Mitigating Code Style Gaps via Reference-Aware Style Transfer

Yingying Jiang[1][0009−0008−9479−3975], Zhiyuan Liu[2][0009−0002−2240−1553], Huiyan Wang[2⋆][0000−0001−6879−1628], and Chang Xu[1][0000−0002−6299−4704]

[1] School of Computer Science, Nanjing University, Nanjing, China
yyjiang1@smail.nju.edu.cn,changxu@nju.edu.cn
[2] Software Institute, Nanjing University, Nanjing, China
zhiyuanliu@smail.nju.edu.cn,why@nju.edu.cn

Abstract. Maintaining consistent code style is essential for software maintainability, yet real-world codebases often exhibit code style gaps arising from different contributions. The widespread adoption of LLMs in programming further exacerbates this issue by introducing stylistic variability and accelerating code production. Existing approaches either focus primarily on formatting while lacking in-depth style investigation or rely on LLM/DL-based modules that often inherit the models' preferred code styles and fail to ensure correct, reference-aware code style consistency. To address these issues, we propose StyleX, an extract-and-apply approach that performs style transfer across 20 widely-investigated styles spanning formatting, syntactic, and semantic aspects. StyleX effectively manages code style consistency by adaptively extracting specific style profiles from reference code and applying style transfer accordingly. Experimental results demonstrate that StyleX outperforms existing learning-based approaches (CodeBuff) and LLM-based approaches (DeepSeek-R1, GPT-4.1), achieving the highest ideal and positive transfer rates in controlled experiments and the highest ideal transfer rate in human evaluations.

Keywords: Code style gaps · Style inconsistency · Style transfer.

1 Introduction

Maintaining code style consistency is essential for improving code comprehension and facilitating collaborative software development. Prior work has shown that consistent style reduces cognitive load [27], streamlines code reviews [36], and can even aid bug localization [15].

Despite its importance, maintaining style consistency in practice remains challenging. Empirical studies have revealed that stylistic inconsistencies are common in real-world projects. Zou et al. [36] analyzed 50,092 pull requests across 117 projects and found that multiple code style criteria showed inconsistencies in over 10% of the requests. Wang et al. [34] identified a wide range of stylistic inconsistencies across both human-written and LLM-generated code. These findings point to the challenge of systematic code style gaps that arise when integrating contributions from diverse, unaligned sources. LLM-assisted programming introduces stylistic variability due to its inherent non-determinism and significantly increases code production velocity [17]. As a result, code style gaps are more likely to accumulate and impose additional burdens on code review, making automated style consistency management more critical than ever for maintainability and seamless integration [36].

Existing approaches for maintaining code style consistency can generally be classified into three categories: (1) Rule-based tools, such as widely used formatters and linters [18, 33], reliably enforce formatting consistency through predefined rules. However, many project-specific styles, such as naming conventions and control-flow idioms, are implicitly reflected in the surrounding code and context-dependent, making them difficult to adapt flexibly using predefined, static rules. (2) Learning-based approaches like CodeBuff [26] and DuetCS [11] offer flexibility by learning from reference code. However, they are limited either in learning only formatting styles or in lacking reliable semantic preservation during style transfer, often violating functional correctness, with instability causing over 44% test failures in some deep learning models [11]. (3) LLM-based approaches leverage LLMs (e.g., GPT-4) that robustly encode stylistic features of code [13], making them promising for code style transfer. However, those approaches tend to favor styles prevalent in

⋆ Corresponding author.

their training corpora, lack adaptive transfer toward specific reference code, and incur non-negligible token consumption.

These limitations point to the need for approaches that capture diverse code styles from reference code and apply them for style consistency while preserving functional correctness. To address this, we propose StyleX, an extract-and-apply approach for adaptive, cost-effective style consistency. StyleX utilizes a meta-representation (*style-schema*) to model diverse stylistic features (five for format, nine for syntax, and six for semantics). Based on the style-schema, StyleX extracts features directly from reference code and performs controlled style transfer through target-directed transformations.

We evaluated StyleX on 387 style consistency tasks derived from real-world GitHub projects. Experimental results demonstrate that StyleX achieves both the highest ideal and positive transfer rates (14.9% and 50.3%), outperforming existing learning-based methods such as CodeBuff (13.2% and 35.5%) and large language models including DeepSeek-R1 (9.2% and 49.0%) and GPT-4.1 (12.5% and 48.4%). Moreover, StyleX achieves 100% functional correctness and offers high cost-efficiency, completing each task in about two seconds without additional token consumption. These findings are further supported by a human evaluation conducted on a subset of tasks, in which StyleX achieves the highest ideal transfer rate of 54.8%.

The remainder of this paper is organized as follows. Section 2 illustrates the motivation and the style investigation. Section 3 details the design of StyleX and its core components. Section 4 describes the experimental setup and presents the evaluation results. Section 5 discusses threats and study limitations. Section 6 reviews related work, and Section 7 concludes the paper.

2 Preliminary

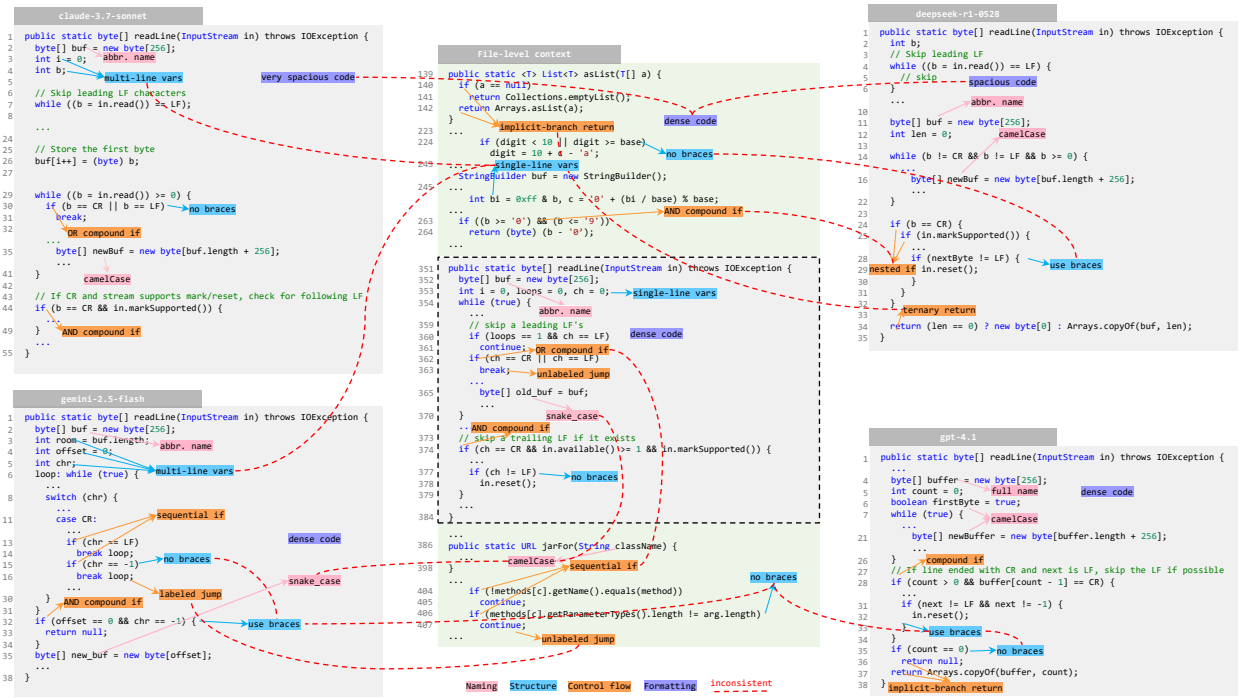


Fig. 1: An illustrative example for code style gaps of context-aware code completions

2.1 Motivating Example

Modern software projects often integrate code from multiple developers and LLM-based assistants. While these contributions are functionally correct and meet basic formatting standards, they may still introduce

stylistic inconsistencies with the surrounding code in many other aspects. To illustrate this challenge, we focus on the scenario where an LLM is tasked with completing a method from an open-source Java project while following the style of the surrounding code. Fig. 1 shows these completions, provided context, and the style annotations of all displayed code.

