LLM-Based Multi-Artifact Consistency Verification for Programming Exercise Quality Assurance

Felix T.J. Dietrich
Technical University of Munich
Munich, Germany
felixtj.dietrich@tum.de

Yuchen Zhou
Technical University of Munich
Munich, Germany
yuchen.zhou@tum.de

Tobias Wasner
Technical University of Munich
Munich, Germany
tobias.wasner@tum.de

Stephan Krusche
Technical University of Munich
Munich, Germany
krusche@in.tum.de

Maribel Acosta
Technical University of Munich
Munich, Germany
maribel.acosta@tum.de

Abstract

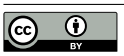
Auto-graded programming exercises involve multiple interconnected artifacts including problem statements, code templates, reference solutions, and test suites. Inconsistencies between these artifacts create extraneous cognitive load, forcing students to reconcile contradictory information rather than focus on learning objectives. Educators must manually detect such inconsistencies, which is time-consuming and error-prone.

We present a theoretically grounded approach to automated consistency verification for multi-artifact programming exercises, building on Mayer's Coherence Principle and Biggs' Constructive Alignment. The methodology combines a novel educational artifact ontology with a specialized Large Language Model (LLM) pipeline detecting Structural and Semantic inconsistencies across heterogeneous exercise components. The ontology defines five consistency categories (Structural, Semantic, Assessment, Temporal, Scope), and this release operationalizes the first two while reserving the remaining three for future instantiation.

We evaluate on 91 perturbed variants from three Java exercises that contain 93 annotated issues across six sub-categories. The reference o4-mini configuration yields 63% precision, 91% recall, F1 0.75, and span F1 0.68, recovering nine in ten inconsistencies with accurate spans; Structural mismatches peak at F1 0.87, whereas Semantic naming remains the main source of noise at F1 0.72. Grok 3 Mini halves latency to 14.3 s and cost to $0.006 per run while retaining F1 0.63. We release the PECV-bench replication package to support reproducible and extensible baselines.

CCS Concepts

• **Applied computing** → **E-learning**; Document management and text processing; • **Computing methodologies** → **Natural language processing**; • **Information systems** → *Retrieval models and ranking*; • **Software and its engineering** → *Software verification and validation.*



Koli Calling '25, Koli, Finland

ACM ISBN 979-8-4007-1599-0/25/11
https://doi.org/10.1145/3769994.3770042

Keywords

Programming Education, Automated Assessment, Large Language Models, Exercise Design, Educational Quality Assurance, Computer Science Education, Consistency Verification, Ontology

ACM Reference Format:
Felix T.J. Dietrich, Yuchen Zhou, Tobias Wasner, Stephan Krusche, and Maribel Acosta. 2025. LLM-Based Multi-Artifact Consistency Verification for Programming Exercise Quality Assurance. In *25th Koli Calling International Conference on Computing Education Research (Koli Calling '25), November 11–16, 2025, Koli, Finland*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3769994.3770042

1 Introduction

Programming exercises are central to computer science education and play a critical role in fostering practical problem-solving skills. They typically consist of multiple interconnected artifacts, such as problem statements, code templates, example solution, test suites, and occasionally supporting materials like Unified Modeling Language (UML) diagrams.

Modern learning management systems such as Artemis [9], Moodle, and Canvas increasingly support such multi-artifact programming exercises, enabling more interactive and automated learning experiences. Each artifact serves a specific pedagogical function: the problem statement defines the task in natural language, the *Template Repository* provides scaffolded starter code to guide student implementation, and the test suite repository evaluates correctness based on predefined criteria.

When multiple instructors or teaching assistants collaborate in designing or updating these artifacts, inconsistencies can arise. For example, the problem statement may instruct students to implement a function named `calculateAverage`, but the provided template already includes an empty implementation of a function named `computeMean`, leading to confusion about whether students should modify the existing code or start from scratch.

These unintentional discrepancies can confuse students, disrupt learning flow, and compromise assessment fairness and instructional alignment. They can also trigger misinterpretations of the task, inflate or deflate perceived difficulty, and even produce incorrect grading outcomes when test suites do not conform to specifications. Informal surveys of educational practitioners indicate that instructors spend significant time on quality assurance (QA) tasks, with consistency checking being a recurring challenge in

collaborative exercise development environments. Despite their impact, inconsistency detection and resolution remain largely manual, time-consuming, and error-prone — especially at scale.

While prior work has investigated consistency checking in both natural language requirements [4, 21] and structured software artifacts such as source code [8, 22] and UML models [11], these efforts typically focus on single-artifact analysis. Educational platforms, however, lack mechanisms for verifying consistency across multiple interconnected artifacts, even though modern learning environments present these artifacts together.

Existing static analysis tools and formal verification approaches are not well suited to this task, as they often rely on complete formal specifications, which are rarely available in an educational context. Translating descriptions in natural language into formal constraints would require a considerable amount of manual effort [4]. Recent advances in LLMs have demonstrated strong capabilities in natural language understanding [5], logical reasoning [25], and code generation [2, 26], motivating their use for consistency checks in educational settings.

Educational research has produced rich error taxonomies rooted in learning theory [1, 12, 18, 19], yet these rarely align with inconsistency types observed in software-based educational artifacts. This absence of alignment leaves no systematic categorization of cross-artifact inconsistencies that integrates both pedagogical and technical perspectives. Currently, there is a lack of automated tools that can identify such inconsistencies across heterogeneous artifacts. The absence of benchmark datasets and evaluation frameworks makes it difficult to compare methods or establish baselines for consistency verification in educational content.

This paper presents an automated **approach for multi-artifact consistency verification in programming education**, leveraging LLMs to detect pedagogically significant inconsistencies. It promotes automated QA in programming education through three contributions, conceptual and empirical:

(1) An **ontology-based model** captures the structure and interdependencies of programming exercise artifacts. The model defines five theoretically grounded consistency categories rooted in learning theory and software engineering (SE) research. We instantiate and evaluate Structural and Semantic categories and outline Assessment, Temporal, and Scope extensions for future work. The implemented checkers presently cover only the subtypes detailed in Section 3, leaving the broader taxonomy for subsequent releases.

(2) An **LLM-based detection pipeline** performs cross-artifact analysis through specialized checkers, producing localized inconsistency reports with a short pedagogical rationale and a suggested remediation.

(3) A **benchmark for educational consistency verification** uses systematic perturbations in three real-world Java programming exercises as input data, yielding 91 variants with 93 annotated issues across six sub-categories. The evaluation spans multiple LLM reasoning models as described in Section 5 and provides reusable assets through the PECV-bench replication package, which bundles code, dataset, prompts, ontology, and evaluation scripts as detailed in Section 6.

The remainder of this paper proceeds as follows. Section 2 reviews related work in consistency checking, LLM-based verification methods, and educational error taxonomies. Section 3 presents the conceptual model, including the ontology-based representation and taxonomy of consistency issues. Section 4 describes the LLM-based detection system architecture and implementation. Section 5 reports empirical evaluation results on the benchmark dataset. Section 6 documents the reproducibility package and released assets. Section 7 discusses implications and limitations, and Section 8 concludes with directions for future work.

