Data Leakage

Spring 2025 Report

Team Members

- Jeffrey Busold
- Ryan Lee
- Arnav Marchareddy
- Michael Socas
- Owen Truong
- Terrence Zhang

Client Information

- Dr. Eman Abdullah AlOmar
- Email - ealomar@stevens.edu
- Phone - (201) 216-8561

Abstract 4
Introduction 4
Data Leakage Original Tool 4
PyCharm Plugin 6
Our Project 7
Website 12
Extension 13
Logo 13
Marketplace 14
Workflow 14
Installation 14
Analyzing Leakage 17
Fixing Data Leakage 18
UI Components 19
Main Page 19
Button: Run Data Leakage Analysis 20
Settings Page 21
Table Panel 22
Data Extraction 24
Leakages Class 25
Differences from the Previous Team 28
Default Quick Fix 30
Overlap Quick Fix 31
Preprocessing Quick Fix 33
Multi-test Quick Fix 36
Copilot Quick Fix 38
Tool Architecture 39
Main Extension 40
View Components 40

Data Processing 40
Execution Modes 41
Docker-Free Solution 41
Docker Solution 43
Development Process for the Main.dl Executable 43
Porting the Python Code 45
Packaging the Pyright Module 46
Testing 48
Tool Usefulness 49
Publishing 50
Steps For Setting Up Publishing 50
Troubleshooting 52
Token Expiration 52
Comparative Analysis 53
Introduction 53
Advantages and Disadvantages List 53
Leakage Analysis Python Program 53
Leakage Analysis PyCharm Plugin 54
Leakage Analysis VS Code Extension 54
Analysis 55
Limitations and Future Work 57
LLM Integration 57
Combining Download and Installation Button 57
Allow Analysis of Python Files 57
More Comprehensive Quick Fix 57
Loading Bar in Extension View for Better UX 58
Group ID 58
Team Experience and Final Thoughts 59
References 61

Abstract

We have developed a new Visual Studio Code extension that detects data leakage — mainly preprocessing, overlap and multi-test leakage — from Jupyter Notebook files. Data leakage happens when a model training data set makes use of test data in data science code, leading to inaccurate performance estimates. Beyond detection, we implemented two correction mechanisms named Quick Fix: a conventional approach that manually fixes the leakage and an LLM-driven approach that guides ML developers toward best practices for building ML pipelines.

According to the paper "Data Leakage in Notebooks: Static Detection and Better Processes", many model designers do not effectively separate their testing data from their evaluation and training data. We are developing an extension for the VS Code IDE that identifies instances of data leakage in ML code and provides suggestions on how to remove the leakage.

Introduction

Data Leakage Original Tool

The Leakage Analysis tool is a static analysis framework specifically designed to detect test data leakage in Python notebooks. Developed as part of the ASE 2022 paper titled "Data Leakage in Notebooks: Static Detection and Better Processes", the tool targets machine learning workflows, where improper separation of training and testing data can lead to misleading model evaluations [7]. It operates by parsing Python code, transforming it into a simplified representation, performing type inference, and executing static analysis using a Datalog engine (Soufflé) [1]. The goal is to identify patterns of data leakage by tracing the flow of data, particularly from test sets, through the program.

Main Components

- AST Parsing and GlobalCollector
  - The analysis begins with parsing the input Python file into an Abstract Syntax Tree (AST) using Python's built-in tools. This provides a structural representation of the source code, which serves as the basis for further transformations. The global_collector.py module scans the AST to collect global variables. These are variables that must be preserved during transformation to avoid unintended renaming, ensuring analysis consistency.

- CodeTransformer (SSA Transformation)
    - The irgen.py module simplifies the original code by breaking down complex statements into smaller atomic ones. Converting the code into Static Single Assignment (SSA) form, where each variable is assigned exactly once. This aids in precise tracking of data flow across the program.
- Customized Pyright
    - The tool integrates a customized version of Pyright, a static type checker for Python, to perform type inference. This helps annotate variables with types which improves the precision of subsequent analysis.
- Fact Generation
    - With type-annotated SSA-form code, the factgen.py module converts the transformed code into Datalog facts. These facts represent the program's data flow and are the input for the core analysis.
- Datalog Analysis
    - The main analysis logic resides in main.dl, written in Datalog, a declarative logic programming language. The Datalog rules identify taint flows and detect whether test data influences training or prediction logic — a clear sign of data leakage.
- HTML Rendering
    - The render.py module outputs an annotated HTML file showing the original code, detected leakage issues, and a summary table with interactive navigation for users to review each leakage case.
- Taints
    - Taints are labels applied to sensitive data, such as test data, that allow the analysis tool to track how that data flows through a program. When variables like X_test or y_test are first introduced, they are marked as tainted. This taint is then propagated to other variables and operations that derive from or interact with the tainted data. The purpose of this mechanism is to detect improper use of test data, such as its inclusion in training steps or model fitting, which would constitute a data leakage issue. By following the flow of tainted data through the code, the tool can statically identify scenarios where test data influences the training process, thus compromising the validity of machine learning evaluation results.

Taint tracking is essential for catching these subtle bugs that often go unnoticed in traditional testing or review processes.

PyCharm Plugin

The Data Leakage Plugin is a PyCharm Community Edition plugin designed to assist data scientists in detecting data leaks within their code [9]. The plugin was created by the previous SPARK Data Leakage Team. The plugin functionality consists of a human readable GUI that consists of two tables, Leakage Summary and Leakage Instances, an analysis button and finally a quick fix, or suggestion, on the resolution of the leakages.

First, the Leakage Summary table gives a summary of the leakages by categorizing the leakages into three types, Multi-Test, Overlap and Preprocessing, and telling us the frequency/occurrences of each type.

Next, the Leakage Instances table is a table with four columns, type, line, variable and cause. Type denotes one of the three leakage categories. Line represents the specific line in the Python file that is causing the leakage error. Variable represents the specific Python variable that the leakage is observed to be happening. Cause denotes some general description on why the leakage occurred.

Next, the "Run Data Leakage Analysis" button runs the analysis on a Python file by using a Dockerized version of the plugin's core algorithm. The core algorithm is called leakage-analysis created by Chenyang Yang et al. for their paper, "Data Leakage in Notebooks: Static Detection and Better Processes" [7]. The core algorithm detects and categorizes data leakages into three types shown below.

- Multi-Test Leakage:
    - Multi-test leakage refers to a situation in which information from multiple tests or experiments is unintentionally shared or used in a way that compromises the validity or independence of the tests.
    - The potential cause for multi-test leakage arises because the tokenization and padding processes are applied to the entire dataset before splitting it into training, validation, and test sets.
- Overlap Leakage:

    - Overlap leakage refers to a situation in which there is unintentional sharing or overlap of information between the training and test datasets in a machine learning model.
    - This can occur when the same or highly similar data points are present in both the training and test sets. When the model is trained on a dataset that shares information with the test set, it may lead to overly optimistic performance evaluations and may not generalize well to new, unseen data.
- Preprocessing Leakage:
    - Preprocessing leakage occurs when information from the test set influences the preprocessing steps applied to the training set.
    - The cause of preprocessing leakage is self-explanatory, leakage that results from the preprocessing step. A potential cause could be sharing of data between training and testing dataset.

Lastly, there is the quick fix offered by the plugin. Quick fix is a naive implementation of a resolution to the leakage problems. When leakage is detected by the plugin, the plugin comes up with some resolution for the leakage, albeit it does not work most of the time due to the vast number of different scenarios a developer will require to implement for.

Our Project

At present, the plugin can scan Python files (.py), but it lacks the capability to process Jupyter Notebook files (.ipynb). This limitation arose due to time constraints faced by the previous development team, along with the fact that PyCharm Community Edition does not support .ipynb file editing. As a result, the primary objective of this project is to develop the plugin for a different code editor or IDE (VS Code) that can support the read and write of .ipynb files. In addition, secondary objectives like supporting the running of the leakage-analysis algorithm without Docker and improving the quick fix solution.

First, Microsoft's Visual Studio Code (VS Code) was chosen for simply having a lot more Jupyter Notebook users than JetBrain's PyCharm IDE. JetBrain had 15.9M users in 2023 [2] and 11.4M concurrent users in 2024 [3], while VS Code has 82M users who have downloaded the Jupyter Notebook Extension for use on VS Code [4]. Not to mention the 15.9M and 11.4M numbers for JetBrain comprise the total combined user of all JetBrain's services [3-4].

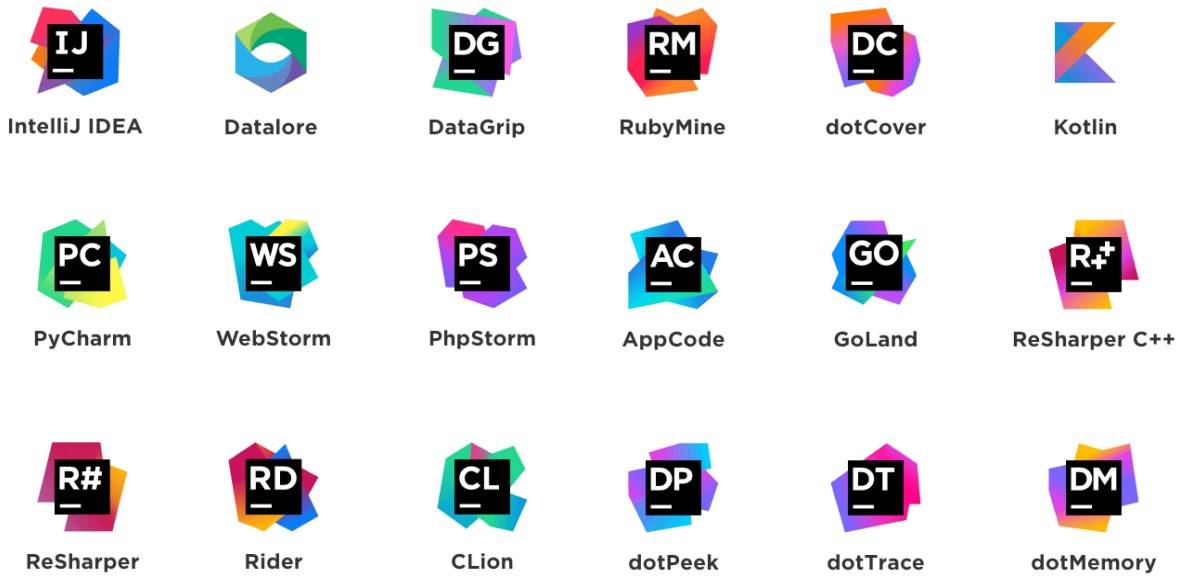


Figure 1: Icon of all JetBrains IDE since 2023

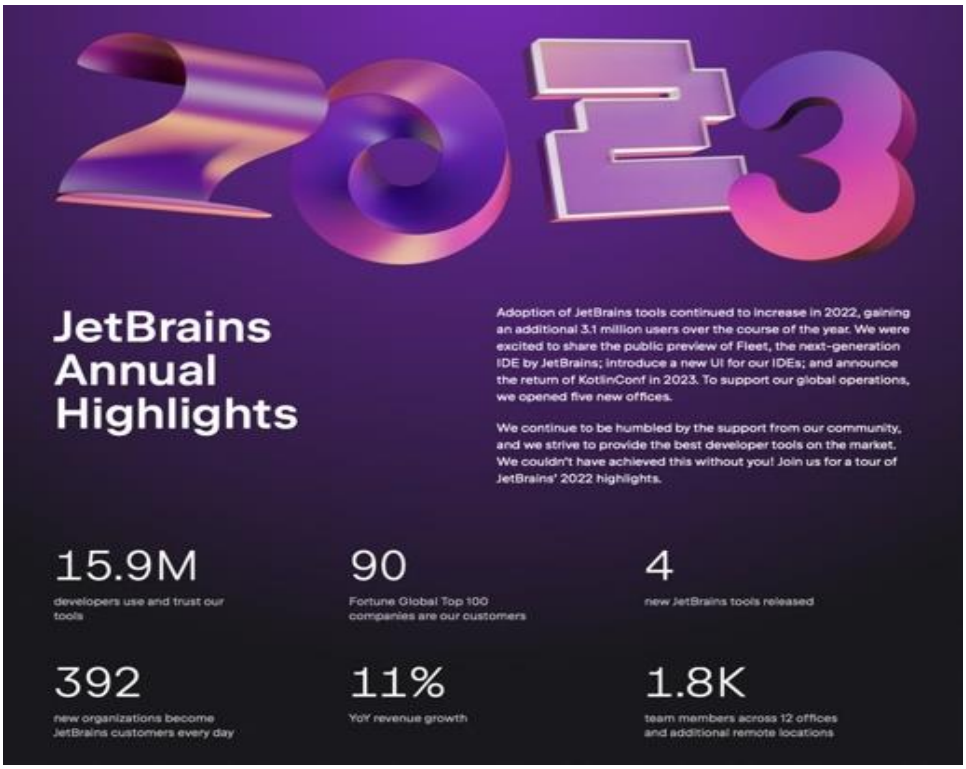


Figure 2: JetBrains Annual Highlights for 2023 - 15.9M developers [2]

Figure 3: JetBrains Annual Highlights for 2024 - 11.4M concurrent developers [3]

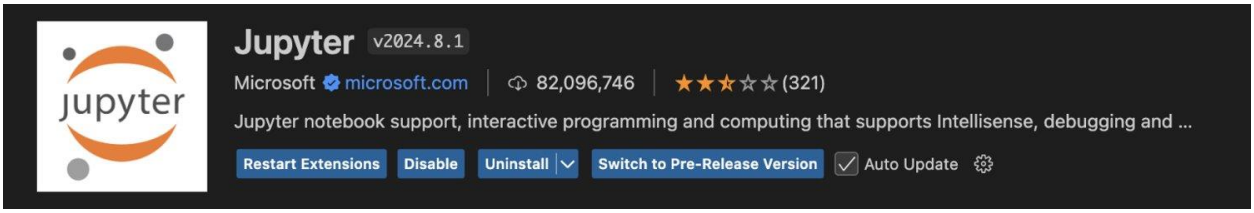


Figure 4: VS Code Marketplace Extension for Jupyter Notebook [4]

In Visual Studio Code (VS Code), plugins are referred to as extensions.