However, we observe style gaps between the context and generated code across multiple aspects, including naming, structure, control flow, and format. In terms of naming, the context favors shorter, abbreviated identifier names in `camelCase`, while generated code sometimes deviates by adopting longer, fully spelled-out names in `snake_case`. Structurally, the context declares multiple variables within a single statement and consistently omits optional braces. In contrast, generated code often deviates by declaring one variable per line and retaining optional braces. Regarding control flow, the context realizes check-then-return logic in an implicit-branch style, whereas generated code may use ternary return expressions. Moreover, the context realizes conditional control logic with equivalent conjunctive semantics as a single `if` statement, whereas generated code sometimes distributes the logic across nested `if` statements. As for format, the context adopts a compact vertical layout, whereas generated code exhibits clear variation in vertical density, ranging from compact to widely spaced.

We also observe style gaps across different models. In the provided context, conditional control logic with equivalent disjunctive semantics is realized both as sequential `if` statements and as a single compact `if` statement. Faced with this ambiguity, Gemini adopts the former while other LLMs adopt the latter, suggesting distinct stylistic tendencies across models.

Moreover, stylistic inconsistencies are also observed within individual sources. In generated code, these inconsistencies typically arise from structural differences, whereas in human-written code, they more commonly occur in naming and control flow.

Integrating new contributions into existing code can introduce style inconsistencies that are not addressed by standard formatters. Although these inconsistencies do not affect functional correctness, they can increase cognitive load by forcing developers to switch between different coding habits and further reduce development and review efficiency.

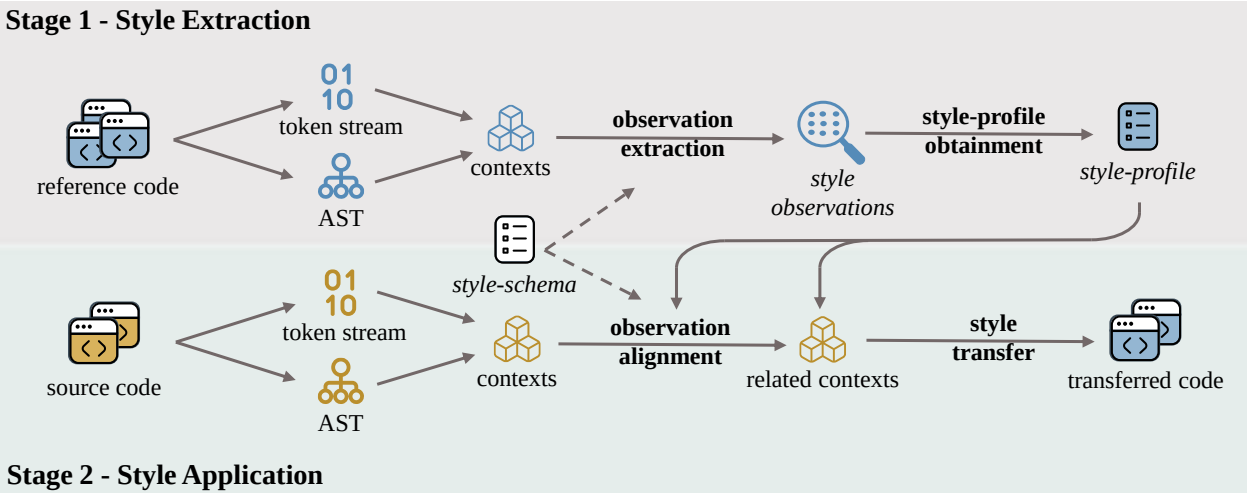


Fig. 2: The workflow of StyleX

2.2 Style Investigation

We followed a three-step pipeline: (i) identifying sources and selection criteria; (ii) extracting candidate style aspects; (iii) applying exclusion criteria and consolidating candidates into the twenty implemented sub-styles.

Sources Selection. We surveyed prior work on code style by searching major academic sources using keyword combinations such as "code style", "stylistic consistency", and related terms. Sources were included if they provided concrete, actionable stylistic rules or transformations, such as identifier naming patterns or behavior-preserving code rewrites. We collected work from the past eight years, spanning empirical studies of style inconsistency, authorship attribution based on coding style, and approaches to convention learning and automated style transfer [11, 12, 20, 21, 23, 28, 34, 36]. Widely used Java style guides were also included [3, 4].

Candidate Extraction. Candidate style aspects were collected from surveyed literature and coding conventions. Each aspect was documented with a name, a brief description, and its source. Redundant or overlapping rules were merged. This process produced a pool of 47 candidate aspects, organized into three categories: formatting, syntactic, and semantic. Formatting aspects concern the visual layout of code, such as indentation style, spacing around lexical tokens, and blank lines. These aspects affect readability without changing syntactic structure. Syntactic aspects involve choices that change the grammatical structure of code within the constraints of the language, thereby altering the AST, but without affecting its semantic behavior. Typical examples include loop forms (`for` vs. `while`), optional braces in control statements, and variable declaration layout. Semantic aspects reflect preferences at the level of design, intent expression, and logical structure beyond syntax, such as the organization of code blocks and identifier naming.

Consolidation. The extracted candidate aspects were reviewed to ensure that they are feasible for automated style transfer. Aspects that require cross-file edits or depend on program analysis were excluded. After this process, we identified five formatting, nine syntactic, and six semantic style aspects, spanning *token*, *expression*, *statement*, and *block* levels, as summarized in Table 4.

3 Methodology

In this section, we elaborate on how we model code style and propose StyleX to maintain code style consistency in an extract-and-apply workflow.

3.1 Overview

We give the StyleX overview in Fig. 2. It consists of two main stages. The first stage extracts and instantiates specific code styles, denoted as a *style-profile*, from reference code. This process is guided by our *style-schema* modeling, which defines the representation of each code style and the mechanism for identifying the underlying contexts that manifest it.

Then, the second stage applies the obtained style-profile to transform a given source code, ensuring its style becomes consistent with the reference. The final output is a stylistically transferred code version.

Informed by the style investigation, we formalize the inherently subjective concept of code style into a unified structure, so as to better extract and represent specific code styles. We here introduce our style modeling first.

3.2 Style Modeling

We define a *style-schema* as a collection of *sub-styles*, where each sub-style is further divided into multiple *sub-style facets* according to its contextual characteristics. These sub-styles are abstracted from the 20 representative style aspects investigated in Section 2.2. Formally, a style-schema is defined as:

$$S = (s_1, \ldots, s_k) \tag{1}$$

Here, s_i represents a family of sub-style facets $s_i^{(l)}$, each defined as:

$$s_i^{(l)} = (C_i^{(l)}, F_i) \tag{2}$$

l denotes a concrete assignment of the attribute set used to fine-grain sub-style s_i. C_i denotes the corresponding contexts from which the sub-style s_i is derived, including token-based contexts (e.g., lexical token

Table 1: Attributes for fine-graining sub-styles

Sub-style	**Attribute description**
Space	Sequence of language unit types in a context.
Newline	(1) Sequence of language unit types in a context; (2) line count of a context.
Line length	Character length of a context.
Body layout	(1) Type of body structure in a context (i.e., statement body, function body, or type-declaration body); (2) Number of statements in the body (i.e., none, one simple, one compound, or multiple); (3) Whether a context has a right sibling node in the AST.
If-statement control style	Semantics of conditional control logic (i.e., conjunction or disjunction).

