International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 792-798
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0113

# AURORA – An AI-Augmented Code Refactoring Advisor

*K. Kavya[1], K. Lakshmi Varaprasad Chakri[2], M. Renuka[3], T. Dhanush Kumar[4], K. Rajya Guru Sai Sri[5]*
*[1,2,3,4]UG – Computer Science and Engineering, SRK Institute of Technology, Vijayawada, Andhra Pradesh*
*[5] Assistant Professor, Computer Science and Engineering, SRK Institute of Technology, Vijayawada, Andhra Pradesh, India*
***Emails:*** *kavyakondaveeti2023@gmail.com[1], kodidasuchakri2546@gmail.com[2], renukamaddi99@gmail.com[3], dhanushtummala10@gmail.com[4], kancharagunta1998@gmail.com[5]*

## Abstract
*Modern software systems often suffer from poor readability, high complexity, and increased technical debt due to inefficient coding practices. Manual code refactoring is time consuming and highly dependent on developer expertise, while existing automated tools are mostly rule-based and lack contextual understanding. To address these limitations, this paper presents AURORA, an AI-Augmented Code Refactoring Advisor that combines static program analysis with transformer-based artificial intelligence models to deliver safe and optimized refactoring suggestions. The proposed system employs Abstract Syntax Tree (AST) analysis to structurally understand source code and identify refactoring opportunities without altering program semantics. AI augmentation is applied to improve code quality by reducing redundancy and enhancing readability while preserving syntactic and logical correctness. A Python-based FastAPI backend performs code analysis and refactoring, while a Visual Studio Code extension provides real-time interaction for developers. Experimental evaluation on sample Python programs demonstrates effective reduction in code complexity and improved maintainability. AURORA offers a reliable and developer-friendly solution for intelligent code refactoring.*

***Keywords:*** *Code Refactoring; Static Code Analysis; Abstract Syntax Tree; Artificial Intelligence; Software Maintainability*

## 1. Introduction
Software maintainability usually gets worse over time as quick fixes and feature additions introduce **code smells** - structural weaknesses in the source that makes complicative for user to understand. For example, a *Long Method* (excessive lines of code in one function) or *Nested Ifs* (deeply nested conditional statements) make maintenance harder. These smells are recognized as symptoms of technical debt. Industry-standard tools like SonarQube and PMD scan code for such patterns, but they rely on handcrafted rules applied to metrics (e.g. number of lines, cyclomatic complexity). Consequently, they cannot reason about what the code *means*-only how it is structured. This leads to many false positives, inconsistent alerts across tools, and no automated fixes. Recent advances in **AI for code** suggest a path beyond rigid rules. Pre-trained models like CodeT5 have shown strong performance on code understanding and generation tasks by leveraging semantic cues in the code. AURORA harnesses this capability: it uses Python's AST parser to detect potential smells, then invokes CodeT5 to perform semantics-driven refactoring. For instance, where a static analyzer might flag a long method, AURORA can actually suggest an *Extract Method* refactoring [3], splitting the code into well-named functions. Our contributions are:

- an end-to-end architecture combining a VS Code UI with a FastAPI ML backend;
- integration of AST-based smell detection with a CodeT5 refactoring engine;
- a conceptual database design (Users, Projects, RefactoringHistory) for tracking usage and facilitating future CI/CD integration and
- a comparative discussion showing how AURORA addresses the limitations of static analyzers.

## 2. Related Work
Static analysis tools such as **SonarQube**, **PMD**, and **Checkstyle** have long played a central role in

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 792-798
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0113

