Internship Thesis

Automatic Code Refactoring with Deep Learning

Alejandro Argüelles Trujillo
Master 1 Student, Université de Toulouse
alejandro.arguelles@univ-tlse.fr
alejandro.ar.tj@gmail.com

August 2025

Abstract

This report outlines the work conducted during my internship at Infotel Conseil in the summer of 2025. The project centered on the development of an AI-driven automatic code refactoring system, which I fully designed and implemented from the ground up. As the primary objective was to conduct research, develop a proof of concept, and initiate the implementation process, this document reflects each of these stages. It presents the research and prototyping efforts, along with selected sections detailing the initial implementation work.

Key topics addressed in this work include the detection of code antipatterns and their automated replacement with improved, semantically correct code segments. The approach combines traditional static analysis techniques—such as metric extraction—with modern AI-driven methods. These include the use of semantic embeddings, fine-tuned transformer-based language models, neural network classifiers, and agent-based orchestration. The integration of these components aims to bridge symbolic and statistical reasoning to enhance the robustness and scalability of the refactoring process.

Contents

1 Introduction and background

Over the past few years, large language models (LLMs) have reshaped software development workflows, moving from autocomplete to capable code-generation assistants integrated directly into the IDE. Controlled studies report substantial productivity gains when developers use assistants such as GitHub Copilot, with randomized experiments showing faster task completion and positive effects on perceived flow and satisfaction [1]. Meanwhile, industry deployments and further field studies continue to quantify the impact in enterprise settings [2].

In parallel, commercial offerings (e.g. GitHub Copilot and Amazon's CodeWhisperer—now part of Amazon Q Developer [3]) have normalized the idea that natural-language prompts can yield working code snippets inside editors like Visual Studio Code. Yet generation alone is not sufficient for modern engineering practice: high-quality software also demands systematic review, restructuring, and optimization after the code exists. This includes automated refactoring, detection of "code smells," and continuous improvements in readability, performance, and maintainability.

Refactoring provides a principled foundation for this post-generation phase. The classical literature frames "bad smells" as indicators—often stemming from complexity, duplication, or poor modularity—that motivate targeted refactorings rather than ad-hoc edits. Contemporary static analysis platforms operationalize these ideas with quantitative metrics (e.g., complexity, duplication, maintainability indices) that can be enforced in CI pipelines [6]. Together, they define actionable signals that an AI system can leverage when proposing code transformations.

At the same time, recent advances in deep learning for code provide complementary capabilities to traditional static rule-based approaches. Pre-trained transformer models specialized for programming languages—such as GraphCodeBERT and its successors [7]—are able to capture semantic structures and data-flow properties, enabling tasks like code search, summarization, and refinement. When combined with vector-search systems such as FAISS [8] or neural-network classifiers, these semantic embeddings allow for scalable retrieval of similar code idioms or canonical patterns and better smell detection, which are essential for learning-guided refactoring and pattern-preserving transformations.

This report situates our product within that evolving landscape: an AI-driven system focused on the analysis and improvement of existing source code. It integrates static analysis techniques—such as metric extraction and rule-based validation—with learned components, including semantic embeddings, fine-tuned transformer models, neural classifiers, and agent-based orchestration. These elements work together to detect code antipatterns and suggest or apply safe, context-aware refactorings.

While the current prototype operates at the level of isolated code snippets—performing local, single-file transformations—it is architected with future scalability in mind. The system is designed to eventually support more complex scenarios such as multi-file reasoning, cross-repository consistency, and the generation of coherent, large-scale patches that align with real-world development constraints.

Finally, the tool is engineered for unobtrusive integration into the developer's workflow. Our current implementation is a Visual Studio Code extension, leveraging the VS Code Extension API for editor integration, with a modular backend that allows new front ends and additional language adapters. The initial focus is TypeScript, but the architecture supports incremental support for languages such as Python or C via narrowly scoped adapters [4].

Table 1: Categories of Anomalies (Code Smells) for Detection and Remediation

Category	Examples	Impact
Structural / holistic (system level)	God Class / Blob, cyclic dependencies	Impede system evolution by increasing architectural complexity.
Component / class level	Feature Envy, overly deep inheritance	Degrade encapsulation and increase coupling.
Local / method level	Long methods, excessive parameters, duplication	Hinder readability and are correlated with defects.

2 Problem formulation and objectives

2.1 Scope and terminology

In this work, *code smell* denotes any recurring design or implementation symptom that suggests a deeper maintainability problem and motivates refactoring (e.g., long methods, excessive parameter lists, feature envy). We use the term broadly to subsume what the literature also calls *antipatterns*—named, commonly observed but ineffective solutions such as God Class/Blob or Spaghetti Code. Foundational accounts of smells and refactoring originate with Fowler and colleagues [5], while the antipatterns literature was formalized in the late 1990s and traced back to Koenig's coinage of the term "anti-pattern."

We interpret smell accumulation as a form of technical debt: a useful metaphor introduced by Ward Cunningham to describe how shortcuts in internal quality incur "interest," i.e., extra effort required to modify and extend systems later. This framing clarifies why remediation must be prioritized and measured, not just performed opportunistically.[1][2][3]

Categories of anomalies for detection and remediation, we organize smells into three practical categories (Table 1).

This taxonomy provides clear targets for automated proposals such as Extract Method, Split Class, or Introduce Dependency Injection, drawing on established refactoring catalogs.[4][5]

Problem. We consider a code repository—initially written in TypeScript—composed of multiple files, internal dependencies, and a build/test system. Our goal is to design a system that supports four core capabilities:

- **Detection:** Identify code smells or quality issues within individual elements (e.g., classes, methods), along with associated confidence scores.
- **Recommendation:** For each detected issue, propose a set of possible refactorings, accompanied by explanations and estimated impact.
- **Transformation:** Apply small, semantics-preserving changes to the code in order to fix the issues while maintaining its original behavior.
- **Evaluation:** Compare the original and refactored versions of the codebase, and measure quality improvements across multiple dimensions.

Any transformation applied must ensure that the resulting code compiles successfully and passes all existing tests. In cases where no tests are available, the system enforces stricter

[1] https://martinfowler.com
[2] https://insights.sei.cmu.edu
[3] https://agilealliance.org
[4] https://dl.ebooksworld.ir
[5] https://refactoring.guru

safety measures, such as type correctness and context-aware constraints, to reduce the risk of unintended side effects.

Objectives and Success Criteria. We evaluate the system against the following:

O1 Accurate, actionable detection. High precision/recall on labeled smells; prioritize findings with clear, automatable remedies.

O2 Safety and semantic preservation. All patches compile and pass existing tests. In the absence of tests, apply conservative guards (e.g., static type validation, scope/alias analysis).

O3 Measurable quality improvement. Demonstrate reductions in established metrics post-refactor (e.g., McCabe cyclomatic complexity, CK coupling/cohesion, maintainability indices, instability/cohesion where applicable).

O4 Standards alignment. Proposals move code toward widely accepted guidelines (e.g., SOLID for OO design; Python PEP 8/20/257; Java Language Specification conventions; C++ Core Guidelines / Google C++ Style Guide).

O5 Developer-in-the-loop ergonomics. Integrate in-IDE with transparent diffs and one-click accept/rollback; retain architectural decisions under human control.

O6 Extensibility. Support additional languages by swapping metric extractors and model adapters without re-architecting the core.

3 System Overview and Architecture

The overall architecture of our AI-assisted refactoring system is structured around two complementary paradigms: a *Sequential Modular Pipeline* (SMP) for deterministic and production-friendly refactoring, and an *Agentic Orchestration* (AO) approach for more flexible, iterative workflows. The figure 1 illustrates the high-level flow of code and decisions through the SMP.