Table 2: γ predicates

Type	**Description**
$\gamma_{newline}$	Any two adjacent AST nodes.
γ_{indent}	Any token at line-start positions.
γ_{vardec}	An `{ localVariableDeclarationStmt, fieldDeclaration }` AST node.
γ_{loop}	An AST node ∈ `{ForStatement, WhileStatement}`.
γ_{naming}	An `Identifier` AST node declared in `{formalParameter, lambdaParameters, localVariableDeclarationStmt }`.
γ_{check_return}	(1) An `ifElseStmt` AST node, with exactly one `returnStmt` in each branch; or (2) A `returnStmt` containing a ternary expression; or (3) An `ifStmt` AST node immediately followed by a `returnStmt`, where the if branch contains exactly one `returnStmt`.

Note: complete definitions of all γ functions can be found in our repository.

pairs) and AST-based contexts (e.g., expressions and statements). $C_i^{(l)}$, a subset of C_i, is the context set corresponding to the sub-style facet $s_i^{(l)}$. F_i denotes a set of parameterized variables encoding the sub-style s_i.

We observe that, for some sub-styles, contexts associated with the same sub-style often exhibit distinct stylistic patterns. For example, for naming, variable names, function names, and class names may follow different conventions; for spacing, different types of lexical units (e.g., operators, parentheses) may require separate treatment. To capture such nuances, we define a set of attributes for each sub-style. By instantiating these attributes from given contexts, we obtain different attribute assignments for decomposing a sub-style into multiple sub-style facets, while simultaneously partitioning the corresponding contexts into several subsets. In our style modeling, we design attributes for sub-styles spanning diverse context types. These attributes capture factors influencing stylistic decisions and are summarized in Table 1. In particular, all language unit types (i.e., lexical unit types and syntactic unit types) are derived from the ANTLR grammar [5]. Using all ANTLR unit types directly would result in excessive fine-graining and low context coverage, making it difficult to model formatting patterns robustly. To address this, we group unit types with similar syntactic roles or functionalities into higher-level categories, with each category representing the corresponding unit types (e.g., lexical units grouped into categories such as binary operators, unary operators, and keywords; syntactic units grouped into categories such as loop statements and branching statements).

For each specific sub-style s_i, we now present a unified formal definition of its corresponding context set C_i:

$$C_i = \{c \mid c \subseteq U_{\text{tok}} \cup U_{\text{ast}}, \gamma_i(c) = \text{True}\} \tag{3}$$

Here, U_{tok} denotes all lexical units in a program's token stream, and U_{ast} denotes all syntactic units in its abstract syntax tree. Each $c \in C_i$ denotes a relevant context of sub-style s_i, consisting of language units from U_{tok} or U_{ast}, with the context's structural and style-relevant constraints enforced by γ_i. To support the following modeling for diverse sub-styles, we provide a few γ functions listed in Table 2, along with their general descriptions.

Table 3: Definitions of types

Type	Description
`Bool`	True or False.
`Int`	An integer.
`Char`	A character.
`Enum`$\{e_1, ..., e_n\}$	A finite set of certain elements.
`List`$(e_1, ..., e_n)$	An ordered sequence of elements.

While $C_i^{(l)}$ describes where a sub-style is extracted, F_i is defined as a set of parameterized feature variables that encode the sub-style s_i. Each variable is instantiated from StyleX's predefined type system (as detailed in Table 3) and characterizes the variation space of a sub-style along a specific dimension.

Table 4 summarizes all sub-styles and their corresponding γ functions and F variables. To facilitate understanding of our style modeling, Section 3.3 provides concrete examples illustrating the instantiation of F_i for a specific sub-style from individual contexts. We then describe how StyleX performs style transfer from a source code C_S (to be transferred) to a given reference code C_R (the target style).

3.3 Stage 1: Style Extraction

This stage aims to instantiate *style-schema* defined in Section 3.2 and derive a *style-profile* from the given reference code. To do so, the token stream and the abstract syntax tree (AST) of the reference code are first obtained via ANTLR [25]. These ASTs and token streams are then analyzed to identify style-relevant contexts, such as statements, expressions, or token sequences.

Observation Extraction. Given any reference code C_R, all relevant contexts for each sub-style s_i can be extracted from either the token streams or the ASTs based on its C_i defined in the style-schema. Each context occurrence and its corresponding f value (an instantiation of F_i) constitutes a sub-style observation, which reflects an individual stylistic choice. All observations serve as the data foundation for deriving a style-profile that captures the dominant trends of each sub-style.

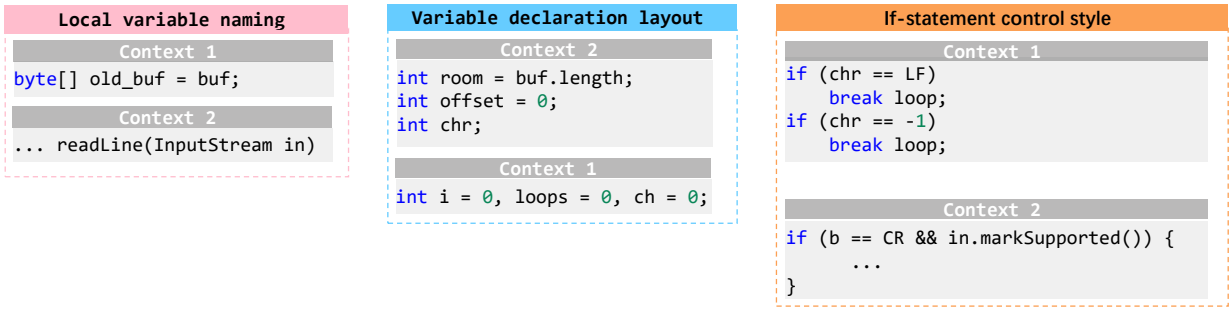


Fig. 3: Example of context instances

To illustrate observation extraction, we select three representative sub-styles: *local variable naming*, *variable declaration layout*, and *if-statement control style* (Fig. 3). For local variable naming, γ_{naming} identifies identifiers introduced by local variable declarations (e.g., *old_ buf*) and formal parameter declarations (e.g., *in*). For each context instance, we extract casing and delimiter patterns, along with the character length, from the identifier text to instantiate F_i. For variable declaration layout, contexts are AST-based structures and satisfy γ_{vardec}. We show split declarations (e.g., *room*, *offset*, *chr*) and merged declarations (e.g., *i*, *loops*, *ch*), where *merge_ var_ declaration* in its F_i is instantiated as *false* and *true*, respectively. For if-statement

