Cascading Effect-Aware Reuse of Conformance Test Suites Under Rapidly Evolving Regulations

ANONYMOUS AUTHOR(S)

Test suite reuse is a linchpin for accelerating conformance testing in software under rapidly evolving regulations, yet existing methods fall short: they struggle to accurately capture the semantic nuances of regulatory changes and their ripple effects across interrelated rules. This deficiency gives rise to two key issues: a low reuse rate of existing test suites and the failure to generate necessary new test suites. The root cause lies in the failure of current approaches to effectively and automatically capture the cascading effects triggered by regulatory changes: modifications in rules propagate top-down through requirements, test scenarios, and concrete test cases, altering the system's testable behavior and making it difficult to maintain consistency between test assets and the latest compliance requirements. To bridge this gap, we present a novel cascading effect-aware framework for test suite reuse. We first construct a fine-grained traceability map linking regulatory rules to historical test repositories. Leveraging this map, we trace cascading impacts of rule changes to pinpoint affected scenarios, enabling targeted reuse of valid historical tests, systematic generation of new suites to fill coverage gaps, and consistent preservation of test asset interpretability and maintainability. Evaluations on real-world compliance tasks validate three key advances: an average of over 90% F1 score for test validity, 189% higher reuse rate than general large language models (LLMs), and 76% reduction in testing cycle time. These results position our approach as a reliable, automated solution for continuous compliance testing evolution in regulated domains, resolving the longstanding challenge of maintaining software compliance in dynamic regulatory landscapes.

1 Introduction

In highly regulated industries such as fintech, automotive electronics, and medical software, compliance testing has become a critical link to ensure product compliance and stable enterprise operations [9, 10, 48, 49, 57, 61]. As regulations and business rules are updated with increasing frequency, a single clause change often triggers multilevel, granular cascading effects: from regulation to business requirements, and further to test scenarios and cases. Complex dependencies exist among various rules, where even minor oversights may lead to significant compliance risks. Currently, the industry largely relies on manual analysis of regulatory differences and updates to test cases. This manual approach is not only inefficient and error-prone but also results in a practical dilemma where testing activities struggle to keep pace with regulatory changes, severely constraining the timeliness and reliability of compliance testing.

Despite high expectations for automated test generation, current methods [37, 50, 54, 57] remain reliant on a "one-size-fits-all" regeneration approach, rebuilding all test suites even for isolated rule changes. A test suite, namely a collection of interrelated test cases collectively validating a specific system function or behavior, is often regenerated in its entirety. This practice introduces two critical issues: First, compromised coverage integrity. Blanket regeneration often removes historically valid test cases essential for unchanged rules while failing to precisely target new regulations. This erodes test suite representativeness and introduces compliance gaps. Second, significant resource waste. Redundant test execution consumes substantial computational and temporal resources, reducing testing efficiency and delaying delivery cycles [22, 44]. Enabling precise, incremental test updates to frequent regulatory changes thus remains a pressing challenge in highly regulated industries.

Test case reuse is regarded as a key strategy to overcome the challenges of "full retesting" [38]. Its core idea is to accurately identify the actual impact scope of the changes and perform additions, deletions, or modifications only in affected test cases while leaving unaffected cases completely undisturbed. In this way, it not only avoids unnecessary full regeneration and re-execution but also concentrates valuable computational and human resources on truly changed scenarios, thereby significantly shortening delivery cycles and improving responsiveness to high-frequency regulatory rule iterations. Toward this concept, test case reuse has been extensively studied, with existing research primarily following three technical directions: reuse based on test case characteristics [5, 41, 46, 47], reuse based on code similarity [29, 56], and reuse based on requirements similarity [6, 30].

However, existing approaches fall short in handling regulatory rule-driven testing scenarios, mainly due to "abstraction level mismatch" and "chain change blind spot". Characteristic and code similarity-based methods focus excessively on implementation details and are therefore insensitive to changes in high-level regulatory rules. Approaches based on requirement similarity seek to distill functional commonality, yet they either compare functional points in isolation [6], overlooking dynamic interdependencies, or focus only on static variation points within the product [30], leaving them unable to capture the cascading impacts and trace cascading effects triggered by rule changes.