The architecture is designed around independent but interacting modules, where data artifacts (e.g., code files, smells, patches) are explicitly represented and passed between components. This allows both modular reuse and functional composition of workflows.

3.1 Sequential Modular Pipeline (SMP)

Figure 1 depicts the system as three cooperating phases: (i) *project preprocessing*, (ii) an iterative *refactoring loop*, and (iii) *project reconstruction*. Each box represents a replaceable module with a narrow responsibility and typed I/O contracts (files, smells, plans, and candidate edits). The design emphasizes protocol-level coupling rather than concrete implementations.

3.2 Core Data Artifacts

Code files/snippets Logical views of the project at file and snippet granularity. Snippets carry a unique ID, a kind (e.g., function, class, module), and source locations; they enable fine-grained retrieval and patching.

Smells Detected quality issues, each with a unique ID, location, kind, severity, and optional human-readable message and suggestions.

Plan An ordered list of smell identifiers to be addressed by the loop. The plan is serializable and replayable.

Proposed code Candidate patches produced for a target smell and validated before materialization.

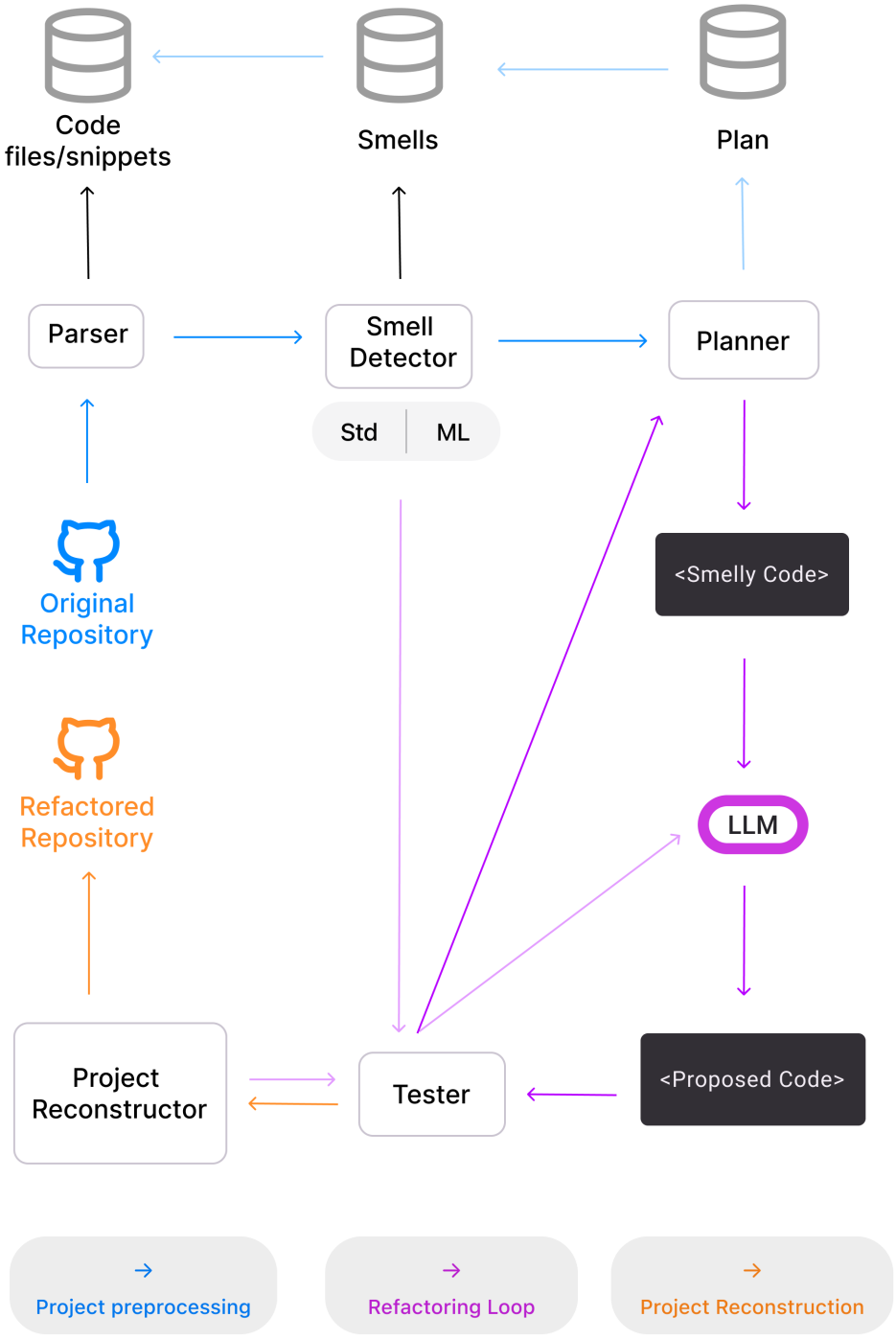


Figure 1: System-level architecture. The diagram highlights three main flows: (1) repository preprocessing (yellow), (2) refactoring loop (blue), and (3) final project reconstruction (red).

We generate unique snippet identifiers (IDs) by hashing structural metadata such as `FileName`, line numbers, and surrounding context. For this we employ the SHA-3 family (Keccak), rather than SHA-256, since SHA-3 is more resistant to collision and preimage attacks due to its sponge construction. This sponge-based design ensures that, for an output length n, the collision resistance is $\mathcal{O}(2^{n/2})$ and the preimage resistance is $\mathcal{O}(2^n)$. Hence, using SHA-3 for ID generation yields stronger guarantees against accidental collisions between snippets than SHA-256.

3.3 Modules (Responsibilities, Inputs, Outputs)

Project Parser. Extracts an initial view of the repository and, optionally, a snippet-level index used for retrieval. Parsers are language-aware but interface-compatible; one instance focuses on TypeScript snippets and produces symbolized regions and locations suitable for downstream retrieval.

Smell Detector. Maps files (or snippets) to a set of smells. Two families are supported: (i) rule/linter–based analysis that operates at project scope and emits structured findings, and (ii) learning-based detectors (binary or multi-class) that score code fragments using a lightweight classifier head over a transformer encoder with optional parameter-efficient adapters. Detectors expose the same interface and can be composed.

Planner. Transforms a pool of smells into an actionable order. It encodes prioritization policies (e.g., severity-first, risk-aware batching, dependency-aware ordering) but remains agnostic to the detector's internals. The output is a plan (list of smell IDs).

LLM Refactoring Agent. Given a target smell and retrieved context, the agent proposes a concrete edit (patch or replacement snippet). Context is constructed from the snippet index and nearby code; prompts may include detector messages and planner rationale.

Tester (Quality Gates). Applies a candidate edit in a transient workspace and validates with project-specific gates (build, tests, linters, type checks). Results (pass/fail plus diagnostics) drive either acceptance, re-prompting, or deferral.

Project Reconstructor. Materializes the accepted edits into a coherent repository. It supports both full reconstruction and incremental file updates to keep the working tree consistent across iterations.

Stores (Persistence). Lightweight stores persist the evolving project state (files/snippets), analysis artifacts (smells), and decisions (plans). Using opaque identifiers across stores decouples producers and consumers and enables plan replay and audit.

3.4 End-to-End Pipeline

Phase A: Project preprocessing. (1) The parser ingests the original repository into a file view and optional snippet index. (2) If a retrieval index is enabled, embeddings are computed and cached for later context selection.

