DABCheck: Um Plugin para Detecção de Mudanças Disruptivas em Argumentos-Padrão de Bibliotecas Python

Gabriel Lima Barros*
Departamento de Ciência da Computação, Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
gabriellimab@ufmg.br

João Vítor Bicalho*
Departamento de Ciência da Computação, Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
joao.silva@dcc.ufmg.br

João Eduardo Montandon
Departamento de Ciência da Computação, Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
joao@dcc.ufmg.br

RESUMO

No ecossistema Python, os valores-padrão são amplamente utilizados para simplificar o uso de métodos e funções em APIs. Esse recurso permite que métodos sejam chamados sem a necessidade de especificar todos os argumentos, atribuindo automaticamente valores-padrão aos argumentos omitidos. Acontece que, ao modificar o valor-padrão de um argumento, o comportamento dos clientes que dependem deste valor-padrão também são afetados. Essa mudança disruptiva é conhecida como Default Argument Breaking Change (DABC). Neste artigo, apresentamos o DABCheck, um plugin para detectar chamadas de métodos expostas a DABCs. Desenvolvido para as IDEs PyCharm e Visual Studio Code, o plugin funciona como um linter, analisando o código-fonte em tempo real e destacando chamadas vulneráveis. Atualmente, ele suporta a detecção de DABCs em três bibliotecas Python: Scikit-Learn, NumPy e Pandas. O artigo detalha as funcionalidades do plugin, sua arquitetura em cada IDE, e apresenta um exemplo de uso.
Demo Video: https://youtu.be/Ht_oRvqnsFs.

PALAVRAS-CHAVE

Mudanças Disruptivas, Argumentos-Padrão, APIs, Python.

1 Introdução

O uso de componentes desenvolvidos por terceiros se tornou uma prática comum no desenvolvimento de software moderno [12]. Atualmente, todo software de médio ou grande porte utiliza bibliotecas e frameworks na implementação de suas funcionalidades. Por exemplo, testes de unidade são implementados com auxílio de bibliotecas como o PyTest [2], enquanto interfaces gráficas são criadas por meio de frameworks GUI como o React [5].

No ecossistema Python, a utilização de bibliotecas é uma prática amplamente difundida. De acordo com o PyPI[1]—principal repositório de pacotes Python—6,8 milhões de releases estão disponíveis para download, abrangendo mais de 600 mil pacotes distintos. Este ecossistema tem se destacado como uma das principais plataformas para desenvolvimento de soluções voltadas para Ciência de Dados e Aprendizado de Máquina [24]. Enquanto cientistas de dados utilizam bibliotecas como NumPy, Pandas e Scikit-Learn para análise de dados e construção de modelos preditivos, frameworks como PyTorch e Tensorflow se tornaram a escolha padrão para o desenvolvimento de modelos de aprendizado profundo [1, 16, 25].

Geralmente, essas ferramentas disponibilizam uma vasta API com milhares de métodos e funções para quem quiser utilizar suas funcionalidades. Alguns desses métodos podem ser complexos serem utilizados, exigindo um número significativo de parâmetros para serem chamados [25]. Por exemplo, o método construtor `SVC` da biblioteca Scikit-Learn—utilizado para instanciar um classificador baseado em SVM—requer 14 argumentos para criar o modelo. Para contornar essa complexidade, os mantenedores definem um conjunto de valores-padrão para esses parâmetros, eliminando a necessidade de especificá-los em todas as chamadas. Na prática, o construtor `SVC()` pode ser invocado sem argumentos, pois todos seus parâmetros possuem valores-padrão definidos.

Por outro lado, depender dos valores-padrão definidos para esses parâmetros aumenta o grau de acoplamento entre a biblioteca e as aplicações que a utilizam; agora os clientes dependem não apenas da compatibilidade a nível de sintaxe, mas também dos valores-padrão atribuídos pelos mantenedores. Em um estudo realizado anteriormente [13, 14], foi analisado a ocorrência desse tipo de violação—denominada *Default Argument Breaking Change* (DABC)—em três bibliotecas populares do ecossistema Python: Scikit-Learn, Pandas, e NumPy. Dos mais de 500 mil clientes analisados, o estudo revelou que aproximadamente 142 mil clientes estão expostos a pelo menos um argumento-padrão modificado pelos mantenedores. Além disso, detectamos que mais de 70% das mudanças realizadas nos valores-padrão alteram o comportamento do método, podendo levar a mudanças disruptivas nas aplicações cliente.