To address the research gap mentioned above, this paper proposes a novel cascading effect-aware test suite reuse framework aimed at tackling the challenges of test asset reuse in dynamic regulatory environments. The core innovation of this framework lies in constructing a fine-grained traceability map that connects regulatory rules, compliance requirements, test scenarios, and historical test cases. This map characterizes semantic dependencies among rules to construct scenarios, not only captures static mappings between rules and test cases, but also aggregate test cases mapped to the same scenario into test suites. Based on this map, the system can automatically trace the cascading impacts caused by regulation changes and accurately identify the affected requirements, scenarios, and test suites for update and reuse. This framework significantly improves reuse efficiency while guaranteeing compliance testing coverage, providing an automated testing solution for highly regulated industries such as finance and healthcare to cope with high-frequency rule changes.

We have applied this approach in the financial field and evaluated it on six practical cases. Experimental results show that our approach achieves strong performance, with an average F1 above 90% for updated test cases against ground-truth assets. It also attains a reuse rate 189% higher than general LLMs, yielding up to 76% efficiency gains, thus demonstrating both effectiveness and broad applicability. Our code is available at https://github.com/AnonymousAuthorsForFSE2026/ReTool.

In summary, this paper makes three major contributions:

(1) We propose a cascading effect-aware test reuse framework. This work innovatively proposes a reuse framework capable of perceiving and tracing the cascading effects of rule changes, providing a practical pathway for test evolution in highly dynamic regulatory environments.
(2) We establish a fine-grained rule-test traceability and impact analysis mechanism. A multi-dimensional traceability map integrating regulatory rules, test scenarios, and test cases is designed, enabling automated propagation analysis of change impacts and significantly improving the precision of test reuse and coverage completeness.
(3) We evaluate our approach in real-world financial compliance scenarios, empirically demonstrating a 90% for updated test cases, a 189% higher test suite reuse rate compared to general LLMs, which results in a 76% reduction in testing cycle time while maintaining excellent interpretability and test coverage.

2 Related Work

In this section, we briefly discuss the reuse of test cases, the construction of reusable test assets, and the analysis of the dependency and propagation of requirement change.

2.1 Test Case Reuse

Many studies have explored reusing test cases based on either test case characteristics or code. For test case characteristics, some extract reusable templates via structural abstraction [2, 3], while others employ similarity metrics such as test case clustering [46], test strategies and steps [47], or semantic matching of functions [26]. For code-based approaches, reuse is achieved by comparing code paths [56], execution trace coverage [29], or function and variable similarities [31]. Although direct, these methods represent a low-level form of test case reuse, making them insensitive to high-level requirement changes and thus less effective in practice.

The work most closely related to ours is reuse based on requirements, where test cases are identified by similarities in requirement documents, functions, or products. For example, Chen et al. transformed functional descriptions into vectors and used ESIM (Enhanced Sequential Inference Model) for text matching to detect similar requirements and enable reuse [6].However, this approach compares functional points in isolation, ignoring dependencies among requirements. Kang et al. focused on product lines [30], extracting domain test requirements and identifying variation points within them to transform platform cases into product-specific cases. Yet, this method targets static variation points and cannot handle dynamic requirement dependencies or cascading rule changes.

2.2 Reusable Test Asset Construction

To facilitate test case reuse, many studies have integrated product information, requirements, test steps, test cases, and execution results to build test asset repositories. For example, Devos et al. developed QPattern [7], a domain-specific repository for smart card applications that provides reusable templates derived from structured test strategies. Li et al. developed a reuse model that extracts documents, test terms, and test cases from repositories [32], and refines design methods of test cases with similar requirement features. Their model comprises five components: reuse index, basic information, test data, test protocol, and execution body. Patel et al. designed Harmony [43], which organizes test cases by products, modules, and business scenarios to provide both standard and specialized test cases. Nan et al. extracted patterns from historical warship software test cases [39], linked them into a repository, and applied semantic retrieval to locate reusable cases.