Phase B: Refactoring loop. (1) The smell detector populates a pool of issues. Implementations can be linter-style (project-wide) or model-based scoring of code fragments. (2) The planner orders the smells into a plan. (3) For the current plan item, the system retrieves local and cross-file context (via the snippet index and asks the LLM agent to propose an edit. (4) The tester validates the edit. Passing edits update the working state; failing edits trigger guided re-prompts or are skipped based on policy. The loop proceeds until the plan is exhausted or a stopping rule is met.

Phase C: Project reconstruction. The reconstructor writes the accepted edits into a refactored repository, preserving structure and metadata. Because the loop operates on stable identifiers, reconstruction is deterministic given a plan and a sequence of accepted edits.

3.5 Design Principles

- **Protocol-first modularity.** Each box in Figure 1 corresponds to an interface with minimal assumptions about implementations, making detectors, planners, and LLMs interchangeable.
- **UID-centric tracking.** Files, snippets, and smells are referenced by opaque IDs, enabling precise traceability across modules and reproducible plan execution.
- **Tight feedback, loose coupling.** The tester provides fast, local feedback to the LLM agent, while persistence layers decouple long-running analyses from generation.
- **Retrieval-augmented edits.** A dedicated snippet index turns the pipeline into a context-aware system that scales to large repositories and reduces prompt ambiguity.

3.6 Agentic Orchestration (AO)

In the AO path an LLM-based agentic loop is activated for smells that require more nuanced or context-aware changes. However, this is still in very experimental stages and has not been fully implemented. Nevertheless, it will be discussed in more detail in section 7.

4 Smell Detection

4.1 Static detection with metrics

Our static smell detector combines ESLint with rule/threshold checks to surface anomalies that correlate with maintainability risks. It is particularly effective for TypeScript projects thanks to `@typescript-eslint` (parsing/rules) and `sonarjs` (cognitive/duplication) [6].

Core metrics considered.

- **Cyclomatic complexity** (`complexity`): independent control–flow paths per function.
- **Cognitive complexity** (`sonarjs/cognitive-complexity`): human-oriented effort to understand nested/control logic.
- **Size measures**: file/function LOC (`max-lines`, `max-lines-per-function`).
- **Structural thresholds**: `max-depth`, `max-params`, `max-statements`, `max-nested-callbacks`.

ESLint-based implementation. We execute ESLint across the repository and parse violations into `Smell` items with UID, severity, and precise source locations. The configuration enables a curated rule set oriented to structure, duplication, clarity, and safety.

Smell catalog detected (grouped by concern).

Structure & size

- **Long Method** — functions too large/complex to read or test
  Rules: `max-lines-per-function`, `complexity`, `max-statements`, `sonarjs/cognitive-complexity`.
- **Large File (proxy for Blob)** — files exceeding reasonable LOC; cohesion risk
  Rules: `max-lines`.
- **Long Parameter List** — routines with many parameters
  Rules: `max-params`.
- **Deep Nesting** — heavily nested `if`/`switch`/loops
  Rules: `max-depth`, `sonarjs/cognitive-complexity`.

- **Callback Hell** — nested callbacks harming linear reasoning
  Rules: `max-nested-callbacks`.

Control flow & logic

- **High Cyclomatic / Cognitive Complexity** — intricate control paths or mentally taxing logic
  Rules: `complexity`, `sonarjs/cognitive-complexity`.
- **Collapsible Conditionals** — nested/redundant conditionals
  Rules: `sonarjs/no-collapsible-if`.
- **Overengineered Small Switch** — trivial `switch` better expressed otherwise
  Rules: `sonarjs/no-small-switch`.
- **Redundant Control Flow** — unnecessary `else`, returns, or ternaries
  Rules: `no-else-return`, `no-useless-return`, `no-unneeded-ternary`.
- **Inverted/Confusing Boolean Checks** — non-idiomatic boolean handling
  Rules: `sonarjs/no-inverted-boolean-check`, `sonarjs/no-redundant-boolean`, `sonarjs/prefer-single-`
- **Nested Template Literals** — readability loss due to template nesting
  Rules: `sonarjs/no-nested-template-literals`.

Duplication & repetition

- **Duplicate Strings / Conditions / Functions** — repetition signalling copy–paste or divergence risk
  Rules: `sonarjs/no-duplicate-string`, `sonarjs/no-identical-conditions`, `sonarjs/no-identical-func`

Safety & dead code

- **Dead/Unsafe Logic** — unreachable code, constant conditions, duplicate cases, fall-through
  Rules: `no-unreachable`, `no-constant-condition`, `no-duplicate-case`, `no-fallthrough`.
- **Useless Catch** — catch blocks that add no value or hide issues
  Rules: `sonarjs/no-useless-catch`.

Readability & hygiene

- **Equality & Mutability Hygiene** — prefer immutability and strict equality
  Rules: `prefer-const`, `no-var`, `eqeqeq`.

Compact rules list: `complexity`, `max-lines`, `max-lines-per-function`, `max-params`, `max-statements`, `max-depth`, `max-nested-callbacks`, `sonarjs/cognitive-complexity`, `sonarjs/no-duplicate-string`, `sonarjs/no-identical-functions`, `sonarjs/no-identical-conditions`, `sonarjs/no-small-switch`, `sonarjs/no-collapsible-if`, `sonarjs/no-inverted-boolean-check`, `sonarjs/no-useless-catch`, `sonarjs/prefer-single-boolean-return`, `sonarjs/no-nested-template-literals`, `sonarjs/no-redundant-boolean`, `no-constant-condition`, `no-unreachable`, `no-else-return`, `no-unneeded-ternary`, `no-useless-return`, `no-duplicate-case`, `no-fallthrough`, `prefer-const`, `no-var`, `eqeqeq`.

4.2 Deep Learning detection

4.2.1 Architecture and models

We complement rule-based analysis with a learned detector that scores code snippets using a transformer encoder followed by a lightweight neural classifier. This enables (i) *binary* decisions (*smell* vs. *no-smell*) and (ii) *type-aware* decisions (multi-class or multi-label over specific smell categories). For parameter efficiency, the encoder is fine-tuned with Low-Rank Adapters (LoRA) [13].

Figure 2: PCA over the embeddings of the dataset. 1: Blob Class, 0: Not smelly.

4.2.2 Dataset

We train and evaluate on the **MLCQ** dataset (*Madeyski–Lewowski Code Quest*), a large, industry-relevant corpus manually annotated by professional developers. MLCQ contains *class-level* smells (Blob, Data Class) and *method-level* smells (Long Method, Feature Envy) taken from contemporary GitHub Java projects; each sample was reviewed on a four-level severity scale (*critical*, *major*, *minor*, *none*) [10]. The public release on Zenodo includes the primary annotations and reviewer background survey to enable stratified analyses [11].

Scope used here. In this work we only focus on one high-impact smell: **Blob** (*class-level*), often associated with low cohesion and excessive responsibilities.

Why MLCQ. MLCQ offers (i) *contemporary* code drawn from actively maintained repositories, (ii) *manual* annotations by practitioners (26 reviewers) on a standardized severity scale, and (iii) sufficient volume for modern learning methods. Prior work adopts MLCQ as a benchmark for smell detection (including Blob and Long Method), reinforcing its role as a reliable testbed for model comparison and ablation studies [12].

4.2.3 Fine Tuning Qwen-LLM

Base encoder. We use the **Qwen/Qwen1.5-1.8B** checkpoint as a lightweight transformer-based encoder for code. The model is instantiated via the `transformers` library from Hugging Face. All inference and fine-tuning operations are executed locally on a high-performance **NVIDIA RTX 4090 GPU**, allowing for fast experimentation and low-latency embedding generation.