2 Related Work

We review related efforts across three intertwined streams to surface the gaps that motivate this work. First, we examine how SE research handles consistency across models, requirements, and source code, highlighting strengths and blind spots. Second, we catalog methodological approaches ranging from formal verification to LLM-enabled pipelines and note the assumptions that limit their reach. Third, we connect these software perspectives to educational error taxonomies grounded in learning theory, then synthesize the insights to position contributions at the intersection of consistency verification and instructional design.

2.1 Consistency Checking in SE

In the SE domain, numerous studies have focused on the consistency checking of software models. Early work by Nuseibeh offers a critical review of approaches that explicitly tolerate and manage inconsistencies, highlighting the various types of inconsistencies that can emerge throughout different stages of the development process [14]. Similarly, Spanoudakis and Zisman provide a survey of inconsistency issues in software models and summarize the techniques proposed to manage them [16]. More recently, Kim and Kim investigate inconsistency problems in source code identifiers within large-scale software systems and propose an automated detection method based on a custom code dictionary [8].

In parallel, the consistency of requirement specifications and their associated use cases has also been extensively studied. Vuotto et al. examine the consistency of functional requirements, demonstrating their approach through a robotic arm case study [21]. Similarly, Chen et al. focus on verifying safety requirements for railway interlocking systems using formal methods grounded in a domain-specific language [4].

Beyond traditional requirements and source code, consistency checking has also been applied to visual and structural design artifacts. One notable line of work focuses on UML, which supports diverse aspects of software modeling but often suffers from internal inconsistencies due to interdependent features [11]. Researchers address this challenge with a unified checking approach that translates UML models into Constraint Logic Programming (CLP) clauses using meta-modeling techniques, enabling the automated detection of inconsistencies via a CLP solver. In parallel, recent work in the context of LLMs explores new consistency challenges: Wang et al. examine discrepancies in coding style between LLM-generated code and human-written code, highlighting the evolving landscape of consistency verification in AI-assisted software development [22].

In summary, existing work on consistency checking primarily focuses either on requirements captured in natural language [4, 21] or on structured artifacts such as source code [8, 22] and UML models [11].

2.2 Consistency Checking Methods

Formal approach is a typical method for consistency checking [4, 21, 27]. A common strategy across recent work involves translating natural language (NL) requirements into a more constrained representation – often termed structured natural language [21], or a domain-specific specification language [4]. This intermediate form reduces ambiguity while preserving human readability. The structured statements are then automatically translated into formal logic expressions (e.g., temporal logic or constraint specification languages [4, 27]) suitable for consistency verification via formal methods such as model checking [4] or realizability synthesis [27].

Nonetheless, while formal approaches can efficiently verify properties over sets of natural language (NL) requirements, they rely on formal specifications that are difficult for non-experts to provide [3].

In recent years, LLMs have emerged as promising tools for natural language understanding [5] and have demonstrated growing capabilities in tasks involving logical reasoning [25] and code generation [26]. Building on these strengths, recent work has explored the use of LLMs to address the challenges of consistency checking in natural language requirements. For instance, [3] proposes a hybrid approach to verifying the satisfiability of string-related NL requirements. The method leverages LLMs in two ways: (1) to derive satisfiability outcomes and propose consistent string instances, and (2) to generate both declarative (Satisfiability Modulo Theories (SMT)) and imperative (Python) checkers to validate these outcomes. Experimental results show that LLM-generated checkers not only achieve high accuracy – reaching perfect test accuracy in some Python-based cases – but also significantly improve the overall success rate and F1-score of satisfiability prediction. This indicates that LLMs can serve as both reasoning agents and generator-assistants in formal verification pipelines, reducing reliance on manual formalization and enabling more scalable verification of NL requirements.

2.3 Error Taxonomies and Educational Assessment

Error classification in educational settings is closely tied to learning theory and instructional design. Biggs' theory of constructive alignment [1] emphasizes aligning learning objectives, teaching activities, and assessments around meaningful performances, offering a foundation for designing error taxonomies that reflect intended cognitive outcomes. Mayer's Cognitive Theory of Multimedia Learning [12] contributes a taxonomy of cognitive overload scenarios, highlighting how instructional design can prevent errors arising from excessive cognitive demand. Complementary insights come from Cognitive Load Theory. Sweller argues that conventional problem solving can hinder schema acquisition due to overlapping cognitive demands, leading to surface-level errors [18]. Further, Sweller introduces element interactivity to explain how intrinsic content complexity influences error patterns [19]. Together, these theories support error taxonomies that distinguish between conceptual misunderstandings and cognitive overload, informing both assessment design and instructional improvement.

Despite the richness of these educational error taxonomies, a gap remains in connecting them with inconsistency classifications from SE. While learning theories provide principled ways to interpret student errors in terms of cognitive processes and instructional alignment, they are rarely integrated with the practical categorization of inconsistencies found in software-based educational artifacts. Addressing this gap is important for building systems that support automated assessment across heterogeneous artifacts by aligning pedagogically meaningful error types with observable inconsistencies in artifact design.

2.4 Research Positioning

The related work reveals several key research gaps that this work addresses. First, existing consistency checking approaches focus either on natural language requirements or structured software artifacts, but not on the unique combination of both found in educational programming exercises. Second, while LLMs have shown promise for consistency verification in SE contexts, their application to cross-artifact educational consistency checking remains unexplored. Third, existing error taxonomies from educational research lack integration with technical inconsistency classifications from SE, limiting their applicability to digital learning environments.

This work bridges these gaps by introducing the first framework for automated consistency verification across heterogeneous educational artifacts, grounded in both learning theory and SE practices. We extend LLM-based consistency checking to the educational domain while developing a novel ontology that integrates pedagogical and technical perspectives on inconsistency classification.

Complementary taxonomies from SE research further motivate the ontology design. Kim and Kim classify identifier inconsistencies into semantic, syntactic, and part-of-speech categories to automate detection in large-scale systems [8]. Wang et al. extend this perspective to LLM-generated code and catalogue 24 inconsistency types across formatting, semantic, expression, control-flow, and fault-tolerance dimensions [22]. These taxonomies reinforce the need to articulate both structural violations and semantic aliasing, yet they rarely connect to pedagogical constructs. Foundational definitions of inconsistency remain heterogeneous: some emphasise conflicting model assertions [16], whereas others describe rule violations more informally [14]. The ontology aligns these perspectives with educational theory, enabling automated analysis that respects both technical precision and instructional coherence.

3 Conceptual Modelling

This section presents the conceptual foundation of this work by first describing the nature of consistency issues in programming exercises and the educational artifacts involved. We then introduce a definition of consistency grounded in learning theory, emphasizing its cognitive and pedagogical implications. Finally, we describe an ontology-based model that captures the structure, dependencies, and inconsistency types across heterogeneous artifacts, forming the basis for automated consistency detection and analysis.