enforcing software quality through syntactic rule enforcement. These tools convert source code into abstract syntax trees (ASTs) and apply rule-based detectors to flag issues such as unused variables, deeply nested blocks, or naming violations. While they are efficient and scalable, their effectiveness is often limited to surface-level issues that can be defined through deterministic patterns [5]. As Fowler et al. emphasize in their foundational work on refactoring [3], addressing deeper design flaws such as code smells requires structural improvements that go beyond formatting or variable declarations. Static analyzers are not equipped to interpret higher-level design intent or offer intelligent improvement suggestions. Pecorelli et al. [2] highlight the challenges in traditional smell detection. Their study reveals that smell detectors suffer from inconsistencies across tools, require manually tuned thresholds, and lack objective assessment. For example, what one tool flags as a "Long Method" might be ignored by another, leading to a lack of trust among developers. These limitations have motivated the search for more robust, learning-based techniques. Recent progress in **machine learning for code** has introduced models that can better represent and reason about program semantics. Earlier efforts focused on using classifiers trained on handcrafted features (e.g., method length, nesting depth) to detect smells, but these approaches often lacked generalizability. The emergence of large-scale **pre-trained models for code**, such as **CodeBERT**, **GraphCodeBERT**, and **CodeT5**, marked a shift toward semantic understanding of source code. These models leverage both textual and structural signals from code and outperform prior approaches on code-related NLP tasks. Among these, **CodeT5** stands out as an identifier-aware, encoder-decoder Transformer designed specifically for code understanding and generation [1]. It combines structural tokenization with naming context, allowing it to generate meaningful and syntactically valid code suggestions. While CodeT5 has been applied to tasks like code summarization, translation, and bug detection, its application to **refactoring assistance** remains relatively unexplored. AURORA builds on these insights by bridging the gap between static analysis and deep learning. Rather than replacing traditional analyzers, it augments them by embedding an AI-powered code agent into the developer workflow. Implemented as a VS Code extension backed by a FastAPI server [4], AURORA performs AST-based smell detection and provides contextual suggestions informed by transformer models. This hybrid approach aims to reduce technical debt through **semantic refactoring recommendations**, bringing together the precision of static analysis with the adaptability of machine learning.

## 3. System Architecture and Methodology

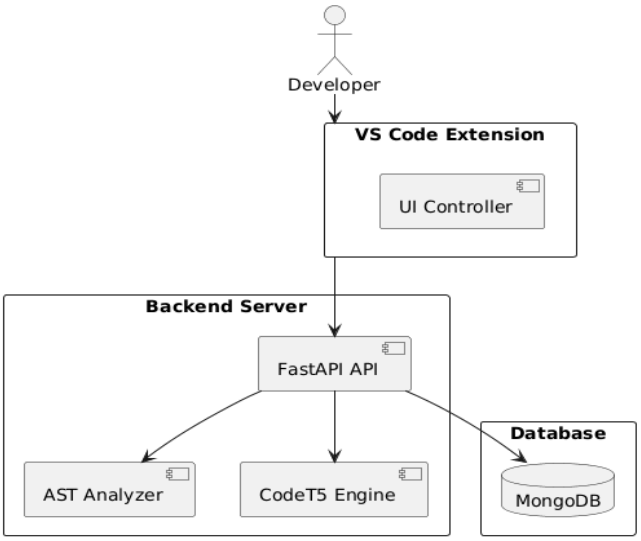


**Figure 1 AURORA System Architecture – Extension + Backend + Model flow**

Figure 1 shows AURORA's layered architecture. On the client side, a **VS Code extension** (written in TypeScript) provides the user interface. It highlights detected code smells in real time and displays refactoring suggestions interactively. When activated by the user, the extension sends the current code (or project) to the backend via REST API calls. The backend is implemented in Python using **FastAPI [4]**, a modern high-performance web framework. This API server handles requests to analyze code or apply refactorings. Upon receiving code, the server first invokes an **AST-based static analyzer**: using Python's ast module (or language-appropriate parsers), it builds an Abstract Syntax Tree and scans for patterns indicating smells. For example, it checks method length, nested control-flow depth, or other smell heuristics. Detected smells and their source

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 792-798
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0113

code ranges are returned to the extension for UI annotation. If the user chooses to refactor a flagged segment, the backend then engages the **CodeT5 model**. Relevant code (e.g. the body of a long method) is packaged into a prompt and fed into a CodeT5 encoder-decoder Transformer. CodeT5 has been pre-trained and fine-tuned for code tasks. It generates a refactored version of the code, such as splitting a method or simplifying conditionals. The backend parses this output into a syntactically correct refactoring suggestion and returns it to the extension. The user can review the suggested change before it is applied. Throughout this flow, AURORA logs activity to a conceptual **database** for scalability. This database enables future features such as multi-user support, historical trend analysis, and integration into CI pipelines. *Implementation of Methodology:* AURORA's methodology is illustrated by this workflow:

- **Trigger Analysis** – The user opens code in VS Code. AURORA's extension calls the FastAPI /analyze endpoint with the source code.
- **AST Parsing and Detection** – The server parses the code into an AST (leveraging libraries such as Python's built-in ast). It walks the AST to detect smell patterns: e.g. if a function node has >50 lines (Long Method) or contains nested If nodes beyond a threshold (Nested Ifs).
- **Report Smells** – The server returns a JSON list of found smells. The extension highlights these in the editor (e.g. underlining a long method, shading nested blocks).
- **Refactor Request** – When the developer requests a refactoring, the extension sends the targeted code snippet to the /refactor endpoint.
- **CodeT5 Refactoring** – The backend encodes the snippet using CodeT5 and decodes a transformed snippet. The model's output is post-processed to ensure compilable code and align naming.
- **Apply Suggestion** – The refactored code is sent back to the extension. The user can accept it (which replaces the code) or discard it. The action is logged in the RefactoringHistory table.

In this way, AURORA realizes **semantic refactoring**: the model uses learned representations to propose changes that respect code intent, while the AST analysis anchors detection in reliable syntactic structure.

## 4. Implementation Details

AURORA was implemented using a modular architecture that separates the UI, analysis logic, AI-based suggestion engine, and persistence layer. This design supports extensibility and scalability, and enables the integration of modern ML components alongside traditional AST parsing.

### 4.1. VS Code Extension (Frontend)

The frontend is implemented in **TypeScript** using the official **VS Code Extension API**. It performs the following tasks:

- Listens for commands like "Analyze Code" or "Refactor Selection"
- Collects the code segment and sends it to the FastAPI backend via REST
- Receives code smell detections and suggestions
- Renders diagnostics and interactive buttons using the editor API

### 4.2. Backend API (FastAPI Server)

The backend is developed using **FastAPI** (Python 3.9+). It exposes the following key endpoints:

- /analyze: Accepts code and returns AST-based smell detections
- /refactor: Sends a prompt to the AI model and returns generated suggestions
- /validate: Performs syntax validation on generated suggestions (optional)

FastAPI was chosen for its high performance, type hinting support, and auto-generated OpenAPI documentation.

### 4.3. Static Code Smell Analyzer

This module uses Python's ast module to parse and analyze code. It implements custom subclasses of ast.NodeVisitor for:

- **Long Method Detection**: Flags methods with more than *N* lines of code
- **Nested If Detection**: Flags methods with multiple levels of conditional nesting

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 792-798
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0113

- **Print Statement Detection**: Flags use of print() in favor of logging.info()

This modular detection logic can be extended to support Java/C++ using language-specific parsers like javalang or libclang.

### 4.4. AI Refactoring Engine

The core refactoring intelligence in AURORA is powered by the **CodeT5 Transformer**, loaded using the **Hugging Face Transformers** library.

- The model is prompted with comments like: "Refactor this function to remove code smell: Long Method"
- The prompt and code are encoded and passed to the model's encoder-decoder stack
- The decoded output is returned and displayed in the VS Code editor

**CodeT5** was selected due to its support for identifier-aware modeling, making it well-suited for semantic transformations of code.

### 4.5. Validation and Post-Processing

Once the model outputs a suggestion, AURORA optionally validates the syntax using:

- Python's built-in compile() for AST sanity check
- Optional formatting via black or autopep8

These steps ensure that generated suggestions are not only semantically meaningful but also syntactically valid.

### 4.6. Database Design

The following table shows the schema designed to capture user sessions, project references, and refactoring history. The implementation uses MongoDB for flexibility, but the schema remains compatible with relational designs.