Task head and pooling. Given tokenized code snippets X—wrapped with lightweight `<code>` tags for formatting consistency—the encoder generates contextual hidden

states $H \in \mathbb{R}^{T\times d}$, where $d = 2048$ is the embedding dimension. To obtain a fixed-size representation for each snippet, we apply last-token pooling, selecting the hidden state corresponding to the last non-padding token using the attention mask.

This pooled embedding is then passed through a lightweight multi-layer perceptron (MLP) head with GELU activation and dropout regularization to produce output logits suitable for classification. Our system supports three modes: (i) binary classification to detect the presence or absence of a code smell, (ii) multi-class classification to predict a single smell type from a predefined set, and (iii) multi-label classification for detecting multiple smells simultaneously within the same snippet.

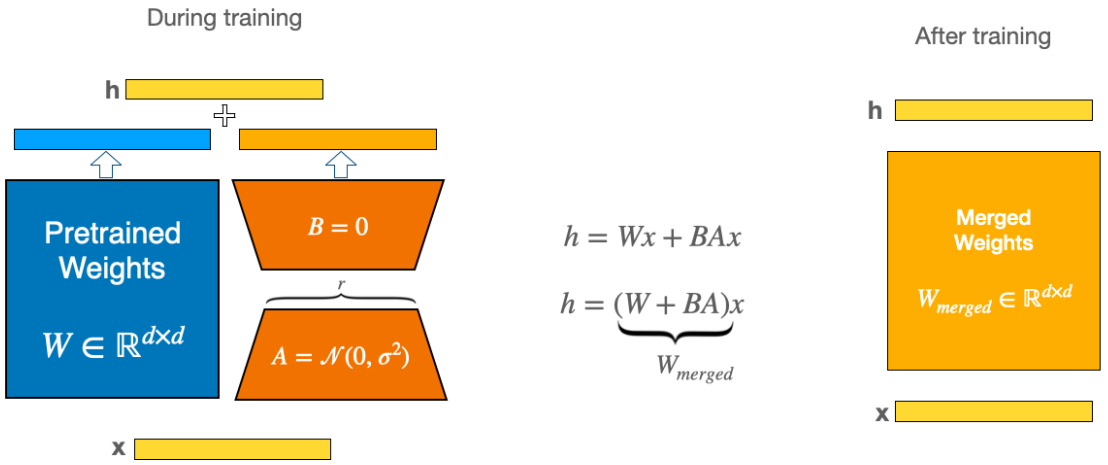


Figure 3: LoRa Diagram

Parameter-efficient adaptation (LoRA). (

To adapt the encoder at low cost, we insert low-rank adapters into attention projections (`q_proj`, `k_proj`, `v_proj`, `o_proj`) and MLP projections (`gate_proj`, `up_proj`, `down_proj`) [13]. For a base linear map W, LoRA learns $\Delta W = \frac{\alpha}{r} AB$ with rank $r = 32$ and scaling $\alpha = 64$ (dropout $p = 0.1$). Only (A, B) and the classifier are trainable; the pretrained weights remain frozen. This reduces trainable parameters to $\mathcal{O}(r(d_{\text{in}} + d_{\text{out}}))$ per adapted layer while preserving the backbone capacity.

Training recipe (our implementation).

- **Checkpoint & tokenizer:** `Qwen/Qwen1.5-1.8B` via `Qwen2Model` + `Qwen2TokenizerFast`; `pad_token` set to `eos` for stable padding.
- **Sequence length:** `MAX_LENGTH` = 1024 tokens; truncation with `padding='max_length'`.
- **Precision & memory:** FP16 forward; **AMP** for mixed precision; **gradient checkpointing** enabled on the encoder.
- **Optimization:** AdamW; `LR`= 3×10^{-4} for LoRA + head; `ENC_LR`= 5×10^{-6} for any unfrozen base params; weight decay 0.01; global grad clip 1.0.
- **Schedule:** cosine decay with 10% warmup via `get_cosine_schedule_with_warmup`.
- **Batching:** effective batch size = `BATCH_SIZE` × `ACCUM_STEPS` (here 8×4); epochs = 12.
- **Regularization:** dropout inside the head; label smoothing $\varepsilon = 0.1$ for BCE-with-logits targets.

- **Saving & selection:** we track validation F1 each epoch; the best state dict (by F1) is saved and reloaded.

Thresholding and calibration. For binary/multi-label settings, we sweep a threshold over raw logits on a held-out split and pick the value that maximizes F1 (reporting precision/recall curves alongside). For multi-class, we use arg max over softmax probabilities and optionally temperature-scale on validation.

Why Qwen + LoRA. Qwen1.5–Qwen2 checkpoints offer strong tokenization and transformer stacks for code/text while remaining light enough for rapid iteration (1.8B parameters). LoRA—and its quantized variant QLoRA—are established approaches for reducing memory and compute during adaptation with competitive quality, enabling us to fine-tune effectively under modest GPU budgets and to swap encoders with minimal code changes. Although the only experiments carried out are binary detectors (snippet has a certain smell/does not have it), in the future it would be advisable to set up a multi-class classifier with many types of smells and, of course, the class without smells.

5 Evaluation of Deep Learning smell detection

5.1 Binary separability with and without fine-tuning (embeddings only)

We quantify how well the two classes separate directly in the model *embeddings* (no PCA), using (i) cosine similarity and (ii) Euclidean distance. For each split (train/validation/test) we compute the mean intra-class (μ_{intra}) and inter-class (μ_{inter}) affinity and report a single separability score per split.

Cosine similarity. We define the cosine-based separability as

$$\Delta_{\cos} \;=\; \mu^{\cos}_{\text{intra}} - \mu^{\cos}_{\text{inter}},$$

so that larger values indicate that same-class embeddings are, on average, more similar than cross-class embeddings.

Table 2: Cosine separability $\Delta_{\cos}$ on embeddings (higher is better).

Split	No fine-tuning	With fine-tuning
Train	0.0628	0.0646
Validation	0.1994	0.1994
Test	0.1679	0.1681

Euclidean distance. We define the distance-based separability as

$$\Delta_{\text{euc}} \;=\; \mu^{\text{euc}}_{\text{inter}} - \mu^{\text{euc}}_{\text{intra}},$$

so that larger values indicate that cross-class pairs are, on average, farther apart than same-class pairs.

Table 3: Euclidean separability Δ_{euc} on embeddings (higher is better).

Split	No fine-tuning	With fine-tuning
Train	19.9393	20.0654
Validation	58.5305	58.5305
Test	50.1862	50.4839

Takeaways. Fine-tuning yields a *small but consistent* improvement in embedding-space separability. Cosine $\Delta_{\cos}$ increases on Train (+0.0018, $\sim 2.9\%$ relative) and Test (+0.0002, $\sim 0.1\%$), while Validation is unchanged. Euclidean Δ_{euc} shows analogous gains on Train (+0.1261, $\sim 0.6\%$) and Test (+0.2977, $\sim 0.6\%$), with Validation again unchanged. Overall, the margins move in the right direction but the effect size is modest, suggesting that additional data and/or stronger adaptation (e.g., larger LoRA ranks or relaxing more layers) may be required to substantially enlarge class gaps in the embedding space.