Com intuito de mitigar tais efeitos, este artigo apresenta um plugin para detectar quais chamadas realizadas por um cliente a uma biblioteca estão expostas a DABCs. O plugin, denominado DABCheck, funciona da seguinte forma. Primeiro, ele analisa o código-fonte do cliente e identifica as chamadas de métodos referentes à API de uma biblioteca cadastrada na base de dados da ferramenta. Para cada chamada, ele verifica quais valores-padrão dos argumentos estão sendo utilizados, e se um desses valores foi modificado pelos mantenedores da biblioteca, i.e., se a chamada está exposta a um DABC. Caso essa situação ocorra, o plugin emite um alerta ao desenvolvedor, informando-o sobre a mudança realizada e sugerindo definir um valor para o argumento detectado. Atualmente, o plugin foi implementado para as IDEs PyCharm[2] e Visual Studio Code,[3] e analisa as chamadas realizadas às bibliotecas Scikit-Learn, NumPy e Pandas.

Este artigo está organizado da seguinte forma. A Seção 2 apresenta a fundamentação teórica necessária para entendimento do

*Ambos os autores contribuíram igualmente para este trabalho.
[1]https://pypi.org/, acessado no dia 15/04/2025.
[2]https://www.jetbrains.com/pycharm/
[3]https://code.visualstudio.com/

trabalho. Já a Seção 3 descreve o plugin DABCheck, detalhando seu funcionamento, arquitetura, e dificuldades encontradas durante o desenvolvimento. Um exemplo de uso do plugin é apresentado na Seção 4. Por fim, as Seções 5 e 6 discutem trabalhos relacionados e apresentam as considerações finais do trabalho.

2 Fundamentação Teórica

Valores-padrão de Argumentos. Esse mecanismo permite que o desenvolvedor defina um valor-padrão para os argumentos de um método ou função que, por sua vez, serão utilizados caso o usuário não forneça valores para esses argumentos durante a chamada do método. Por exemplo, o método `round(number, ndigits=None)`, presente na API padrão da linguagem Python,[4] recebe dois parâmetros e realiza o arredondamento de `number` para o número de casas decimais especificado em `ndigits`. Como se pode observar, o parâmetro `ndigits` possui `None` como valor-padrão. Isso significa que a chamada `round(3.1415)` não somente é válida, como é equivalente a `round(3.1415, None)`. Em outras palavras, o valor `None` foi atribuído automaticamente ao argumento `ndigits`. Neste caso, a chamada `round(3.1415)` irá arredondar o valor para `3`, o inteiro mais próximo. Os valores-padrão são amplamente utilizados em APIs de bibliotecas e frameworks, pois promovem a flexibilidade e reúso de seus métodos e funções [13, 14].

Mudanças Disruptivas (Breaking Changes). Mudanças disruptivas representam alterações realizadas em uma biblioteca que, de alguma forma, afetam sua compatibilidade com versões anteriores [8]. A literatura classifica essas mudanças em dois tipos: *sintáticas* e *semânticas*. Mudanças sintáticas ocorrem quando é realizada uma mudança na assinatura do método ou função utilizada [3, 19]. Já mudanças semânticas ocorrem quando o comportamento do método é alterado mas sua estrutura permanece intacta [11, 20]. Mudanças semânticas representam um grande desafio para os desenvolvedores, pois geralmente passam despercebidas durante a atualização da biblioteca, i.e., não levam a erros explícitos de compatibilidade [13, 14]. Isso faz com que a mudança no comportamento do método seja detectada muito posteriormente.

Default Argument Breaking Changes. Os mantenedores de bibliotecas e frameworks podem atualizar os valores-padrão dos argumentos de seus métodos e funções para atender a novas demandas. Esse tipo de mudança não altera a estrutura do método—ele continua a aceitar os mesmos argumentos—mas pode modificar seu comportamento, levando a mudanças disruptivas nas aplicações que utilizam a biblioteca. Considere o construtor método `SVC()`[5] da biblioteca Scikit-Learn como exemplo. Na versão 0.22 da biblioteca, o valor-padrão do argumento `gamma` foi alterado de `"auto"` para `"scale"`. Esse argumento define a fórmula matemática para calcular o coeficiente do kernel a ser utilizado pelo classificador. Indiretamente, a mudança no valor-padrão deste argumento levou à mudança da fórmula utilizada para calcular o valor de `gamma` em todos os clientes que não especificaram um valor para esse argumento.

O termo *Default Argument Breaking Change* (DABC) foi introduzido em estudos anteriores para caracterizar esse tipo de mudança em três bibliotecas populares do ecossistema Python: Scikit-Learn, NumPy e Pandas [13, 14]. Já o presente trabalho propõe uma ferramenta para detectar as chamadas de métodos expostas as esses DABCs, automaticamente.