**Table 1.** Conceptual Database Schema Used in AURORA

| Field | Type | Description |
|---|---|---|
| user_id | string | Primary key of the user |
| email | string | User's login or identity |
| project_id | string | Foreign key referencing the project |
| repo_link | string | GitHub or GitLab repository URL |
| refactor_id | string | Unique ID for each refactoring instance |
| smell_type | string | Type of detected code smell (e.g., Long Method) |
| timestamp | datetime | Timestamp of when refactoring occurred |

**Storage Considerations:** The Users collection stores user profiles and preferences. Projects links each user to one or more repositories. RefactoringHistory logs each suggestion generated and optionally accepted, supporting future dashboards or analytics. In this setup, the frontend remains lightweight, and heavy operations are deferred to the backend model, allowing scalability and minimal resource consumption on the user machine. The modular architecture also enables future extension to other IDEs beyond VS Code.

## 5. Implementation Details

This figure (Figure 2) demonstrates how users can directly interact with code smells via the right-click context menu inside VS Code.

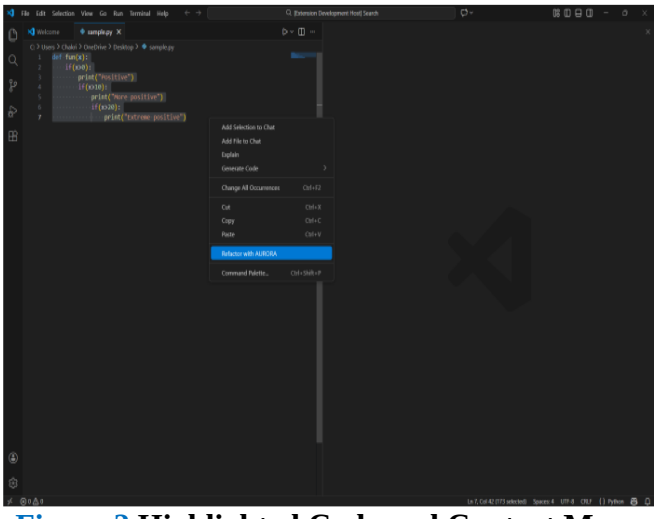

**Figure 2 Highlighted Code and Context Menu Option to Refactor Using AURORA in VS Code**

The figure (Figure 3) illustrates how CodeT5 model detects smells and returns human-readable suggestions with updated code.

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 792-798
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0113

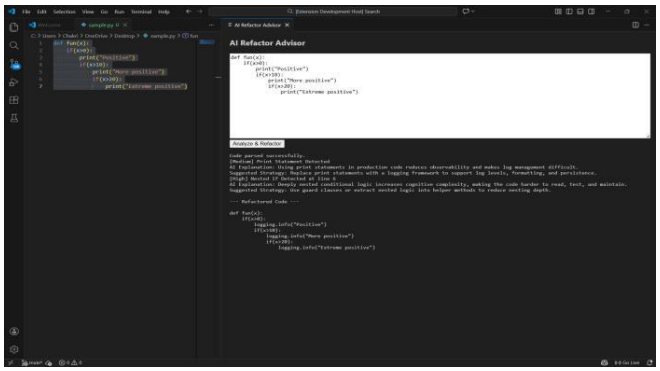

**Figure 3 Output Generated by AURORA Showing AI Explanation, Strategy, and Refactored Code**

The figure (Figure 4) shows the initial extension load view, representing client-side deployment and model integration.

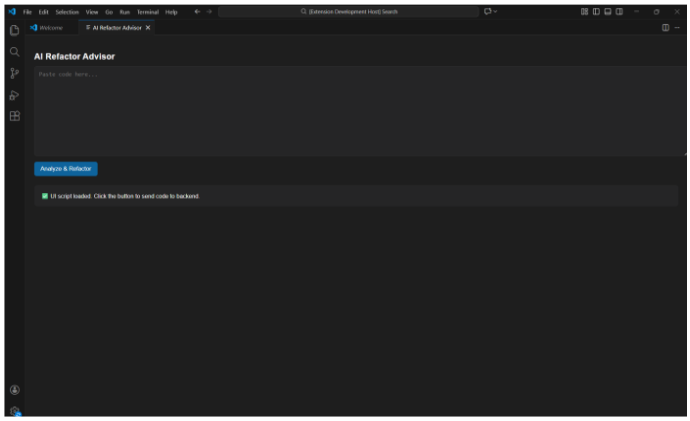


**Figure 4 AURORA Extension UI Deployed Inside Visual Studio Code on Developer Machine**

AURORA was evaluated based on its integration within the Visual Studio Code environment, response behavior, and initial qualitative assessment of its refactoring suggestions. Since the prototype is in its early stage, our focus was on validating the core functionality and potential of the system rather than conducting large-scale quantitative experiments.