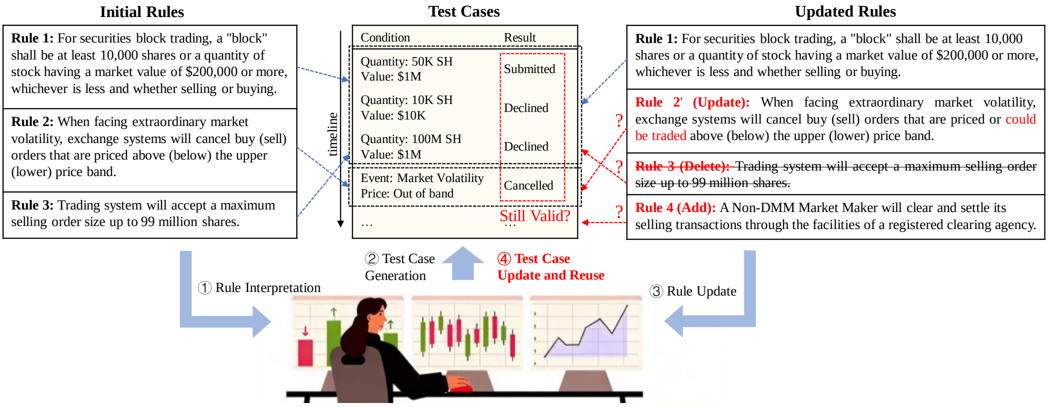
Some scholars have further leveraged ontologies and knowledge graphs to design test asset repositories. Cai et al. defined ontologies based on software functionality to annotate test cases [5], enabling reuse via semantic similarity between queries and cases. Yang et al. built a knowledge graph connecting software components, test products, and defects [59], and used collaborative filtering to recommend relevant test cases.

In summary, test asset repositories provide valuable historical test information and enhance test efficiency and reusability. However, no repository directly establishes traceability from regulatory rules to test cases. Moreover, most reusable assets rely on manual extraction and maintenance by domain experts, which is time-consuming and labor-intensive.

2.3 Change Propagation and Dependency Analysis

Another relevant work is about requirement dependencies and their role in change propagation. Early studies refined dependency models to better capture intrinsic and additional dependencies [60], analyzed large-scale change request data in complex systems to reveal ripple-like propagation patterns [23], or applied fuzzy logic to classify strong and weak dependencies for prioritizing regression test cases [8]. While these methods provide valuable insights into dependency modeling, they primarily operate at the requirement level and rely on manually established requirement-test links, leaving downstream assets such as test cases outside the scope of automated analysis.

Broader research in requirements traceability and dependency extraction has explored information retrieval, machine learning, and dependency parsing to derive requirement-requirement links [27], and surveys highlight their use in impact analysis [52]. Some approaches even link regulatory requirements to system-level requirements for security testing [53]. Nevertheless, automated reuse or update of test cases remains unaddressed.



Rule 1: For securities block trading, a "block" shall be at least 10,000 shares or a quantity of stock having a market value of $200,000 or more, whichever is less and whether selling or buying.

Rule 2: When facing extraordinary market volatility, exchange systems will cancel buy (sell) orders that are priced above (below) the upper (lower) price band.

Rule 3: Trading system will accept a maximum selling order size up to 99 million shares.

Condition	Result
Quantity: 50K SH Value: $1M	Submitted
Quantity: 10K SH Value: $10K	Declined
Quantity: 100M SH Value: $1M	Declined
Event: Market Volatility Price: Out of band	Cancelled
…	…

Rule 1: For securities block trading, a "block" shall be at least 10,000 shares or a quantity of stock having a market value of $200,000 or more, whichever is less and whether selling or buying.

Rule 2' (Update): When facing extraordinary market volatility, exchange systems will cancel buy (sell) orders that are priced or could be traded above (below) the upper (lower) price band.