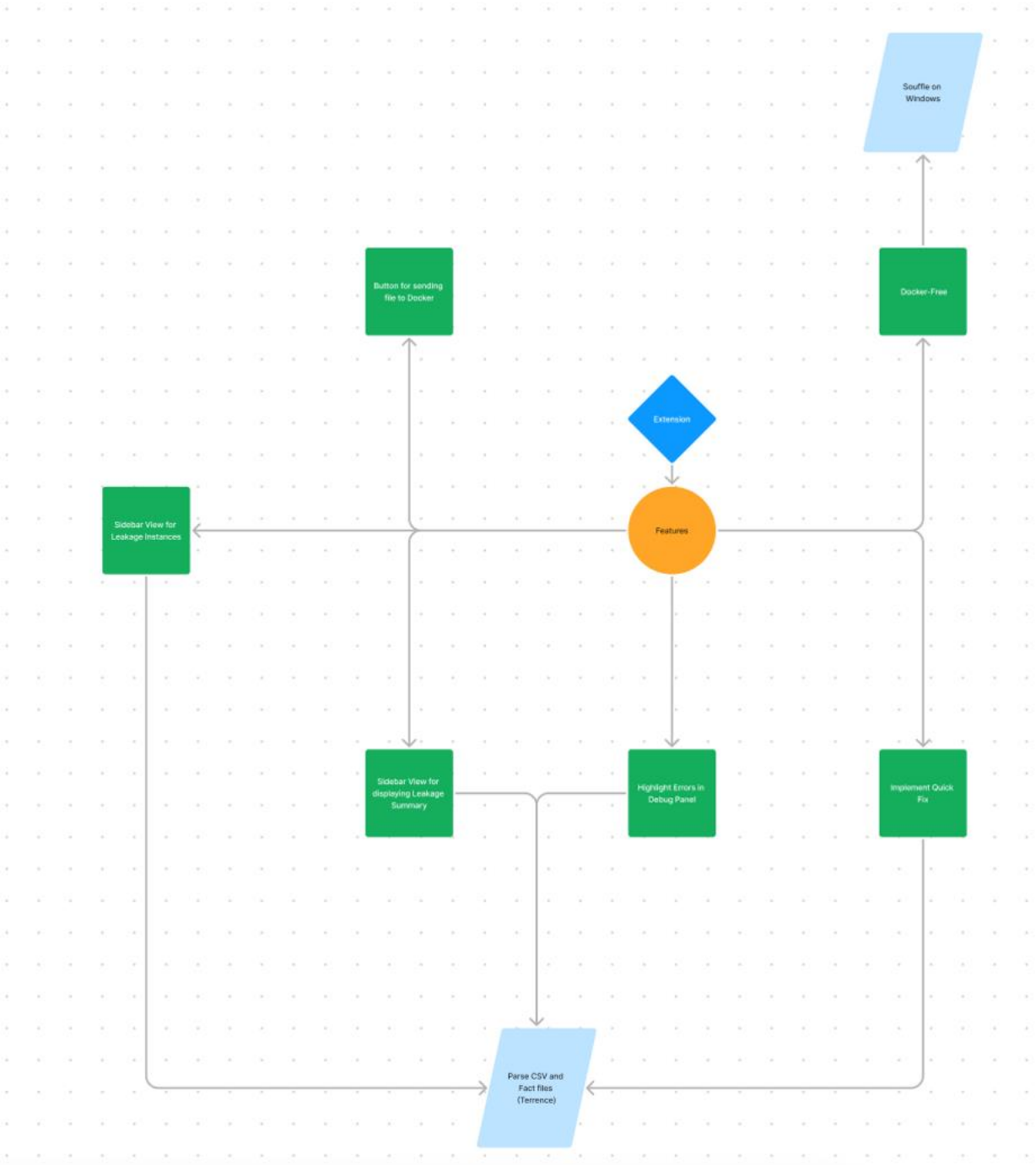


Figure 5: Feature Diagram

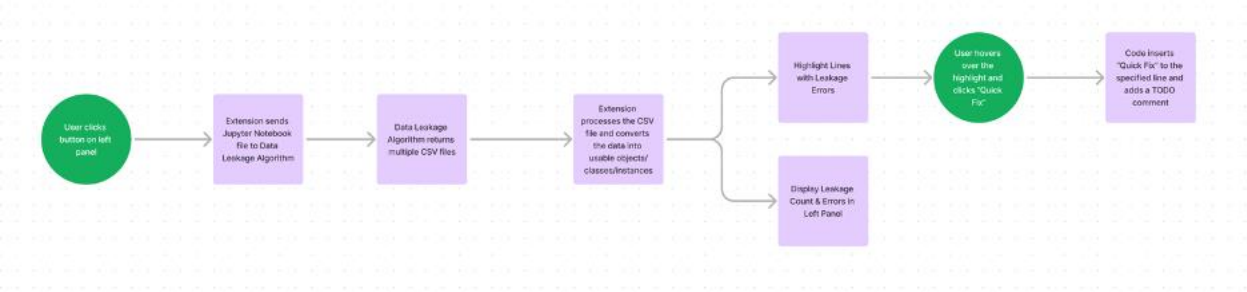


Figure 6: User Flow

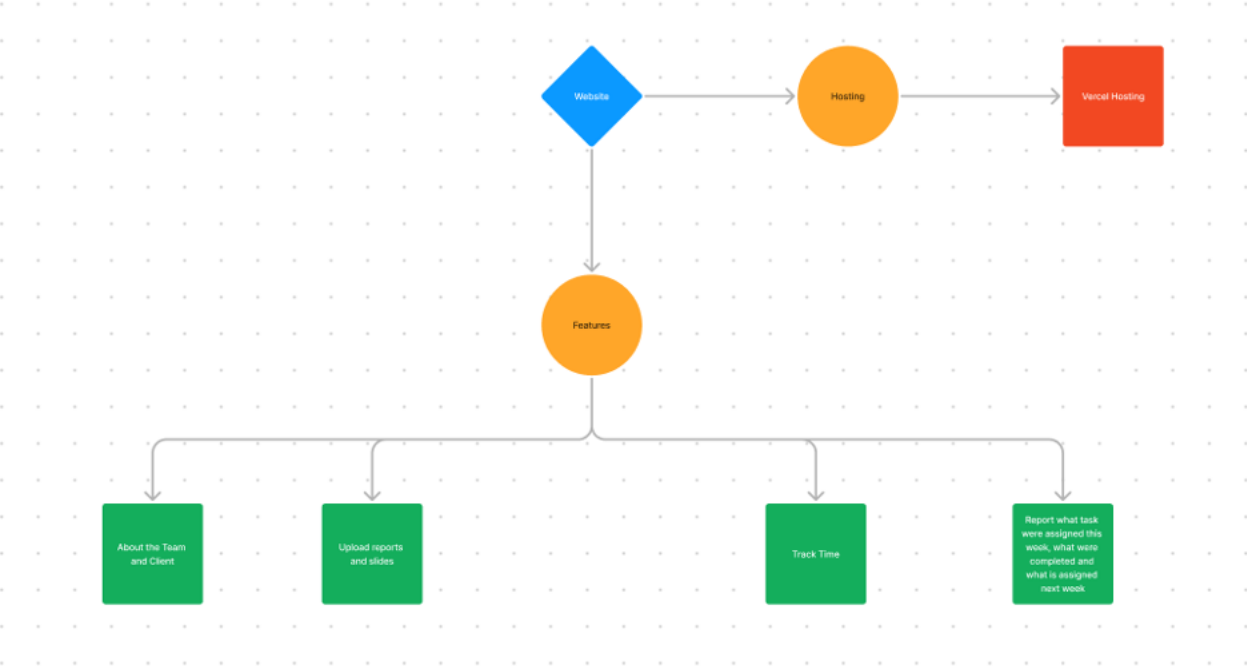


Figure 7: Website Feature

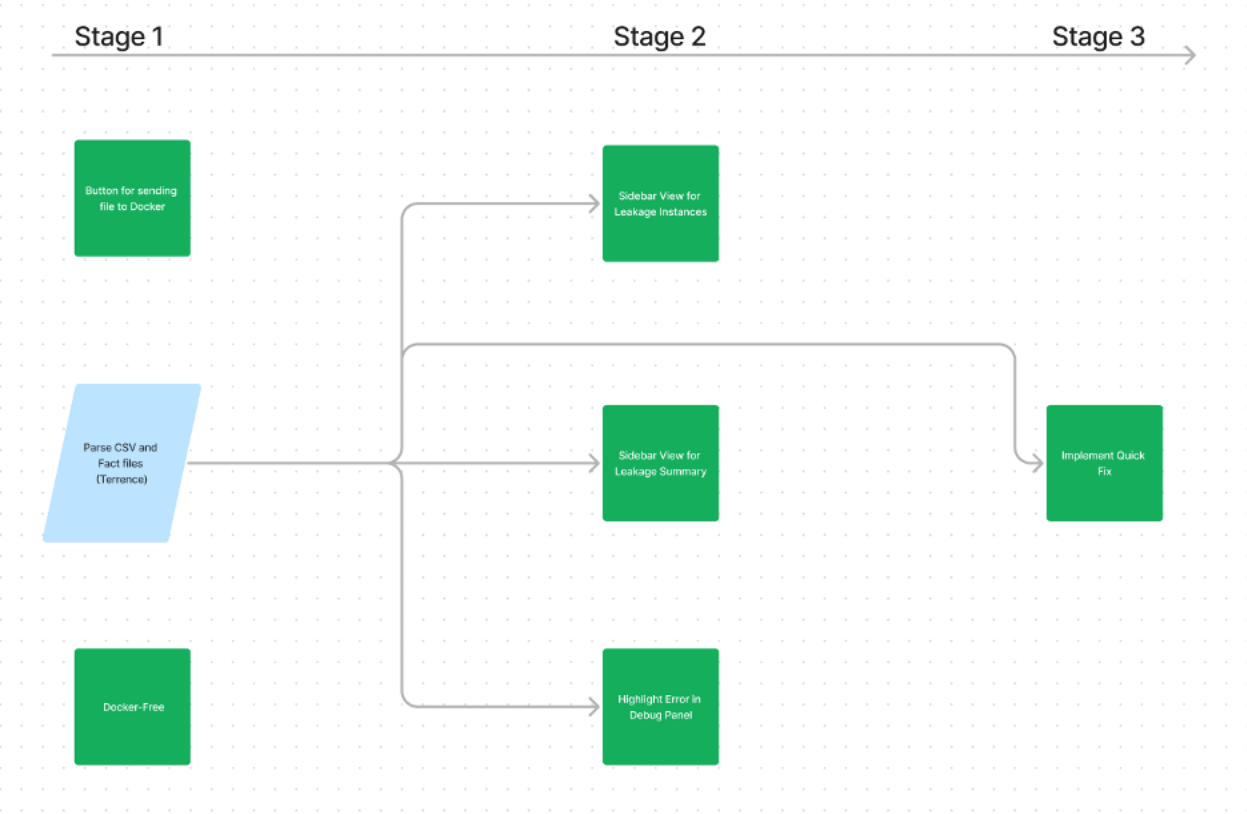


Figure 8: Extension Feature Priority

Website

Documentation website deployed at https://leakage-detector.vercel.app/ [8].

This website provides:

1) VS Code extension's purpose
2) Downloadable binaries to use in our extension
3) Downloadable Python files and Jupyter Notebook files that contain data leakage
4) Installation guide and run guide
5) Tutorial videos for installing extension dependencies and installing/using our extension
6) Information about the three data leakage types and how the extension or GitHub Copilot resolve the data leakage
7) Links to source code and previous documentation

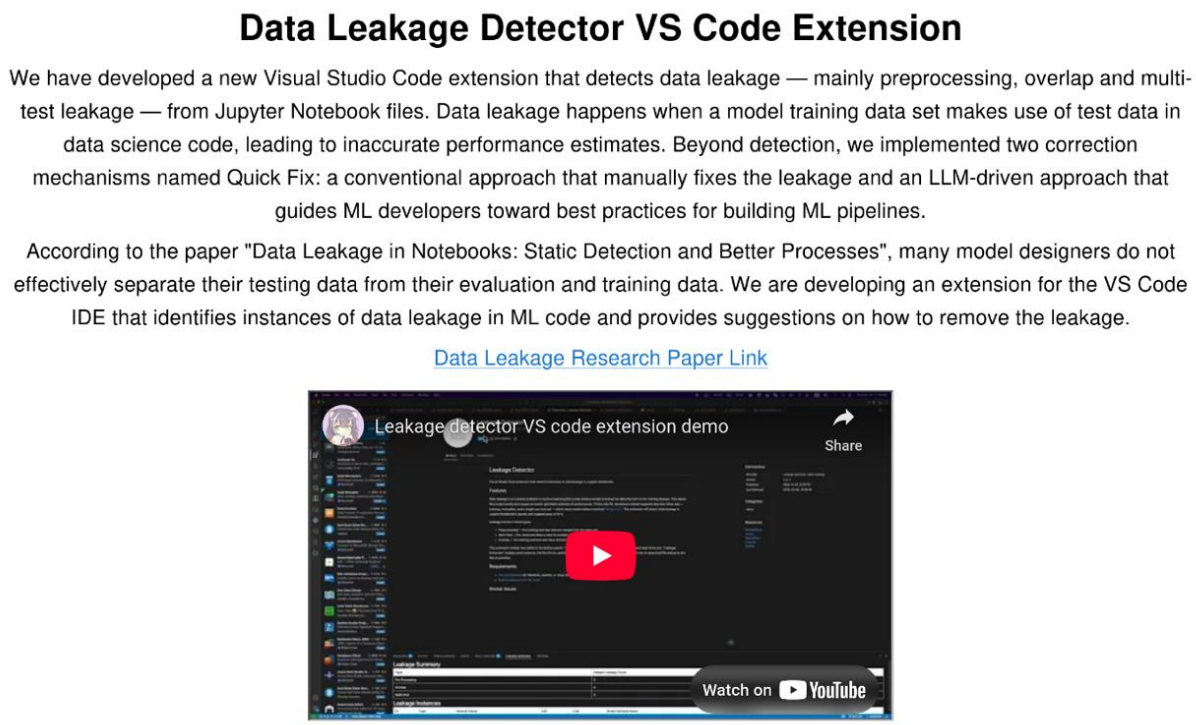


Figure 9: Website Home Page [8]

Extension

Logo

Figure 10: Main Logo

The poster logo went through several iterations before settling on the final design. Prior versions were more embellished, featuring the word “Leakage” crossed out in some form to represent the intent of the extension. It was decided that the more simplistic design would be the best fit, however, which is represented above.

Figure 11: Extension Icon

The extension icon is small and simple, made to reflect the intent of removing sources of data leakage found within the provided code.

Marketplace

Our extension is readily available on the VS Code Marketplace as seen below with our logo. Using GitHub Action, whenever we merge code into production branch, our extension will automatically update with the newest version.

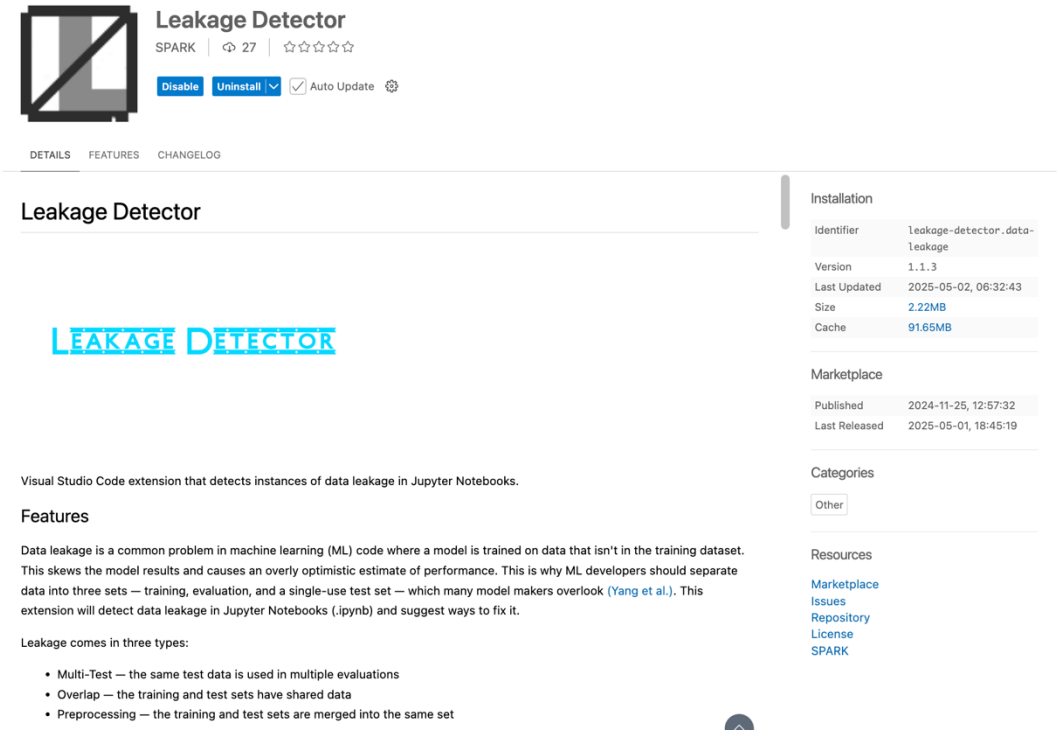


Figure 12: Extension Store inside VS Code

Workflow

Installation

1. Install Leakage Detector Extension from the VS Code marketplace, found at https://marketplace.visualstudio.com/items/?itemName=leakage-detector.data-leakage
2. Launch Visual Studio Code. From the activity bar on the left, choose the "Data Leakage" extension [step 2].
3. Step 3: Click on the gear icon in the top right to open the User Settings tab of the extension [step 3].