[4] https://docs.python.org/3/library/functions.html#round
[5] https://scikit-learn.org/1.1/modules/generated/sklearn.svm.SVC.html

3 O Plugin

3.1 Introdução

Conforme mostrado na Seção 2, os DABCs são violações introduzidas durante o desenvolvimento do sistema quando um usuário da biblioteca não define um valor para um argumento-padrão que foi modificado pelos mantenedores. Dessa forma, seria mais adequado se uma ferramenta para detectar DABCs auxiliasse os desenvolvedores durante a implementação do sistema, não a posteriori. Portanto, decidimos implementar o DABCheck como um plugin para IDEs. Essa abordagem traz duas grandes vantagens: (a) o desenvolvedor recebe *feedback* imediato sobre a implementação de uma chamada exposta a um DABC, e com isso pode corrigir o problema no instante exato em que ele foi inserido, e (b) a ferramenta tem acesso as informações contextuais disponibilizada pela própria IDE, o que pode auxiliar na detecção de DABCs.

Optamos por desenvolver plugins para duas IDEs: PyCharm e Visual Studio Code. Essas IDEs foram escolhidas devido à sua popularidade entre a comunidade de desenvolvedores. Enquanto o Visual Studio Code é o editor mais utilizado pelos profissionais de desenvolvimento de software—74% utilizam essa IDE regularmente—o PyCharm é a IDE específica para desenvolvimento em Python mais popular, regularmente usada por 15% dos profissionais [22].

A seguir são apresentadas as funcionalidades básicas do plugin, o algoritmo utilizado para detectar as chamadas expostas a DABCs, e a arquitetura de implementação para cada uma das IDEs.

3.2 Funcionamento Básico

O plugin DABCheck foi projetado para funcionar como um *linter*. Um *linter* é uma ferramenta de análise estática de código usada para encontrar erros e inconsistências de programação, como bugs, problemas de estilo, e violações de boas práticas [15, 21, 23]. Esse tipo de ferramenta analisa o código-fonte e fornece feedback imediato ao desenvolvedor, permitindo que ele identifique e corrija problemas antes mesmo de executar o programa.

O DABCheck foi implementado em ambas as IDEs para suportar as seguintes funcionalidades:

(1) **Análise de Código:** O plugin analisa estaticamente o código-fonte aberto no editor da IDE, buscando por chamadas de funções que estão expostas aos DABCs previamente cadastrados na base de dados. A busca e detecção das chamadas é feita com base no algoritmo de correspondência apresentado na Seção 3.3. É importante mencionar que a análise é realizada em tempo real, ou seja, enquanto o desenvolvedor escreve o código no arquivo. Com isso, o plugin fornece *feedback* imediato sobre a presença dos DABCs.
(2) **Destaque da Violação:** Caso uma chamada de função esteja exposta a um DABC, o plugin aplica um destaque visual na linha da chamada, e.g., sublinhar a linha detectada em vermelho. Essa funcionalidade, semelhante a forma como os *linters* destacam os erros detectados, permite que o desenvolvedor localize rapidamente as chamadas expostas a DABCs.

Figura 1: DABCheck em funcionamento na IDE Visual Studio Code.

(3) **Exibição de Mensagem de Alerta:** Ao passar o mouse sobre a chamada destacada, o plugin exibe uma mensagem de alerta informando-o sobre a presença do DABC. Essa mensagem contém informações detalhadas sobre a mudança realizada pelos mantenedores, como o valor-padrão que foi modificado e a versão da biblioteca onde essa mudança ocorreu. Além disso, a mensagem sugere que o desenvolvedor defina um valor para o argumento afetado, evitando assim a exposição ao DABC.

A Figura 1 apresenta uma captura de tela do DABCheck para a IDE Visual Studio Code, destacando cada uma das funcionalidades descritas acima conforme sua enumeração.

3.3 Algoritmo de Detecção

Ambas as ferramentas se basearam no algoritmo implementado por Montandon et al. [13, 16] para detectar as chamadas expostas a DABCs. Para cada chamada de função, o algoritmo realiza os seguintes passos:

(1) Recupera o nome da função que está sendo chamada e verfica se seu nome está presente na base de dados dos DABCs, obtida previamente.
(2) Caso esse nome esteja presente, o algoritmo realiza o pareamento entre os argumentos utilizados na chamada e os parâmetros definidos na assinatura da função. Esse pareamento é feito de duas formas:
   (a) **Posicional:** os argumentos são pareados com base na ordem em que são passados na chamada.
   (b) **Nomeado:** a correspondência é feita a partir do nome passado junto ao valor do argumento.
(3) O algoritmo verifica se o argumento sujeito ao DABC foi definido na chamada. **Se o argumento não foi definido**, o algoritmo considera que a chamada está exposta a um DABC e portanto deve ser corrigida pelo desenvolvedor. Caso contrário, o algoritmo ignora a chamada.