~~**Rule 3 (Delete):** Trading system will accept a maximum selling order size up to 99 million shares.~~

Rule 4 (Add): A Non-DMM Market Maker will clear and settle its selling transactions through the facilities of a registered clearing agency.

Fig. 1. A testing engineer analyzes what test cases can be reused and what should be composed newly.

3 Cascading Effects in Regulatory Test Case Reuse: A Stock Exchange Example

This paper addresses rule changes with the goal of achieving precise reuse of test cases, which involves accurately identifying reusable cases and effectively updating them in response to modifications. However, changes in rules indirectly affect test cases through a dependency chain: rules influence requirements, requirements shape scenarios, and scenarios determine test cases. This interconnected dependency (also named **cascading dependency**) amplifies the impact of changes, resulting in a "**cascading effect**" that poses challenges to the safe reuse and updating of test cases. Specific issues include:

(1) **Accurately identifying substantive changes rather than just superficial edits.** Regulatory texts often undergo minor edits, such as grammatical adjustments, reordering of clauses, or synonymous substitutions. These subtle modifications usually do not affect test cases. At the same time, changes to core elements, such as constraints, requirements, or business process logic, propagate through dependency chains to the corresponding test cases. Distinguishing between superficial and meaningful changes is therefore challenging but essential. Failing to do so may result in either unnecessary updates or overlooked critical modifications.

(2) **Tracing cascading change impacts rather than just localized effects.** Even when substantive changes are identified, their impact is rarely confined to a single rule or test case. Modifications can propagate through a network of interdependent rules to related test cases that are not directly linked, generating cascading effects that are difficult to predict and control. The extent and intensity of these propagations depend on both the topology of the dependency network and the semantic strength of the connections, making it challenging to determine which test cases are truly affected and which remain consistent.

(3) **Selective test case reuse and targeted updates rather than indiscriminate regeneration.** Given the propagation of impacts, determining which test cases can be safely reused and which require updates is far from trivial. Updating all test cases indiscriminately is costly and time-consuming, while failing to update affected cases risks non-compliance. Achieving a balance between efficient test case reuse and correctness update requires a precise understanding of both the impact scope and the underlying dependency structure.

To illustrate how these challenges manifest in practice, we consider a concrete example in a stock exchange context. A trader's activities, the rules they must follow, and trade operations are tightly interconnected in this example. Changes to the rules can influence many related trades simultaneously through propagation.

> **An Illustrative Example (A Stock Exchange Scenario)** *By the original rules, the trader begins the day by planning several block trades. For example, she wants to sell 50,000 shares worth $1,000,000. She will carefully check whether the order meets both the minimum block size and value requirements and does not exceed the maximum selling limit of 99 million shares. During periods of extraordinary market volatility, she must closely monitor the price bands: any sell orders priced above or below the allowed limits are automatically canceled by the exchange system. Each of these actions, including approved trades, potentially canceled orders, and boundary cases, is captured and validated, representing scenarios where operations are either permitted or blocked.*
>
> *After the rules are updated, the trader's workflow becomes more complex. The modification to the volatility rule has no impact as it is merely an optimization of expression. The deletion of the maximum selling limit opens the possibility of executing larger trades that were previously forbidden, requiring her to reassess order sizes. At the same time, the newly introduced clearing obligation for Non-DMM Market Makers adds a step that must be followed before settlement. From the trader's perspective, these updates require her to adjust not just individual actions but all related trades, each of which is simultaneously constrained by multiple rules. Consequently, her behavior across the entire set of transactions must be coordinated and recalibrated, demanding careful planning and close attention to how each rule collectively shapes her trading decisions.*

The changes in the rules require updating certain test cases to reflect the new constraints. Example 1 partially shows the test cases for original rules in Fig. 1. For instance, Cases 1-3 focus on stock trading quantities and are used to test the quantity constraints specified in Rule 3. Case 4 concerns order cancellations during periods of market volatility and is used to test cancellations following the trades in Case 3. After the deletion of Rule 3, the quantity constraints in Cases 1-3 are no longer valid, so these test cases need to be modified or removed. Case 4, which is unrelated to quantity and thus unaffected by the deletion of Rule 3, does not require modification but should be retested to ensure correct behavior under the updated conditions.