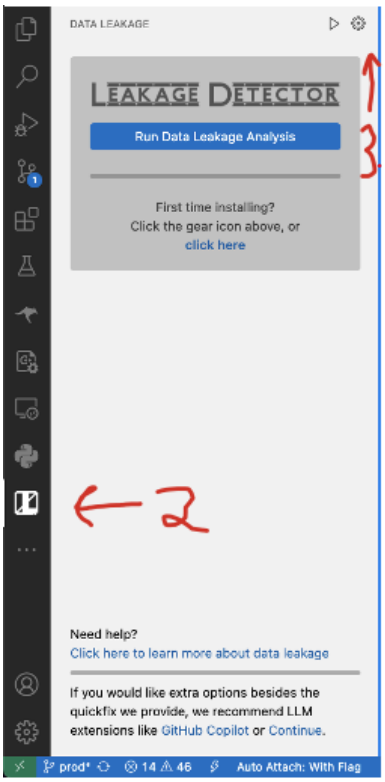


Figure 13: Step 2 and 3 of installation

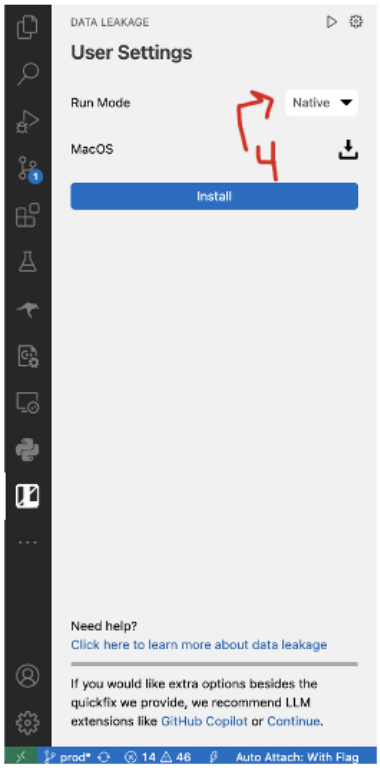


Figure 14: Step 4 of installation

4. Choose your preferred run mode from the dropdown menu, either "Native" or "Docker" [step 4]
    a. Native Mode: This mode uses a downloaded binary specific to your operating system. Selecting Native will display a link to this file. Download this zip file and proceed to step 5.
        i. Pick Native [Step i]
        ii. Download binary [Step ii]
            1. Unzip the binary (you should get a folder called “main”)
            2. Create a new folder called “leakage”
            3. Move the “main” folder inside “leakage folder”
        iii. Click the install button [Step iii]
        iv. Highlight the “leakage” folder [Step iv]
        v. Press the “Select Leakage Program Analysis Folder” button [Step v]
        vi. You should now see a notification that says “Leakage program successfully installed” [Step vi]

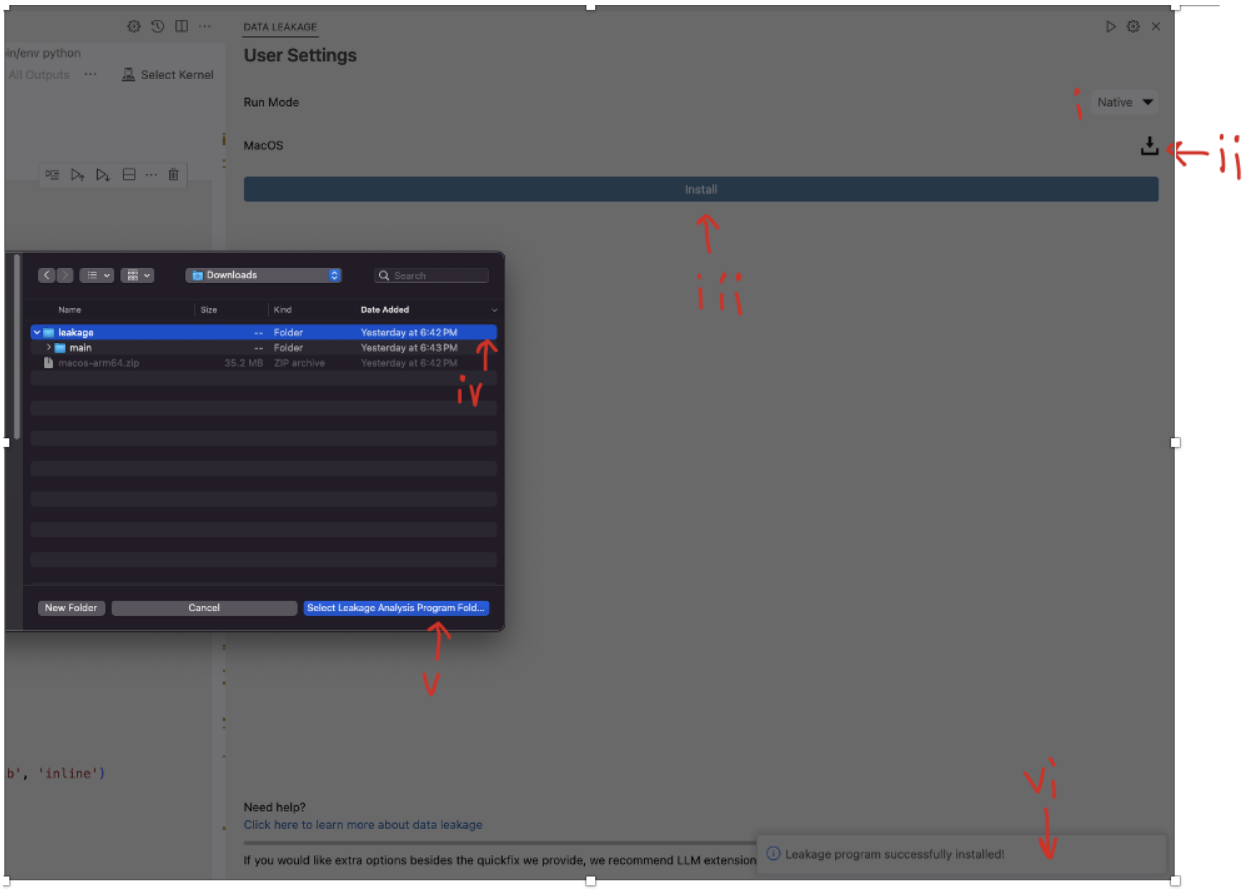


Figure 15: Steps for setting up native solution

b. Selecting Docker requires no further steps and you may skip to step 6. Ensure you have Docker Desktop running in the background.

5. Once you have downloaded the zip file, locate and extract the folder. Then, return to VS Code and click the "Install" button. Navigate to where the folder was extracted and select it.
6. Return to the main extension page by clicking the run icon.
7. Open a Jupyter Notebook file in the active tab of VS Code [Step 7].

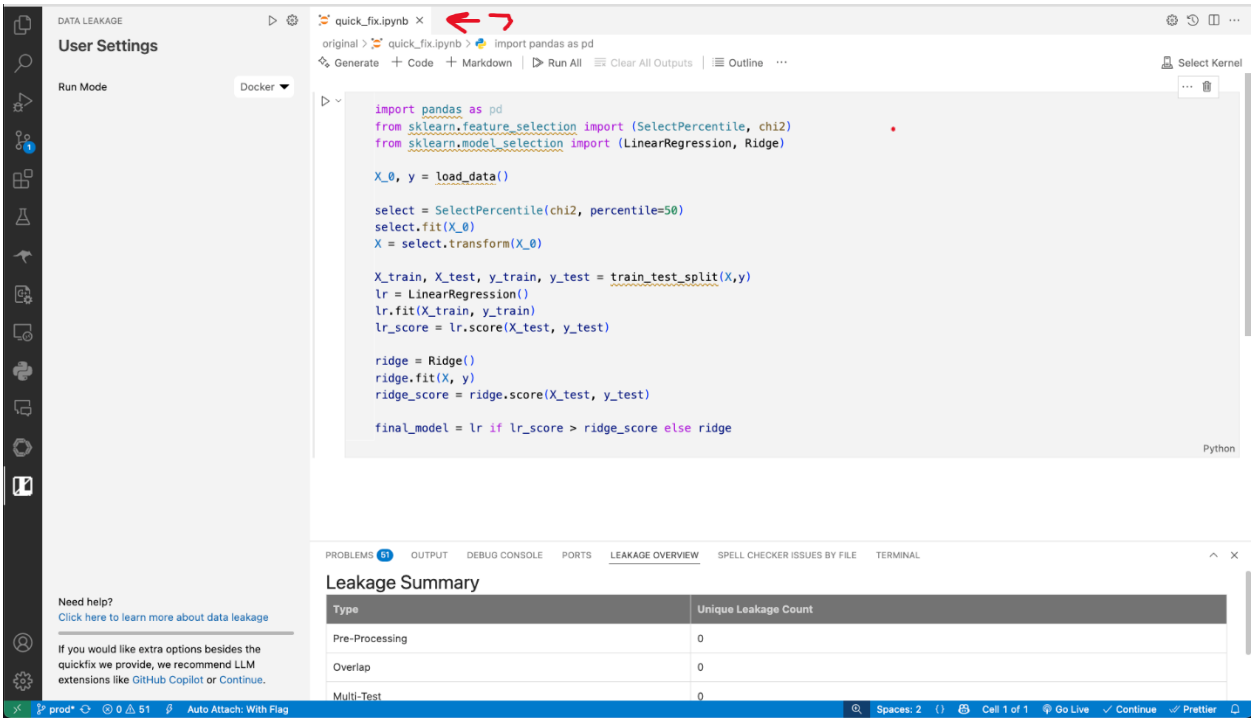


Figure 16: Open and make a notebook file active

Analyzing Leakage

1. In the extension window, click "Run Data Leakage Analysis" to start the process [Step 1].
    a. Allow time for the extension to analyze the notebook for instances of data leakage. This may take a few seconds.
2. Once the analysis is complete, you will receive a notification at the bottom right of VS Code [Step 2].
3. Review the "Leakage Overview" tab in the bottom panel of VS Code. It will show a summary of detected leakages and provide a detailed table of instances. Each instance can be examined by clicking on a row in the table [Step 3].

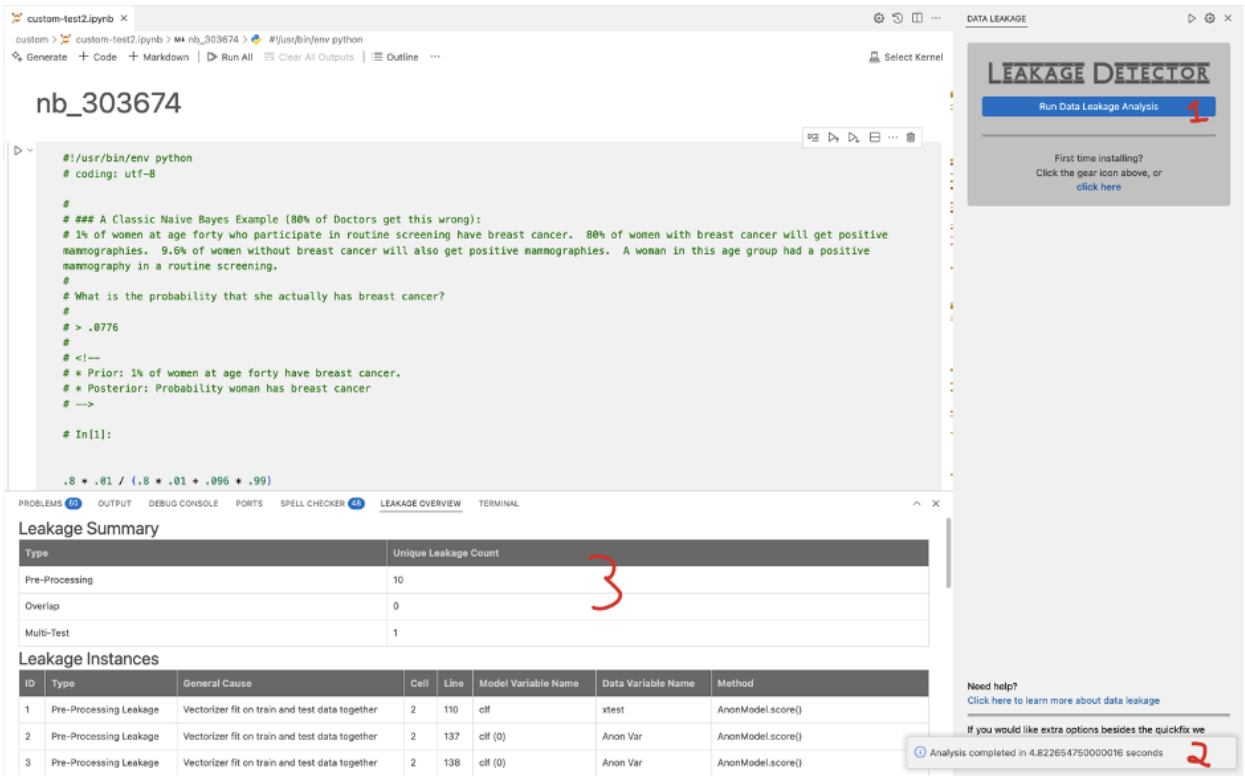


Figure 17: Analysis process

Fixing Data Leakage

1. Navigate to a data leakage instance by selecting a row in the leakage instances table.
2. The selected leakage instance will be highlighted in your Jupyter Notebook file [Step 2].
3. Hover over the highlighted line with the red error to reveal the "Quick Fix" option [Step 3].
4. Click on "Quick Fix" to see several potential solutions [Step 4].
    a. Then, you may select the light bulb icon to perform the manual Quick Fix or select the option “Fix using Copilot” to perform Copilot’s AI-based Quick Fix.
    b. You must have the GitHub Copilot VS Code extension to fix using Copilot, which is discussed in the installation guide and linked above as an optional requirement. These options attempt to resolve the data leakage.

Manual Quick Fix: Select any light bulb icon to perform the manual Quick Fix. Your Jupyter Notebook will be updated to remove the data leakage instance. Note that these fixes are rudimentary and might not always be the optimal solution.

Copilot Quick Fix: Select the option “Fix using Copilot” to perform GitHub Copilot’s AI-based Quick Fix. This will prompt a Copilot window to "Accept”, “Close” (reject), or “Accept & Run.” Please be aware that while GitHub Copilot can provide helpful suggestions, it might occasionally generate incorrect or suboptimal code solutions. Always review its recommendations critically before applying them.

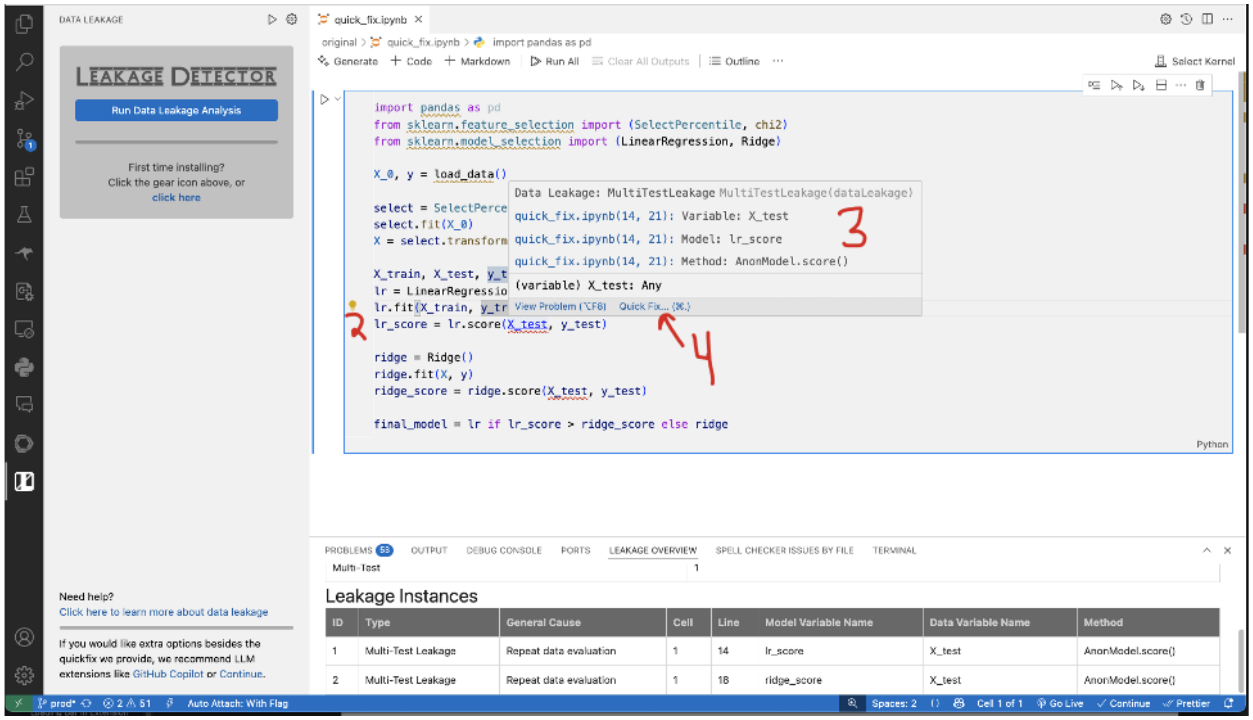


Figure 18: Quickfix guide

UI Components

Main Page

There is a button for running analysis with links to the settings page, to our website and to the VS Code extension store for AI Quick Fix extensions, i.e. Copilot and Continue extensions (Continue is open source and privacy-friendly alternative to Copilot).