3.1 Problem Description

Modern programming exercise platforms, e.g. AnonLMS, support a variety of heterogeneous artifacts that make up a complete exercise. Some typical artifacts include: *Learning Objectives*, which define the intended competencies or skills students are expected to acquire through the exercise; *Problem Statements*, which describe the task

to be completed, often in natural language, and specify the requirements that guide the student's implementation; *Solution Repository*, which is a code repository containing an example solution that fulfill the task described in the Problem Statement and serve as the basis for correctness and performance evaluation; *Template Repository*, which provides starter code or scaffolding to guide students in their implementation; and *Test-Suite Repository*, which includes test cases used to evaluate the correctness of student submissions.

All of the above artifacts are created and maintained by instructors and teaching staff, such as tutors or course instructors. Among them, only the Problem Statement and the Template Repository are typically visible to students during the exercise phase. When multiple contributors are involved in the development process, *consistency issues* can emerge. These are unintended contradictions or misalignments between artifacts that may confuse students or undermine the learning objectives.

For example, consider a scenario where the problem statement instructs students to implement a method `calculateTotal()` that computes the sum of all items, but the provided template contains a stub for `getPrice()` instead. Students must reconcile this discrepancy by determining whether to modify the existing stub or create a new method, diverting cognitive resources from the intended learning objective of implementing the calculation logic. Such inconsistencies can reduce learning effectiveness and create unfair assessment conditions when students make different assumptions about the intended implementation approach.

We enable automated assessment of consistency issues across these artifacts by first understanding the dependencies among them. It is also essential to model how information flows between artifacts and how discrepancies might affect instructional coherence or evaluation validity. We introduce a high-level definition of consistency issues in the context of programming exercises, grounded in learning theory, to scope the problem. Finally, a clearer understanding of the typical inconsistency patterns is needed. This calls for a principled categorization scheme that can distinguish pedagogically meaningful inconsistencies from superficial or intentional variations.

3.2 Theoretical Foundation and Definition of Consistency

The definition of consistency issues is grounded in established theories from educational psychology and instructional design, which explain how learners process information and how inconsistencies impact cognitive load. Mayer's Coherence Principle [12] and Biggs' Constructive Alignment theory [1] emphasize aligning instructional components to avoid cognitive distractions. Inconsistencies across educational artifacts can therefore disrupt pedagogical design by forcing learners to reconcile conflicting information rather than focus on intended learning objectives. Cognitive Load Theory [18, 19] further demonstrates that learners have limited working memory capacity; contradictory content diverts cognitive resources and increases extraneous load [12]. Building on these insights, we define a **consistency issue** as follows, highlighting the importance of distinguishing pedagogically intentional variations from unintentional inconsistencies that hinder learning:

> **Consistency Issue:** *An unintentional violation of instructional coherence that creates extraneous cognitive load by presenting contradictory information about the same educational element.*

3.3 Ontology-Based Modeling of Educational Artifacts and Consistency Issues

In this paper, we define a domain ontology [17] to formally capture the structure and interdependencies of educational artifacts used in programming exercises. This ontology is then instantiated as a knowledge graph [7], where real-world artifacts and detected consistency issues are represented as nodes and edges. The use of an ontology provides a shared schema to model diverse artifact types (e.g., problem statements, template code, test suites) and their semantic roles in the learning process. Initializing this ontology as a knowledge graph with concrete exercise artifacts enables flexible querying, structured metadata annotation (e.g., severity, source location), and support for tracking artifact evolution over time.

Figure 1 shows that the proposed ontology includes core artifact types such as `Learning Objective`, `Problem Statement`, `Template Repository`, `Solution Repository`, and `Test Suite Repository`, all of which are modeled as subtypes of the generic `Artifact` class. Each artifact is associated with specific relationships that reflect their functional dependencies. For example, the `provides_scaffold` relation connects a template to a problem statement, while the `tests` relation connects test suites to code repositories.

Consistency issues are modeled as typed binary relations between artifacts using the term `inconsistent_with`. The ontology provides a five-category taxonomy of consistency issues: Structural, Semantic, Assessment, Temporal, and Scope. These categories were derived from an analysis of established theories in educational psychology (e.g., Mayer's Coherence Principle [12], Biggs' Constructive Alignment [1], Sweller's Cognitive Load Theory [19]) and SE research on consistency verification [16], which collectively address different dimensions in which inconsistencies can hinder learning (coherence, alignment, sequencing, and coverage). While this taxonomy is not exhaustive and remains under active development, it provides a principled foundation for categorizing pedagogically significant inconsistencies. Each category is characterized below.

Structural Inconsistencies occur when overlapping elements across different artifacts make conflicting assertions that cannot be simultaneously satisfied. These prevent students from implementing solutions that satisfy both the natural language specification and the provided code structure, drawing on SE research on syntactic vs. semantic distinctions [16].

Semantic Inconsistencies arise when the same knowledge element is represented differently across educational artifacts, creating cognitive mapping barriers. Students must reconcile conflicting conceptual representations rather than focus on learning objectives, increasing extraneous cognitive load as explained by Mayer's theory of multimedia learning [12] and semantic consistency distinctions from SE [16].