Em ambas extensões, o algoritmo é executado sempre que o desenvolvedor realiza uma alteração no código-fonte do arquivo aberto no editor. Os destaques e mensagens de alerta são aplicados para as linhas que possuam chamadas consideradas expostas pelo algoritmo. Além disso, em ambas as extensões, tanto a identificação dos nomes das funções quanto a extração dos argumentos nomeados e posicionais são realizadas com auxílio de expressões regulares.

3.4 Plugin para PyCharm

O desenvolvimento de plugins para a IDE PyCharm é realizado utilizando a arquitetura IntelliJ Platform, que permite a criação de ferramentas e funcionalidades adicionais para as IDEs da JetBrains. A estrutura do desenvolvimento de plugins é baseada em componentes extensíveis e pontos de integração, conhecidos como Extension Points (EPs), que são utilizados para registrar comportamentos personalizados de acordo com as necessidades do desenvolvedor.

3.4.1 Arquitetura do plugin. A Figura 2 apresenta a arquitetura do plugin DABCheck para o PyCharm. É possível observar que o plugin é composto por dois componentes principais (*MethodHighlighter* e *MethodGutterIconRenderer*), que são responsáveis por suas funcionalidades centrais e uma base de dados (*DABCs Database*). Uma explicação detalhada de cada um desses módulos é apresentada a seguir.

DABCs Database. Essa base de dados armazena as definições dos DABCs detectados em bibliotecas de terceiros. Para cada DABC, armazena-se suas informações como a assinatura da função, o parâmetro afetado, e a versão onde o DABC foi introduzido. Essa base

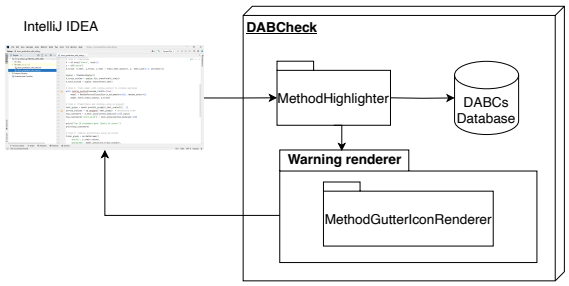

Figura 2: Arquitetura da extensão DABCheck para o PyCharm.

de dados é gerada a partir do trabalho de Montandon et al. [13, 14], que analisaram as bibliotecas Scikit-Learn, NumPy e Pandas.

MethodHighlighter. Este componente realiza a detecção e destaque das funções potencialmente afetadas por DABCs. O MethodHighlighter faz uma análise estática do código, identificando as bibliotecas importadas e correlacionando-as com os arquivos da base de dados que contém informações sobre métodos possivelmente vulneráveis a alterações em argumentos padrão. Quando o código é editado pelo usuário, o módulo atualiza os destaques no editor em tempo real, alertando sobre os riscos dos DABCs.

MethodGutterIconRenderer. Este componente é responsável por gerenciar a exibição de ícones na margem da linha correspondente ao método identificado (*gutter*). Esses ícones não apenas servem como alertas visuais, mas também são interativos, permitindo ao usuário visualizar uma descrição detalhada do aviso ao manter o mouse sobre o ícone. O desenvolvedor ainda pode optar por ignorar o alerta clicando sobre o ícone.

3.5 Extensão para Visual Studio Code

Implementado em TypeScript, a extensão utiliza a API disponibilizada pelo Visual Studio Code para interagir com os recursos do editor. Esses recursos incluem a capacidade de identificar os arquivos abertos no editor, e aplicar destaques visuais nas chamadas de funções que estão expostas a DABCs. O restante da seção descreve as funcionalidades e arquitetura projetada para a extensão.

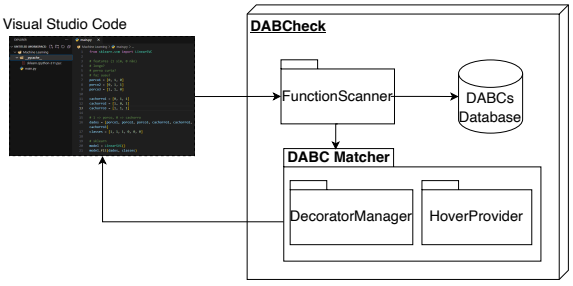

Figura 3: Arquitetura da extensão DABCheck para o Visual Studio Code.

3.5.1 Arquitetura da extensão. A Figura 3 apresenta a arquitetura da extensão para o Visual Studio Code. Como se pode observar, a extensão é composta por dois módulos (*FunctionScanner* e *DABC Matcher*) e uma base de dados (*DABCs Database*). Uma explicação detalhada de cada um desses módulos é apresentada a seguir.