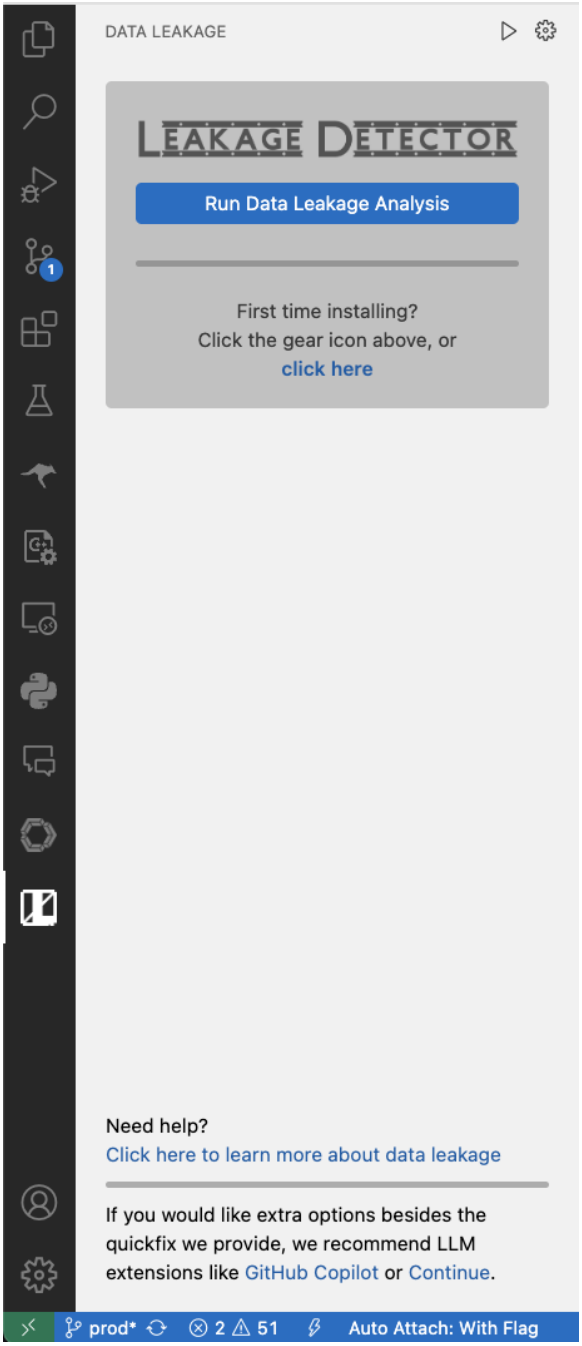
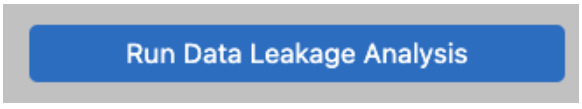


Figure 19: Main Page

Button: Run Data Leakage Analysis



When “Run Data Leakage Analysis” button is clicked, the extension sends the current active Jupyter Notebook file to the algorithm (Docker or native binary) to be run. If a non-ipynb file is

provided, the extension will show an error to the user that the current file is not a Jupyter Notebook.

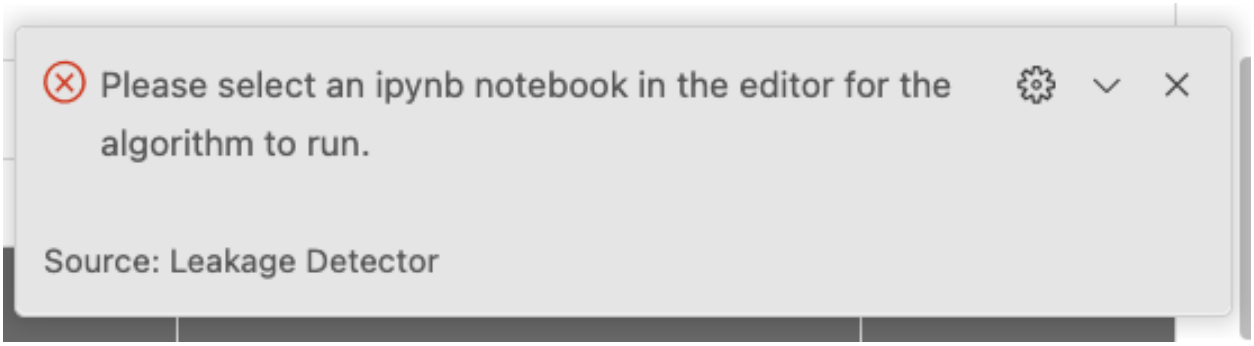


Figure 20: No notebook found error

When a proper ipynb file is focused on, the extension will try to run the selected run mode (native or Docker) to get the CSV and Fact files.
If the analysis fails, the following error is displayed:

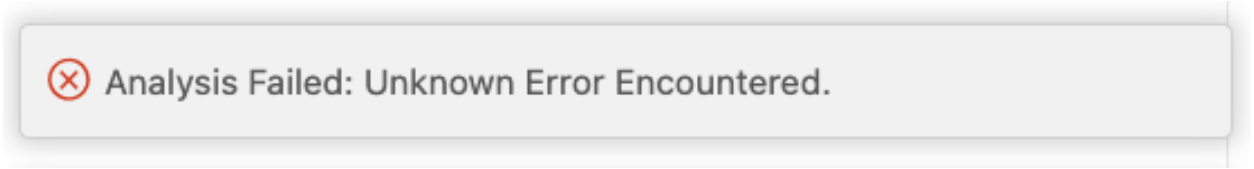


Figure 21: Analysis failed error

Settings Page

In the User Settings page, there are roughly around three parts to configure.
First is the run mode. The user is given the choice of running the leakage detector on their host machine, i.e. with the native binary, or with Docker.

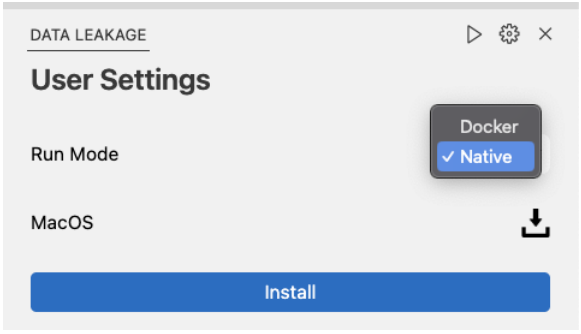


Figure 22: Run Mode

Next, if they choose to run with the native binary, a download button corresponding to their OS will appear for them to download and then install with the “install” button. The extension then opens a folder picker dialog which allows the user to select the folder containing the bundled

Docker-free solution. Our Docker-free solution can be downloaded from our website.
https://leakage-detector.vercel.app/download
The user would unzip the zip file of the Docker-free solution and select that folder in the file picker. Once the folder has been selected and the user clicks “Select Leakage Analysis Program Folder”, it will automatically install the program and its binaries. Once the installation is complete, the user will receive a popup notification stating that the Docker-free solution has been successfully installed.

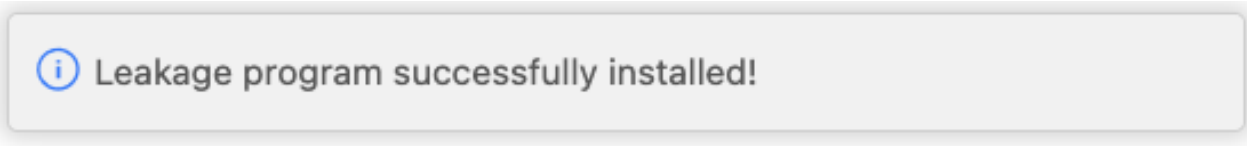


Figure *23*: Successfully installed the binary

Table Panel

Leakage Summary

After running “Run Data Leakage Analysis” successfully, the two tables (Leakage Summary & Leakage Instance) in VS Code’s panel (at the bottom) will be populated with values.
Leakage Summary has 2 columns and 3 rows not including the header. The three rows represent Pre-Processing, Overlap and Multi-Test with their respective frequency/count on the total error in a specific Notebook file.

Leakage Summary

Type	Unique Leakage Count
Pre-Processing	10
Overlap	0
Multi-Test	1

Figure *24*: Leakage Summary Table (nb_303674.ipynb)

Leakage Instances

Like Leakage Summary table, once the user clicks the button, Leakage Instances table will be populated with rows.

Leakage Instances

ID	Type	General Cause	Cell	Line	Model Variable Name	Data Variable Name	Method
1	Pre-Processing Leakage	Vectorizer fit on train and test data together	2	110	clf	xtest	AnonModel.score()
2	Pre-Processing Leakage	Vectorizer fit on train and test data together	2	137	clf (0)	Anon Var	AnonModel.score()
3	Pre-Processing Leakage	Vectorizer fit on train and test data together	2	138	clf (0)	Anon Var	AnonModel.score()
4	Pre-Processing Leakage	Vectorizer fit on train and test data together	2	164	clf (1)	Anon Var	AnonModel.score()
5	Pre-Processing Leakage	Vectorizer fit on train and test data together	2	165	clf (1)	Anon Var	AnonModel.score()
6	Pre-Processing Leakage	Vectorizer fit on train and test data together	3	110	clf (3)	xtest	AnonModel.score()
7	Pre-Processing Leakage	Vectorizer fit on train and test data together	3	137	clf (4)	Anon Var	AnonModel.score()
8	Pre-Processing Leakage	Vectorizer fit on train and test data together	3	138	clf (4)	Anon Var	AnonModel.score()
9	Pre-Processing Leakage	Vectorizer fit on train and test data together	3	164	clf (6)	Anon Var	AnonModel.score()
10	Pre-Processing Leakage	Vectorizer fit on train and test data together	3	165	clf (6)	Anon Var	AnonModel.score()
11	Multi-Test Leakage	Repeat data evaluation	2	109	clf	xtrain	AnonModel.score()
12	Multi-Test Leakage	Repeat data evaluation	2	110	clf	xtest	AnonModel.score()

Figure 25: Leakage Instances Table (nb_303674.ipynb)

The table currently has 5 columns:

- Type
    - Tells us the type of the leakage.
    - The type can be one of the three: Overlap, Preprocessing, Multi-Test
- General Cause
    - Tells us the general reason for the cause of each leakage.
- Cell
    - Tells us which Notebook cell this error belongs in.
    - Index starts at 1.
- Line
    - Tells us which line in the Notebook cell the error is occuring from.
    - Index 1 starts from the first line in each cell (i.e. each cell has a separate line index).
- Model Variable Name
    - Tells us the associated model's variable name created by the user.
- Data Variable Name
    - Tells us the associated data's variable name created by the user.
- Method
    - Tells us which model's method was used.

Data Extraction

Data extraction evolved significantly through multiple iterations before reaching its current form, with each version introducing major changes. Initially, the approach built on the work of the previous team, which focused exclusively on parsing the generated CSV and Fact files to identify leakage information. However, it became evident that this method often led to incorrect interpretations due to the raw and disjointed nature of these files. For instance, a CSV might list all the training and testing lines present in the user's file but does not indicate which pairs are actually related to each other. Understanding these relationships may then require cross-referencing with another generated file—yet which file exactly, and how to extract or align that information, is unclear and not clearly documented. Essentially, this task was akin to knowing the output of a very complex equation, but being given only the raw numbers and having to reverse-engineer the operations ourselves.

Recognizing these limitations, we transitioned to directly parsing the generated HTML file. This file consolidates the final analysis output and clearly states not only which variables are involved in leakages, but also provides rich context such as the number of leakages detected, their exact locations, the relevant train and test sites, sources of leakage, and other usages of the same variable. This shift significantly improved both the accuracy and reliability of our results, as it

eliminated the need to manually piece together relationships from intermediate, ambiguous data sources.

Leakage	#Detected	Locations
Pre-processing leakage	0	
Overlap leakage	0	
No independence test data	1	20 16

```
1 import pandas as pd
2 from sklearn . feature_selection import (SelectPercentile,
3                                         chi2)
4 from sklearn . model_selection import (LinearRegression,
5                                        Ridge)
6
7 X_0, y = load_data()
8
9 select = SelectPercentile(chi2, percentile=50)
10 select.fit(X_0)
11 X = select.transform(X_0)
12
13 X_train, y_train, X_test, y_test = train_test_split(X,y)
14 lr = LinearRegression()
15 lr.fit(X_train, y_train) train highlight train/test sites no independent test data
16 lr_score = lr.score(X_test, y_test) test validation highlight train/test sites used multiple times highlight other usage
17
18 ridge = Ridge()
19 ridge.fit(X, y) train highlight train/test sites no independent test data
20 ridge_score = ridge.score(X_test, y_test) test validation highlight train/test sites used multiple times highlight other usage
21
22 final_model = lr if lr_score > ridge_score else ridge
```

Figure 26: Example HTML file

This shift in strategy ultimately led to the development of the Leakages class, which now serves as the central component for data extraction.

Leakages Class

Constructor

constructor(outputDirectory: string, filename: string, fileLines: number)

The Leakages class is designed to initialize with three key parameters that provide the necessary context and resources for extracting leakage information:

1. **outputDirectory** – This refers to the parent directory that contains the subdirectory holding the CSV and Fact files. For instance, if the data for the file *nb_303674.py* is in *~/Outputs/nb_303674-fact/*, then outputDirectory should be specified as *~/Outputs*.
2. **fileName** – This is the name of the notebook or Python file to be analyzed, minus the file extension. For example, if the original notebook is *nb_303674.ipynb*, then the file name would be *nb_303674*. This parameter helps identify which file the leakage data corresponds to.

3. **fileLines** – This is the total number of lines in the analyzed file. It is particularly important for parsing the HTML file, as the line numbers are often used to pinpoint the exact locations of detected leakages and related information.

getLeakages()

```
public async getLeakages(): Promise<LeakageOutput>
```

The Leakages class exposes a single public asynchronous method: **getLeakages()**. As the name implies, this method retrieves all leakage instances detected for the specified file. It leverages the contextual data provided during initialization—such as the file path, file name, and line count—to accurately parse and extract leakage-related information from the corresponding HTML, CSV, and Fact files. The structure and format of the output returned by getLeakages() is detailed in the following section.

Output Structure

```
export enum LeakageType {
 OverlapLeakage = 'OverlapLeakage',
 PreProcessingLeakage = 'PreProcessingLeakage',
 MultiTestLeakage = 'MultiTestLeakage',
}

export type LeakageOutput = {
 leakageInstances: LeakageInstances;
 leakageLines: LeakageLines;
};

export type LeakageInstances = Record<
 LeakageType,
 {
  count: number;
  lines: number[];
 }
>;

export type LeakageLines = Record<number, LineInfo>;

export type LineInfo = {
 metadata?: Metadata;
 tags: LineTag[];
};

export type Metadata = {
 model: string;
 variable: string;
 method: string;
};

export type LineTag = {
 name: string;
 isButton: boolean;
 highlightTrainTestSites?: number[];
 markLeakSources?: number[];
 highlightOtherUses?: number[];
};
```

The getLeakages() method returns an object with two distinct properties: **LeakageInstances** and **LeakageLines**, each serving a specific role in representing leakage-related data.

LeakageInstances

The LeakageInstances property contains mappings from each **LeakageType** to an object detailing the total count of that type of leakage and the line numbers where they occur. For instance, if there are 2 overlap leakages spread across lines 20, 30, and 40, the LeakageInstances entry would look like:

```
{
 "overlap": {
  "count": 2,
  "lines": [20, 30, 40]
 }
}
```

This data directly corresponds to the summary table found at the top of the generated HTML files, providing a quick overview of the types and spread of leakages detected in the file.