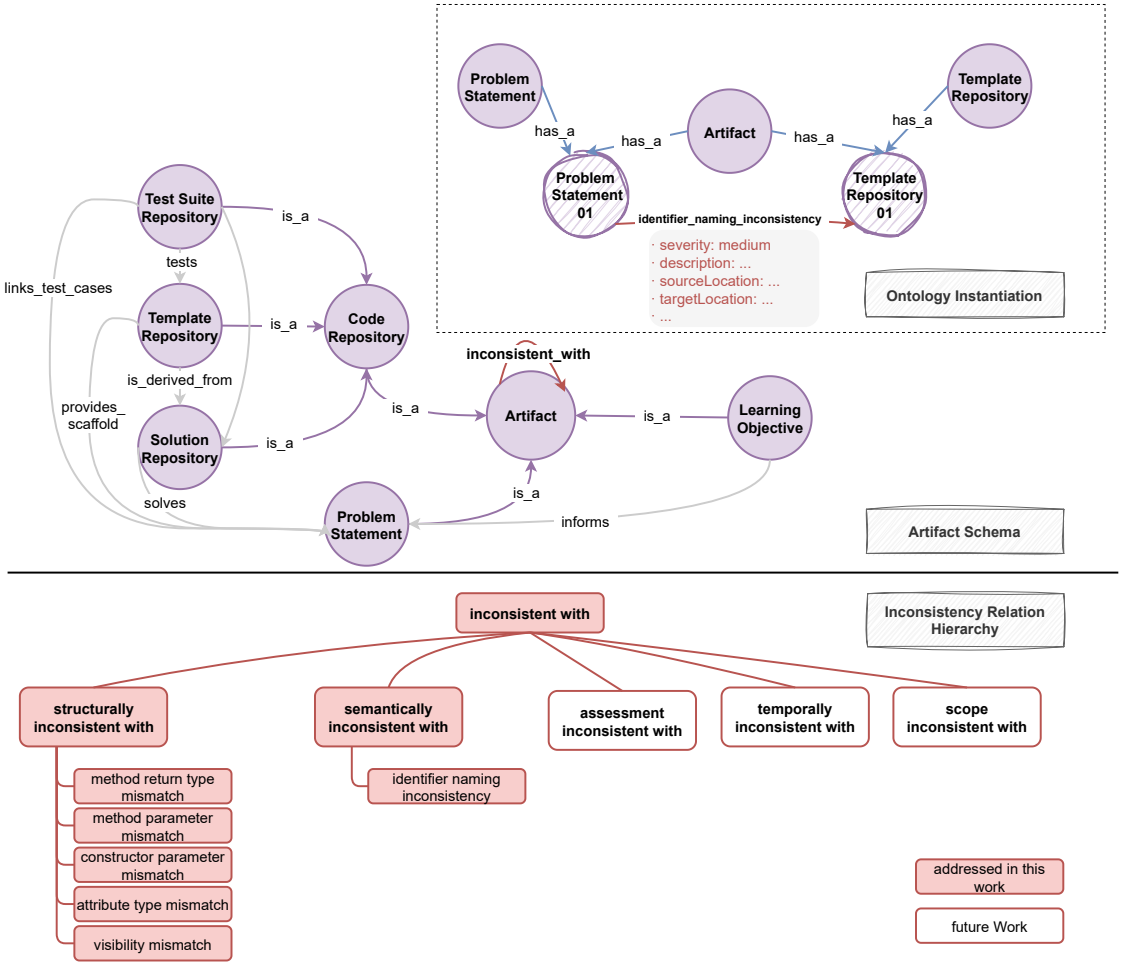


Figure 1: Educational artifact inconsistency ontology modeling five artifact types (Problem Statement, Template Repository, Solution Repository, Test-Suite Repository, Learning Objective) and five theoretically grounded inconsistency categories; Structural and Semantic sub-categories are partially instantiated and evaluated, while Assessment, Temporal, and Scope remain conceptually defined for future work

Assessment Inconsistencies occur when instructional content and evaluation criteria are misaligned. Students cannot demonstrate learning effectively when assessment demands conflict with instructional design, violating Biggs' constructive alignment principles [1] and Webb's alignment of standards and assessments [23, 24].

Temporal Inconsistencies emerge when sequencing, pacing, or prerequisite relationships conflict with pedagogical design principles. These force simultaneous processing of elements that should be learned serially, creating cognitive overload as predicted by Sweller's cognitive load theory [19] and Webb's sequencing criteria for learning progression [24].

Scope Inconsistencies appear when breadth, depth, or coverage mismatches create uncertainty about learning expectations. Students cannot determine appropriate knowledge boundaries or performance levels, leading to incomplete learning in line with Webb's depth-of-knowledge and content coverage frameworks [23, 24].

Figure 1 illustrates the ontology structure and shows detailed subcategories for Structural and Semantic inconsistencies. These two categories were selected for initial instantiation and evaluation because they are the most frequent in programming exercises and most directly confound student implementation. The present release covers a narrow slice of the envisioned taxonomy: Structural inconsistencies currently include method return type mismatches, method parameter mismatches, constructor parameter mismatches, attribute type mismatches, and visibility mismatches, while Semantic inconsistencies are limited to identifier naming inconsistencies with additional sub-categories left for future work. The remaining three categories are conceptually defined in the ontology but reserved for future instantiation and validation, ensuring that the evaluation remains focused while keeping the ontology extensible.

Each consistency issue is linked to its source and target artifacts and enriched with descriptive properties such as severity and

Listing 1: Context rendering with line numbers and file paths for precise issue localization.

```
===== Problem Statement =====
problem_statement.md
 1 | # H01E02 - Lectures
 2 | Implement a class representing a lecture.
 3 | 1. Add attributes: `lectureName`, `numberOfInscribedStudents`
 4 | 2. Create a constructor that initializes every attribute.

===== Template Repository =====
template_repository
`-- src
  `-- com
    `-- example
      `-- exercise
        |-- Lecture.java
        `-- Main.java

template_repository/src/com/example/exercise/Lecture.java
 1 | package com.example.exercise;
 2 | public class Lecture {
 3 |     // TODO 1.1: Add attributes from the UML diagram
 4 |     // TODO 1.2: Add the constructor
 5 | }
```

location. This structure allows us to represent, organize, and analyze pedagogically significant inconsistencies across heterogeneous educational components in a unified and extensible framework.

4 LLM-Based Inconsistency Detection System

We implement an automated system that detects consistency issues across programming exercise artifacts using LLMs. The system processes problem statements, code templates, and test files to identify contradictions that confuse students, reducing manual QA effort for instructors. Building on the ontology-based model described in Section 3, this system integrates the ontology-based data model in two ways. First, the ontology enriches the instruction given to the LLM with context-aware guidance, such as the taxonomy and definitions of consistency issues, enabling the model to interpret various types of consistency issues more effectively. Second, the ontology provides a structured schema that guides the format of the LLM's output. Specifically, the properties defined in the `inconsistent_with` relation are used as keys for the model's response. The resulting outputs are interpretable and machine-actionable for downstream processing and evaluation.

4.1 System Architecture

The system employs two specialized checkers operating in parallel: a Structural checker that detects signature, parameter, and visibility conflicts within the currently implemented subtypes, and a Semantic checker that identifies naming inconsistencies between problem descriptions and code implementations. Each checker executes the same three-step pipeline optimized for reliability and cost efficiency. The modular architecture supports extensibility, letting additional Structural refinements as well as Assessment, Temporal, or Scope modules plug in without altering existing components.

Step 1: Context Rendering. The system transforms heterogeneous exercise artifacts into a unified prompt that retains markdown structure for problem statements, adds line numbers to code files for precise localization, and summarizes repository organization for cross-artifact analysis (Listing 1). Language-specific filters restrict the rendered context to relevant files such as Java sources under `src/`, and optional solution and test repositories are included when present.

Step 2: Reasoning Prompt Construction. Each checker receives prompts that encode educational reasoning principles and include few-shot decisions. The structural prompt enumerates the five Structural sub-categories currently supported and clarifies which pedagogical scaffoldings remain intentional. The semantic prompt establishes the validator role, references cognitive load theory, and guides the model through a four-step analysis that spans entity identification, cognitive mapping, contextual validation, and inconsistency classification. Listing 2 sketches the condensed prompt template and its decision criteria, while a shared context prefix lets LLM-providers reuse cached context across runs to reduce cost.

Step 3: Structured LLM Invocation. The system invokes reasoning models with JSON schemas aligned to the ontology, enabling deterministic parsing and downstream processing. Parallel structural and semantic chains emit typed `ConsistencyIssue` objects, which the pipeline merges before serializing results with timing and cost summaries. Figure 2 lists the fields each checker must fill: severity, the `description` used as the pedagogical rationale, a suggested remediation, and the referenced artifact locations. All prompts, schemas, and checker implementations are released via the PECV-bench replication package (Section 6).