FunctionScanner. O módulo *FunctionScanner* se comunica com a API do Editor do Visual Studio Code, e realiza uma varredura das chamadas de função presentes no arquivo que está em edição. Para cada chamada detectada, o módulo realiza uma consulta ao *DABCs Database* para verificar se existe um DABC associado a função chamada. Caso exista, o *FunctionScanner* encaminha a chamada detectada e a definição do DABC para o módulo *DABC Matcher*.

DABC Matcher. Este módulo verifica se a chamada detectada está exposta a um DABC. Para isso, ele analisa tanto a chamada detectada quanto a definição do DABC recebidos do módulo *FunctionScanner*. Como já mencionado na Seção 3.3, uma heurística é aplicada para verificar se um valor para o argumento sujeito ao DABC foi definido na chamada. Caso o valor não tenha sido definido, o *DecoratorManager* é acionado para aplicar um destaque visual na linha correspondente a chamada. Além disso, toda vez que o desenvolvedor passa o mouse sobre a chamada detectada, o *HoverProvider* é acionado para exibir uma mensagem de alerta sobre a existência do DABC.

3.6 Dificuldades Encontradas

As principais dificuldades enfrentadas durante o desenvolvimento estiveram relacionadas à escassez de informações e materiais de apoio. Embora o Visual Studio Code disponha de uma documentação oficial sobre a criação de extensões, ela é genérica e não abrange com profundidade funcionalidades mais específicas da ferramenta. Além disso, há poucos exemplos práticos e tutoriais disponíveis, o que torna o processo de implementação mais complexo.

Para o caso do IntelliJ Platform, as dificuldades com escassez de informações e materiais de apoio se repetiram. A documentação oficial da JetBrains sobre a criação de extensões é eficaz para as primeiras configurações, apenas. Além disso, a versão community do IntelliJ não permite utilizar o parser do código Python. Esse recurso, disponível apenas na versão Ultimate, teria facilitado a análise sintática do código e a detecção dos DABCs projeto.

4 Exemplo de Uso

Neste artigo, é apresentado um exemplo de uso do funcionamento do plugin para a IDE PyCharm. Já a extensão para o Visual Studio Code é apresentada no vídeo de demonstração, cujo link foi disponibilizado no início do artigo.

Código de Demonstração. Para ilustrar o exemplo de uso, utilizamos o script presente na Figura 4. Esse mesmo script foi adotado por Montandon et al. [13, 14] para explicar o que é um DABC. Este código possui 21 linhas e realiza a classificação de notícias entre 20 categorias distintas. O script obtém o dataset e as variáveis preditivas (linhas 07–09), para em seguida dividi-los entre conjunto de treinamento e teste (linhas 12–13). Na linha 16, uma nova instância de `SVC` é criada fornecendo apenas o argumento `random_state` para garantir o mesmo grau de aleatoriedade no treinamento e