Table 4: Style-schema of sub-styles

Category	Sub-style	γ	F	Granularity	Source
Formatting	Indention	γ_{indent}	indention_char: Enum, indention_length: Int	token	[3, 4]
	Space	γ_{space}	has_space: Bool	token	[3, 4]
	Newline	$\gamma_{newline}$	newline_amount: Int	token	[3, 4]
	Max line length	γ_{line_length}	relative_indentation: String, avg_splited_line_len: Int, min_splited_line_len: Int	token	[3, 4]
	Body layout	γ_{body_layout}	before_body: Bool, after_body: Bool	token	[3, 4]
Syntactic	Optional braces	γ_{brace}	use_brace: Bool	token	[3, 4, 28]
	Modifiers order	$\gamma_{modifier}$	order_of_modifier: List	token	[4]
	Boolean expression	γ_{parexp}	pattern_id: Enum	expression	[21]
	Literal position	γ_{parexp}	pattern_id: Enum	expression	[21]
	INC/DEC statement	$\gamma_{expstmt}$	pattern_id: Enum	statement	[20]
	Assignment statement	$\gamma_{expstmt}$	pattern_id: Enum	statement	[20]
	Array declaration	γ_{vardec}	pattern_id: Enum	statement	[4]
	Loop syntax	γ_{loop}	pattern_id: Enum	statement	[20, 21, 28]
	Variable declaration layout	γ_{vardec}	merge_var_declaration: Bool	block	[3, 4, 20, 21]
Semantic	Local variable naming	γ_{naming}	case_style: Enum, max_length: Int	token	[3, 4, 20]
	Logical style of if-else	γ_{ifelse}	short_logic_first: Bool	block	[21]
	If-statement control style	γ_{if}	pattern_id: Enum	block	[20, 21, 28]
	Loop tail branch	γ_{tail_branch}	pattern_id: Enum	block	[11]
	Check then return	γ_{check_return}	pattern_id: Enum	block	[3]
	Check then assign	γ_{check_assign}	pattern_id: Enum	block	[21]

control style, contexts are AST-based structures and satisfy γ_{if}. This sub-style is fine-grained into conjunction and disjunction facets, each illustrated by a representative context instance in Fig 3. Each facet has a predefined template that enumerates all possible equivalent writing patterns. Each context instance is matched to a specific pattern in its template, and its F_i variable *pattern_id* records the identifier of the matched pattern. For all sub-styles whose F_i includes *pattern_id*, observations are extracted following the same template-matching procedure, ensuring consistency of the extraction method.

Style-profile Obtainment. To accurately capture the stylistic preferences of the reference code, we determine the final F_i value for each sub-style facet $s_i^{(l)}$ by selecting the most frequent one from its associated sub-style observations, requiring a minimum occurrence of 70%. In this way, we can eliminate noise and isolate the dominant stylistic features. Then, for all considered sub-styles, we regard C_R's particular *style-profile* as the set of dominant F_i values across all sub-style facets: style-profile$(C_R) = \{f_1^{(l_1)}, \ldots, f_1^{(l_{n1})}, f_m^{(l_1)}, \ldots, f_m^{(l_{nm})}\}$, where each $f_i^{(l)}$ denotes the dominant F_i value extracted from the context set $C_i^{(l)}$, and $m \leq k$ is the number of sub-styles whose dominant F_i values are successfully extracted. Thus, we successfully extract the style-profile from the reference code C_R.

3.4 Stage 2: Style Application

This stage applies the extracted style-profile to transform a given source code so that its style aligns with the reference. The process involves two steps:

Table 5: Real-world projects for task construction

Project	Stars (K)	KLOC*	Contributors	Domain
Jedis	12.0	31.0	231	Middleware
Stirling-PDF	58.4	29.0	242	Office
Arthas	36.5	41.0	206	Diagnostics
NewPipe	34.4	43.8	942	Multimedia
Zookeeper	12.5	51.3	245	Middleware
RxJava	48.2	78.6	299	Reactive Library

* KLOC = thousand lines of Java code

Observation Alignment. Given a source code C_S to be transformed, StyleX identifies style-relevant contexts by using the same context definitions C_i as those used during style extraction. However, only the sub-styles that have been successfully instantiated in the style-profile are considered. This step establishes a one-to-one mapping from each context instance in C_S to its target F_i value, enabling localized and deterministic transformations for subsequent style transfer.

Style Transfer. For each aligned context instance identified in the previous step, StyleX modifies the context to match the target F_i value whenever the current F_i value differs. This process begins by transforming all AST-based contexts, followed by transforming token-based contexts derived from the modified AST. The final output is the transformed code.

The type of transformation operation depends on the sub-style and may involve lexical edits (e.g., inserting or removing whitespace or newlines), structural adjustments (e.g., adding or removing braces, merging or splitting declarations), identifier renaming, reordering of AST nodes, or template-based rewriting using predefined code skeletons. All transformations are applied to identified style-relevant contexts and are guided by the extracted style-profile from the reference code. Our localized, schema-guided approach ensures controlled and predictable style transfer.

4 Experiments

To comprehensively evaluate StyleX, we design the following research questions (RQs):

- **RQ1**: How effective is StyleX in maintaining code style consistency as measured by automated validation tools?
- **RQ2**: How does the overhead of StyleX compare with other state-of-the-art style consistency approaches?
- **RQ3**: How do developers perceive StyleX's ability to maintain code style consistency?

4.1 Experimental Setup

To evaluate the effectiveness of StyleX in real-world scenarios, we aim to construct code style consistency tasks, each containing reference code snippets C_R and a source code snippet C_S. C_R and C_S differ in their code styles. In this way, each task is designed to transfer C_S to C_T, whose code style is consistent with C_R.

Preparation. In order to extract suitable and practical code snippets with different code styles, we selected six open-source Java projects from GitHub, considering their scales (29–79 KLOC), popularity (over 10K stars), maintenance activity (actively as of 2025), contributor diversity (over 200 contributors), and domain difference (five distinct domains), as shown in Table 5.

Based on the natural insight that "*different developers tend to exhibit distinct code styles*" [8], we first partitioned codes across all six projects according to their responsible contributors (using `git blame`), and excluded contributors with fewer than 1,200 LOC to ensure enough code materials behind each contributor.

In this way, we derived 72 authorship pairs (A, B), where A and B represent two distinct developers. To mitigate potential threats arising from two developers sharing overly similar code styles, we employ Forsee [16], a code authorship attribution model that achieves 96.7% accuracy on Java datasets, to identify authorship pairs whose code can be reliably distinguished. We trained a classifier for each authorship pair using their corresponding code materials (with a 4:1 split between training and test sets) and retained 48 authorship pairs for which the corresponding classifier reported classification accuracy exceeding 80%, indicating a marked style difference.

Task Construction. For each authorship pair, we performed code segmentation from all their related code materials at the function level and excluded methods shorter than 20 effective lines of code (when blanks and comments were excluded). To construct a specific code style consistency task (C_R, C_S) for the authorship pair (A, B), we choose C_R from A's code segments in the classifier's training set and C_S from B's code segments in the classifier's test set. This avoids data leakage when evaluating the authorship of C_T, which is derived from C_S by style transfer. We sampled up to ten functions per source style of each authorship pair, resulting in a total of 424 code style consistency tasks. After filtering out tasks whose C_R or C_S had already been incorrectly classified by the corresponding classifiers, we obtained 387 tasks. For each task (C_R, C_S), a style consistency approach is required to transfer C_S's style and output C_T, i.e., (C_R, C_S) $\rightarrow$ C_T.

Oracle. Ideally, C_T should be consistent with C_R in style. However, code style is inherently subjective, making it difficult to verify the success of a transfer with precision [8]. Therefore, we use both automated evaluation (in RQ1 and RQ2) and developer judgment (in RQ3). For automated evaluation, we employ a trained binary authorship classifier for each authorship pair, as described in task construction. For each consistency task (C_R, C_S) and the output C_T, the classifier examines whether C_T is attributed to C_R's corresponding developer A (i.e., ideal style transfer) or C_S's developer B (i.e., unsatisfactory style transfer). For human judgement, participants would be asked to compare and choose the one between C_S and C_T that is closer to C_R in code style. C_T is considered an ideal style transfer if chosen by the majority of participants.

