applied consistently to both predictions and ground truth.

Positional offsets are ignored for matching, and no span-level scoring is performed.

III.2.3 Metrics Outputs

The reported effectiveness metrics are micro-averaged precision, recall, and F_1 computed under the same matching criterion for all treatments.

When judge-centric indicators are reported for the hybrid pipeline (P2), acceptance rate and true-positive rejection rate are computed from the archived candidate pool and the per-candidate keep/drop decisions.

III.3 EXPERIMENTAL CONFIGURATION FILES

All experiments are parameterized through declarative YAML configuration files that define dataset paths, detection strategies, model parameters, evaluation criteria, and output locations.

This configuration-driven design ensures that experimental treatments are executed under consistent conditions, with differences arising only from explicitly declared experimental factors.

To preserve auditability, every execution archives the resolved configuration used for the run, including all defaults applied during loading and validation.

III.3.1 Configuration Directory Organization

Configuration files are organized by pipeline/treatment family and stored alongside the codebase.

Each configuration is intended to be self-contained and portable, relying on relative paths and a standardized run directory structure for outputs.

III.3.2 Top-Level Configuration Schema

The top-level configuration schema includes: dataset snapshot identifiers/paths, model backends (when applicable), detector parameters, evaluation settings, and output directories.

The schema is designed to make all relevant experimental choices explicit and machine-readable.

III.3.3 Comparability and Control Policies

Comparability across treatments is enforced by holding constant the dataset snapshot and the evaluation protocol, while varying only the intended treatment factors (e.g., detection strategy and backend model selection).

Any treatment-specific post-processing steps are treated as part of the declared pipeline configuration and are therefore captured in archived run artifacts.

III.3.4 Path Resolution and Portability Assumptions

Paths are resolved relative to a project root to improve portability across environments.

To reproduce a run, users must preserve the directory layout and ensure that configuration files resolve to the same dataset snapshot and model resources referenced in the archived run metadata.

III.4 MODELS, BACKENDS, AND IDENTIFIERS

This section records the concrete model identifiers and execution backends required to reproduce the reported treatments, including the exact tags used for open-weight models.

III.4.1 Classifier Models (Pipeline 1)

Pipeline 1 applies six transformer-based PII detectors to each snippet.

The specific models are:

- StarPII (`bigcode/starpii`) (36, 37)
- AB-AI/PII (`ab-ai/pii_model`) (38)
- Piiranha v1 (`iiiorg/piiranha-v1-detect-personal-information`) (39)
- DeBERTa PII – H2O (`h2oai/deberta_finetuned_pii`) (40)

- DeBERTa PII – lakshyakh93 (`lakshyakh93/deberta_finetuned_pii`) (41)
- Multilingual DistilBERT-based PII detector (`yonigo/distilbert-base-multilingual-cased-pii`) (42)

III.4.2 LLM Models and Ollama Tags (Pipelines 2–3)

All LLM-based experiments use open-weight models executed via Ollama.

To ensure reproducibility, each model is identified by its exact Ollama tag.

The evaluated tags include:

Code-specialized models:

- Qwen3-Coder (`qwen3-coder:30b-a3b-q4_K_M`) (43)
- CodeGemma (`codegemma:7b-instruct`) (44)
- Code Llama (`codellama:13b-instruct-q4_K_M`) (45)
- Codestral (`codestral:22b-v0.1-q4_K_M`) (46)
- Devstral-Small-2 (`devstral-small-2:24b-instruct-2512-q4_K_M`) (47)

General-purpose / reasoning-oriented models:

- Qwen3 (`qwen3:14b-q4_K_M`) (43)
- Qwen3 (`qwen3:30b-a3b-instruct-2507-q4_K_M`) (43)
- DeepSeek R1 (`deepseek-r1:14b`) (48)
- Phi-4 (`phi4:14b-q4_K_M`) (49)
- Gemma 3 (`gemma3:12b-it-q4_K_M`) (50)
- GPT-OSS (`gpt-oss:20b`) (51)
- Mistral (`mistral:7b-instruct`) (52)

III.5 EXPERIMENTAL EXECUTION AND GENERATED ARTIFACTS

Experimental runs are orchestrated through a dedicated execution driver that consumes the YAML configurations and produces a structured set of reproducibility artifacts for each run.