Listing 2: Simplified semantic checker prompt showing four-step reasoning framework and decision criteria.

```
# MISSION
You are a Semantic Consistency Validator for programming exercises.
Detect
UNINTENDED semantic inconsistencies where the same conceptual
entity is
referenced with different names across artifacts.

# ANALYSIS FRAMEWORK
## STEP 1: Conceptual Entity Identification
For each entity in the problem statement:
1. What is the core concept being described?
2. How is this concept represented in template and solution?
3. Is the naming relationship clear for students?

## STEP 2: Cognitive Mapping Assessment
- INTRINSIC LOAD: Names should support core learning objectives
- EXTRANEOUS LOAD: Inconsistent naming creates additional mental
effort
- GERMANE LOAD: Names should help build coherent mental models

## STEP 3: Contextual Validation
Check for educational necessity and technical requirements

## STEP 4: Inconsistency Classification
Raise an issue only if: same conceptual entity + clear mapping
confusion +
unintentional oversight + cognitive burden + no technical necessity

# EXAMPLES
REPORT: Problem "calculateTotal()" -> Template "getPrice()"
DO NOT REPORT: Problem "student name" -> Template
"getName()/setName()"
```

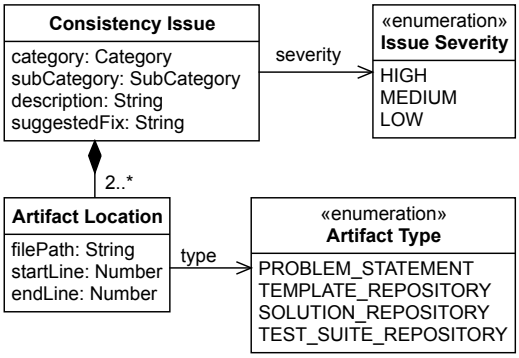


Figure 2: Consistency issue data model defining the structured schema for detected inconsistencies. Each consistency issue contains four textual attributes and severity assessment. The composition relationship (a whole–part association linking each issue to its reported artifact locations) requires at least two artifact locations to establish inter- or cross-artifact relationships, with each location specifying file path, line interval, and artifact type.

4.2 System Characteristics

Computational Performance. The system processes real Java exercises from programming courses, with mean analysis times spanning a few dozen seconds per exercise depending on the reasoning model and per-run costs remaining in the low-cent range. The parallel architecture reduces total processing time by roughly half compared to sequential checking, making the approach viable for routine educational use while maintaining detection quality. Detailed performance benchmarks across different models are presented in Section 5.

Conceptual Strengths and Limitations. The LLM-based approach enables cross-modal reasoning between natural language and code, handling cases where the same concept appears differently across artifacts. Detection accuracy depends on prompt quality and model capability, and the system must distinguish genuine issues from intentional pedagogical variations. The current implementation focuses on Structural and Semantic sub-categories, leaving further refinements and additional categories for future work, while the modular architecture supports independent development of new checkers.

Scope and Scalability Constraints. The system currently handles Java exercises with specific artifact structures (problem statements, template repository, solution repository). Extension to other programming languages requires prompt adaptation and validation. Token limits constrain analysis of large codebases, though educational exercises typically fall within manageable bounds. Context handling capabilities vary significantly across reasoning models: while the reference deployment with o4-mini supports up to 200,000 tokens, other models may have different limits affecting exercise complexity that can be processed. The reasoning model costs and performance characteristics depend heavily on the selected provider, necessitating model-specific optimization for different institutional deployment scenarios.

5 Empirical Evaluation

We benchmark the LLM-based inconsistency detector on synthetically perturbed programming exercises to answer the following research questions (RQ):

RQ1 **Detection Performance:** How accurately does the system detect *and localize* consistency issues across categories?

RQ2 **Deployment Feasibility:** What accuracy-latency-cost trade-offs do different LLMs offer for practical use?

5.1 Dataset, Exercises, and Gold Standard Creation

We evaluate the detector on three Java programming assignments that span the spectrum from simple to advanced. Each assignment provides a problem statement, template repository and solution repository. We create a controlled benchmark by synthetically injecting consistency issues into these artifacts. *H01E01–Lectures* is a basic object-oriented design exercise requiring implementation of a single *Lecture* class with simple attributes and accessor methods. *H02E02–Panic_at_Seal_Saloon* is a medium-complexity scenario with several interacting classes (e.g., "Seal" and "Saloon") plus shared state. *H05E01–Space_Seal_Farm* is an advanced assignment involving inheritance hierarchies and abstract classes to model different seal species on a farm.

For *Lectures*, *Panic_at_Seal_Saloon*, and *Space_Seal_Farm*, we created **30**, **29**, and **32** perturbed variants respectively, totaling **91** variants. Each variant contains one injected consistency issues (two farm variants contain two dependent issues), yielding **93** issues in the gold standard. Issues fall into six sub-categories: *Identifier Naming Inconsistency* (23), *Method Return-Type Mismatch* (19), *Attribute Type Mismatch* (17), *Method Parameter Mismatch* (12), *Visibility Mismatch* (12) and *Constructor Parameter Mismatch* (10). The issues are intentionally distributed across artifacts to require cross-artifact reasoning: 89 issues touch the problem statement, 90 the solution repository, and 40 the template code, and many span multiple artifacts. Analyzing injection patterns, 44% of variants modify only the solution repository, 32% alter problem statements, and 21% change template code, while 3% span multiple repositories. On average, each variant changes 11.3 lines of code. The PECV-bench replication package (Section 6) preserves the variant bundles, prompts, and annotations for direct reuse.

Injections were generated semi-automatically. GitHub Copilot Agent powered by Claude Sonnet 4 proposed diverse patches via unified diffs. The authors reviewed, edited or discarded proposals to ensure exactly one primary issue per variant and to avoid residual problems. We built the gold annotations by first letting the detector propose spans and then manually refining and aggregating them: spans were narrowed, merged or shifted to the most relevant lines so that instructors can locate issues quickly. Consequently the gold spans are "human-readable" rather than exhaustive, introducing fuzziness in span boundaries that motivates the overlap-based evaluation described next.

5.2 Metrics and Matching

We answer RQ1 Detection Performance by inspecting both how often the detector spots an issue and how precisely it locates that issue. We count a detection as a true positive (TP) when it matches

a gold issue, a false positive (FP) when it has no gold counterpart, and a false negative (FN) when a gold issue remains unmatched. Precision $P = \frac{TP}{TP+FP}$ therefore captures how many reported issues are correct, while recall $R = \frac{TP}{TP+FN}$ captures how many genuine issues the detector recovers. We summarize the balance between precision and recall with the harmonic mean $F1 = \frac{2PR}{P+R}$.

Precision measures the share of model findings that truly exist in the gold standard, telling instructors how many suggested issue reports deserve attention. Recall measures the share of gold issues the detector successfully surfaces, indicating how many problematic cases reach the instructor without extra searching.

F1 provides a single summary by combining precision and recall through their harmonic mean. High F1 therefore signals that the detector keeps false alarms in check without missing many genuine issues, which aligns with instructors' need to balance confidence in automated issue reports against the time they spend reviewing them.