The example illustrates that the impact of a single rule change can propagate through multiple related test cases, demonstrating the importance of analyzing cascading effects to determine which test cases need updates and which can be safely reused.

Example 1: Test Cases for Initial Rules in Fig. 1.

	Market	Variety	Method	Operation	Quantity/Price	Direction	Event	Result	Next
Case 1	NYSE	Security	Block Trading	Trade	10K SH, $200K	Sell	-	Succeed	**Case 5,6**
Case 2	NYSE	Security	Block Trading	Trade	20K SH, $300K	Sell	-	Succeed	**Case 5,6**
Case 3	NYSE	Security	Block Trading	Trade	5K SH, $200K	Sell	-	Fail	-
Case 4	NYSE	Security	Block Trading	Trade	10K SH, $100K	Sell	-	Fail	-
Case 5	NYSE	Security	Block Trading	Cancel	Above Upper Band	Sell	Market Volatility	Succeed	-
Case 6	NYSE	Security	Block Trading	Cancel	Below Upper Band	Sell	Market Volatility	Fail	-
Case 7	NYSE	Security	Block Trading	Trade	10K SH, $300K	Buy	-	Succeed	**Case 10,11**
Case 8	NYSE	Security	Block Trading	Trade	20K SH, $200K	Buy	-	Succeed	**Case 10,11**
Case 9	NYSE	Security	Block Trading	Trade	100M SH, $5K	Buy	-	Fail	-
Case 10	NYSE	Security	Block Trading	Cancel	Below Lower Band	Buy	Market Volatility	Succeed	-
Case 11	NYSE	Security	Block Trading	Cancel	Above Lower Band	Buy	Market Volatility	Fail	-

4 The Cascading Effect-Aware Approach

To address the aforementioned challenges, it is essential to accurately understand the dependency structures and impact scope between underlying rules and test cases. This requires establishing a fine-grained tracing mechanism to model the multi-layered dependencies among test cases, requirements, and scenarios, and to construct a mapping relationship graph among them. Meanwhile, an impact analysis algorithm should be introduced to automatically identify affected requirement clauses and related scenarios when system changes occur, thereby accurately pinpointing the set of test cases that require updates and enabling efficient reuse and precise maintenance of test cases.

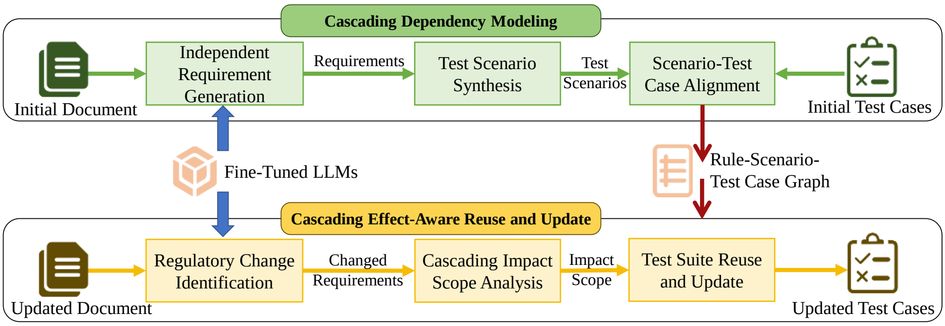

Cascading Dependency Modeling
Initial Document
Independent Requirement Generation
Requirements
Test Scenario Synthesis
Test Scenarios
Scenario-Test Case Alignment
Initial Test Cases
Fine-Tuned LLMs
Rule-Scenario-Test Case Graph
Cascading Effect-Aware Reuse and Update
Updated Document
Regulatory Change Identification
Changed Requirements
Cascading Impact Scope Analysis
Impact Scope
Test Suite Reuse and Update
Updated Test Cases

Fig. 2. The two-phase cascading effect-aware framework for test suite reuse and update.