LeakageLines

The LeakageLines property contains detailed mappings from line numbers to associated metadata and tags, offering fine-grained insight into how each line contributes to potential data leakage. Each entry corresponds to a line number and contains two fields: metadata and tags.

1. Metadata

- **Purpose**: Describes the key components used within that line.
- **Fields**:
    - variable: The variable involved.
    - method: The method or function invoked.
    - model: The model used.
- **Note**: This field is **nullable** since not all lines include metadata

2. Tags

- **Purpose**: Indicates special annotations or interactive elements tied to that line.
- **Structure**: An array of tag objects, each with its own properties.

Tag Types:

Each tag falls into one of the following five categories:

i. Pure Text Tag

- isButton: false

- Contains only a name field.
- Non-interactive, used purely for annotation.

ii. Highlight Train/Test Sites Button

- isButton: true
- highlightTrainTestSites: An array of line numbers to be highlighted when the tag is clicked.

iii. Show Leakage Sources Button

- isButton: true
- markLeakSources: An array of line numbers related to leakage's source.

iv. Highlight Other Uses Button

- isButton: true
- highlightOtherUses: An array of lines where the same variable is used elsewhere.

Important Note

Only line numbers with tags appear in the LeakageLines object. If a line is absent, it is safe to assume it has no tags or metadata.

Differences from the Previous Team

As mentioned earlier, while the previous team opted to manually extract and interpret raw data from the CSV and Fact files, we ultimately shifted toward directly parsing the generated HTML file. This decision led to a significantly simpler and more streamlined program design. Whereas the previous team's solution relied on multiple abstracted classes and interfaces to handle different parts of the data extraction process, our implementation is centered around a single class, the Leakages class, that encapsulates the entire workflow.

Initially, we did attempt to build on the prior team's implementation. However, during this process, we uncovered several inefficiencies and design issues that led us to reconsider. One notable example is their use of a method that is called repeatedly which in turn invokes a secondary constructor method that reads the same file each time

(

```java
/**
 * Looks through the CSV files that provide information about the provided {@code leakageType}.
 * Adds leakage instances to the field {@link #leakageInstances}.
 */
public void findLeakageInstancesInFile(File file) {
    if (file.exists()) {

        try {
            BufferedReader reader = new BufferedReader(new FileReader(file));

            String line;
            while ((line = reader.readLine()) != null) {
                var leakageInstance = createLeakageInstanceFromLine(line);

                addLeakageInstanceIfNotPresent(leakageInstance);


            }
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 27). This approach leads to excessive and redundant file I/O operations, as the same file is opened and scanned multiple times unnecessarily. Not only does this introduce inefficiencies in performance, especially for larger files, but it also complicates the logic and resource management within the program. In contrast, our implementation performs a single pass over the HTML file, loads its content into memory, and reuses it as needed—significantly improving both efficiency and clarity.

```java
/**
 * Looks through the CSV files that provide information about the provided {@code leakageType}.
 * Adds leakage instances to the field {@link #leakageInstances}.
 */
public void findLeakageInstancesInFile(File file) {
    if (file.exists()) {

        try {
            BufferedReader reader = new BufferedReader(new FileReader(file));

            String line;
            while ((line = reader.readLine()) != null) {
                var leakageInstance = createLeakageInstanceFromLine(line);

                addLeakageInstanceIfNotPresent(leakageInstance);


            }
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 27: createLeakageInstanceFromLine called for each line in the file

```java
@Override
protected MultiTestLeakageInstance createLeakageInstanceFromLine(String line) {
    String[] columns = line.split(("\t"));

    final var multiUseTestLeak = new MultiUseTestLeak(columns);
    final var telemetry = new MultiTestLeakageTelemetry(multiUseTestLeak);

    Invocation invocation = new Invocation(columns[getCsvInvocationColumn()]);
    int actualLineNumber = getActualLineNumberFromInvocation(invocation);

    return new MultiTestLeakageInstance(actualLineNumber, invocation, telemetry.getTest());

}
```

Figure 28: createLeakageInstanceFromLine calls the constructor method of MultiTestLeakageTelemetry

```
public MultiTestLeakageTelemetry(MultiUseTestLeak multiUseTestLeak) {
    String filePath = Paths.get(LeakageResult.getFolderPath()).resolve("Telemetry_MultiUseTestLeak.csv").toString();
    File file = new File(filePath);
    this.test = "";
    this.invo = "";
    this.ctx1 = "";

    try {
        BufferedReader reader = new BufferedReader(new FileReader(file));

        String line;

        while (((line = reader.readLine()) != null)) {
```

Figure 29: The constructor method reads the same “Telemetry_MultiUseTestLeak.csv” every time

In addition, the previous team implemented leakage source detection under the assumption that all types of leakages—whether overlap, pre-processing, or multi-test—contain taints, which they used as indicators to identify leakage sources. However, our research later revealed that this assumption was flawed: taints are exclusive to pre-processing leakages (*Figure* 30). This misunderstanding led to incorrect generalizations in their detection logic.

```
/* --------------------
"Taint" inputs (specified locations for reduce/duplicate in pre-processing)
-------------------- */
```

Figure 30: Source code of the leakage detection program. Located in src/main.dl

Given that the generated HTML file already consolidates and presents the processed information from the CSV and Fact files, including accurate leakage sources, it became clear that replicating this behavior by manually interpreting raw data was not only redundant but also error prone. Therefore, we chose to instead directly parse the HTML, allowing us to avoid incorrect assumptions and significantly simplify the implementation while ensuring consistency with the final output format.

Default Quick Fix

In most code editors and IDEs, quick fixes are, as the name suggests, fast and automated ways to resolve issues detected in the code. Although the method of accessing quick fix functionality may vary across different editors, the core idea remains the same: the tool automatically makes small, targeted edits to the user's code to try to correct the specified problem.

Quick fixes are designed for convenience, helping developers maintain their workflow without needing to manually write out every minor correction. However, they are typically rudimentary solutions, they address surface-level symptoms rather than deeply understanding the context of the code. As a result, while quick fixes can be very helpful for simple issues (such as missing

imports, basic syntax errors, or common refactoring tasks), they are not always guaranteed to work and may require manual verification or adjustment afterward.

```
1  import pandas as pd
2  from sklearn.feature_selection import SelectPercentile
       , chi2
3  from sklearn.model_selection import LinearRegression,
       Ridge
4
5  X_0, y = load_data()
6
7  select = SelectPercentile(chi2, percentile=50)
8  select.fit(X_0)
9  X = select.transform(X_0)
10
11 X_train, y_train, X_test, y_test = train_test_split(X,
       y)
12 lr = LinearRegression()
13 lr.fit(X_train, y_train)
14 lr_score = lr.score(X_test, y_test)
15
16 ridge = Ridge()
17 ridge.fit(X, y)
18 ridge_score = ridge.score(X_test, y_test)
19
20 final_model = lr if lr_score > ridge_score else ridge
```

Figure 31: Example file containing all three leakages

Overlap Quick Fix

Overlap leakages occur when some or all test data is mistakenly used during training or hyper-parameter tuning. A typical example is shown in Figure *31*, where an overlap leakage arises between lines 17–18: the variable X, which contains both training and test data, is used to train the ridge model. A proper fix would involve replacing X with X_train, the variable returned by the train_test_split method that contains only the training data. Our quick fix algorithm automates this repair. It follows these steps:

1. Find the latest call to train_test_split in the code.
2. Identify the X_train and y_train outputs (typically the first and third elements of the returned tuple from scikit-learn's train_test_split).
3. Replace the parameters of the offending fitting method with the correct X_train and y_train variables.

After applying the quick fix, the code in Figure *31* would look like Figure *32*:

```python
import pandas as pd
from sklearn.feature_selection import (SelectPercentile, chi2)
from sklearn.model_selection import (LinearRegression, Ridge)

X_0, y = load_data()

select = SelectPercentile(chi2, percentile=50)
select.fit(X_0)
X = select.transform(X_0)

X_train, X_test, y_train, y_test = train_test_split(X,y)
lr = LinearRegression()
lr.fit(X_train, y_train)
lr_score = lr.score(X_test, y_test)

ridge = Ridge()
ridge.fit(X_train, y_train)
ridge_score = ridge.score(X_test, y_test)

final_model = lr if lr_score > ridge_score else ridge
```

Figure 32: Example overlap quick fix

While the quick fix algorithm works well for many common cases, there are important limitations:

- Dependency on scikit-learn's train_test_split:
  Our algorithm assumes the user is utilizing scikit-learn's train_test_split method. If a different splitting method or custom function is used, the fix may fail or produce undefined behavior. For example, if the user chooses to define their own custom train/test split method, our quick fix algorithm will not be able to find that method and as such will perform no action. We chose to build around scikit-learn due to its popularity and widespread use in the data science community, aiming for the highest practical coverage.
- Incorrect train/test split selection:
  The algorithm always selects the latest train_test_split call found. If multiple train_test_split calls exist and the user intended to reference an earlier split, the fix will incorrectly use the most recent one. For example, if three train_test_split methods are used and the correct X_train came from the second call, our algorithm would mistakenly use the third. In these cases, it is critical for the user to manually verify the quick fix and adjust it if necessary.
- Lack of Scope Awareness:
  In a similar vein to incorrect train/test split selection, the algorithm does not currently

account for variable scoping. Because the quick fix logic operates by scanning the file line-by-line, it cannot determine whether variables are actually in scope at the time of replacement. It also does not handle complexities introduced by functions, loops, conditionals, or other control flow structures. As a result, there may be situations where the fix incorrectly replaces a variable that is either out of scope or improperly linked to the wrong context. In addition, this also means that larger files are more prone to possible errors as they typically contain more complex data flows. Users should carefully review the quick fix changes, particularly in files with complex control flow or modular structures.

Preprocessing Quick Fix

Preprocessing leakages happen when training and test data are preprocessed together, allowing information from the test set to improperly influence the training set. A typical example is shown in Figure *31*, where a preprocessing leakage occurs between lines 9–10: the variable X_0, which contains both training and test data, is transformed. This causes the resulting variable X—which is later split—to have its X_train inadvertently influenced by X_test. A proper fix is to move the preprocessing and feature selection steps after the train_test_split. This ensures that only training data is used to learn transformations, preserving the integrity of the evaluation. Our quick fix algorithm handles this correction by following these steps:

1. Locate the latest call to the train_test_split method.
2. Identify the X_train and X_test outputs of the split.
3. Find the lines where preprocessing occurs.
4. Move the train_test_split call upward, placing it before the preprocessing.
5. Assign the outputs of train_test_split to temporary variables (e.g., X_train_0 and X_test_0).
6. Apply all preprocessing and transformation methods to these temporary variables.
7. Have the results of preprocessing be the original X_train and X_test variables.

In essence, the quick fix introduces intermediate variables that "absorb" the transformations, and their processed versions are then assigned back to the expected names.

After applying the quick fix, the code for Figure *31* would look like Figure *33*:

```
import pandas as pd
from sklearn.feature_selection import (SelectPercentile, chi2)
from sklearn.model_selection import (LinearRegression, Ridge)

X_0, y = load_data()

X_train_0, y_train, X_test_0, y_test = train_test_split(X,y)
select = SelectPercentile(chi2, percentile=50)
select.fit(X_train_0)
X_train = select.transform(X_train_0)
X_test = select.transform(X_test_0)

lr = LinearRegression()
lr.fit(X_train, y_train)
lr_score = lr.score(X_test, y_test)

ridge = Ridge()
ridge.fit(X, y)
ridge_score = ridge.score(X_test, y_test)

final_model = lr if lr_score > ridge_score else ridge
```

Figure 33: Example preprocessing quick fix

Just like the overlap quick fixes, several limitations exist for preprocessing quick fixes:

- Same train/test split and scoping limitations:
  The preprocessing quick fix shares the same limitations as the overlap quick fix — namely, the assumption that the latest train_test_split call is the correct one and the inability to properly account for variable scope or control flow (such as functions, loops, or conditionals). The potential undefined behaviors from these limitations are also the same here.
- Reliance on "fit" and "transform" keywords, and limited model detection:
  The quick fix algorithm identifies preprocessing operations by looking for usage of the fit and transform methods. However, accurately detecting which model is the intended one is difficult, as there may be multiple models instantiated, and the correct one is not

always obvious from static analysis. Furthermore, the algorithm only proceeds with a replacement if fit or transform are detected on the model. To partially address this, a temporary fallback mechanism was added: if the algorithm cannot find a matching model or preprocessing operation, it inserts placeholder preprocessing calls into the code (Figure *34*). This allows the user to manually insert the correct operations afterward, ensuring the program remains functional but may require manual attention.

```
import pandas as pd
from sklearn.feature_selection import (SelectPercentile, chi2)
from sklearn.model_selection import (LinearRegression, Ridge)

X_0, y = load_data()

X_train_0, y_train, X_test_0, y_test = train_test_split(X,y)
select = SelectPercentile(chi2, percentile=50)
select.fit_(X_0)
X = select.transform_(X_0)
fit_model.fit(X_train_0)
X_train = transform_model.transform(X_train_0)
X_test = transform_model.transform(X_test_0)


lr = LinearRegression()
lr.fit(X_train, y_train)
lr_score = lr.score(X_test, y_test)

ridge = Ridge()
ridge.fit(X, y)
ridge_score = ridge.score(X_test, y_test)

final_model = lr if lr_score > ridge_score else ridge
```

Figure 34: Example of fallback mechanism

Multi-test Quick Fix

Multi-test leakages occur when the same test data is repeatedly used for evaluation, blurring the line between validation and true testing. A common example is shown in *Figure* 31, where the variable X_test is evaluated multiple times across both the lr and ridge models, leading to an artificial inflation of performance metrics. A standard solution is to introduce a new, independent test set that has not been reused or influenced by earlier evaluations. Our quick fix algorithm addresses multi-test leakages using the following steps:

1. Identify the reused test variable (e.g., X_test) that is evaluated multiple times across different models.
2. Insert a placeholder loading method, such as load_test_data(), to simulate retrieving a new, independent test dataset.
3. Create new test variables derived from the original test variable (e.g., X_X_test_new_0 and y_X_test_new) through the load_test_data() function.
4. Select a user-initialized model that performs transformations (for example, a scaler or preprocessor).
5. Apply the model's transformation method (specifically looking for the transform keyword) to X_X_test_new_0, resulting in a transformed variable X_X_test_new.
6. Select the last user-initialized model capable of evaluation (for instance, a classifier or regressor).
7. Evaluate the selected model using the newly created X_X_test_new and y_X_test_new variables, ensuring that the evaluation does not reuse the original, potentially tainted test set.

After applying the quick fix, the code for Figure *31* would look like Figure *35*:

```python
import pandas as pd
from sklearn.feature_selection import (SelectPercentile, chi2)
from sklearn.model_selection import (LinearRegression, Ridge)

X_0, y = load_data()

select = SelectPercentile(chi2, percentile=50)
select.fit(X_0)
X = select.transform(X_0)

X_train, X_test, y_train, y_test = train_test_split(X,y)
lr = LinearRegression()
lr.fit(X_train, y_train)
lr_score = lr.score(X_test, y_test)

ridge = Ridge()
ridge.fit(X, y)
ridge_score = ridge.score(X_test, y_test)

final_model = lr if lr_score > ridge_score else ridge
X_X_test_new_0, y_X_test_new = load_test_data()
X_X_test_new = select.transform(X_X_test_new_0)
final_model.score(X_X_test_new_0, y_X_test_new)
```

Figure 35: Example of multi-test leakage

As with the other quick fixes, users are strongly encouraged to review and verify the changes applied to ensure the fixes align with their intended code behavior.

- Scoping Limitations:

  Similar to the overlap and preprocessing leakage fixes, the algorithm does not account for scoping rules or control flow structures (such as loops, functions, or conditionals). As the algorithm scans the file line-by-line, it may incorrectly assume variables are always in scope. The potential undefined behaviors from these limitations are also the same here.

- Train/Test and Model Selection Assumptions:

  The quick fix assumes that the most recently initialized models for transformation and evaluation are the correct ones to use. In cases where multiple models exist and the user's intended usage differs, the algorithm may select the wrong model. The potential

undefined behaviors from these limitations are also the same here as in previous limitations.