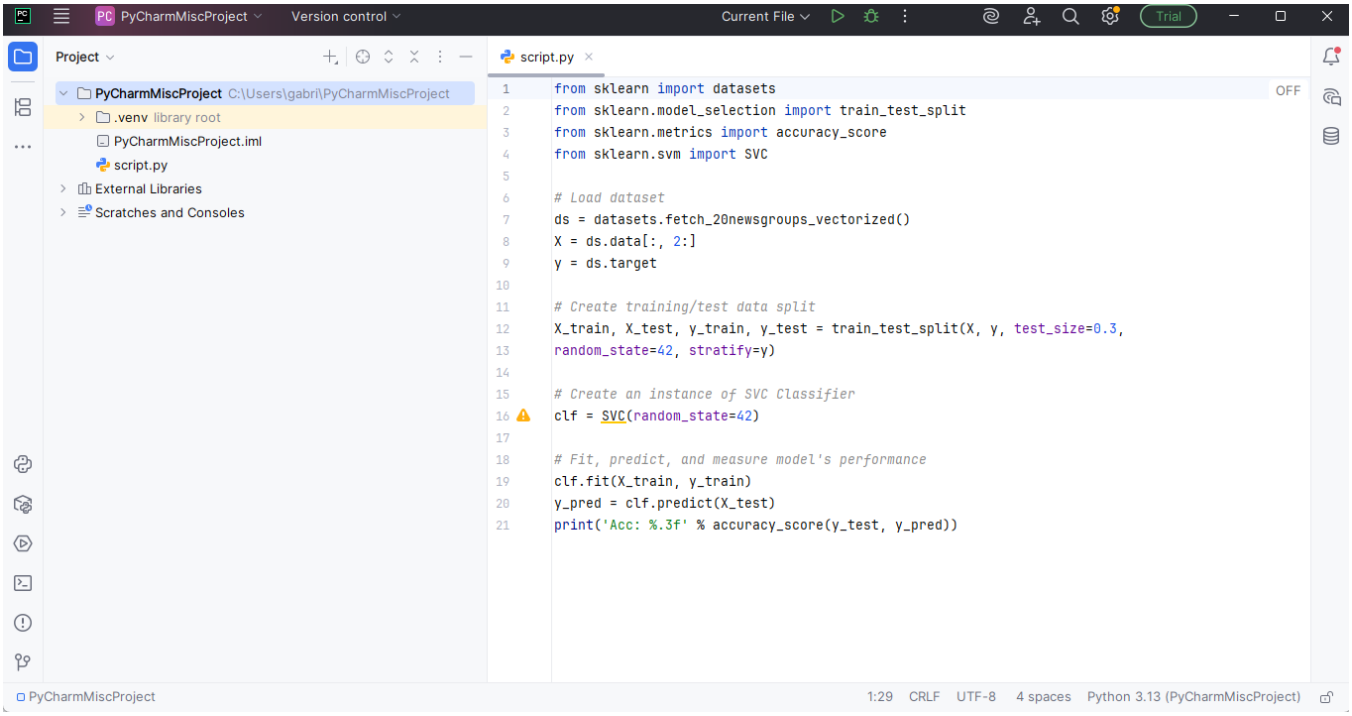

Figura 4: DABCheck em funcionamento na IDE PyCharm.

execução do modelo. Ao final (linhas 19–21), o script realiza o treinamento do modelo e suas predições, e reporta o nível de precisão alcançada.

Neste exemplo, o DABC está presente justamente na linha 16, responsável pela inicialização do classificador SVC. O valor do argumento `gamma`—responsável pela escolha da fórmula a ser adotada no Kernel do classificador—muda de `"auto"` para `"scale"`.

4.1 Exemplo de uso Plugin PyCharm

A utilização do DABCheck na IDE PyCharm tem início com a abertura de um projeto Python, onde o desenvolvedor pode seguir com o fluxo usual de implementação do projeto. No exemplo, o projeto aberto foi inicializado com um ambiente virtual (diretório *venv*) e o arquivo de script em python chamado *script.py*.

O desenvolvedor então seleciona o arquivo para ser aberto no editor da IDE. A partir deste momento, o plugin DABCheck inicia seu processamento analisando as chamadas de métodos presentes no arquivo aberto. O módulo *MethodHighlighter* inspeciona os métodos, identifica se algum deles pertence a uma biblioteca cadastrada no *DABCs Dataset*, e verifica se a chamada é feita sem a definição do argumento exposto ao DABC. Das seis chamadas identificadas no exemplo, apenas a chamada `SVC(random_state=42)` está exposta, i.e., não há valor definido para `gamma`. A chamada é então automaticamente destacada, fornecendo feedback visual ao desenvolvedor.

As informações da linha destacada são enviadas ao módulo *MethodGutterIconRenderer* que, por sua vez, complementa a sinalização renderizando um ícone de alerta na margem da linha correspondente. Uma janela do tipo *tooltip* é exibida ao posicionar o cursor sobre o ícone adicionado; essa janela apresenta detalhes sobre o DABC exposto naquela chamada, como a chamada e o argumento expostos, bem como a versão responsável pela modificação. A Figura 5 apresenta o *tooltip* para o exemplo em questão.

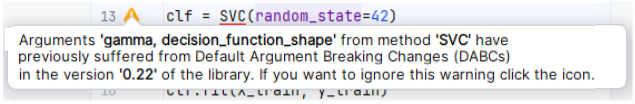

Figura 5: Aviso reportado pelo PyCharm.

Esse mecanismo permite que o desenvolvedor compreenda rapidamente a origem do possível problema e tome a decisão informada. Ele pode optar por ignorar o alerta clicando sobre o ícone ou ajustar o código fornecendo explicitamente um valor ao parâmetro destacado. Em ambas as situações, uma vez solucionado o alerta, o módulo *MethodHighlighter* atualiza o código e remove a marcação correspondente.

5 Trabalhos Relacionados

Análise de Mudanças Disruptivas. Diversos trabalhos analisaram a ocorrência de mudanças disruptivas em componentes de terceiros, i.e., bibliotecas e frameworks [3, 6, 7, 11, 13, 14, 17, 25]. A maioria desses trabalhos se concentrou em analisar mudanças disruptivas em contextos específicos, conforme uma determinada linguagem de programação ou plataforma. Por exemplo, os trabalhos de Brito et al. [3], Ochoa et al. [19], e Mostafa et al. [17] analisaram essas mudanças em bibliotecas do ecossistema Java. Mezzetti et al. [11] analisou

a ocorrência de mudanças semânticas em bibliotecas presentes no ecossistema *npm.js*, do JavaScript. Já Haryono et al. [6], Zhang et al. [25], e Montandon et al. [14] estudaram a introdução dessas violações em bibliotecas de Aprendizado e Máquina e Aprendizado Profundo. Contudo, esses trabalhos se limitaram a caracterizar e quantificar essas mudanças em APIs de bibliotecas e frameworks.