4.1 Rules and Test Cases Characteristics Complicate Cascading Dependency Modeling

Regulatory rules and test cases possess inherent characteristics that make modeling cascading dependencies particularly challenging. Understanding these characteristics is crucial for constructing accurate mappings between rules and test cases, which in turn enables effective tracking of changes and safe reuse of test assets. We summarize the characteristics as follows:

(1) **Content Coupling Across Rules.** A single regulatory rule often includes multiple requirement aspects that may overlap with other rules. For example, *Rule 1* constrains both quantity and market value for certain trades, while *Rule 3* imposes additional quantity limits on the same trade direction. A test case typically corresponds to only one requirement, meaning that multiple rules can simultaneously constrain a single test case. This many-to-many relationship complicates modeling, as overlapping requirements must be decomposed into independent, unambiguous items to avoid redundancy or conflicts.

(2) **Temporal Dependencies Among Rules.** Rules may impose constraints that take effect at different times. For example, the constraints in *Rule 1* apply immediately upon order declaration, whereas those in *Rule 2* become relevant only afterward. This is aligned with test cases, which specify sequential execution through the keyword "Next" (e.g., *Case 1* precedes *Case 4*). Modeling cascading effects therefore requires capturing both independent requirements and their chronological order to maintain correct verification and execution sequences.

(3) **Semantic Gap Between Rules and Test Cases.** Regulatory rules are typically described in abstract natural language and specify high-level constraints, while test cases contain concrete values. For instance, *Rule 1* may require a quantity of "at least 10,000 shares", whereas *Case 2* instantiates this as "20K SH". Bridging this semantic gap is essential to accurately relate rules to test cases and track how changes propagate.

In summary, the content coupling among multiple rules, the temporal dependencies collectively, and the semantic gap between rules and test cases complicate the construction of cascading effect models. These characteristics highlight why accurately mapping rules to test cases is nontrivial and underscore the need for a systematic approach to model and manage cascading effects effectively.

4.2 Framework Overview

We introduce a two-phase cascading effect-aware framework, illustrated in Fig. 2. It takes as input the initial and updated regulatory documents along with the initial test cases, and outputs updated test cases aligned with the updated document.

Phase I aims to construct the rule-scenario-test case graph from historical repositories, through cascading dependency modeling. It consists of three steps for solving the three concrete issues of

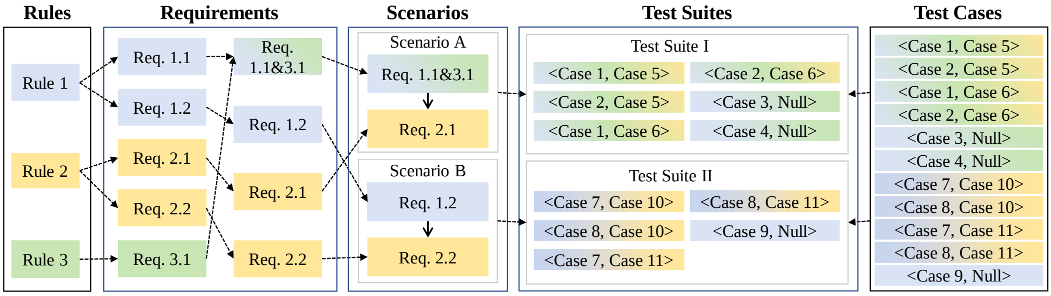


Fig. 3. The cascading dependency modeling result of the running example. The gradient colored boxes represent requirements or test cases obtained through merging multiple rules.

the mapping challenge. First, structured requirements are extracted and merged from regulatory rules to ensure that each requirement is independent and operational, facilitating subsequent automated processing. Second, test scenarios are synthesized based on the relationships among these requirements. Finally, test cases are matched with the constructed scenarios to establish traceable links from rules to test cases, while simultaneously aggregating test cases into test suites.