### 5.1. Code Highlighting and Extension UI

The AI Refactor Advisor extension was successfully deployed inside Visual Studio Code. As shown in Fig. 5, the extension provides an interface where users can paste code and click the "Analyze & Refactor" button to trigger the backend pipeline. This lightweight UI ensures that the tool remains accessible during routine development workflows.

### 5.2. Interaction and Code Smell Detection

When the user selects a block of code and right-clicks, the context menu provides a "Refactor with AURORA" option (Fig. 6). This initiates the AST parsing and code smell detection process in the backend. The tool successfully identified basic smells such as deeply nested if-else statements and usage of raw print() statements - typical indicators of maintainability issues.

### 5.3. Refactoring Suggestions and Model Output

The backend, powered by the CodeT5 transformer, generated semantic refactoring suggestions. As demonstrated in Fig. 7, the AI explains the rationale behind each suggestion (e.g., replacing print statements with logging, reducing nesting), followed by a transformed version of the code. The suggestions are human-readable, context-aware, and structurally valid, making them suitable for integration into real-world codebases.

### 5.4. Deployment Setup

AURORA's architecture is divided between a VS Code frontend and a FastAPI backend hosted locally or remotely. The CodeT5 model runs in a separate runtime, optionally GPU-accelerated. This separation ensures that the developer's IDE remains responsive while heavier inference tasks are offloaded to a dedicated environment. Deployment on a local machine with modest specs yielded satisfactory performance for small to medium code files (Figure 5).

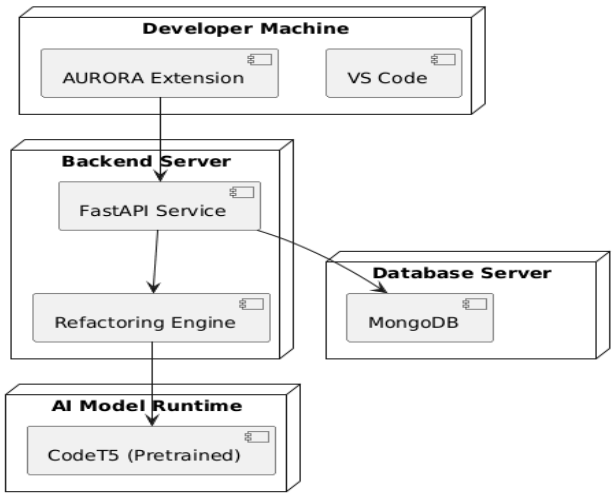


**Figure 5 Deployment Setup of the AURORA**

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 792-798
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0113

Figures (Figures 6 and 7) present additional examples demonstrating how our system detects and suggests improvements for different code smells.

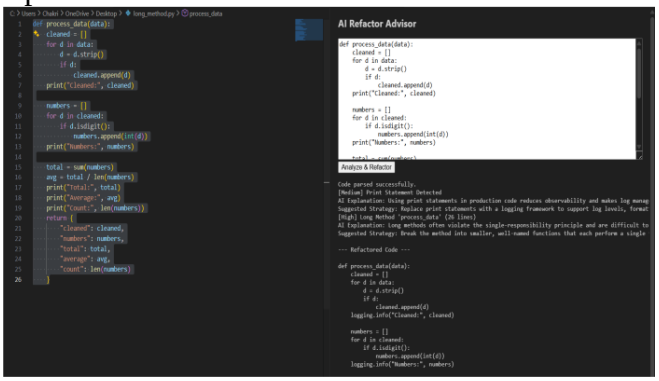


**Figure 6 Long Method Detection in Python**

This figure shows a function process_data containing over 25 lines of code, thereby exceeding the defined threshold for the **Long Method** code smell. In AURORA, we configured a threshold of **20 lines** to identify functions that become difficult to maintain or understand. The static analysis component parsed the AST and calculated the number of statements under the function block. The detection engine labeled it as a *high-priority* smell, suggesting decomposition into smaller helper functions that each perform a single, coherent task. This example also reflects AURORA's ability to combine **line metrics** with **semantic AI suggestions** such as using logging.info() instead of raw print() statements.