Ferramentas para Detecção de Mudanças Disruptivas. Outros trabalhos se concentraram em desenvolver ferramentas para auxiliar a detecção automática de mudanças disruptivas. Ochoa et al. [18] propuseram o Breaking Bot, um bot que analisa pull requests de bibliotecas Java hospedadas no GitHub para identificar, estaticamente, alterações que levem a mudanças disruptivas. Li et al. [9] descreveram uma técnica para detecção dessas violações em Aplicativos Android através de uma análise realizada no bytecode desses aplicativos. Essa técnica—denominada CiD—foi posteriormente implementada como ferramenta. Du and Ma [4] propuseram um protótipo para detectar as possíveis violações introduzidas entre duas versões de bibliotecas escritas em Python. Recentemente, Mahmud et al. [10] apresentou o APICIA, um analisador que verifica o impacto de mudanças de APIs utilizadas em aplicativos Android.

Diferencial do DABCheck. Embora essas ferramentas sejam úteis para detectar mudanças disruptivas, a forma como foram implementadas limita sua aplicabilidade durante o desenvolvimento de software. Por exemplo, a ferramenta Breaking Bot realiza sua análise após o processo de desenvolvimento, ou seja, quando a funcionalidade já foi implementada. Já a ferramenta CiD deve ser executada por linha de comando, após a compilação do código-fonte do aplicativo. Em outras palavras, o desenvolvedor deverá interromper seu fluxo de trabalho para executar essas ferramentas, representando um obstáculo para sua adoção. O plugin DABCheck foi projetado para ser integrado diretamente ao ambiente de desenvolvimento—similar a um *linter*—permitindo que o desenvolvedor receba feedback imediato sobre essas incompatibilidades a medida em que o seu sistema é desenvolvido.

6 Considerações Finais

Este artigo apresentou o DABCheck, uma ferramenta para detectar chamadas de métodos expostas a *Default Argument Breaking Changes* (DABCs) em bibliotecas do ecossistema Python. A ferramenta foi desenvolvida como plugin para as IDEs PyCharm e Visual Studio Code, duas das IDEs mais populares entre os desenvolvedores que trabalham com Python. Estruturamos o plugin como um *linter*, permitindo que os desenvolvedores recebam feedback imediato—i.e., durante o desenvolvimento—sobre quais chamadas de métodos estão expostas. Para isso, o plugin analisa o código-fonte em tempo real do arquivo aberto no editor, destacando as chamadas vulneráveis e exibindo mensagens de alerta com informações detalhadas sobre a mudança realizada pelos mantenedores. Atualmente, o plugin suporta a detecção de DABCs em três bibliotecas populares do ecossistema Python: Scikit-Learn, NumPy e Pandas.

Trabalhos Futuros. Os próximos passos envolvem aprimorar a estrutura da ferramenta para facilitar seu uso contínuo por parte dos desenvolvedores. Por exemplo, pretendemos modificar a estrutura de armazenamento dos DABCs, permitindo que a base de dados seja atualizada automaticamente por meio de um serviço externo. Também planejamos melhorar o processo de detecção de chamadas, realizando a identificação das chamadas através de uma análise sintática da AST do arquivo em aberto. Por fim, pretendemos implementar mecanismos de monitoramento e telemetria para medir o uso da ferramenta, e então avaliar sua eficácia em cenários reais.

DISPONIBILIDADE DE ARTEFATO

Os plugins para PyCharm e Visual Studio Code estão disponíveis publicamente para download e uso no seguinte repositório: https://doi.org/10.5281/zenodo.15442932.

AGRADECIMENTOS

Esta pesquisa foi financiada pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), e pela Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG).