Phase II focuses on addressing the cascading effect analysis of regulatory changes on test suites and enabling their efficient update and reuse. It also consists of three steps. First, the modified, added, or deleted regulatory rules are identified. Second, the rule-scenario-test case relationship graph from Phase I is leveraged to trace the cascading impact, accurately locating affected scenarios and determining the scope of changes. Finally, stable portions of the existing test suites are identified and reused, while only the minimal set of test cases required for the impacted segments are updated or generated, resulting in an updated, rule-aligned test suite.

In the proposed framework, we establish a reliable mapping from natural-language regulatory rules to structured test cases while explicitly accounting for cascading effects. The framework uses a dual-track “LLM + deterministic algorithms” strategy: the LLM structures unstructured rules into normalized representations, and all subsequent steps, including cascading-aware change impact analysis, test case generation, and updating, are handled by deterministic algorithms. This confines LLM uncertainty to the initial stage, ensures faithful tracing of cascading effects, and enables the automatic generation of a traceable chain from rule changes through scenario-level cascading to test case updates, achieving end-to-end explainability and compliance with stringent regulatory requirements, thus forming a trustworthy cascading-aware testing assurance system.

4.3 Phase I: Cascading Dependency Modeling

We use the rules and test cases in Fig. 1 and Example 1 as examples to illustrate steps in this phase. The schematic diagram of these steps is shown in Fig. 3.

4.3.1 Independent Requirement Generation. To bridge the semantic gap between natural-language regulatory rules and structured test cases, we leverage a fine-tuned LLM to generate *structured requirements* from the rules. Details about the syntax definition of structured requirements and the process of model selection and fine-tuning will be detailed in Section 4.5. The model transforms unstructured rule texts into normalized representations that explicitly capture attributes, conditions, and consequences relevant to testing. For example, *Rule 1* is processed to yield *Req. 1.1* and *Req. 1.2*, *Rule 2* yields *Req. 2.1* and *Req. 2.2* (*Req. 2.2* for selling is omitted here), and *Rule 3* yields *Req. 3.1*, as illustrated in Example 2.

Example 2: Part of Independent Requirements for Rules 1-3.

```
Req. 1.1
if Operation is "Trade" and Trading Method is "Block␣Trading" and Trading Variety is
   "Security" and Direction is "Sell"
then Quantity is "10,000␣shares␣or␣more" and Quantity is "$200,000␣or␣more"
Req. 1.2
if Operation is "Trade" and Trading Method is "Block␣Trading" and Trading Variety is
   "Security" and Direction is "Buy"
then Quantity is "10,000␣shares␣or␣more" and Quantity is "$200,000␣or␣more"
Req. 2.1
if Event is "Market␣Volatility" and Price is "Above␣Upper␣Bound" and Direction is "Sell"
then Operation is "Cancel"
Req. 3.1
if Operation is "Trade" and Direction is "Sell"
then Quantity is "99␣million␣shares␣or␣less"
```

To handle content-couple across multiple rules, structured requirements that constrain the same attribute under identical conditions are merged, producing *independent requirements* such as *Req. 1.1&3.1* in Example 3. This step also integrates contextual information, such as "Trading Market" in *Req. 1.1&3.1*, ensuring that all constraints on the same elements are fully represented.

Example 3: Merged Structured Requirement for Req. 1.1 and Req. 3.1.

```
Req. 1.1&3.1
if Trading Market is "New␣York␣Stock␣Exchange" and Trading Variety is "Security" and
   Trading Method is "Block␣Trading" and Operation is "Trade" and Direction is "Sell"
then Quantity is "10,000␣shares␣or␣more" and Quantity is "99␣million␣shares␣or␣less"
   and Quantity is "$200,000␣or␣more"
```

4.3.2 Test Scenario Synthesis. We construct test scenarios by explicitly uncovering relations among requirements in this step. The relations among requirements here fall into two major categories: *explicit temporal relations*, where semantic cues such as "before", "after", or "until" indicate ordering constraints, and *implicit data dependencies*, where the post-condition of one requirement provides the pre-condition for another (e.g., a "Purchase Declaration" must precede a "Cancellation"). We analyze each structured requirement for its pre- and post-conditions as well as temporal expressions, thereby inferring the system states before and after execution. Requirements sharing consistent states are linked to unified scenarios. For example, as illustrated in Example 4, the purchasing constraint in *Req. 1.1&3.1* transitions the system into a "Pending" state, which then enables *Req. 2.1* on order cancellation under market volatility. Together, these requirements form *Scenario A*, a realistic chain of operations reflecting cascade relations in compliance testing.