Importantly, while greater separability typically supports better global classification, our classifier is a flexible neural network. Fine-tuning can induce *local* decision-boundary adjustments that materially improve accuracy even when global separability metrics—based on mean intra/inter similarities—change only slightly. In other words, the model can locally adapt to task-relevant neighborhoods of the embedding manifold and exploit subtle structure that $\Delta_{\cos}$ and Δ_{euc} do not fully capture.“‘

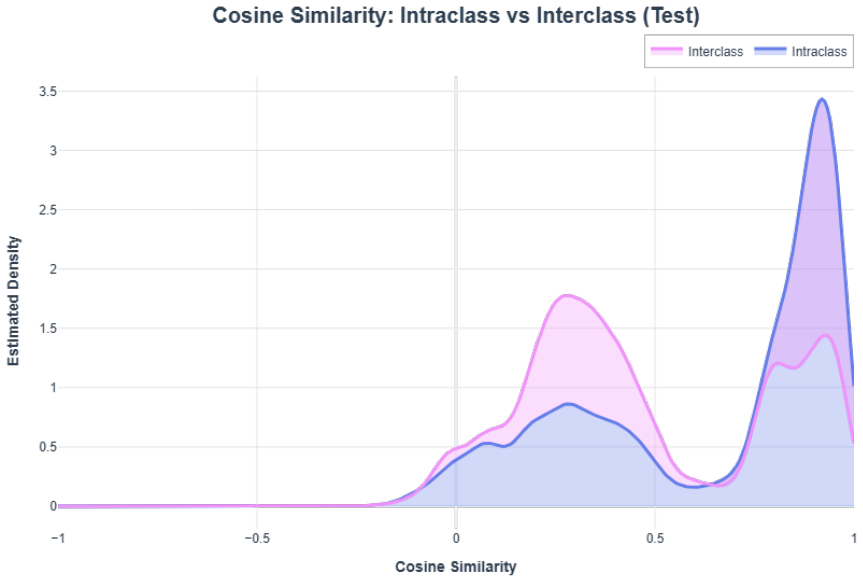
5.2 Is Fine-Tuning Worth It? (0.58 vs. 0.56 F1) and Comparison to Prior Work

Observed effect. With LoRA fine-tuning of a lightweight decoder-only transformer (Qwen1.5–1.8B) used as an encoder with last-token pooling and a small MLP head, our binary smell detector improves from **F1=0.56** (frozen backbone) to **F1=0.58** (fine-tuned). The absolute gain is Δ =0.02 ($\approx$3.6% relative).

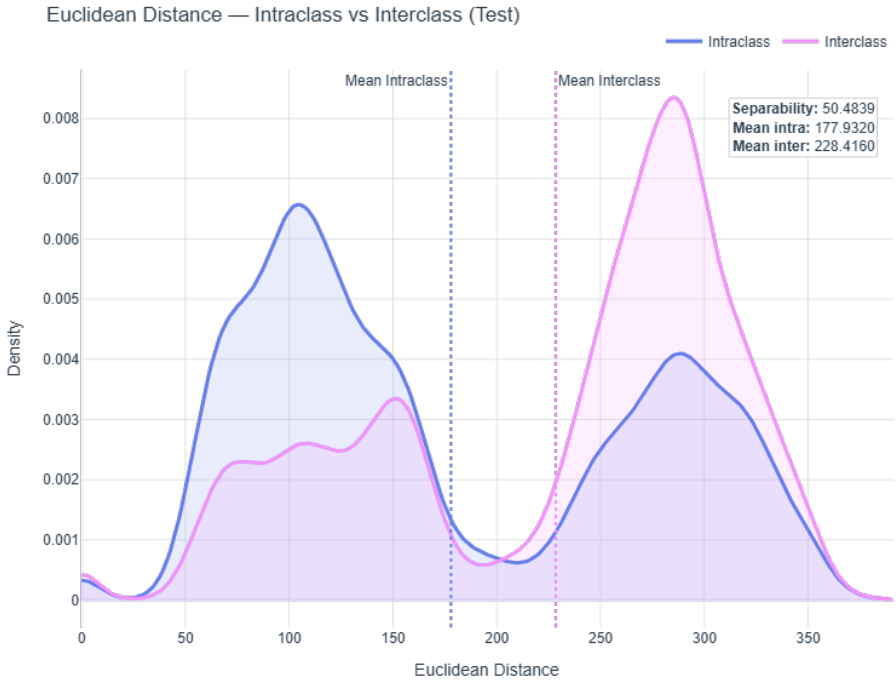
Significance. By itself, $\Delta = 0.02$ is modest and may or may not be statistically significant depending on sample size and class balance. Because F1 is a non-linear functional of the confusion matrix, a simple McNemar test on accuracy is not appropriate. A standard approach is a *paired, stratified bootstrap* over evaluation examples: resample the test set with replacement (e.g., 10,000 times), recompute F1 for both models on each bootstrap sample, and form a confidence interval for Δ from the empirical distribution of the difference between the two results. If the 95% CI excludes 0, the improvement is statistically significant; otherwise it should be reported as a directional but inconclusive gain. (A paired permutation test on per-example contributions to the confusion matrix is an acceptable robustness check.)

Why we likely do better than classical ML baselines. Using the same dataset and the same *proportions* for train/validation/test splits (though not the identical items in each split), we observe a substantially better result on the *class-level* smell than reported by prior work. In particular, on MLCQ the study of Kovačević *et al.* reports F1$\approx$0.53 for God Class (class-level) versus our **0.58** under comparable conditions, while their method-level Long Method reaches $\approx$0.75.[6] We attribute the class-level improvement primarily to:

[6]Split contents are not identical, so numbers are indicative rather than strictly comparable.



(a) Cosine similarity density (inter vs. intra) after fine-tuning



(b) Euclidean distance density (inter vs. intra) after fine-tuning

- **A stronger backbone:** we leverage the Qwen1.5–1.8B transformer (decoder-only) adapted with LoRA, which provides high-quality tokenization and representations for code with efficient adaptation.
- **End-to-end adaptation:** our MLP head is trained jointly with the adapters, allowing the representation to specialize for smell discrimination; in contrast, train *classical* ML models (RF/GBDT/SVM) on fixed embeddings or handcrafted metrics, which limits task-specific representation shaping.

Interpretation and reporting. Given the small margin, we recommend reporting (i) the bootstrap 95% CI for F1 and Δ; (ii) precision/recall with CIs; and (iii) calibration curves or threshold-sweeps to show that the gain is consistent across operating points. If the CI for Δ straddles 0, present the effect as *directionally positive but not statistically significant*, and highlight the stronger *practical* advantages (e.g., better calibration or robustness across projects) if observed.

Caveat on comparability. Results across studies using MLCQ should always note that we matched *split proportions* but did not reproduce the exact per-example splits; annotator disagreement at class level is also non-negligible in MLCQ, which can blur small margins.

5.3 Synthetic-Embedding Augmentation: Methods and Findings

We investigated whether augmenting the *embedding space* (produced by the fine-tuned Qwen encoder + MLP head) with synthetic samples could improve binary smell detection beyond the fine-tuned baseline (F1 $\approx$ 0.58 vs. 0.56 without fine-tuning). Below we summarize each method and its core formulation.

Table 4: Effect of synthetic-embedding augmentation on test performance.

Method	F1	Significant?
Baseline (fine-tuned, no aug.)	**0.58**	Yes
Random *down*sampling	0.56	No
Random *up*sampling	0.51	No
SMOTE	0.56	No
VAE (latent dim = 64)	0.53	No
Diffusion (steps = 1000)	0.579	No

5.3.1 Random Resampling (Down/Up)

Goal. Correct class imbalance and expose the classifier to a more balanced empirical risk.

Procedure. Let $\mathcal{D} = \{(x_i, y_i)\}$ be embedding/label pairs, with minority class $\mathcal{D}_m$ and majority class $\mathcal{D}_M$. Downsampling draws a subset $\tilde{\mathcal{D}}_M \subset \mathcal{D}_M$ to match $|\mathcal{D}_m|$; upsampling draws with replacement $\tilde{\mathcal{D}}_m$ to match $|\mathcal{D}_M|$. The empirical risk stays

$$\hat{\mathcal{L}} = \frac{1}{|\tilde{\mathcal{D}}|} \sum_{(x,y)\in\tilde{\mathcal{D}}} \ell\big(f_\theta(x), y\big),$$

but the effective class prior is altered to reduce bias.