These artifacts are archived to support post-hoc analysis, error inspection, and independent reproduction of reported results, with an emphasis on auditability of both configuration and model interactions.

III.5.1 Execution Orchestration

Each run loads a single YAML configuration, resolves defaults, validates schema constraints, and executes the corresponding pipeline end-to-end.

The execution driver records the resolved configuration and run metadata (timestamps, identifiers, and output paths) as part of the run directory.

III.5.2 Output Directory Layout

Each run produces a self-contained output directory that groups the primary outputs (detections and metrics) and the supporting reproducibility artifacts (resolved configuration, metadata, logs, and optional LLM interaction traces).

The directory layout is structured to make replication straightforward by enabling deterministic lookup of all artifacts for a given run identifier.

III.5.3 Pipeline-Specific Intermediate Artifacts

To support auditability and to enable recomputation of reported metrics from archived evidence, the execution driver retains pipeline-specific intermediate files in addition to final outputs.

For the hybrid pipeline (P2), the candidate pool presented to the judge is stored as `candidates_for_judge.jsonl`.

Candidates are deduplicated using the key `(sample_id, label, value_norm)`, ensuring that each unique candidate corresponds to exactly one keep/drop decision and that the number of judge decisions is invariant across judge models.

The per-candidate judge decisions are archived so that acceptance rate and true-positive rejection rate can be recomputed directly from `candidates_for_judge.jsonl` and the keep/-drop logs without relying on transient runtime state.

III.5.4 Detection Outputs

Detection outputs are serialized in machine-readable formats (e.g., JSONL) and are keyed by the dataset record identifier `id` to support stable joins with the ground truth snapshot.

When applicable, outputs preserve verbatim values and supporting metadata required by the

evaluation protocol, including normalized value fields used for matching.

III.5.5 Metrics and Aggregated Results

Each run emits machine-readable metrics files containing the evaluation outputs used in the dissertation.

Aggregated results reported in the main chapters are derived from these per-run metrics and can be recomputed by re-running the evaluation driver over the archived outputs.

III.5.6 Execution Logs and LLM Interaction Records

For every run, execution logs record key steps, parameters, and any errors.

When a pipeline relies on an LLM backend, the run additionally archives an interaction log (NDJSON), recording per-call raw prompts and raw responses as reproducibility artifacts.

These interaction traces support auditability of the exact model inputs and outputs used during execution.

III.6 PROMPT TEMPLATES, DETERMINISM, AND LOGGING CONTROLS

This section records the concrete controls that must be preserved to reproduce judge behavior and extraction outputs as closely as possible.

III.6.1 Prompt Templates

All prompts are defined as Jinja2 templates that encapsulate both system and user instructions.

Prompt templates are treated as versioned artifacts and are archived or referenced by commit/version identifiers in the reproducibility package.

III.6.2 Determinism and Decoding Settings

Where supported by the inference backend, deterministic settings are applied, including disabled sampling and fixed seeds, to reduce run-to-run variability.

Because residual nondeterminism can still arise from backend behavior, quantization, or environment effects, raw interaction logs are retained to enable independent inspection and diagnosis of deviations.

III.6.3 LLM Interaction Log Format

When LLM backends are involved, the archived NDJSON interaction logs contain, at minimum, timestamps, model identifiers, inference parameters, raw prompts, and unprocessed raw responses or errors.

These logs include health checks, pre-runs, and batch processing calls.

III.7 EXECUTION ENVIRONMENT AND SOFTWARE STACK

All experiments were conducted in two controlled environments to support reproducibility and to reflect realistic resource constraints.

III.7.1 Local Workstation

A desktop machine equipped with an NVIDIA RTX 4060 GPU with 8 GB of VRAM, an Intel i7-class processor, and 32 GB of RAM was used primarily to develop, debug, and validate the framework.

This environment also executed all transformer-based classifiers of Pipeline 1 using the HuggingFace inference stack and baseline executions of Pipeline 2 using Code Llama `codellama:13b-instruct-q4_K_M`) via Ollama.

III.7.2 Cloud Environment

Additional executions used Google Colab Pro with NVIDIA T4 and L4 GPUs, providing between 15 and 24 GB of VRAM.

Cloud executions preserved the same datasets, configuration files, prompt templates, and evaluation procedures adopted in the local environment.