Human annotators draw issue spans differently, so we treat localization as a fuzzy overlap problem. We compute span F1 using the Dice coefficient $\frac{2|A \cap B|}{|A|+|B|}$, where A and B are the sets of affected lines marked by the detector and the gold standard. This overlap rewards predictions that capture the relevant lines even when boundaries differ by a few statements.

We also report Intersection-over-Union (IoU) = $\frac{|A \cap B|}{|A \cup B|}$, the Jaccard overlap used in detection research [6, 13]. IoU complements span F1 by penalizing predictions that cover large spans without matching the instructor-authored core lines.

Predicted and gold issues are paired greedily in a one-to-one manner to maximize the overall Dice score, following established matching procedures for partial-span tasks [10, 20]. Any prediction that fails to match becomes a false positive. Any gold issue that remains unmatched turns into a false negative. For RQ2 Deployment Feasibility we additionally log mean runtime, token usage, and dollar cost per detector run. Suggested remediations and severity levels do not influence these metrics because we evaluate detection capability independently from remediation guidance.

5.3 Protocol

We evaluate four reasoning models: OpenAI o4-mini (2025-04-16), xAI Grok 3 Mini (2025-06-10), Google Gemini 2.5 Flash (2025-06-17) and Google Gemini 2.5 Flash Lite (preview 2025-06-17), setting reasoning effort to `medium`.[1] Each model was run three times on every perturbed variant. With 91 variants and four models, this amounts to 1 092 expected runs. In practice, 42 Gemini 2.5 Flash Lite runs and one Gemini 2.5 Flash run failed because the model entered a self-referential reasoning loop when confronted with the structural prompt. We therefore analyze **1 049** runs: 273 for o4-mini, 273 for Grok 3 Mini, 272 for Gemini 2.5 Flash and 231 for Gemini 2.5 Flash Lite.

5.4 Results

Table 1 exposes a detector that favors recall over precision while still completing in seconds rather than the lengthy manual QA passes instructors typically perform. OpenAI o4-mini is the only model with F1 above 0.70. It reaches F1 0.75 and recall 0.91 but still raises 148 false positives, which means roughly one extra issue report for every 1.7 correct detections. Gemini 2.5 Flash and Gemini 2.5 Flash Lite push recall to 0.95 and 0.91 yet generate 623 and 288 false positives, which would swamp instructor queues without triage support. Grok 3 Mini trims the noise to 222 false positives while keeping recall 0.84, although precision 0.51 still leaves noticeable follow-up work. Precision therefore remains the bottleneck for adoption, even though the runs themselves complete in seconds.

Runtime and cost numbers highlight the speed advantage. Grok 3 Mini finishes a sweep in 14.3 s at roughly $0.006 per run, about one quarter the cost of o4-mini and far faster than manual walkthroughs. Gemini 2.5 Flash Lite sits at 16.9 s and $0.0063 but offers little relief from false alarms. No configuration delivers both low latency and a short issue list, so practical deployments will need either additional filtering or prompt pruning to exploit the raw speed safely.

The model also struggles once descriptions become narrative-heavy; o4-mini's F1 drops from 0.79 on *Lectures* to 0.71 on *Space_Seal_Farm*, and Gemini 2.5 Flash Lite follows the same pattern, indicating that long-form requirements and intertwined classes still confuse the prompts. Category analysis (Table 2) explains the gap. Structural mismatches reach F1 above 0.80 and span F1 near 0.70, so signature-level contradictions remain easy to localize. By contrast, Semantic identifier shifts and method-parameter changes fall to F1 0.72 and 0.58 with precision as low as 0.41. These findings point toward either richer ontology cues or human confirmation for ambiguous naming cases.

Failure analysis underscores the need for safeguards. Gemini 2.5 Flash Lite aborted 42 of 273 scheduled runs because the structural prompt triggered a self-referential loop, and the full Flash model failed once under the same conditions. Production deployments require extra consideration with simplified prompts to avoid silent coverage gaps.

5.5 Findings

We summarize the quantitative evidence per research question, highlighting where the detector already adds value for instructors and which safeguards keep that value reliable.

> **RQ1 Detection Performance.** o4-mini delivers F1 0.75 with recall 0.91 and span F1 0.68, producing a concise shortlist that points instructors to the right lines. Precision 0.63 (148 false positives, roughly one extra issue report per 1.7 correct detections) is the main remaining friction, so light prioritization keeps reviews efficient.

Across 273 runs, o4-mini surfaces 254 of 279 annotated issues while missing 25. The high recall and overlap metrics (IoU 0.57) mean reviewers open issue reports that already sit on the problematic code, turning verification into a quick yes-or-no check. Structural categories hold F1 above 0.84 with IoU up to 0.69, so instructors can approve those reports with minimal hesitation, though these metrics apply only to the subset of Structural subtypes we currently implement. Semantic naming and method-parameter mismatches reach F1 0.72 and 0.58 with precision below

[1] o4-mini was accessed via Azure OpenAI. The other models were queried through OpenRouter with the cheapest available provider first. Latency and cost numbers reflect this setup.

Table 1: Overall results across the 1 049 analysed runs. Span F1 and IoU measure average line-overlap quality for matched issues. Time and Cost are mean per run.

Model	TP	FP	FN	Prec.	Rec.	F1	Span F1	IoU	Time (s)	Cost ($)
OpenAI o4-mini	254	148	25	0.63	0.91	**0.75**	0.68	0.57	32.96	0.0338
xAI Grok 3 Mini	233	222	46	0.51	0.84	0.63	0.64	0.53	**14.31**	**0.0061**
Google Gemini 2.5 Flash	263	623	15	0.30	**0.95**	0.45	0.60	0.47	26.38	0.0244
Google Gemini 2.5 Flash Lite	216	288	21	0.43	0.91	0.58	0.59	0.49	16.97	0.0063

Table 2: Per-ontology subcategory metrics for o4-mini. Structural mismatches (return types, constructor parameters, visibility and attribute types) show higher F1 and localization accuracy than Semantic naming and parameter mismatches.

Category	Prec.	Rec.	F1	Span F1	IoU
Method Return-Type Mismatch	0.80	0.96	**0.87**	0.72	0.62
Constructor Parameter Mismatch	0.74	0.97	0.84	0.61	0.50
Visibility Mismatch	**0.81**	0.81	0.81	**0.80**	**0.69**
Attribute Type Mismatch	0.60	0.94	0.73	0.63	0.51
Method Parameter Mismatch	0.41	1.00	0.58	0.68	0.55
Identifier Naming Inconsistency	0.63	0.83	0.72	0.64	0.53

0.65, yet their localized spans and short rationales still provide helpful guidance when paired with confidence, severity, or batching cues; in this release semantic coverage is limited to identifier naming discrepancies, leaving additional semantic variants for later work. Performance stays strong on compact exercises (F1 0.79 on *Lectures*) and tapers to 0.71 on narrative-heavy *Space_Seal_Farm*, signalling where richer prompts can raise precision further.