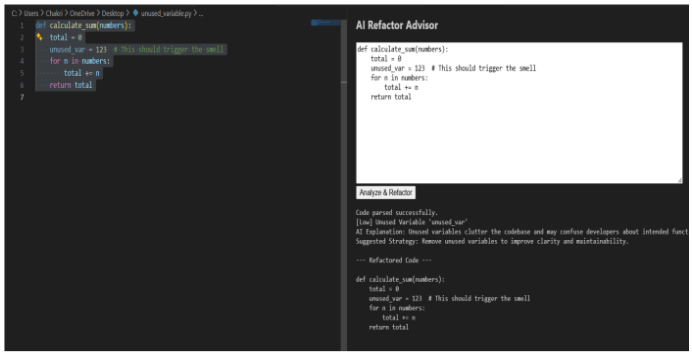


**Figure 7 Unused Variable Detection**

This figure demonstrates detection of the **Unused Variable** code smell. The variable unused_var = 123 is declared but not used anywhere within the function. AURORA flagged this as a *low-priority* smell but still relevant for improving code clarity.

The AST-based static analyzer matches assignments against references within the scope and flags variables that have no downstream impact on computation or logic. This helps prevent confusion for future developers and reduces clutter in the codebase.

### 5.5. Observations and Early Feedback

While a formal user study was not conducted, initial usage revealed that:

- The system detects common structural issues reliably.
- Suggestions improve readability and align with known refactoring patterns.
- Response times were reasonable (under 2 seconds for typical files).

These findings indicate that AURORA has practical value in supporting developers during early code writing and refactoring phases.

## Conclusion and Future Work

We have presented AURORA, an AI-augmented refactoring advisor that integrates static AST analysis with a CodeT5 Transformer to detect code smells and suggest intelligent refactorings. Unlike conventional static analyzers (SonarQube, PMD [5]) which rely on fixed syntactic rules, AURORA leverages machine-learned semantics to craft context-aware fixes. Our architecture (Fig. 1) and implementation demonstrate how a VS Code extension can interact with a FastAPI/CodeT5 backend to provide real-time code guidance. In future work, we plan to expand AURORA in several directions: (1) Support more code smells and languages (e.g. integrate Java AST parsing); (2) Continuously fine-tune CodeT5 on a growing corpus of refactoring examples, and possibly incorporate user feedback into model updates; (3) Integrate AURORA into CI/CD pipelines using the designed database schema, so that refactoring suggestions can be tracked as part of code review; (4) Conduct larger-scale user studies and benchmark comparisons with more tools. We also envision incorporating dynamic analysis hints (e.g. test coverage) to further refine suggestions. By combining rule-based detection with AI-driven code transformations, AURORA represents a step toward reducing technical debt through **semantic refactoring**. As AI models for code continue to evolve, we anticipate tools like AURORA will

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 02 February 2026
Page No: 792-798
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0113

become essential extensions in developer's environments, helping maintain clean and maintainable software.

## Acknowledgement

We would like to express our sincere gratitude to our guide, K. Rajya Guru Sai Sri, Department of Computer Science and Engineering, SRK Institute of Technology, for her constant support, valuable insights, and encouragement throughout the development of the AURORA project. We also thank the faculty and peers who provided feedback during demonstrations and evaluations.

## References

[1]. Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in Proc. EMNLP, 2021.

[2]. F. Pecorelli, S. Luján, V. Lenarduzzi, F. Palomba, and A. Di Lucia, "On the adequacy of static analysis warnings with respect to code smell prediction," Empirical Softw. Eng., vol. 27, no. 3, Mar. 2022, pp. 64–84.

[3]. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[4]. S. Ramírez, FastAPI Documentation, 2020. [Online]. Available: https://fastapi.tiangolo.com/

[5]. PMD Source Code Analyzer, PMD, 2025. [Online]. Available: https://pmd.github.io/ (accessed Jan. 2025)