III.7.3 Software Stack

All pipelines were implemented in Python 3.10, with dependency management handled through Poetry.

Transformer-based classifiers were executed using the HuggingFace Transformers and Accelerate libraries.

Open-weight language models were executed with Ollama version 0.3.x.

Prompting logic was implemented through Jinja2 templates, and run-level parameters were

controlled via YAML configuration files.

III.8 REPRODUCTION GUIDELINES AND CONSTRAINTS

To reproduce the experiments reported in this dissertation, the dataset snapshot, configuration files, and execution procedure described in this appendix must be used without modification.

Differences in hardware, execution environment, or model availability may affect runtime or cost, but do not alter the logical evaluation protocol defined by the configuration artifacts and the dataset snapshot.

III.8.1 Required Artifacts

Reproduction requires: (i) the immutable dataset snapshot (JSONL) and its recorded checksum, (ii) the YAML configuration files for the treatments of interest, (iii) the execution driver, and (iv) the archived run directories containing resolved configuration, run metadata, detection outputs, and metrics.

When LLM backends are involved, the interaction logs are required for full auditability.

III.8.2 Permitted and Non-Permitted Modifications

Permitted modifications are limited to environment-level adaptations that do not change the logical protocol (e.g., installation paths, containerization, or hardware selection).

Non-permitted modifications include changing the dataset snapshot, altering the evaluation protocol, or editing treatment configurations in a way that changes experimental factors without producing a new run directory and updated reproducibility artifacts.

III.8.3 Hardware and Environment Sensitivity

Runtime and cost may vary with compute resources, especially for treatments that invoke LLM backends.

To maintain comparability, the logical configuration (dataset snapshot, parameters, and evaluation settings) must remain identical, and any environment changes must be documented in run metadata.

III.8.4 Threats to Reproducibility

Threats include model availability changes, nondeterminism in external backends, and environment drift.

Mitigations include (i) immutable dataset snapshots with recorded checksums, (ii) archived resolved configurations and run metadata, and (iii) raw LLM interaction traces when applicable, enabling independent inspection and auditability of the inputs that produced each output.

III.9 PII CATEGORY DISTRIBUTION IN DATASET

Table III.1 summarizes the distribution of annotated personal-data fragments across the categories included in the synthetic Java dataset snapshot.

The table is provided for descriptive purposes, to document category coverage and class imbalance, and to support the interpretation of the micro-averaged evaluation metrics reported in the main results.

No performance claims are derived from this distribution alone.

The category counts in Table III.1 sum to 3,596 entities, matching the total number of annotated entities obtained by counting all `fragments[*].category` entries in the dataset snapshot.

Table III.1: Distribution of PII categories in the synthetic Java dataset.

Category	#Entities	Share (%)
LASTNAME	156	4.3%
STATE	155	4.3%
COUNTY	148	4.1%
ZIPCODE	125	3.5%
JOBTITLE	112	3.1%
ACCOUNTNAME	109	3.0%
USERNAME	106	2.9%
FIRSTNAME	106	2.9%
MIDDLENAME	104	2.9%
CITY	99	2.8%
IPV4	90	2.5%
SECONDARYADDRESS	89	2.5%
MAC	89	2.5%
CREDITCARDCVV	89	2.5%
STREET	88	2.4%
ACCOUNTNUMBER	85	2.4%
DRIVERLICENSENUM	85	2.4%
IPV6	84	2.3%
USERAGENT	84	2.3%
PASSWORD	82	2.3%
BITCOINADDRESS	82	2.3%
LITECOINADDRESS	82	2.3%
EMAIL	80	2.2%
ETHEREUMADDRESS	79	2.2%
BIC	79	2.2%
IDCARDNUM	78	2.2%
IBAN	78	2.2%
SSN	77	2.1%
VEHICLEVRM	77	2.1%
PIN	76	2.1%
DATEOFBIRTH	73	2.0%
KEY	73	2.0%
TAXNUM	72	2.0%
IP	71	2.0%
CREDITCARDNUMBER	69	1.9%
DOB	69	1.9%
VEHICLEVIN	64	1.8%
DATE	57	1.6%
PHONEIMEI	57	1.6%
MASKEDNUMBER	51	1.4%
PHONENUMBER	47	1.3%
NEARBYGPSCOORDINATE	20	0.6%