Baselines. We choose CodeBuff [26], a universal learning-based formatter, as the representative learning-based code consistency approach, and select two largely-used large language models, DeepSeek-R1-0528[3] (hereafter referred to as DeepSeek) and GPT-4.1 (hereafter referred to as GPT) as the representative LLM-based approach. The temperature hyperparameter was uniformly set to 0.6 to allow flexible solutions. We exclude widely used rule-based formatters because they rely on predefined style configurations rather than on extracting styles from reference code. Applying them to our setting would require manually deriving configurations from the reference code, introducing human bias into the evaluation. We further exclude DL-based methods in the evaluation, since they typically suffer from poor correctness preservation; for example, the state-of-the-art DUETCS [11] achieves only 55.8% computation accuracy (compile and test pass) during code style transfer.

Experimental Settings. We controlled the following independent variables.

- Code style consistency approach. We study and compare four code style consistency approaches, i.e., CodeBuff, StyleX (ours), DeepSeek, and GPT.
- C_R's size. We varied the size of C_R during task construction to study StyleX's performance on style transfer as the number of lines of code in the reference increased. We controlled 200, 400, 800, and 1000 in our evaluation.

Then, to evaluate the performance of each combination (required by the three research questions), we design three dependent variables.

- Ideal transfer rate (%). It is the proportion of tasks that are ideally transferred, with an ideal transfer defined in Section 4 for both automated and human evaluations.

[3] Accessed via Google Vertex in `https://cloud.google.com/?hl=zh_cn`

- Positive transfer rate (%). It is the proportion of tasks for which the authorship classifier shows a greater tendency to attribute C_T to C_R's developer, compared with attributing C_S to C_R's developer.
- Functional correctness rate (%). We run the covering unit tests for each C_T and report the proportion of C_T that successfully compile and pass all tests, reflecting the preservation of functional correctness.
- Time cost (s). It measures the average time spent on a code style consistency task.

All experiments were conducted on a workstation with an Intel Core i5-13500 CPU (2.5 GHz) and 32 GB of RAM. All Forsee's classifiers were trained on a server with Ubuntu 20.04 with an NVIDIA GPU (CUDA 12.2).

4.2 RQ1: Effectiveness

For research question RQ1, we measure the ideal transfer rate, positive transfer rate, and functional correctness rate for StyleX, in comparison with three existing code style consistency approaches.

Table 6 shows the ideal transfer rates under different reference code sizes. StyleX achieves the highest ideal transfer rate of 14.9% on average, compared with CodeBuff (13.2%), DeepSeek (9.2%), and GPT (12.5%). Although the four approaches exhibit relatively modest ideal transfer rates, StyleX consistently outperforms strong LLMs such as GPT and DeepSeek.

Table 6: Comparison of ideal/positive style transfer rates for the four code style consistency approaches

Method	**Ideal/Positive Transfer Rate**				
	200	**400**	**800**	**1000**	**Avg.**
StyleX	**16.9**/51.2	**17.8**/48.8	**12.1**/**51.4**	12.7/**49.9**	**14.9**/**50.3**
CodeBuff	15.4/38.5	13.2/36.2	11.1/35.4	**13.2**/31.8	13.2/35.5
DeepSeek	9.6/**53.5**	9.0/**51.7**	7.5/44.2	10.6/46.5	9.2/49.0
GPT	13.4/49.9	12.9/45.0	11.6/51.2	11.9/47.3	12.5/48.4

Since the ideal transfer rate metric is strictly consistent with our expectation, it may fail to capture cases where a style transfer meaningfully shifts the code toward the target style, even if it does not achieve perfect alignment. Therefore, we additionally introduce the positive transfer rate to capture such partial improvements. As shown in Table 6, StyleX achieves the highest average positive transfer rate of 50.3%, outperforming CodeBuff (35.5%), DeepSeek (49.0%) and GPT (48.4%). These results indicate that StyleX improves alignment of transferred code with the target style effectively, even when strict consistency is not attainable.

Furthermore, we also compare how different approaches preserve code correctness during their code transfer by running built-in unit tests associated with each project. Table 9 shows our experimental results. Only StyleX achieves 100% functional correctness across experiments, indicating StyleX's strong reliability in preserving code correctness. In contrast, the other three approaches exhibit test failures: 2.5% for CodeBuff, 4.2% for DeepSeek, and 1.9% for GPT, respectively. Among the observed failed tasks, over 90% are due to compiler errors, indicating that formatting transfers like CodeBuff and token-based transfers like DeepSeek and GPT can still non-negligibly result in code that fails to compile. For example, CodeBuff's failures are solely due to formatting issues, such as placing code on the same line as comments. GPT's and DeepSeek's failures primarily result from unsupported language features, the incorrect addition of the `final` modifier to variables, and the use of unsupported logging APIs.

Therefore, we answer RQ1 as follows: *StyleX achieves the highest ideal and positive transfer rates across all code style consistency tasks, with high correctness preservation.*

4.3 RQ2: Overhead

To evaluate the overhead, we measured the time cost of processing all code style consistency tasks and additionally recorded the token consumption for LLM-based approaches. Experiments were conducted with each configuration executed three times.

Table 7: Time cost per consistency task

Method	Time Cost (s)				
	200	**400**	**800**	**1000**	**Avg.**
StyleX	2.0	2.0	2.0	2.1	**2.0**
CodeBuff	2.8	4.2	5.7	6.3	4.8
DeepSeek	28.0	28.3	27.7	30.1	28.5
GPT	5.7	5.9	6.0	5.6	5.8

Table 8: Token consumption across all tasks

Method	Overall Tokens (M)				
	200	**400**	**800**	**1000**	**Total**
DeepSeek	2.7	3.5	4.4	5.3	15.9
GPT	1.2	1.9	2.9	3.6	9.5

We reported the average time to complete a task in Table 7. StyleX consistently required the shortest time among the four approaches, taking about 2 seconds per task, which is 2.4x faster than CodeBuff, 2.9x faster than GPT, and 14.25x faster than DeepSeek. Additionally, considering that LLM-based models also require additional tokens to complete such tasks, we also measured the token consumption for LLM-based models, as shown in Table 8. It reports that in order to complete all 387 tasks, GPT consumed 1.2M–3.6M tokens, while DeepSeek consumed 2.7M–5.3M tokens. These results also exhibit that StyleX is both relatively efficient and cost-effective in code style transfer.

Therefore, we answer RQ2 as follows: *StyleX shows strong cost-effectiveness with the best efficiency and no additional token consumption compared to existing work.*

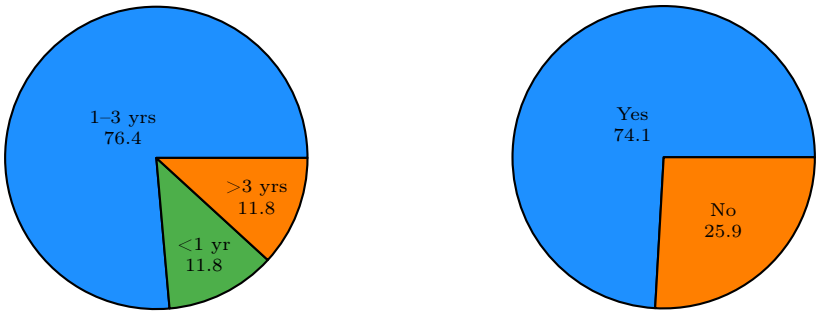
4.4 RQ3: Human Evaluation

Setup. To conduct a human evaluation of our code style consistency tasks, we propose a questionnaire to understand how participants consider each task as ideally transferred or not. To do so, we recruited 88 participants with different programming experiences and Java familiarity, as shown in Fig. 4. To alleviate the pressure on participants to complete the questionnaire, we selected tasks for which at least one studied approach ideally transferred its code style as expected (as studied in RQ1), resulting in 60 tasks in total.