5.3.2 SMOTE in Embedding Space

Goal. Create *interpolated* minority embeddings to densify the decision boundary without duplicating points.

Core rule. For a minority point x_i and one of its k nearest minority neighbors $x_i^{(nn)}$,

$$x_{\text{new}} \;=\; x_i \;+\; \lambda\big(x_i^{(nn)} - x_i\big), \qquad \lambda \sim \mathcal{U}(0,1).$$

Notes. We L_2-normalized embeddings; neighborhood search used cosine similarity (inner product on normalized vectors). Interpolation assumes local linearity of the embedding manifold, which may be only approximately true.

5.3.3 Variational Autoencoder (VAE) Synthesis

Goal. Learn a smooth, class-conditioned latent manifold of embeddings and sample diverse but plausible points.

Objective. With encoder $q_\phi(z|x)$ and decoder $p_\theta(x|z)$ (optionally conditioned on y), maximize the ELBO:

$$\mathcal{L}_{\text{VAE}}(\theta,\phi) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - \text{KL}\big(q_\phi(z|x)\,\|\,p(z)\big),$$

typically with $p(z) = \mathcal{N}(0, I)$ and a Gaussian decoder for continuous embeddings.

Sampling. $z \sim \mathcal{N}(0, I)$, then $\tilde{x} \sim p_\theta(x|z)$; optionally condition on class to draw $\tilde{x}$ for the minority.

5.3.4 Diffusion Models over Embeddings

Goal. Generate high-fidelity synthetic embeddings with better mode coverage than VAEs.

Forward (noising) process.

$$q(x_t \mid x_{t-1}) = \mathcal{N}\big(\sqrt{1-\beta_t}\,x_{t-1},\, \beta_t I\big), \quad t = 1, \ldots, T.$$

Reverse (denoising) objective. Train ε_θ to predict noise:

$$\mathcal{L}_{\text{DDPM}} = \mathbb{E}_{t,\varepsilon}\Big[\big\|\varepsilon - \varepsilon_\theta(x_t, t)\big\|_2^2\Big],$$

then sample x_{t-1} from x_t using the learned mean/variance schedule.

5.3.5 Statistical Assessment

We compared each augmented model to the fine-tuned baseline via the *difference in F1*, $\Delta\text{F1} = \text{F1}_{\text{aug}} - \text{F1}_{\text{base}}$, and computed a 95% bootstrap CI by resampling test instances with replacement (B replicates). We deemed improvements *significant* if the CI did not include 0. For multiple methods, we controlled the familywise error via Holm–Bonferroni.

Conclusion. Across random resampling, SMOTE, VAE, and diffusion augmentations applied directly in embedding space, test F1 scores approached but did not surpass the fine-tuned baseline in a statistically reliable way (Table 4). This suggests either (i) further hyperparameter exploration (e.g., neighborhood size, latent dimension, conditioning strength, synthetic-to-real mixing ratios) is required to unlock benefits, or (ii) the current embedding geometry constrains how much synthetic data can help without additional representation learning (e.g., stronger contrastive/metric objectives or class-conditional priors).

6 LLMs for Snippet Refactoring

In the early stages of development, we explored the use of local large language models (LLMs) for code snippet refactoring, leveraging a workstation equipped with an NVIDIA RTX 4090 GPU (24GB VRAM). This allowed us to experiment with several decoder-only and encoder-based architectures optimized for code understanding and generation.

Among the models tested locally were:

- **Code LLaMA**: an open-weight family of LLMs trained specifically for programming tasks, offering robust performance on generation and completion benchmarks.
- **GraphCodeBERT**: a graph-aware encoder that integrates data-flow information to enhance representation learning, particularly useful for tasks like smell detection and code search.
- **Qwen-7B**: a strong general-purpose decoder model from Alibaba, tested here in a code refactoring context and also fine-tuned using LoRA to support binary and type-aware smell detection.

While these local models offered full control over data and inference behavior, they also introduced significant complexity in terms of resource management, prompt tuning, and scaling.

These hosted models have also been evaluated on challenging bug-fixing benchmarks such as SWE-bench [9], showing that current generation LLMs can already resolve non-trivial, real-world GitHub issues.

In the later stages of the project, we transitioned to using an API-based approach through Mistral's hosted platform, which provides access to multiple specialized LLMs. These models were used specifically for the generation of refactored, *non-smelly* code candidates based on context and planner feedback.

The models available through the Mistral API included:

- **Mistral 3B**: a compact and efficient model for basic editing and scaffolding.
- **Mistral Small 2503**: a refined version with improved language understanding, well-suited for structured code transformation.
- **Mistral Medium 2505**: a more capable model offering better semantic coherence and performance on code reasoning tasks.

7 Agents

7.1 Background: Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) combines a parametric language model with a non-parametric retriever: given a user goal, a retriever selects relevant chunks from an external corpus (codebase, docs, logs), which are then provided as grounded context to the LLM to produce the final answer. This architecture improves factuality, lets systems use up-to-date knowledge without re-training, and provides traceability to sources.

In practice, the retriever is a vector index over embeddings of the corpus; queries are embedded in the same space, and nearest-neighbor search returns the top-k passages. We used FAISS for efficient similarity search (flat and IVF variants), which

supports both L_2 and inner-product metrics. *FAISS documentation.* When all vectors are L_2-normalized, the inner product equals cosine similarity.

7.2 Early experiment: automatic code documentation (agent prototype)

Because full, in-place refactoring is invasive (requires compiling, testing, and strict quality gates), we first explored a simpler but related subject of study: **automatic code documentation**. This task is non-destructive (read-only over the source tree) yet still stresses retrieval quality and LLM instruction following.

We built a *sequential agent* with LangChain whose two Tools were:

1. **RAG**: query embedding → FAISS index → retrieve top-k code snippets.
2. **Markdown writer**: synthesize a final, structured document.

The agent was instructed to *ask itself questions* about the target module (API, responsibilities, data flow, edge cases), use the RAG tool to fetch the exact snippets needed to answer each question, and then assemble a coherent Markdown report. Retrieval used FAISS with *inner product* over L_2-normalized embeddings (i.e., cosine similarity), which we selected for its robustness in high dimensions. *LangChain agent/tooling references. FAISS references.* The qualitative results of this prototype are included in the *Results* section of this report.

Although this experiment was at an earlier stage and not integrated into the PoC, it showed promising behavior: clear sectioning, accurate cross-references to retrieved code, and maintainable output formatting (Markdown).

7.3 Toward a refactoring agent with LangGraph

For the **LLM-driven refactoring** path, we propose moving from classic LangChain agents to **LangGraph**, a stateful orchestration framework designed for controllable, long-running, multi-tool (and multi-agent) workflows. LangChain now recommends LangGraph for new agents, and LangGraph provides graph-structured control flow, persistent state, human-in-the-loop hooks, and recovery policies—features that are valuable for iterative code changes.

A refactoring agent in LangGraph would wire the following Tools (nodes) into a controlled loop:

- **Smell Detector**: static rules + learned classifiers to propose targets.
- **Planner**: prioritize issues, batch by risk/impact, and order dependencies.
- **Context Retriever (RAG)**: fetch local and cross-file context (snippets, call graph).
- **Proposer**: generate candidate patches/snippets conditioned on smells and context.
- **Tester / Quality Gates**: compile, run tests/linters/type checks; return diagnostics.
- **Repository IO**: apply/rollback patches in a sandbox; persist decisions and diffs.
- **Memory/Trace**: persist intermediate state to enable retries and auditability.