- Manual Insertion of Test Loading Method:
  The user is responsible for manually inserting the appropriate data loading logic inside the generated load_test_data() placeholder. Since it is impossible to infer where or how the user wishes to source new test data, the algorithm simply creates the framework but cannot fully automate this step.

Copilot Quick Fix

The extension is fully compatible with other Visual Studio Code extensions that provide AI-assisted code suggestions, such as GitHub Copilot. When the GitHub Copilot extension is installed, its Quick Fix suggestions will appear alongside those generated by our extension.

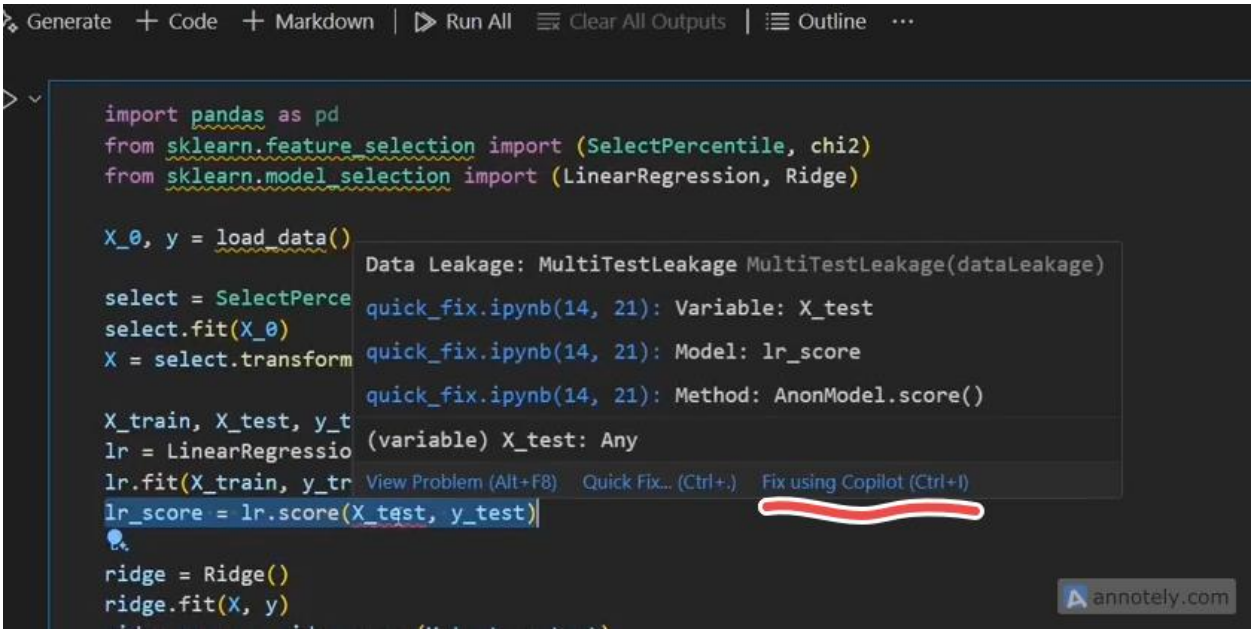


Figure 36: Copilot Option in Popup

Upon selection, a confirmation view of the proposed fix is displayed, with the prompt input bar enabled to facilitate additional user queries or refinement of the suggestion. Any proposed changes are highlighted within the file itself in order to enable easier navigation through the changes. The user is also presented with the option to either accept or dismiss the proposed changes through VS Code's corresponding action buttons ("Accept" or "Close").

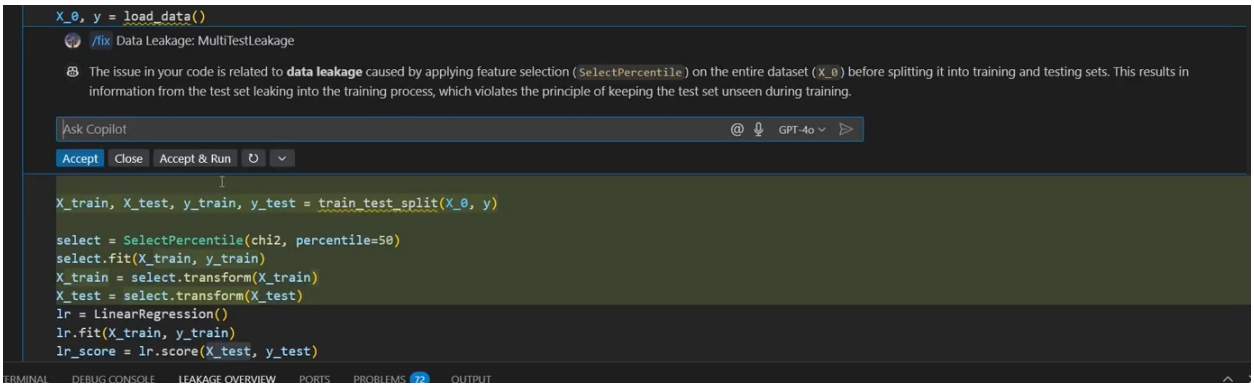


Figure 37: Copilot Diff

Tool Architecture

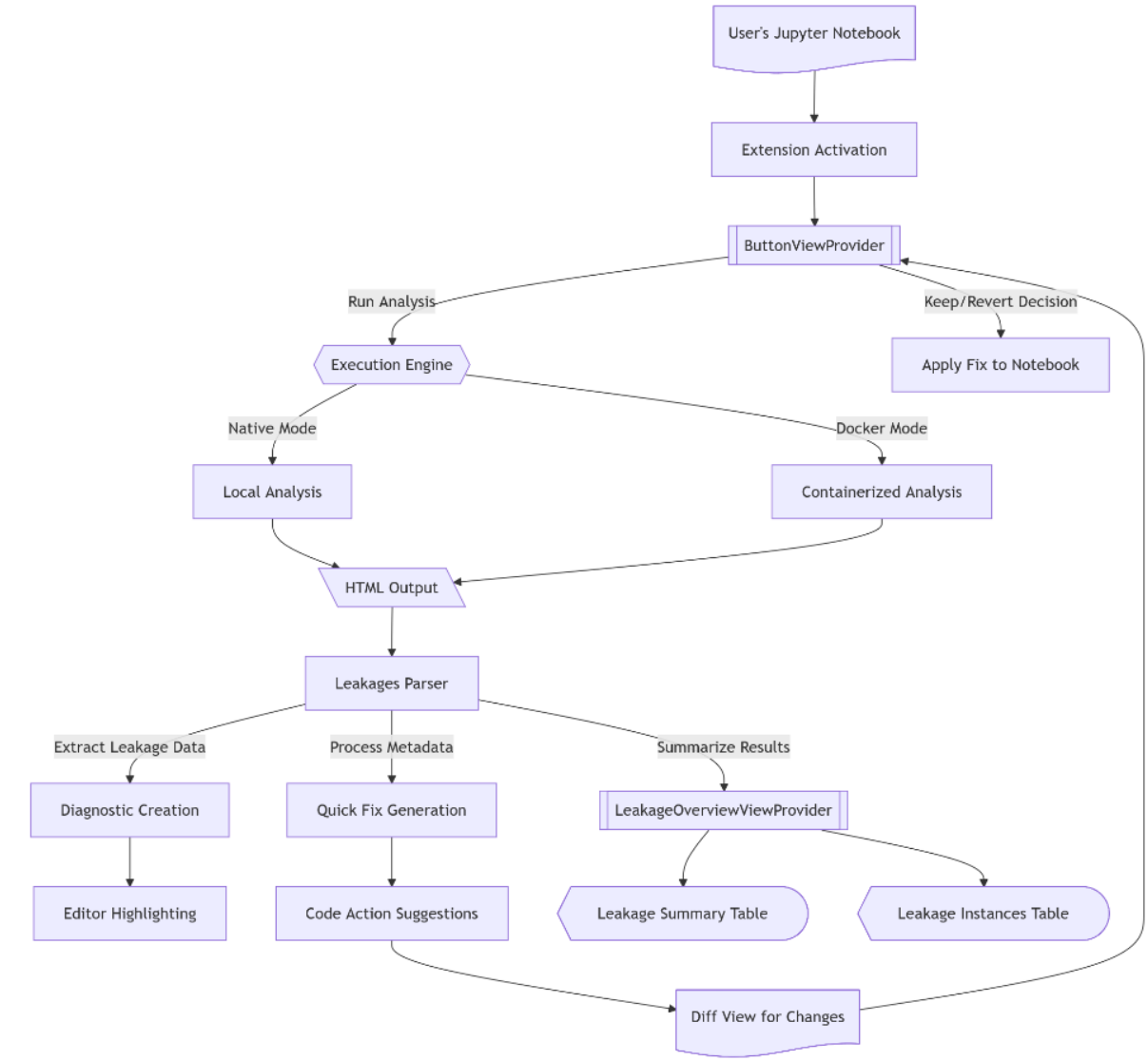


Figure 38: Tool Architecture Made in Mermaid.js

There are four core components to the tool’s architecture:

Main Extension

The main extension module, implemented in extension.ts, serves as the central entry point and coordinator for the entire Leakage Detector system. It initializes the extension context, registers all necessary commands, and establishes the communication channels between different components. This orchestrator is responsible for coordinating the diagnostic collections highlighting leakage issues in notebooks, managing the Quick Fix process through VS Code diff views and decision handling, and connecting user actions in the UI to their corresponding backend operations. When users interact with any part of the extension - whether running an analysis, viewing results, or applying fixes - the main extension component ensures these requests are properly routed and executed.

View Components

The View Components provide the user interface elements that make the extension's functionality accessible and intuitive. The ButtonViewProvider offers a sidebar interface with controls for initiating leakage analysis and managing quick fixes, handling both the main analysis buttons and the dialog for applying or rejecting potential code fixes. Meanwhile, the LeakageOverviewViewProvider creates a panel view at the bottom of the editor displaying detected leakages in two complementary tables - one showing summary statistics of different leakage types, and another providing detailed per-instance information including location, variables involved, and leakage classification. These view components transform complex technical data into actionable information, giving users clear visibility into potential issues and straightforward paths to resolution.

Data Processing

Data processing forms the analytical backbone of the extension, with the Leakages class as its central component for extracting and organizing information from analysis results. This class parses the HTML output from the analysis engine, identifying leakage patterns and extracting crucial metadata like variable relationships, train-test separation details, and line mappings. Supporting this core parser, the Diagnostics Module translates the identified leakages into VS Code diagnostic markers that visually highlight problematic code in the editor with appropriate error messages. The QuickFixManual component completes this workflow by providing practical code fixes for each type of detected leakage, generating modified code that addresses

issues like preprocessing leakage, data overlap, and multi-test data leakage through targeted transformations of the user's Jupyter Notebook code.

Execution Modes

The extension supports two distinct execution modes to accommodate user environments and preferences. Native mode leverages downloaded binaries specific to the user's operating system (Windows, macOS, or Linux) to perform leakage analysis directly on the host machine. This approach provides faster startup times and requires no additional containerization infrastructure, making it ideal for quick analyses and systems with limited resources. Docker mode, on the other hand, runs the analysis engine in a containerized environment that ensures consistent behavior across different platforms by isolating the analysis process from the host system's configuration. This containerized approach provides greater reliability at the cost of requiring Docker Desktop to be installed and running.

We transitioned from a Docker-only approach to include a Docker-free run mode, that is, native mode. The Docker-free run mode gives users the ability to run leakage detection natively on their local machines through downloadable OS-specific binaries; it integrates Pyright type information extraction with custom Python parsing code and a specialized Main.dl executable, eliminating Docker dependencies while maintaining full analytical capabilities. Next, we will explore both execution architectures in more detail, showing how the Docker solution's containerized environment compares to the native implementation's direct system integration.

Docker-Free Solution

The Docker-free solution is a program that runs the leakage algorithm in native mode. The main advantage of this is that running the leakage algorithm will consume significantly less memory and CPU usage over a running Docker container. The solution is packaged as a single zip file that contains all its required dependencies. This allows the installation process of the program to be as simple as possible for the user. Here is a diagram showing the architecture of the Docker-free solution.

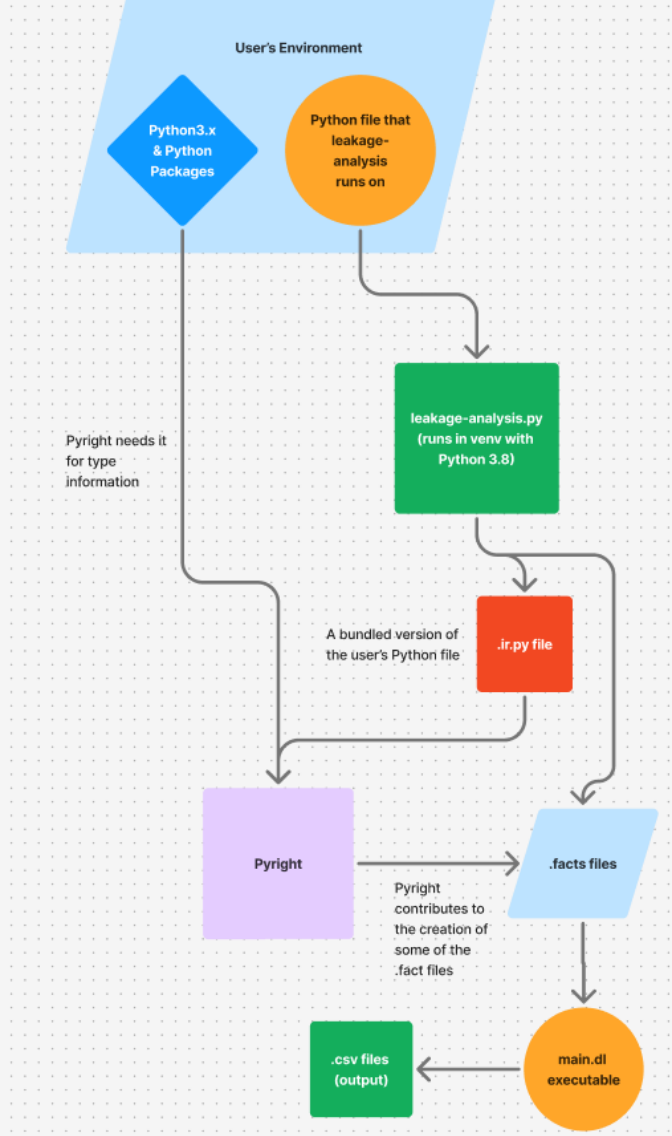


Figure 39: Native Solution Diagram

These are the main components of the Docker-free solution:

1. Pyright Module – A npm package that is run with Node and is responsible for outputting important type information of the Jupyter Notebook code.
2. Python Code – This is the main code that is responsible for collecting and parsing program input as well as calling all the other components as sub processes.
3. Main.dl Executable – This is the executable that runs the leakage algorithm from the input supplied by the program and outputs a set of csv files.

Docker Solution

The Docker solution runs the leakage algorithm within a Docker container environment [14]. The main advantage of this approach is that it provides a consistent and isolated environment that works across different operating systems and configurations. The solution only requires the user to download Docker Desktop and run it. All the complexities of downloading images and running containers are handled by the application, making the process straightforward for users. Here is a diagram showing the architecture of the Docker solution.

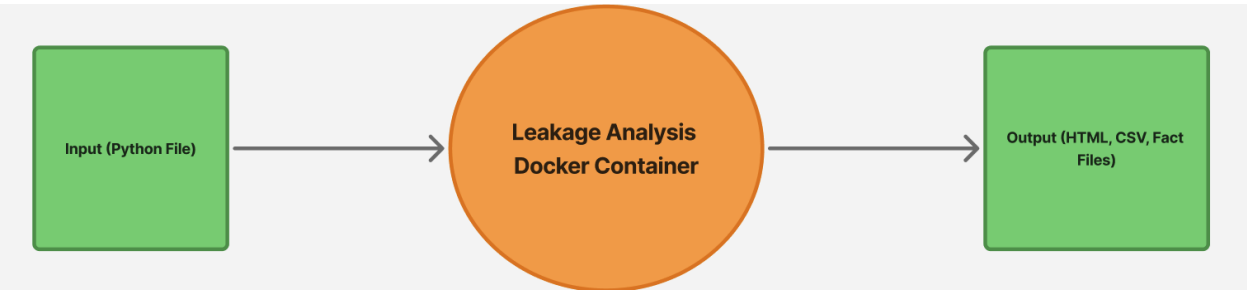


Figure *40*: Docker Solution Data Flow

These are the main components of the Docker solution:

1. Docker Image – A pre-configured environment containing all required dependencies, libraries, and executables needed to run the leakage algorithm.
2. Container Runtime – Managed by the application, this handles the creation and execution of the Docker container that runs the leakage algorithm.
3. Volume Mounting – Facilitates access to local files from within the container, allowing seamless input and output between the host system and the containerized environment.