> **RQ2 Deployment Feasibility.** Grok 3 Mini answers the feasibility question with speed and cost (14.3 s per run, $0.006, recall 0.84), while o4-mini trades time and budget (32.9 s, $0.0338) for higher precision 0.63 and span F1 0.68. Instructors can therefore pick between cheaper exploratory passes and more trustworthy confirmations.

Measured against o4-mini, Grok 3 Mini cuts latency by 56% and cost by 82% yet keeps most recall (0.84 versus 0.91). The trade-off shows up in precision and false positives: Grok 3 Mini raises 222 spurious issue reports (precision 0.51), whereas o4-mini holds that number to 148 at precision 0.63. Both models deliver similar localization quality (span F1 0.64 vs. 0.68). These results indicate that resource-constrained settings gain from Grok 3 Mini's throughput, while high-stakes reviews still benefit from o4-mini's cleaner issue list. The detector is deployable today so long as instructors align the chosen model with their tolerance for triage effort versus runtime and cost.

5.6 Threats to Validity

We discuss the study limitations following Runeson and Höst's validity framework [15].

Internal Validity: The benchmark relies on synthetic perturbations proposed by Claude Sonnet 4 through GitHub Copilot Agent and edited by the authors. These diffs can leave lexical fingerprints that differ from organic instructor edits, potentially biasing the detector toward easier cases. We filtered prompts that surfaced trivial cues and enforced single-issue variants, yet we lack inter-rater agreement statistics because the same authors injected and reviewed issues. Gold spans originated from model suggestions, so anchoring bias may persist despite iterative shrink-to-fit passes.

Construct Validity. The evaluation reports precision, recall, F1, span F1 (Dice), and IoU on line spans to approximate instructor review. True positives are predicted issues matched one-to-one with a gold issue, false positives are unmatched predictions, and false negatives are unmatched gold issues, yet severity labels do not influence the metrics. Dice overlap tolerates boundary differences, but it cannot gauge whether suggested remediations meaningfully lower cognitive load. We evaluate only injected Structural and Semantic categories, so the study does not assess pedagogical impact or downstream effects on student submissions.

External Validity. All exercises stem from a single German institution's Java curriculum that follows Artemis repository conventions. The pipeline has not been validated on exercises authored in other languages than Java. Only Structural and Semantic inconsistencies were instantiated, so findings cannot generalize to assessment rubrics, pacing plans, or scope misalignments. Prompts assume English artefacts; multilingual deployments or accessibility-first templates require fresh evaluation.

Technological Limitations. The detector depends on closed-source reasoning models whose providers can revise token limits, safety filters, or pricing without notice. The prompts target medium reasoning effort and large context windows; courses with bigger repositories or providers with tighter limits may experience silent truncation. We did not benchmark open-weight baselines, so portability to sovereign hosting environments remains unverified.

6 Reproducibility Artifacts

Reference implementation: The PECV-bench replication package (v1.0.0) at commit 188a6d6 archives the code, prompts, ontology assets, and evaluation outputs used in this paper. The artefact is published on Zenodo[2] and mirrored on GitHub.[3] Installing the repository as a Python package exposes the `pecv-bench` orchestration CLI and the `pecv-reference` checker described in Section 4, while the bundled `.env.example` documents provider credentials, model identifiers, and tracing flags. The software component is released under the MIT license to enable reuse and extension.

[2]DOI: https://doi.org/10.5281/zenodo.17260262
[3]https://github.com/ls1intum/PECV-bench

Benchmark dataset: Under `data/`, the archive contributes 91 perturbed Java variants across three programming exercises. Each bundle aligns problem statements, templates, solutions, test-suite repositories, and annotations that tag the injected inconsistency sub-category, matching the evaluation reported in Section 5. The dataset is released under CC-BY-4.0 so that instructors can adapt the materials for their own studies while preserving attribution.

Evaluation and reporting: The preset `pecv-reference.yaml` captures the runs reported in Table 1. Invoking the `run-benchmark` subcommand with the `pecv-reference` target and recorded model arguments reproduces the evaluation, and the `report` subcommand regenerates aggregate tables and metrics by aligning predictions with gold annotations. Together these scripts provide a single, versioned entry point that satisfies ACM reproducibility guidance for implementation, data, evaluation harness, and regenerated reports. Figure 3 provides a high-level view of this end-to-end pipeline.

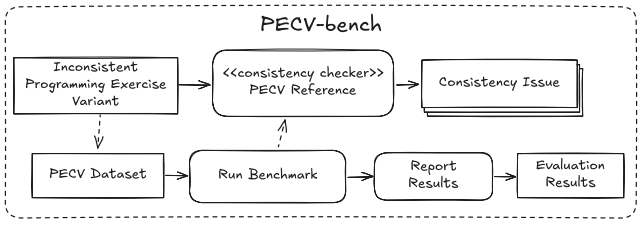


Figure 3: PECV-bench distills the study into a versioned, single-entry pipeline that ties perturbed exercises to consistency checking and reproducible reporting, enabling end-to-end replication of our results.

7 Discussion

The quantitative study confirms that LLM pipelines recover the majority of injected inconsistencies, yet a precision ceiling around 0.63 leaves instructors with a persistent triage burden. Reviewers therefore receive value only when the detector feeds scalable prioritization, such as severity gating or lightweight grouping by artifact scope, rather than a raw issue list; without that support, triage effort quickly erodes much of the promised time savings.

Category-level evidence exposes a maturity gap within the ontology. Structural checkers already capture return-type, visibility, and constructor mismatches with stable localization, but Semantic reasoning degrades once narratives become longer or naming patterns drift across artifacts. Both categories remain underdeveloped, covering only a subset of the theorized subtypes, while the Assessment, Temporal, and Scope classes stay purely conceptual, so the ontology still falls short of comprehensive coverage for multi-artifact pedagogy.

The evaluation design further constrains the conclusions. We rely on synthetic perturbations authored with LLM assistance inside a single Java curriculum, which risks distribution shift when facing organically produced inconsistencies, alternative programming languages, or different institutional conventions, and the manually authored severity and suggested-fix fields remain unvalidated, so end-to-end automation claims stay tentative.

Model-specific failure modes such as Gemini's self-referential loops demonstrate the need for careful considerations such as adaptive prompt selection and downstream filters that learn from past instructor decisions. Collecting evidence with real instructors will clarify acceptable false-positive rates, the usefulness of confidence estimates, and how prompts should evolve alongside reviewer tooling. Section 6 provides the reproducibility scaffolding for that work, but turning the detector into a sustainable QA assistant still demands evaluation of severity and confidence scoring, iterative workflow refinement, and deliberate expansion of both the dataset and the ontology.

8 Conclusion

The presented ontology, detector pipeline, and PECV-bench corpus demonstrate that automated cross-artifact consistency verification for programming exercises is feasible with current reasoning models. Structural checks already deliver actionable precision-recall trade-offs that point instructors to signature-level violations, while Semantic coverage remains fragile and depends on disciplined human review. Together, these contributions establish the first reproducible baseline for multi-artifact educational QA and expose the design space for ontology-guided LLM reasoning.