LangGraph's explicit state machine (graph) allows us to branch on validation outcomes (accept, re-prompt with diagnostics, or defer), throttle risky edits, and maintain reproducibility across iterations—capabilities that are difficult to guarantee with ad-hoc prompting alone.

Summary. The agent track remains exploratory and was not merged into the PoC, yet the documentation-agent prototype delivered encouraging results. The next step is to promote the refactoring loop to a LangGraph application with strict quality gates and RAG-grounded context, leveraging FAISS-backed retrieval and agentic control to make edits that are safe, auditable, and reproducible.

8 Visual Studio Code extension

During this period I also learned JavaScript and, with the help of another Infotel intern, we've begun implementing a VS Code extension for the Infotel team. The plan is to integrate it via a button that opens a tree view in the left panel showing all detected smells; any smelly code will be highlighted with a red background for easy identification. Although we've only just started development, this is the vision of what the tool will offer once it's finished.

9 Log Analysis and Error Tracing System Design

During the final phase of the internship at Infotel, a company specializing in providing IT solutions to major financial institutions, we were tasked with drafting a *cahier des charges* for a potential future internship project. The objective of this project is to automatically analyze logs in order to detect and trace the root causes of application errors occurring within a banking client's infrastructure.

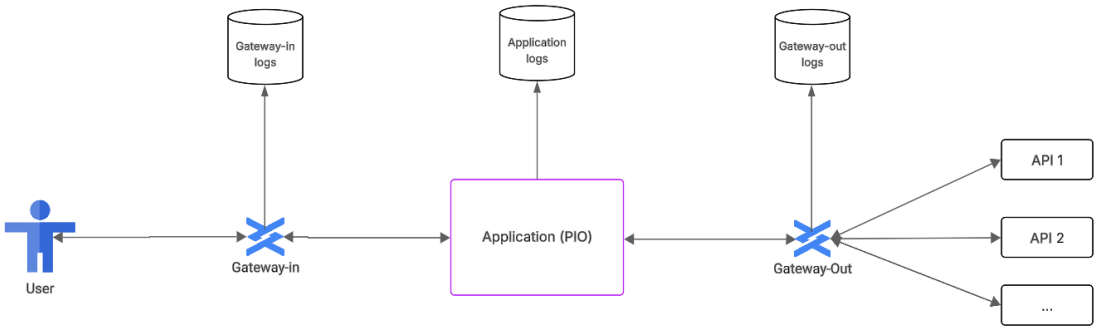


Figure 5: Gateway Logs Diagram

As illustrated in Figure 5, the application system is composed of three main log generators: **Gateway-in**, **Application (PIO)**, and **Gateway-out**. Each of these components generates dedicated logs—namely, *Gateway-in logs*, *Application logs*, and *Gateway-out logs*—which can be correlated using *correlation IDs* embedded in each request. These logs are crucial for reconstructing the complete execution flow and identifying the point of failure when an error arises.

The *cahier des charges* outlines two possible directions for the solution:

1. **Classical algorithmic approach**: Constructing directed log graphs where each node represents a log entry and edges represent causal relationships inferred via correlation IDs. By applying traversal algorithms or anomaly detection techniques on these graphs, it becomes possible to localize error origins and propagate impact zones.
2. **Advanced multi-agent system**: If necessary, a more sophisticated architecture could be implemented using AI agents equipped with Retrieval-Augmented Generation (RAG) capabilities, leveraging frameworks like LangChain or LangGraph. These agents would be specialized in tasks such as database interaction, log parsing, or natural language interfacing, allowing users to query the system in plain English or French (e.g., "What caused error 500 on request X?").

This specification serves as a foundational document to guide the development of a robust and scalable system for error diagnosis and traceability. It reflects both Infotel's commitment to innovation and the high standards expected by its banking clients.

10 Conclusion

This internship set out to explore automatic code refactoring driven by a hybrid of static analysis and modern deep learning, and to translate those ideas into a developer-friendly prototype. The resulting system combines (i) ESLint-based metric and rule checks for fast, explainable detection, (ii) a learned smell detector built on a lightweight transformer adapted with LoRA, and (iii) an iterative loop that proposes small, semantics-preserving edits guarded by quality gates. The architecture is deliberately modular—protocol-first, UID-centric, and retrieval-augmented—so that detectors, planners, and LLM backends can evolve independently.

What we achieved.

- A sequential modular pipeline (SMP) and data artifacts (files/snippets, smells, plans, patches) that enable reproducible refactoring loops and plan replay.
- A practical static detector for TypeScript (ESLint + `@typescript-eslint` + `sonarjs`) covering structure, control flow, duplication, and hygiene.
- A learned detector for the Blob/Class-level smell trained on MLCQ, with fine-tuning improving F1 from 0.56 to 0.58 and modest—but consistent—embedding separability gains.
- A first VS Code integration concept to surface findings and staged edits inside the IDE, aligning with developer-in-the-loop ergonomics.
- An agentic pathfinder: a documentation agent (RAG + FAISS) that validated retrieval quality and prompted the migration toward LangGraph for controllable refactoring workflows.
- A *cahier des charges* for a follow-on project on log analysis and error tracing in banking systems, outlining both a classical graph approach and an AI multi-agent alternative.

Key lessons. Static analysis and learning are complementary: rules provide high-precision, actionable anchors and cheap guardrails, while learned models help generalize beyond hand-coded patterns. Retrieval is not optional—scoped, snippet-level

context reduces hallucinations and improves patch locality. Finally, synthetic data in *embedding space* (SMOTE, VAE, diffusion) did not reliably surpass the fine-tuned baseline, suggesting that representation learning, not resampling, is the main bottleneck at current scale.

Limitations. The prototype focuses on local, single-file edits and one high-impact smell (Blob). Evaluation is constrained to TypeScript and to MLCQ for the learned component. Statistical significance of small F1 gains depends on dataset size and stratification; broader, project-level benchmarks with paired bootstrap CIs are needed. Safety currently relies on compilation/tests/linters; absence of tests increases conservatism and reduces coverage.

Future work.

- **Multi-file reasoning & consistency:** cross-reference analysis (call/dep graphs), global symbol rewrites, and patch sets that preserve architectural invariants.
- **Agentic orchestration with LangGraph:** explicit state, retries with diagnostic feedback, human checkpoints, and policy-driven risk throttling.
- **Broader smell coverage & languages:** extend to Long Method, Feature Envy, Data Class; add Python/Java adapters with shared I/O contracts.
- **Stronger learning objectives:** contrastive/metric learning, curriculum over difficulty, and calibration; revisit augmentation at the *source* level rather than in embedding space.
- **End-to-end IDE workflow:** deterministic diffs, one-click accept/rollback, telemetry for UX, and CI hooks for gates in real repositories.
- **Log analysis project:** instantiate the specified graph-based pipeline first (correlation-ID causal chains, anomaly scoring), then layer RAG/agents for interactive diagnosis.

In sum, this work delivers a principled foundation and a functioning path to AI-assisted refactoring: modular where it must be, conservative where it matters, and ready to scale from snippet-level edits toward repository-wide, auditable transformations—while opening a second, adjacent line on automated error tracing for production systems.

11 Acknowledgements

I thank Frolo and Josua for their guidance and vision.

References

[1] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. *The Impact of AI on Developer Productivity: Evidence from GitHub Copilot.* arXiv:2302.06590, 2023.