Process. In each questionnaire, participants would be asked to compare, for each code style consistency task, how the source code C_S and its corresponding C_T after code style transfer can be closer to the original reference code C_R in terms of code style. Altogether, we distributed 88 questionnaires and collected 80 valid responses, resulting in 58 code style consistency tasks evaluated; each has been answered by at least two participants. Interested readers can refer to our questionnaire example on the website for details.

Table 9: Comparison of functional correctness rates across reference code sizes

Method	Functional Correctness Rate (%)				
	200	**400**	**800**	**1000**	**Avg.**
StyleX	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**
CodeBuff	98.4	98.9	94.0	98.9	97.5
DeepSeek	97.2	96.7	95.5	93.9	95.8
GPT	99.2	98.4	97.4	97.4	98.1



(a) Programming experience (b) Java familiarity

Fig. 4: Details about participants in human evaluation.

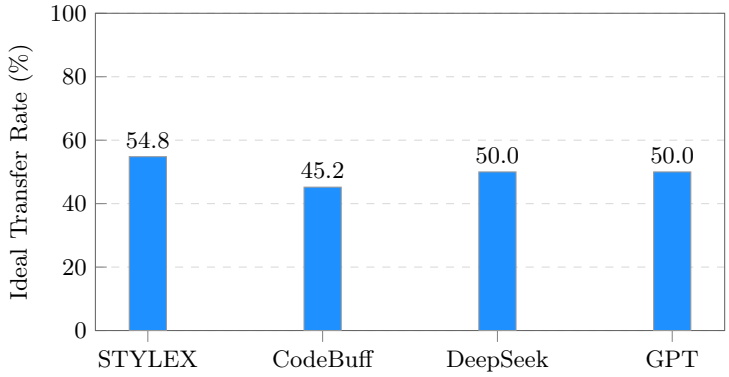


Fig. 5: Comparison of ideal style transfer rates under human evaluation.

Results. Fig. 5 shows the ideal style transfer rate of each approach when evaluated by participants. StyleX achieves the highest ideal transfer rate of 54.8%, exceeding CodeBuff (45.2%) and both LLM-based approaches (both 50%). Note that human evaluation yields slightly different results from those obtained with automatic tools in RQ1. For example, CodeBuff is evaluated as the worst by participants, while it is regarded as the second-best in Table 6. This is mostly because developers tend to focus on deeper and more diverse stylistic expressions rather than mere formatting. Nevertheless, in both automated and human evaluations, StyleX consistently demonstrates the most desirable, comparable to or better than the other approaches.

To assess capability comparisons across different approaches from multiple style perspectives, we also asked participants to rate the stylistic similarity between randomly selected code snippets (source or transferred by any approach) and the reference code across different aspects on a 5-point scale (the higher the score, the more similar). We computed the average similarity score for each approach across all tasks, as shown in Fig. 6. StyleX and GPT improve style similarity in formatting and syntactic aspects relative to Original (no style transfer applied), while maintaining comparable semantic style similarity, indicating multi-aspect style transfer capability. DeepSeek improves syntactic similarity but shows decreased similarity in the other two aspects, suggesting its uneven performance in style transfer. CodeBuff results in lower similarity than Original across all three aspects. Case studies show that CodeBuff occasionally produces poorly formatted code, which reduces readability and may further negatively influence human judgments of syntactic and semantic style similarity.

To investigate whether participant background influences style similarity judgments, we reported the average similarity scores for the three style aspects, grouping participants by programming experience and Java familiarity (Fig. 7). StyleX is the only approach that exhibits stable performance across all programming experience levels, achieving higher scores than Original in formatting and syntactic aspects while maintaining comparable scores in semantic aspects. These results indicate that StyleX provides consistent and

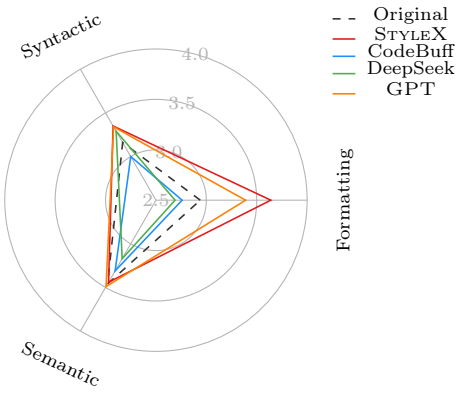


Fig. 6: Average similarity scores for the three aspects

reliable style transfer across participants with varying programming experience, suggesting its robustness and broader applicability relative to other evaluated approaches.

Therefore, we answer RQ3 as follows: *StyleX's superiority has been perceived and confirmed by developers, with its diverse stylistic considerations.*

5 Discussion

Threats to Construct Validity. Our style model is derived from a synthesis of well-studied code style aspects in prior literature, rather than from a comprehensive and systematic survey of all stylistic variations observed in practice. As a result, our style model may not fully capture emerging, project-specific, or less-studied stylistic aspects, potentially limiting the completeness of our style representation. However, our style model is designed to be extensible, allowing additional style aspects to be incorporated by defining new contexts and associated style feature variables. Another threat arises from the inherent subjectivity of code style and the absence of a universally accepted ground truth for style consistency. In our automated experiments, we employ authorship models to assess whether the transferred code is attributed to the target author, using the models' predictions as a proxy measure of style consistency. However, this consistency reflects agreement with the models' learned stylistic patterns rather than human perception of stylistic consistency. To mitigate this limitation, we complement the automated evaluation with a human study, in which developers directly judge the stylistic similarity.

Threats to External Validity. StyleX is currently implemented and evaluated only for Java. However, the underlying methodology is not inherently language-specific and can be adapted to other programming languages with moderate effort by defining language-specific style-schemas and corresponding parsing infrastructure. A second threat arises from the construction of code style consistency tasks, in which the source and reference code are not restricted to the same project. While this strategy ensures sufficient stylistic diversity and enables more reliable evaluation of style transfer effects, it may not fully reflect within-project style transfer scenarios, where code typically follows a uniform convention. As a result, the generalizability of our findings to individual projects may be limited. Another threat concerns our human evaluation. Participants assessed code from unfamiliar projects and may therefore have been less sensitive to human-specific stylistic variations. Future work could investigate StyleX in real-world development settings where developers assess style transfers on their own repositories. Finally, although the rise of LLM coding assistants partially motivates this work, StyleX addresses code style inconsistency regardless of code source. Our evaluation focuses on human contributors because human-authored repositories provide explicit authorship information and relatively stable stylistic preferences, enabling reliable developer-specific style profiles and a well-defined ground truth. Nevertheless, LLM-generated code may exhibit additional stylistic characteristics not fully captured in our experimental dataset. Evaluating StyleX on repositories containing AI-generated contributions is an important direction for future work.