Development Process for the Main.dl Executable

Developing and prototyping the Docker-free solution involved lots of trial and error. The initial blocker that was encountered was that the leakage algorithm itself was written in a language called datalog. That language relied on an interpreter called “Souffle” and although it was very simple to install on MacOS and Linux, the main issue was that it is incredibly challenging to install on Windows due to the lack of support for it.

One possible solution considered was SWIG. From the official documentation on the Souffle website [13], it stated “Soufflé uses SWIG as a programming interface such that other languages can interface it. This enables a program written in other languages to read in the Datalog input file through the wrapper files and output the corresponding CSV files.”

The main problem with this approach is that it requires Souffle to already be installed on the user's machine which is something that needs to be avoided. The next approach that was considered which ended up working was that one of Souffle's features is to convert any Datalog program into an equivalent C++ program. If that was the case, that C++ program can be compiled to run on every platform. In this way, the installation of Souffle would no longer be required on the end user's machine.

The initial approach that was done for this was to install Souffle's C++ code and libraries inside a Visual Studio project with the intention that Visual Studio would properly link the C++ code to the necessary libraries during compilation. This didn't work at all since there were way too many issues with Visual Studio recognizing those files as a valid project. Based on this, it seems that Visual Studio has its own strict structure for setting up C++ projects. Another approach that was attempted was using a Cross Compiler such as MingW to compile the C++ code for Windows through Linux but there were again issues with finding all the necessary libraries. This was attempted because MingW has ported most of the core functions from the GNU Compiler Collection.

Luckily, the maintainers of the Souffle repository set up a GitHub actions pipeline that would automatically build Souffle on Windows and run many unit tests with it. The windows container that the actions was run on had an environment with all the necessary dependencies and configurations to build souffle. It was also helpful to learn that whenever a repository was forked, all its GitHub Actions would be forked with it. The idea was to make a fork of the Souffle repository and modify its GitHub Action so that it builds Souffle on Windows and uses that environment to compile the main.dl C++ file into an executable that can run on Windows. The main.dl's C++ file was added to the repository so it can easily be used by GitHub Actions.

```yaml
- name: Compile and Test Algo
  working-directory: ${{github.workspace}}
  shell: cmd
  run: |
    pushd "%ProgramFiles(x86)%\Microsoft Visual Studio\2019\Enterprise\VC\Auxiliary\Build" &
    c:/hostedtoolcache/windows/Python/3.12.6/x86/python3.exe d:/a/souffle/souffle/build/src/s
    cd d:/a/souffle/souffle/compilation
    dir
    mkdir d:\a\souffle\souffle\compilation\output
    leakage-algo.exe -F input -D output
    cd d:/a/souffle/souffle/compilation/output
    dir

- name: Upload Compiled Algo and Dlls
  uses: actions/upload-artifact@v4
  with:
    name: leakage-algo
    path: |
      d:/a/souffle/souffle/compilation/leakage-algo.exe
      d:/a/souffle/souffle/build/src/sqlite3.dll
      d:/a/souffle/souffle/build/src/zlib1.dll
    retention-days: 7
```

Figure *41*: GitHub Workflow for Souffle

This is the GitHub Actions code that compiles the C++ code for Windows and automatically uploads the executable and it's necessary dlls to the repository so it can be downloaded later. A similar process was done to compile the C++ code into an executable for Linux inside the Docker container from the leakage-analysis GitHub repository, which already had Souffle installed. When attempting that process on Mac, there were some difficulties, so it was decided to just stick with having users install Souffle due to it simply being done with Brew in a single command.

Porting the Python Code

The main component of the Docker-free solution is that it's built with Python. The issue is that this code is more than 2 years old and doesn't work at all on any recent versions of Python that are higher than Python 3.8. For this use case, there is a python package called "Pyinstaller" which allows you to bundle python code into a single file or folder on the given platform. The major benefit of this solution is that the bundled output is self-contained and will run on the target system, even if that system does NOT have Python installed.

Packaging the Pyright Module

Pyright is a node module that can output type information for python code. One initial issue was that the leakage program uses a custom version of Pyright. This module has small changes to the core code to include some additional types and handle edge cases that are specific to this leakage program. One approach that we considered was using a tool called pkg. This would allow a Node.js module or project to be converted into an executable to run on any target system, even without Node installed. When attempting this, there were too many difficulties due to the syntax of Pyright's code being incompatible with pkg's configurations. We finally decided to use npm to bundle the module into a .tgz file which can be installed on other machines via "npm install". In this way, we can simply distribute that file with our Docker-free solution as a dependency. This would mean that the end user would need to have Node installed on their system, but it isn't much of an issue because of Node's excellent support for all platforms and extremely easy installation process.

Additional GitHub Actions were created to automatically build the Docker-free solution with Pyright on MacOS, Windows, and Linux. The workflow defines a single job called build, which is configured to run in parallel on three operating systems: macOS 13, macOS 14, and Ubuntu 22.04. It uses the actions/checkout@v4 action to clone the repository onto the runner so that the build job can access the project files. It then installs and configures Python 3.8 on the runner using actions/setup-python@v4. This is done because the leakage algorithm code is written using Python 3.8.

A virtual environment (venv) is created and activated in order to safely install additional packages. The environment path is stored in the GitHub Actions environment variable VIRTUAL_ENV for use in later steps. It upgrades pip and installs all required Python packages listed in the requirements.txt file into the virtual environment. The required packages are pyinstaller, to bundle the code into an executable, as well as all the required packages to run the leakage algorithm. Pyinstaller is then run on src/main.py, which compiles the Python script and its dependencies into a standalone executable. This typically creates a dist/main folder containing the compiled application. The workflow then creates a folder at dist/main/_internal/pyright and copies the leakage author's custom version of the Pyright static type checker into it. This is done because Pyright is an internal dependency of the python program.

If the job is running on a Linux runner, it copies all files from res/linux/ to the internal directory of the build (dist/main/_internal). Similarly, if the job is running on macOS, it copies files from

res/mac/ to the same internal directory. Finally, the compiled build (the entire dist directory) is uploaded as an artifact for later use using actions/upload-artifact@v4.

Figure *42*: GitHub Action UI

The generation of HTML output was also reimplemented due to adopting a new method of parsing. More specifically, rather than parsing the CSV files, the HTML file is parsed directly. Although the process of reimplementing HTML generation went smoothly on MacOS and Linux, some difficulties were encountered with Windows. This is because HTML generation required additional packages to be added and compiled which included NumPy. Pyinstaller itself

had some issues compiling NumPy for Python 3.8 on Windows for some unknown reason related to processing DLLs.

```
Traceback (most recent call last):
  File "src\main.py", line 13, in <module>
  File "PyInstaller\loader\pyimod02_importers.py", line 450, in exec_module
  File "src\render.py", line 5, in <module>
  File "PyInstaller\loader\pyimod02_importers.py", line 450, in exec_module
  File "pandas\__init__.py", line 16, in <module>
ImportError: Unable to import required dependencies:
numpy:

IMPORTANT: PLEASE READ THIS FOR ADVICE ON HOW TO SOLVE THIS ISSUE!

Importing the numpy C-extensions failed. This error can happen for
many reasons, often due to issues with your setup or how NumPy was
installed.

We have compiled some common reasons and troubleshooting tips at:

    https://numpy.org/devdocs/user/troubleshooting-importerror.html

Please note and check the following:

  * The Python version is: Python3.8 from "C:\Users\orcio\Downloads\windows-2022-build\main\r
  * The NumPy version is: "1.24.4"

and make sure that they are the versions you expect.
Please carefully study the documentation linked above for further help.

Original error was: DLL load failed while importing _multiarray_umath: The specified module c

[PYI-7480:ERROR] Failed to execute script 'main' due to unhandled exception!
```

Figure *43*: PyRight Error

The way this was addressed was to isolate the code that is responsible for HTML generation and compile it with a newer version of Python 3.10. This isolated code exists as its own Windows executable and is invoked from the original code whenever HTML generation is run.

Testing

Once a pull request is submitted, another person is assigned to review and test the branch before merging into the target branch.

Tests include the following:

- Does button work for Docker solution?
- Does button work for native solution?
- Can you navigate between settings and main page?
- Can you switch between Docker and native solution?
- Can you download binaries?
- Can you install binaries?
- Does diagnostic appear after clicking the run button?

- Can you hover over the diagnostics and select the Quick Fix solution?
- Can the quick fix solution be applied?

Tool Usefulness

The ubiquity and invisibility of data leakage and the convenience of an automated solution are why we chose Leakage Detector as our project. Data leaks can lead to inaccurate predictions, distrust, and, at worst, legal issues, so we hope that having a tool dedicated solely to detecting it will be a major boon to data security. Data scientists, ML engineers, and statisticians can integrate the extension into their regular environments with no issue. By automatically detecting leakage and providing solutions, and by showing the exact type of leakage and the variable that caused it, the developers will get a detailed look on what generally causes this type of leakage. The quick fixes offered by the extension double as teaching good coding practices, so if a developer uses the extension consistently for a long time, they could adopt the suggested coding practices habitually.

We've worked hard and tested the extension thoroughly to ensure the analysis is as in-depth and accurate as possible, and it's our hope that that can improve the quality of research. It is known that data leakage is a common problem in predictive models, but Leakage Detector can be used to give an idea of just how often it occurs and its consequences on model performance. By automating detection, the extension will help greatly in examining several files' worth of datasets and finding patterns of leakage. Seeing common causes of each type and recognizing those patterns can inform experts on how to improve training models. Even after the leakage instances are fixed, they can be logged for personal reference to recognize similarities in future instances to past ones. Findings that benefit one developer or researcher can be shared to benefit the entire community.

Instructors in machine learning or statistics can also benefit by using the extension to teach students about the types of leakage and why they occur. Showing the type of leakage and directing the user to the variable that caused it makes the lesson more interactive and intuitive. That interactivity can be further enhanced by asking students to download the extensions for themselves to use on example files or making assignments to use the output from the extension to explain individual occurrences of leakage. With enough usage, Leakage Detector could become a mainstay in university courses this way.

Publishing

We have automated the publishing of our extension onto VS Code marketplace on Microsoft Azure with the usage of GitHub Action in `/.github/workflows/publish.yml` in our DataLeakage_JupyterNotebook_Fall2024 repository. More is explained below.

Steps For Setting Up Publishing

1. Created Azure DevOps organization and received Personal Access Key (PAT).
2. Created Visual Studio Marketplace publisher.
3. Installed node-loader to package .node files.
4. Got main branch up to date and logged in to vsce with the access key in the embedded VS Code terminal.
5. vsce publish
6. Set up automated CI/CD with .github/workflows/publish.yaml
7. Add vsce PAT as a secret in Settings tab
    a. Navigate to “Settings” tab [Step a]
    b. Open “Secrets and variables” [Step b]
    c. Click “Actions” [Step c]

d. Click “new repository secret” [Step d]

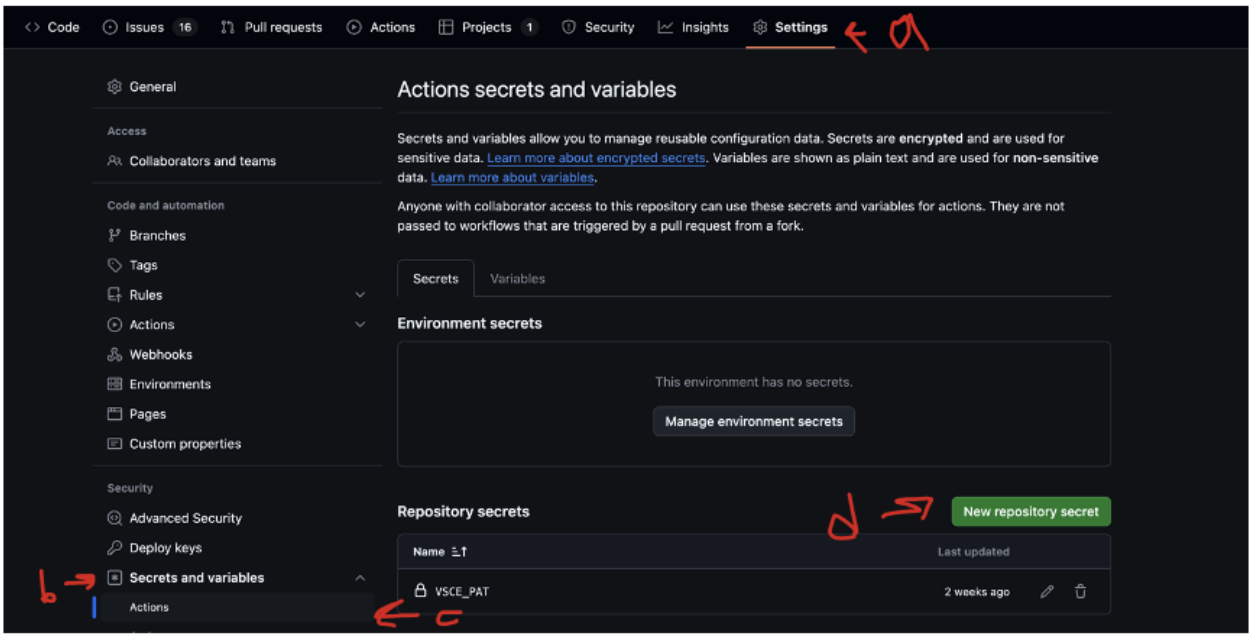


Figure *44*: Actions secrets and variables page in GitHub

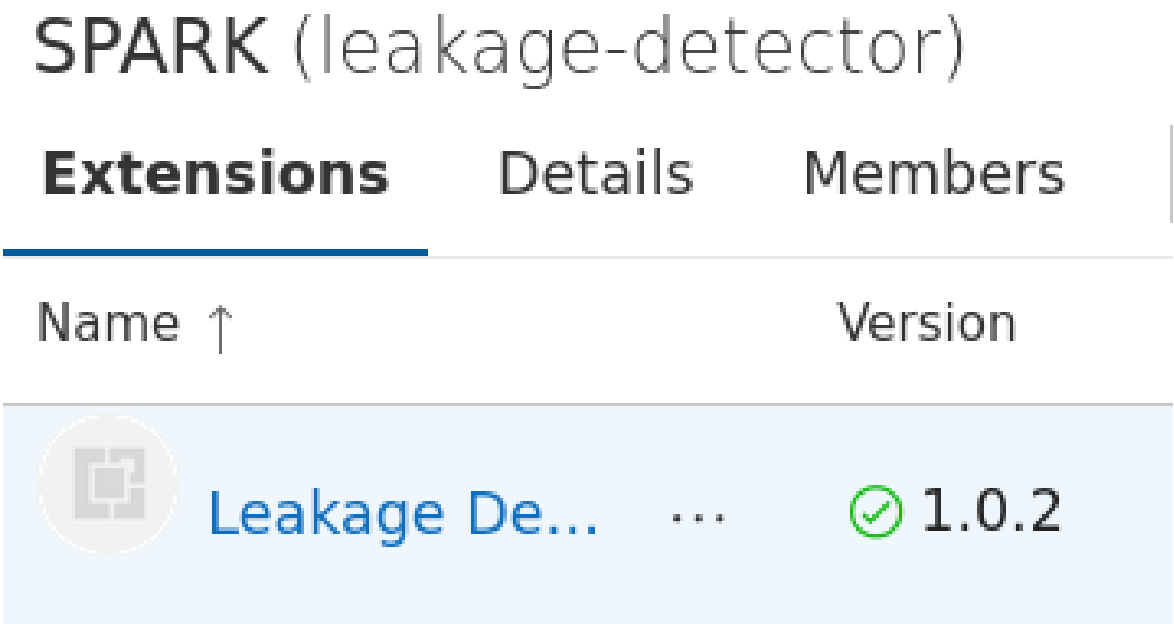


Figure *45*: Azure Page for VSCode Publishing

GitHub Actions publishes automatically on each push to the `prod` branch, using the publish.yaml file. It is recommended to create a pull request from `main` branch to `prod` branch

for the push so that another person can check for any problem before pushing and publishing to production. The changelog and the version number in package.json still need to be updated manually beforehand. A personal access token generated by Dr. AlOmar on the Azure DevOps page is used to log in to vsce.