Nevertheless, the study reflects a Java-only curriculum, relies on synthetic perturbations, and validates only subsets of the Structural and Semantic taxonomies. The companion severity and suggested-fix fields stay human-authored and outside the quantitative evaluation, limiting the ability to claim end-to-end automation. Bridging these gaps requires broader datasets, richer ontology instantiations, and empirical feedback from authentic deployments.

Future work focuses on six directions. Benchmarking additional reasoning models such as GPT-5 mini and open-weight alternatives will map precision, recall, latency, and cost envelopes. Calibrating severity and confidence scoring against instructor judgments comes next before integrating suggested remediations. We will refine prompts, aggregation tooling, and reviewer interfaces so instructors can prioritize the most urgent findings instead of sifting through repetitive issue reports. Extending PECV-bench with Python, C, and multi-institution datasets will capture authentic instructor-authored inconsistencies. Completing the Structural and Semantic subtypes while instantiating Assessment, Temporal, and Scope categories expands coverage. Running classroom pilots will instrument real instructor use and surface integration needs. The replication package (Section 6) scaffolds these studies and invites the community to evolve the ontology, approaches, and datasets toward production-ready, multi-language consistency QA.

Acknowledgments

We acknowledge the teaching staff and students whose experiences with programming exercise inconsistencies motivated this research. This work was supported by the Technical University of Munich and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) within the Collaborative Research Centre (SFB 1608 - Project-ID 501798263).

This paper includes sections revised with the help of GitHub Copilot and ChatGPT to enhance writing clarity. We have carefully reviewed all AI-generated content.

References

[1] John Biggs. 1996. Enhancing Teaching through Constructive Alignment. *Higher Education* 32, 3 (Oct. 1996), 347–364.

[2] Anastasiia Birillo, Elizaveta Artser, Anna Potriasaeva, Ilya Vlasov, Katsiaryna Dzialets, Yaroslav Golubev, Igor Gerasimov, Hieke Keuning, and Timofey Bryksin. 2024. One Step at a Time: Combining LLMs and Static Analysis to Generate Next-Step Hints for Programming Tasks. In *Proceedings of the 24th Koli Calling International Conference on Computing Education Research*. ACM, 1–12.

[3] Boqi Chen, Aren A. Babikian, Shuzhao Feng, Dániel Varró, and Gunter Mussbacher. 2025. LLM-based Satisfiability Checking of String Requirements by Consistent Data and Checker Generation. arXiv:2506.16639

[4] Xiaohong Chen, Zhiwei Zhong, Zhi Jin, Min Zhang, Tong Li, Xiang Chen, and Tingliang Zhou. 2019. Automating Consistency Verification of Safety Requirements for Railway Interlocking Systems. In *27th International Requirements Engineering Conference*. IEEE, 308–318.

[5] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. 2019. Unified Language Model Pre-training for Natural Language Understanding and Generation. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc.

[6] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. 2010. The Pascal Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision* 88, 2 (June 2010), 303–338.

[7] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia D'amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. 2022. Knowledge Graphs. *Computing Surveys* 54, 4 (May 2022), 1–37.

[8] Suntae Kim and Dongsun Kim. 2016. Automatic Identifier Inconsistency Detection Using Code Dictionary. *Empirical Software Engineering* 21, 2 (April 2016), 565–604.

[9] Stephan Krusche and Andreas Seitz. 2018. ArTEMiS: An Automatic Assessment Management System for Interactive Learning *(49th Technical Symposium on Computer Science Education)*. ACM, 284–289. doi:10.1145/3159450.3159602

[10] Zhengzhong Liu, Teruko Mitamura, and Eduard Hovy. 2015. Evaluation Algorithms for Event Nugget Detection : A Pilot Study. In *Proceedings of the 3rd Workshop on EVENTS: Definition, Detection, Coreference, and Representation*, Eduard Hovy, Teruko Mitamura, and Martha Palmer (Eds.). Association for Computational Linguistics, 53–57.

[11] H. Malgouyres and G. Motet. 2006. A UML Model Consistency Verification Approach Based on Meta-Modeling Formalization. In *Proceedings of the Symposium on Applied Computing*. ACM, 1804–1809.

[12] Richard E. Mayer and Roxana Moreno. 2003. Nine Ways to Reduce Cognitive Load in Multimedia Learning. *Educational Psychologist* 38, 1 (Jan. 2003), 43–52.

[13] Annamaria Mesaros, Toni Heittola, and Tuomas Virtanen. 2016. Metrics for Polyphonic Sound Event Detection. *Applied Sciences* 6, 6 (June 2016), 162.

[14] B. Nuseibeh. 1996. To Be and Not to Be: On Managing Inconsistency in Software Development. In *Proceedings of the 8th International Workshop on Software Specification and Design*. 164–169.

[15] Per Runeson and Martin Höst. [n. d.]. Guidelines for conducting and reporting case study research in software engineering. 14, 2 ([n. d.]), 131–164. doi:10.1007/s10664-008-9102-8

[16] George Spanoudakis and Andrea Zisman. 2001. Inconsistency Management in Software Engineering: Survey and Open Research Issues. In *Handbook of Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, 329–380.

[17] Steffen Staab and Rudi Studer (Eds.). 2009. *Handbook on Ontologies*. Springer Berlin Heidelberg.

[18] John Sweller. 1988. Cognitive Load During Problem Solving: Effects on Learning. *Cognitive Science* 12, 2 (1988), 257–285.

[19] John Sweller. 1994. Cognitive Load Theory, Learning Difficulty, and Instructional Design. *Learning and Instruction* 4, 4 (Jan. 1994), 295–312.

[20] Marc Vilain, John Burger, John Aberdeen, Dennis Connolly, and Lynette Hirschman. 1995. A Model-Theoretic Coreference Scoring Scheme. In *Sixth Message Understanding Conference*.

[21] Simone Vuotto, Massimo Narizzano, Luca Pulina, and Armando Tacchella. 2019. Poster: Automatic Consistency Checking of Requirements with ReqV. In *12th Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 363–366.

[22] Yanlin Wang, Tianyue Jiang, Mingwei Liu, Jiachi Chen, Mingzhi Mao, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2025. Beyond Functional Correctness: Investigating Coding Style Inconsistencies in Large Language Models. *Proceedings of the ACM on Software Engineering* 2, FSE (June 2025), 690–712.

[23] Norman L. Webb. 1997. *Criteria for Alignment of Expectations and Assessments in Mathematics and Science Education.* Technical Report. Council of Chief State School Officers, Attn: Publications, One Massachusetts Avenue, NW, Ste.

[24] Norman L. Webb. 1999. *Alignment of Science and Mathematics Standards and Assessments in Four States.* Technical Report. Wisconsin Center for Education Research, 1025 W.

[25] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903

[26] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th SIGPLAN International Symposium on Machine Programming*. ACM, 1–10.

[27] Rongjie Yan, Chih-Hong Cheng, and Yesheng Chai. 2015. Formal Consistency Checking over Specifications in Natural Languages. In *Design, Automation & Test in Europe Conference & Exhibition*. 1677–1682.