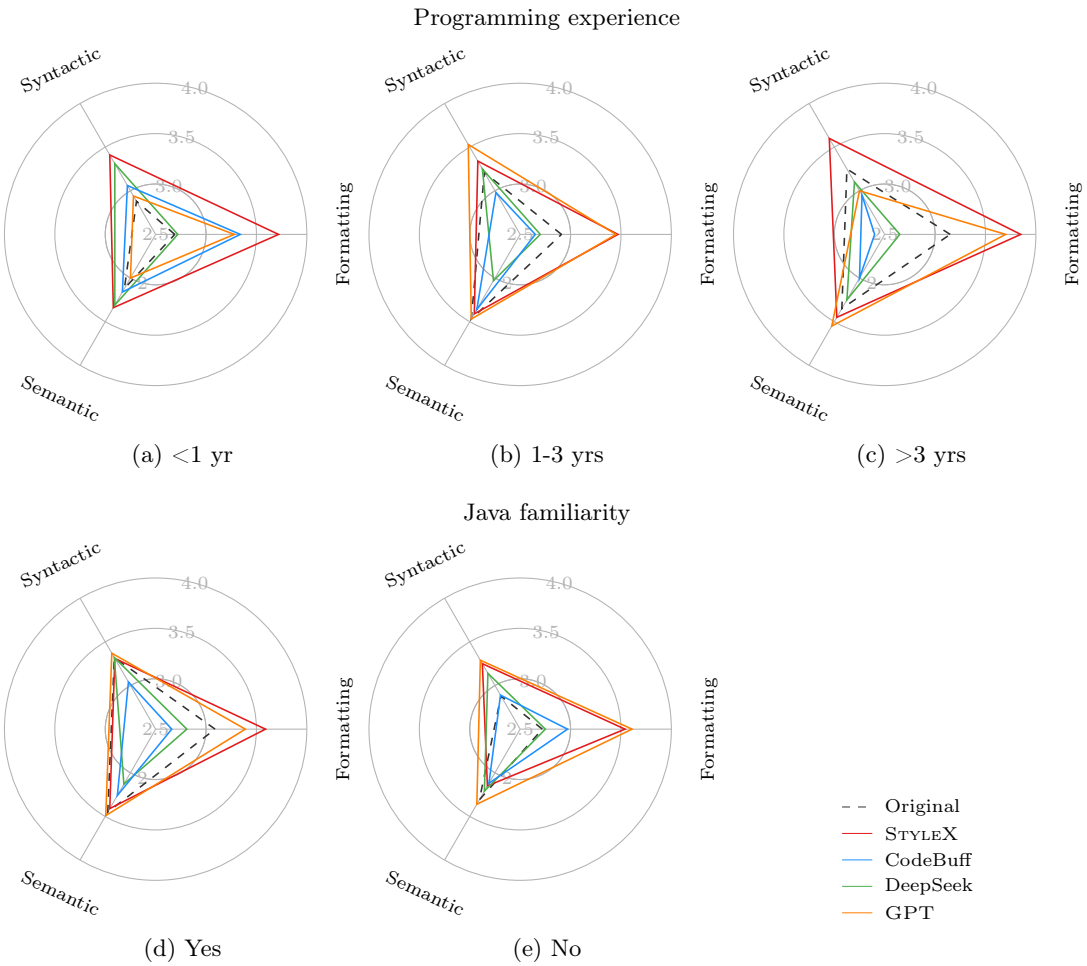


Fig. 7: Average similarity scores across three style aspects under different participant backgrounds

Threats to Conclusion. Our task selection strategy for the human study introduces a potential threat by considering only tasks for which at least one approach is assessed as ideal transfer in automated evaluation. This measure reduces participants' burden and avoids uninformative cases with no discernible stylistic changes. However, this selection may bias the human evaluation toward easier or more transferable cases, potentially inflating the perceived effectiveness of the studied approaches. Another threat arises from the limited number of participants and the inherent subjectivity of human judgments. To mitigate this threat, we recruited participants with diverse programming experience and language backgrounds, and ensured that each task was independently evaluated by at least two participants.

6 Related Work

6.1 Large Language Models

Large language models have demonstrated remarkable capabilities in code generation, promoting their integration into real-world development environments to assist developers and improve development efficiency [9]. In recent years, numerous code assistants powered by LLMs have emerged, such as GitHub Copilot [14], ChatGPT [1], and AlphaCode [19]. Most existing LLMs are trained on large, general-purpose programming corpora [35] with diverse code styles. Although they can produce functionally correct code, they do not account for project- or developer-specific coding styles. As a result, the generated code may exhibit stylistic inconsistencies with the existing codebase.

6.2 Code Style Gaps

In software development, a single program procedure can be implemented in multiple ways, reflecting varied coding styles. In large-scale software projects involving multiple developers, such diversity often leads to inconsistent code styles. Maintaining style consistency is not merely an aesthetic concern but also significantly influences multiple factors that directly affect development efficiency. These include code readability [32], fault injection rate [10], effort in code review [29], as well as pull request merge decisions and resolution time [36]. Therefore, stylistic consistency plays a critical role in ensuring code quality.

Stylistic inconsistencies are prevalent in real-world programming scenarios. For instance, Mi et al. [24] observed that a considerable proportion of code snippets from Stack Overflow violate standard coding style guidelines. With the growing adoption of LLMs in code generation, new dimensions of stylistic inconsistency have emerged. Wang et al. [34] revealed noticeable style gaps between human-written and LLM-generated code, while Siddiq et al. [30] show code quality issues in transformer-based code generation techniques by focusing on code smells. Building on these findings, our work further investigates the problem of stylistic inconsistency and proposes to mitigate the style gaps introduced by modern code generation paradigms.

6.3 Code Style Consistency

To address code style inconsistency problem, one promising solution is to apply code style transfer. Some traditional rule-based approaches include well-known code style linters and formatters such as CheckStyle [2], SonarLint [7], and Pylint [6]. They usually conduct code style transfer based on static transformation rules and mainly focus on fixed formatting and syntactic styles. In contrast, another line of research adopts data-driven approaches like CodeBuff [26], style-analyzer [23] and Styler [22] focus on leveraging formatting styles from codebase and apply them adaptively to source code, while DuetCS [11] and CodeStylist [31] further utilize deep learning techniques to achieve better style leveraging, but show instability to preserve the original program's functional behavior due to model uncertainty [11]. While large language models further broaden the applicability of style transfer, they still suffer from stylistic differences [13] and incur non-negligible overhead. To bridge this gap, we propose a novel adaptive and cost-effective transfer approach, which incorporates diverse stylistic features and demonstrates superiority compared to existing techniques.

7 Conclusion

In this paper, we addressed the problem of aligning code with target styles to mitigate code style gaps by capturing and applying stylistic features derived from reference code. We proposed StyleX, an extract-and-apply framework that models code styles through a structured *style-schema* and performs controlled, localized transformations. Our results show that StyleX achieves performance comparable to prior learning-based approaches and state-of-the-art LLMs, while offering improved efficiency and lower cost. We have also implemented StyleX as an IDE plugin for code style inconsistency analysis and automatic style transfer. In future work, we plan to extend it to more programming languages and incorporate style distribution analysis to help developers make more informed style transfer decisions. Our artifact and supplementary materials can be found at `https://github.com/YingyingJiang1/STYLEX`.