Azure DevOps only allows personal access tokens with an expiration date, for added security. If the token expires, the extension will not be able to be published, so a new one will need to be generated using.

Troubleshooting

Token Expiration

Personal Access Tokens have an expiration date of 1 year. 1 week before the token expires, Azure will send an email reminder about the token expiration [13].

Changing Token

1. Generate a new Personal Access Token on Azure.
2. Navigate to “Actions secrets and variables” page [Figure 38].
3. Click the edit icon button for the VSCE_PAT repository secret [Step 3].

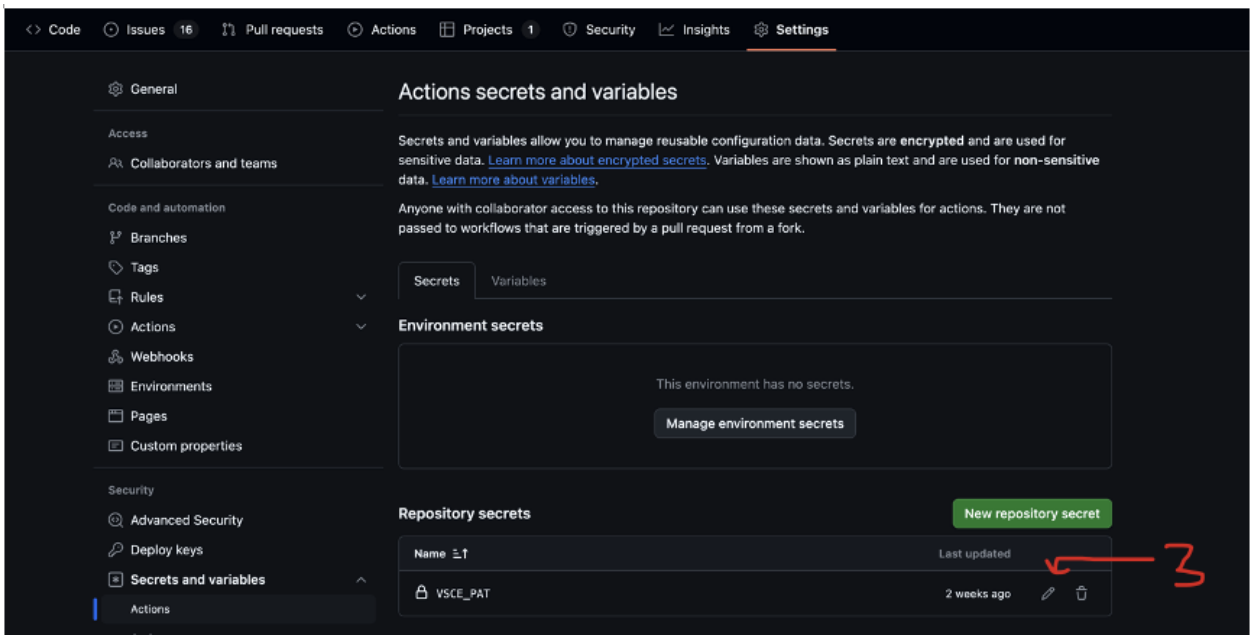


Figure 46: Edit VSCE_PAT Secret

GitHub Action Failing

After the token expires, when push is made to `prod` branch, the "publish" GitHub Action will fail. To resolve this issue, follow the same steps as the previous section, "Changing Token", and then do the following in VS Code while inside Leakage Detector folder:

1. Open terminal with `Ctrl + ~` (or `Command + ~` on Mac).
2. Type and run `git checkout main`
3. Type and run `git pull`
4. Type and run `git branch -D prod`
    a. If there is an error message that `prod` branch does not exist, ignore the message and go to the next step.
5. Type and run `git checkout -b prod`
6. Type and run `git push --force`

Comparative Analysis

Introduction

Our extension has been built upon two previous generations of the leakage detector tool. First, there was the python program, "Leakage Analysis," developed by Chenyang Yang et al [7]. For their "Data Leakage in Notebooks: Static Detection and Better Processing" paper, there was the PyCharm plugin developed by the previous SPARK team from Stevens Institute of Technology in 2023-2024 that built upon the original Python program, "Leakage Analysis," by not only providing a graphical user interface for running and analyzing the leakages using "Leakage Analysis" algorithm inside Docker, but also providing quick fix solutions [9]. Then, finally, we have the Leakage Detector VS Code Extension developed between 2024-2025 that is like the PyCharm plugin, but instead, does analysis on Jupyter Notebooks with a choice of either running the core algorithm, "Leakage Analysis" natively or in Docker [8]. Not only that, but the extension also has automatic deployment so future teams and developers will have an easier time with working on the extension.

Advantages and Disadvantages List

What are the advantages and disadvantages of the three?

Leakage Analysis Python Program

- Advantages
    - IDE independent / Portable.

        - The original Leakage Analysis Python Program is not attached to any IDE or code editors. Because of that, the program can work anywhere as long as the dependencies are installed correctly.
- Disadvantages
    - Souffle is a required dependency, and it is very difficult to install on Windows [1].
        - May have to use Docker.
    - UI is poor
        - If the user wishes to analyze the leakages, they will need to parse the generated HTML file, CSV and Fact files with their eyes.
    - Does not work with Jupyter Notebook files without converting to a Python file.

Leakage Analysis PyCharm Plugin

- Advantages
    - Naive quick fix
        - Provides a rudimentary solution to leakages inside PyCharm GUI. Does not work most of the time.
    - Table display for analyzing leakages.
        - Contains 2 tables, “Leakage Summary” and “Leakage Instances,” for the convenience of the user to assist them in analyzing the leakages.
            - In the PyCharm Plugin, there are only 4 columns
- Disadvantages
    - Requires Docker.
        - The PyCharm plugin requires Docker in order to run the Leakage Analysis Python program regardless of the host operating system.
    - IDE dependent.
        - Only usable in PyCharm Plugin.
    - Does not work with Jupyter Notebook unless the notebook is converted to a Python file.

Leakage Analysis VS Code Extension

- Advantages
    - Slightly less naive quick fix

        - Provides a less rudimentary solution to leakages inside VS Code GUI by providing fixes to some extra edge cases. The solution provided does not work sometimes.
        - Also provides a UI for showing quick fix changes before and after change called “diff.” The user can choose to either discard or approve the changes to their Jupyter Notebook file caused by the quick fix solution in the main page of the extension.
    - AI Quick Fix
        - VS Code has a wide variety of extensions including Copilot and Continue (Continue is open source and privacy-friendly alternative to Copilot), which are two extensions that specialize in LLM integration with VS Code. Leakage Analysis VS Code Extension is compatible with these extensions.
    - Table display for analyzing leakages.
        - Similar to Leakage Analysis PyCharm Plugin. However, “Leakage Instances” contain more columns like “Data Variable Name”, “Model Variable Name” and “Method”. For more detail, see the Extension section of the document.
    - Docker & Native binaries (i.e. binaries executable by host operating system) version of algorithm for all OS.
        - The user can choose to either use Docker for the Leakage Analysis Python program or use our distributed binaries of the Python program to analyze leakages.
- Disadvantages
    - IDE dependent.
        - Only usable in VS Code.
    - Only works with Jupyter Notebook files and unusable with Python files.

Analysis

The original Python program is very portable and flexible, but dependency is annoying to install on Windows and so may require Docker. Not only that, it also only works on Python files, so the need to convert Jupyter Notebooks to Python files is a must before analysis. Beyond that, it is

hard to get a glimpse of all of the leakages from the HTML file since the leakages are spread out in the HTML file.

This was what the PyCharm plugin went out to solve – provide a way for data scientists an overview of all of the leakages, and also a quick fix solution. The problem however is that it only works in PyCharm and like the original python program, it only works with Python files, and it had dependency issues similar to the original Python program. Because of that, the PyCharm Plugin requires users to run Docker to use the plugin.

Then there is the VS Code extension, which solves the dependency issue by distributing binaries for each of the three major operating system, Linux, Windows, MacOS, and half solves the problem of only working Python files by allowing Jupyter Notebooks to be used too. Docker is no longer a necessity but can still be used instead of these binaries, and the user is provided with the option to choose their preferred method within the extension.

The extension also provides a less naïve solution of quick fix compared to the plugin. After running an analysis on the Jupyter Notebook file, the user may select the “quick fix” option for a given line of code to receive suggested edits that will remove the associated leakage instance. This allows users to quickly identify and correct instances of data leakage using the given corrections. In addition to the built-in quick fix methods, the extension also offers integration with Github Copilot to provide LLM-powered solutions to leakage instances. This creates a more dynamic system that provides users with multiple options to resolve the issues in their code. The only problem is that the extension only allows Jupyter Notebooks - this reverses the original problem. However, this can very much be solved by future teams that work on the project as VS Code allows both usage of Jupyter Notebook and Python files unlike PyCharm Community Edition. Beyond that, VS Code extension tables have more data than the PyCharm equivalent so more analysis can be done in VS Code than in PyCharm. However, both have the inherent problem of not being portable – the extension is only usable in VS Code and the plugin is only usable in PyCharm.

Therefore, it depends on what you need. If you use Jupyter Notebook or don’t want to use Docker, you may want to use the VS Code extension. If you use Python, you may want to use PyCharm plugin. If you use Python and don’t mind having to manually look over the HTML files and prefer the flexibilty of the IDE, use the original Python program.

Limitations and Future Work

In our implementation of Leakage Detector in VS Code, there are some bugs and features that we were not able to get to.

LLM Integration

For better control of how an LLM interacts with our extension, we should consider integrating LLM directly into our application. An example of better control is that by having full control, we will be able to not only send the ipynb file to the LLM, but we can also send our table data and the associated HTML, CSV and Fact files for potential parsing by the LLM. Future team could work upon this by adding an LLM section in the settings page where the user can fill in OpenAI API compatible endpoints (many LLM services and open-source software related to hosting LLM are compatible with OpenAI's API) for the extension to query LLM models from.

Combining Download and Installation Button

The requirement for the user to download the binary, unzip it, create a "leakage" folder and move the unzipped folder inside the "leakage" folder, and finally have the user press the installation button to then select the "leakage folder" is convoluted. Future teams could work upon this by eliminating the need for an installation button. The moment the user clicks the download button, it should do all the steps for the user (i.e. download, unzip, create a "leakage folder", etc.).

Allow Analysis of Python Files

A major drawback of our extension is that it only works for Jupyter Notebooks. Unfortunately, due to major refactoring needs and priority for other features, the current team was not able to incorporate the analysis of Python files as a feature. Future teams could work upon this by refactoring the codebase to allow both Jupyter Notebook and Python file analysis. Specifically, the diagnostic related code in "src/data/Diagnostics/" and the Python line conversions in "helpers/Leakages/createLeakageAdapters.ts".

More Comprehensive Quick Fix

Our quick fix algorithm still has room for improvement. Most notably, it currently lacks awareness of variable scope within the user's file. As a result, it may attempt to reference variables that are out of scope in the context of the quick fix, potentially leading to undefined behavior. Future teams could enhance the algorithm by incorporating scope analysis to ensure

that suggested fixes only reference variables available in the appropriate context. Since many of our other known limitations stem from this core issue, addressing it could also pave the way for resolving those secondary problems more easily.

Loading Bar in Extension View for Better UX

The current loading circle at the bottom of VS Code may not be enough as it is quite tiny. The disabling of the run button may also not be enough. Future teams could work upon this by adding a fake loading bar to the main page. A real one is not possible as there is no way to judge how long the leakage analysis will take.

Group ID

Group ID could be added in the future to prevent confusion with relationships between different rows. Currently, it is possible for multiple rows in Leakage Instances table to be from the same leakage. There is no way to tell if multiple rows are related to each other and this may cause confusion. Future teams could work upon this by researching CSV & Fact files to figure out how to group the rows in Leakage Instances.

Team Experience and Final Thoughts

Owen - Initially, I found it challenging to comprehend team members' roles and track project progress. I addressed this challenge by actively asking questions and involving myself in various aspects of the project. The implementation of GitHub issues and the Kanban board proved instrumental in monitoring progress and facilitating effective task management.

Jeffrey - This project was a very valuable experience and taught me a lot that I will take with me in my future career. I think the team worked well together and everyone did their part to contribute towards the final product. I'm glad I had this opportunity and wish everyone involved the best going forward.

Michael - Working with a team and a professional client was thrilling, even if I wasn't accustomed to it at first. I got the chance to learn alongside everyone else, and everyone having assigned tasks helped us with productivity. When I had a question or trouble with a task, I was never hesitant to ask or has it out with someone.

Ryan - My time working on the VS Code extension team was challenging, particularly with adapting to the VS Code API and UI framework. Initially, I encountered teething issues navigating these platforms, but valuable feedback from previous team members and comprehensive online documentation helped me overcome these challenges [10–12]. Throughout the project, I frequently needed to make decisive calls to cut through ambiguity and keep development moving forward. Drawing on my prior web development background, I built the project documentation website using Next.js and Material UI, creating an intuitive front-end experience for users. While the overall process was rewarding, I wish we implemented the HTML parsing approach sooner. This delay limited our capacity to refine quick fixes and other quality-of-life extension improvements that would have enhanced the user experience. Despite these constraints, the project significantly expanded my understanding of extension development across multiple IDE ecosystems.

Arnav – My time working on this extension with the team was incredibly challenging and a great learning process. A lot of the work I was given was very different from the things I studied in my classes. I had to do a lot of research and experimentation whenever tackling challenges head on,

especially figuring out how to develop a working docker free solution. One big challenge I tackled was removing Souffle as a dependency due to its difficult installation process. Overall, it was a great experience getting to work with a team as well as learning about VS Code extension development.

Terrence – Like the rest of the team, I found working on this extension to be a significant challenge. Much of the difficulty stemmed from trying to interpret the leakage program's generated CSV and Fact files. I initially focused too much on replicating the previous team's approach and overlooked a simpler, more effective solution—parsing the generated HTML file directly. Fortunately, I eventually recognized this and was able to get the analysis working successfully. Overall, this year has been an incredible learning experience, giving me valuable insight into collaborative software development.

References

1. https://souffle-lang.github.io/
   - Official Documentation for the Souffle Interpreter
2. https://www.jetbrains.com/lp/annualreport-2023/
   - JetBrains 2023 Report
3. https://www.jetbrains.com/lp/annualreport-2024/
   - JetBrains 2024 Report
4. https://marketplace.visualstudio.com/items?itemName=ms-toolsai.jupyter
   - VS Code's Jupyter Notebook Extension
5. https://marketplace.visualstudio.com/items?itemName=ms-python.python
   - VS Code's Python Extension
6. https://github.com/malusamayo/leakage-analysis
   - Original Leakage Analysis Python Program
7. https://dl.acm.org/doi/10.1145/3551349.3556918
   - Leakage Analysis Python Program Paper
8. https://leakage-detector.vercel.app
   - Our website URI
9. https://se4airesearch.github.io/DataLeakage_Website_Fall2023/pages/resources/
   - Previous Data Leakage team's website URI
10. https://code.visualstudio.com/api/ux-guidelines/overview
    - UX guide of VS Code's UI architecture and elements
11. https://code.visualstudio.com/api/working-with-extensions/continuous-integration#automated-publishing
    - VS Code guide to automate publishing of VS Code extension
12. https://code.visualstudio.com/api/references/VS Code-api
    - VS Code API for VS Code extensions
13. https://developercommunity.visualstudio.com/t/get-notifications-for-expiring-personal-access-tok/501413
    - VS Code Publishing Token Expiration Thread
14. https://hub.docker.com/r/owentruong/leakage-analysis
    - Docker Image for Leakage Analysis Algorithm