[2] Ya Gao and GitHub Customer Research. *Research: Quantifying GitHub Copilot's Impact in the Enterprise with Accenture.* GitHub Blog, 13 May 2024. URL: https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-in-the-enterprise-with-accenture/.

[3] Amazon Web Services. *Amazon Q Developer and Amazon CodeWhisperer Documentation*. 30 Apr 2024. URL: https://docs.aws.amazon.com/toolkit-for-vscode/latest/userguide/amazonq.html.

[4] Microsoft. *Extension API — Visual Studio Code Documentation*. Accessed 7 Aug 2025. URL: https://code.visualstudio.com/api.

[5] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[6] SonarSource. *Understanding Measures and Metrics*. SonarQube Docs v10.8, accessed 7 Aug 2025. URL: https://docs.sonarsource.com/sonarqube-server/10.8/user-guide/code-metrics/metrics-definition/.

[7] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang *et al. GraphCodeBERT: Pre-training Code Representations with Data Flow*. arXiv:2009.08366, 2020.

[8] Meta AI. *Faiss — Library for Efficient Similarity Search*. Online documentation, accessed 7 Aug 2025. URL: https://faiss.ai.

[9] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* arXiv:2310.06770, 2023.

[10] Leszek Madeyski and Tomasz Lewowski. *MLCQ: Industry-Relevant Code Smell Data Set*. In *Proceedings of the Evaluation and Assessment in Software Engineering (EASE)*, 2020. PDF: https://madeyski.e-informatyka.pl/download/MadeyskiLewowski20EASE.pdf.

[11] Leszek Madeyski and Tomasz Lewowski. *MLCQ: Madeyski–Lewowski Code Quest Dataset*. Zenodo record (all versions), 2020. URL: https://zenodo.org/records/3666840.

[12] Various Authors. *Recent Studies Leveraging MLCQ for Long Method and Blob Detection*. Software Engineering Research, 2022. Note: discusses the benchmark status of MLCQ as a standard dataset.

[13] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. *LoRA: Low-Rank Adaptation of Large Language Models*. arXiv:2106.09685, 2021. URL: https://arxiv.org/abs/2106.09685.

12 Example of Automatic Refactoring in a Real Repository

This section presents a real example of exported logs from the software while refactoring a file in a repository. In the original file, the same symbol appeared multiple times (37 occurrences): `icon: 'nav-icon-bullet'`. The system detected this redundancy and replaced it with a variable `const ICON_BULLET = 'nav-icon-bullet'`, improving the maintainability of the code.

RefactorIA version v0

Session started at: 2025-06-16 14:40:48

Cloning repo coreui...

From https://github.com/coreui/coreui-free-angular-admin-template.git

Cloned to /home/admindev/project/IA-Refactorisation/data/raw/coreui

Using local repo coreui

From /home/admindev/project/IA-Refactorisation/data/raw/coreui

Creating CodeStore

📄 First File Uploaded: `/home/admindev/project/IA-Refactorisation/data/raw/coreui/src/declarations.d.ts`

🔍 Executing the Smell Detector...

✅ 94 smells detects and saved in `smells.json` 📊 Total smells: 94 • Severidad 1: 94

Creating a Plan...

✅ Plan created with 94 steps ordered by severity.

🎯 First UID to tratar: f1f0bd556698

Reconstructing Proyect...

✅ Project reconstructed in /home/admindev/project/IA-Refactorisation/data/reconstructed/coreui

Loading Local LLM...

Qwen/Qwen2.5-Coder-14B-Instruct loaded...

New Cycle Start

Next Smell To Refactor

🧪 Code Smell Detected

- **Rule**: `sonarjs/no-duplicate-string`
- **Message**: Define a constant instead of duplicating this literal 37 times.
- **Severity**: 1
- **Location**: `/home/admindev/project/IA-Refactorisation/data/raw/coreui/src/app/layout/default-layout/_nav.ts`
  - Line: 40
  - Column: 15

Current file

📄 File Summary

- **UID**: `64a917c7a5a9d60a`
- **Path**: `/home/admindev/project/IA-Refactorisation/data/raw/coreui/src/app/layout/default-layout/_nav.ts`

```
import { INavData } from '@coreui/angular';

export const navItems: INavData[] = [
  {
    name: 'Dashboard',
    url: '/dashboard',
    iconComponent: { name: 'cil-speedometer' },
    badge: {
      color: 'info',
      text: 'NEW'
    }
  },
  {
    title: true,
    name: 'Theme'
  },
  {
    name: 'Colors',
    url: '/theme/colors',
    iconComponent: { name: 'cil-drop' }
```

```
    },
    {
      name: 'Typography',
      url: '/theme/typography',
      linkProps: { fragment: 'headings' },
      iconComponent: { name: 'cil-pencil' }
    },
    {
      name: 'Components',
      title: true
    },
    {
      name: 'Base',
      url: '/base',
      iconComponent: { name: 'cil-puzzle' },
      children: [
        {
          name: 'Accordion',
          url: '/base/accordion',
          icon: 'nav-icon-bullet'
        },
        {
          name: 'Breadcrumbs',
          url: '/base/breadcrumbs',
          icon: 'nav-icon-bullet'
        },
        {
          name: 'Cards',
          url: '/base/cards',
          icon: 'nav-icon-bullet'
        },
        {
          name: 'Carousel',
          url: '/base/carousel',
          icon: 'nav-icon-bullet'
        },
        {
          name: 'Collapse',
          url: '/base/collapse',
          icon: 'nav-icon-bullet'
        },

        ...


      ]
    },
    {
      title: true,
      name: 'Links',
      class: 'mt-auto'
    },
    {
      name: 'Docs',
```

```
    url: 'https://coreui.io/angular/docs/',
    iconComponent: { name: 'cil-description' },
    attributes: { target: '_blank' }
  }
];
```

Refactorized file

```
import { INavData } from '@coreui/angular';

const ICON_BULLET = 'nav-icon-bullet';

export const navItems: INavData[] = [
  {
    name: 'Dashboard',
    url: '/dashboard',
    iconComponent: { name: 'cil-speedometer' },
    badge: {
      color: 'info',
      text: 'NEW'
    }
  },
  {
    title: true,
    name: 'Theme'
  },
  {
    name: 'Colors',
    url: '/theme/colors',
    iconComponent: { name: 'cil-drop' }
  },
  {
    name: 'Typography',
    url: '/theme/typography',
    linkProps: { fragment: 'headings' },
    iconComponent: { name: 'cil-pencil' }
  },
  {
    name: 'Components',
    title: true
  },
  {
    name: 'Base',
    url: '/base',
    iconComponent: { name: 'cil-puzzle' },
    children: [
      {
        name: 'Accordion',
        url: '/base/accordion',
        icon: ICON_BULLET
      },
```

```
        {
          name: 'Breadcrumbs',
          url: '/base/breadcrumbs',
          icon: ICON_BULLET
        },
        {
          name: 'Cards',
          url: '/base/cards',
          icon: ICON_BULLET
        },
        {
          name: 'Carousel',
          url: '/base/carousel',
          icon: ICON_BULLET
        },
        {
          name: 'Collapse',
          url: '/base/collapse',
          icon: ICON_BULLET
        },
        {
          name: 'List Group',
          url: '/base/list-group',
          icon: ICON_BULLET
        },

        ...


      ]
    },
    {
      title: true,
      name: 'Links',
      class: 'mt-auto'
    },
    {
      name: 'Docs',
      url: 'https://coreui.io/angular/docs/',
      iconComponent: { name: 'cil-description' },
      attributes: { target: '_blank' }
    }
  ];
```

Eventual Errors in new code

```
- None
```