Acknowledgments. This work was supported by the Natural Science Foundation of China (Grant Nos. 92582201 and 62302209), the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118), and the "111 Center" (No. B26023). The authors gratefully acknowledge the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Chatgpt, `https://chatgpt.com/`, last accessed 2025/10/17
2. Checkstyle, `https://checkstyle.sourceforge.io/`, last accessed 2025/10/17

3. Code conventions for the Java programming language, https://www.oracle.com/java/technologies/javase/codeconventions-contents.html, last accessed 2025/09/28
4. Google Java style guide, https://google.github.io/styleguide/javaguide.html, last accessed 2025/09/28
5. Grammars-v4, https://github.com/antlr/grammars-v4, last accessed 2025/10/17
6. Pylint: a python static analysis tool, https://pypi.org/project/pylint/, last accessed 2025/10/17
7. SonarQube for IDE plugin, https://plugins.jetbrains.com/plugin/7973-sonarqube-for-ide, last accessed 2025/10/17
8. Allamanis, M., Barr, E.T., Bird, C., Sutton, C.: Learning natural coding conventions. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. p. 281–293. FSE 2014, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2635868.2635883
9. Barenkamp, M., Rebstadt, J., Thomas, O.: Applications of AI in classical software engineering. AI Perspectives **2**(1), 1 (2020). https://doi.org/10.1186/s42467-020-00005-4
10. Boogerd, C., Moonen, L.: Evaluating the relation between coding standard violations and faults within and across software versions. In: Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories (MSR '09). pp. 41–50. IEEE, New York, NY, USA (2009). https://doi.org/10.1109/MSR.2009.5069479
11. Chen, B., Abedjan, Z.: Duetcs: Code style transfer through generation and retrieval. In: Proceedings of the 45th International Conference on Software Engineering. p. 2362–2373. ICSE '23, IEEE Press (2023). https://doi.org/10.1109/ICSE48619.2023.00198
12. De Ruvo, G., Tempero, E., Luxton-Reilly, A., Rowe, G.B., Giacaman, N.: Understanding semantic style by analysing student code. In: Proceedings of the 20th Australasian Computing Education Conference. p. 73–82. ACE '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3160489.3160500
13. Dipongkor, A.: Can large language models comprehend code stylometry? p. 2429–2431. ASE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3691620.3695370
14. Dohmke, T.: The economic impact of the AI-powered developer lifecycle and lessons from github copilot, https://github.blog/news-insights/research/the-economic-impact-of-the-ai-powered-developer-lifecycle-and-lessons-from-github-copilot/, last accessed 2025/09/28
15. Fakhoury, S., Roy, D., Ma, Y., Arnaoudova, V., Adesope, O.: Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. Empirical Softw. Engg. **25**(3), 2140–2178 (May 2020). https://doi.org/10.1007/s10664-019-09751-4
16. Guo, X., Fu, C., Chen, J., Liu, H., Han, L., Li, W.: Enhancing robustness of code authorship attribution through expert feature knowledge. p. 199–209. Proceedings of the 2024 ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3650212.3652121
17. Harding, W., Kloster, M.: Coding on copilot: 2023 data shows downward pressure on code quality (2024), https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality, last accessed 2025/10/17
18. JetBrains: Configuring code style schemes, https://www.jetbrains.com/help/idea/configuring-code-style.html#configure-code-style-schemes, last accessed 2025/10/26
19. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al.: Competition-level code generation with alphacode. Science **378**(6624), 1092–1097 (2022). https://doi.org/10.1126/science.abq1158
20. Li, Z., Chen, G.Q., Chen, C., Zou, Y., Xu, S.: Ropgen: towards robust code authorship attribution via automatic coding style transformation. In: Proceedings of the 44th International Conference on Software Engineering. p. 1906–1918. ICSE '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3510003.3510181
21. Liu, Q., Ji, S., Liu, C., Wu, C.: A practical black-box attack on source code authorship identification classifiers. IEEE Transactions on Information Forensics and Security **16**, 3620–3633 (2021). https://doi.org/10.1109/TIFS.2021.3080507
22. Loriot, B., Madeiral, F., Monperrus, M.: Styler: learning formatting conventions to repair checkstyle violations. Empirical Software Engineering **27** (08 2022). https://doi.org/10.1007/s10664-021-10107-0
23. Markovtsev, V., Long, W., Mougard, H., Slavnov, K., Bulychev, E.: Style-analyzer: fixing code style inconsistencies with interpretable unsupervised algorithms. In: Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). pp. 468–478. IEEE (2019). https://doi.org/10.1109/MSR.2019.00073
24. Mi, Q., Bai, H., Wang, X., Liu, W., Song, X.: An empirical study of coding style compliance on stack overflow. In: Proceedings of the 2003 International Conference on Software Engineering and Knowledge Engineering. pp. 130–135 (2023). https://doi.org/10.18293/SEKE2023-125
25. Parr, T.J., Quong, R.W.: Antlr: a predicated-ll(k) parser generator. Softw. Pract. Exper. **25**(7), 789–810 (Jul 1995). https://doi.org/10.1002/spe.4380250705

26. Parr, T., Vinju, J.: Towards a universal code formatter through machine learning. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering. p. 137–151. SLE 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2997364.2997383
27. Peitek, N., Bergum, A., Rekrut, M., Mucke, J., Nadig, M., Parnin, C., Siegmund, J., Apel, S.: Correlates of programmer efficacy and their link to experience: a combined eeg and eye-tracking study. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 120–131. ESEC/FSE '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3540250.3549084
28. Quiring, E., Maier, A., Rieck, K.: Misleading authorship attribution of source code using adversarial learning. In: Proceedings of the 28th USENIX Security Symposium (USENIX Security 19). pp. 479–496 (2019)
29. Rigby, P.C., Storey, M.A.: Understanding broadcast based peer review on open source software projects. In: Proceedings of the 33rd International Conference on Software Engineering. p. 541–550. ICSE '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1985793.1985867
30. Siddiq, M.L., Majumder, S.H., Mim, M.R., Jajodia, S., Santos, J.C.S.: An empirical study of code smells in transformer-based code generation techniques. In: Proceedings of the IEEE 22nd International Working Conference on Source Code Analysis and Manipulation. pp. 71–82 (2022). https://doi.org/10.1109/SCAM55253.2022.00014
31. Ting, C.K., Munson, K., Wade, S., Savla, A., Kate, K., Srinivas, K.: Codestylist: a system for performing code style transfer using neural networks. Proceedings of the AAAI Conference on Artificial Intelligence **37**(13), 16485–16487 (Jul 2024). https://doi.org/10.1609/aaai.v37i13.27087
32. Tysell Sundkvist, L., Persson, E.: Code styling and its effects on code readability and interpretation (2017)
33. Tómasdóttir, K.F., Aniche, M., Van Deursen, A.: The adoption of javascript linters in practice: a case study on eslint. IEEE Transactions on Software Engineering **46**(8), 863–891 (2020). https://doi.org/10.1109/TSE.2018.2871058
34. Wang, Y., Jiang, T., Liu, M., Chen, J., Mao, M., Liu, X., Ma, Y., Zheng, Z.: Beyond functional correctness: Investigating coding style inconsistencies in large language models. Proc. ACM Softw. Eng. **2**(FSE) (Jun 2025). https://doi.org/10.1145/3715749
35. Zheng, Q., Xia, X., Zou, X., Dong, Y., Wang, S., Xue, Y., Shen, L., Wang, Z., Wang, A., Li, Y., Su, T., Yang, Z., Tang, J.: Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In: Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. p. 5673–5684. KDD '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3580305.3599790
36. Zou, W., Xuan, J., Xie, X., Chen, Z., Xu, B.: How does code style inconsistency affect pull request integration? an exploratory study on 117 github projects. Empirical Software Engineering **24**(6), 3871–3903 (2019). https://doi.org/10.1007/s10664-019-09720-x