REFERÊNCIAS

[1] Mohamed Raed El aoun, Lionel Nganyewou Tidjon, Ben Rombaut, Foutse Khomh, and Ahmed E. Hassan. 2022. An Empirical Study of Library Usage and Dependency in Deep Learning Frameworks.
[2] Livia Barbosa and Andre Hora. 2022. How and Why Developers Migrate Python Tests. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 538–548.
[3] Aline Brito, Marco Tulio Valente, Laerte Xavier, and Andre Hora. 2020. You Broke My Code: Understanding the Motivations for Breaking Changes in APIs. *Empirical Software Engineering* 25, 2 (March 2020), 1458–1492.
[4] Xingliang Du and Jun Ma. 2022. AexPy: Detecting API Breaking Changes in Python Packages. In *33rd International Symposium on Software Reliability Engineering (ISSRE)*. 470–481.
[5] Fabio Ferreira, Hudson Borges, and Marco Tulio Valente. 2024. Refactoring React-based Web Apps. *Journal of Systems and Software* 1 (2024), 1–36.
[6] Stefanus A. Haryono, Ferdian Thung, David Lo, Julia Lawall, and Lingxiao Jiang. 2021. Characterization and Automatic Updates of Deprecated Machine-Learning API Usages. In *International Conference on Software Maintenance and Evolution (ICSME)*. 137–147.
[7] Raula Gaikovina Kula, Ali Ouni, Daniel M. German, and Katsuro Inoue. 2018. An Empirical Study on the Impact of Refactoring Activities on Evolving Client-Used APIs. *Information and Software Technology* 93, C (jan 2018), 186–199.
[8] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. 2021. A Systematic Review of API Evolution Literature. *Comput. Surveys* 54, 8 (Oct. 2021), 171:1–171:36.
[9] Li Li, Tegawendé F. Bissyandé, Haoyu Wang, and Jacques Klein. 2018. CiD: Automating the Detection of API-related Compatibility Issues in Android Apps. In *27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) (ISSTA 2018)*. 153–163.
[10] Tarek Mahmud, Meiru Che, Jihan Rouijel, Mujahid Khan, and Guowei Yang. 2024. APICIA: An API Change Impact Analyzer for Android Apps. In *46th International Conference on Software Engineering: Companion Proceedings*. 99–103.
[11] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.Js Libraries. In *32nd European Conference on Object-Oriented Programming (ECOOP)*. 1–24.
[12] João Eduardo Montandon, Luciana Lourdes Silva, and Marco Tulio Valente. 2019. Identifying Experts in Software Libraries and Frameworks Among GitHub Users. In *16th International Conference on Mining Software Repositories (MSR)*. 276–287.
[13] João Eduardo Montandon, Luciana Lourdes Silva, Cristiano Politowski, Ghizlane El Boussaidi, and Marco Tulio Valente. 2023. Unboxing Default Argument Breaking Changes in Scikit Learn. In *23rd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 1–11.
[14] João Eduardo Montandon, Luciana Lourdes Silva, Cristiano Politowski, Daniel Prates, Arthur de Brito Bonifácio, and Ghizlane El Boussaidi. 2025. Unboxing Default Argument Breaking Changes in 1 + 2 Data Science Libraries. *Journal of Systems and Software* (2025), 1–38.
[15] João Eduardo Montandon, Silvio Souza, and Marco Tulio Valente. 2011. Study on the Relevance of the Warnings Reported by Java Bug-Finding Tools. *IET Software* 5, 4 (2011), 366–374.
[16] João Eduardo Montandon, Marco Tulio Valente, and Luciana L. Silva. 2021. Mining the Technical Roles of GitHub Users. *Information and Software Technology* 131 (March 2021), 1–19.
[17] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. Experience Paper: A Study on Behavioral Backward Incompatibilities of Java Software Libraries. In *26th International Symposium on Software Testing and Analysis (ISSTA)*. 215–225.

[18] Lina Ochoa, Thomas Degueule, and Jean-Rémy Falleri. 2022. BreakBot: Analyzing the Impact of Breaking Changes to Assist Library Evolution. In *2022 IEEE/ACM 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 26–30.

[19] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. 2022. Breaking Bad? Semantic Versioning and Impact of Breaking Changes in Maven Central. *Empirical Software Engineering* 27, 3 (March 2022), 61.

[20] A. Ponomarenko and V. Rubanov. 2012. Backward Compatibility of Software Interfaces: Steps towards Automatic Verification. *Programming and Computer Software* 38, 5 (2012), 257–267.

[21] SonarSource. 2025. SonarQube. https://www.sonarsource.com/.

[22] Stack Overflow. 2024. Stack Overflow Developer Survey. https://survey.stackoverflow.co/2024/.

[23] Kristín Fjóla Tómasdóttir, Mauricio Aniche, and Arie van Deursen. 2017. Why and How JavaScript Developers Use Linters. In *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 578–589.

[24] Jiawei Wang, Tzu-Yang KUO, Li Li, and Andreas Zeller. 2020. Assessing and Restoring Reproducibility of Jupyter Notebooks. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 138–149.

[25] Zejun Zhang, Yanming Yang, Xin Xia, David Lo, Xiaoxue Ren, and John Grundy. 2021. Unveiling the Mystery of API Evolution in Deep Learning Frameworks: A Case Study of Tensorflow 2. In *43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 238–247.