Example 4: Test Scenario for Req 1.1&3.1 and Req 2.1.

```
Scenario A
Req. 1.1&3.1
if Trading Market is "New␣York␣Stock␣Exchange" and Trading Variety is "Security" and
   Trading Method is "Block␣Trading" and Operation is "Trade" and Direction is "Sell"
   and Status is "Undeclared"
then Quantity is "10,000␣shares␣or␣more" and Quantity is "99␣million␣shares␣or␣less"
   and Quantity is "$200,000␣or␣more" and Status is "Pending"
Req. 2.1
if Trading Market is "New␣York␣Stock␣Exchange" and Event is "Market␣Volatility" and
   Price is "Above␣Upper␣Bound" and Direction is "Sell" and Status is "Pending"
then Operation is "Cancel" and Status is "Cancelled"

Order Req. 1.1&3.1 -> Req. 2.1
```

4.3.3 Scenario-Test Case Alignment. To address the semantic gap between scenarios and test cases, we adopt a context-aware alignment algorithm. For background constraints, domain-specific terms such as Trading Variety require exact matches, while generic terms allow semantic similarity (e.g.,

Algorithm 1: Scenario-Test Case Alignment within Cascading Dependency Modeling

```
Input  : $T$: set of test case chains; $S$: set of identified test scenarios.
Output : $R$: mapping from each scenario $s \in S$ to its corresponding test suite.
1  $R \leftarrow \emptyset$ ;                                  // Initialize the mapping from scenarios to test suites
2  foreach $s \in S$ do
3      Let $s = \langle r_1, r_2, \ldots, r_n \rangle$ ;                              // Requirements in scenario s
4      foreach $tc \in T$ do
5          Let $tc = \langle t_1, t_2, \ldots, t_m \rangle$ ;                         // Test cases in the chain
6          $C \leftarrow \emptyset$ ;                            // Mapping between requirements and test cases
7          for $a = 1$ to $n$ do
8              $M_a \leftarrow \emptyset$ ;                       // Keys of $t_a$ matched with requirement $r_a$
9              foreach $(key_a^i, value_a^i) \in t_a$ and $(label_a^j, elem_a^j) \in r_a$ do
10                 if $label_a^j$ is a background constraint then
11                     if $key_a^i$ matches $label_a^j$ exactly or semantically similar then
12                         $M_a \leftarrow M_a \cup \{key_a^i\}$ ;            // Mark as matched background key
13                 else if $label_a^j$ is a numerical constraint for the same attribute then
14                     if context of $t_a$ matches scenario context and $value_a^i \in$ requirement interval and expected
                         result consistent then
15                         $M_a \leftarrow M_a \cup \{key_a^i\}$ ;   // Mark as matched numerical key, positive/negative instance
16             if $M_a = \{keys(t_a)\}$ then
17                 $C[r_a] \leftarrow t_a$ ;                                  // All keys in $t_a$ satisfy $r_a$
18         if $tc$ covers all $r_1, \ldots, r_n$ in $C$ (one-to-one) or $tc$ is a prefix of $r_1, \ldots, r_n$ and $t_m.result = Fail$ then
19             $R[s] \leftarrow R[s] \cup \{tc\}$ ;                      // Add $tc$ to the test suite for scenario s
20 return $R$ ;
```

buy vs. purchase). For numerical constraints, we first check contextual relevance (e.g., "quantity" applies to trading but not cancellation), then compare test case values against scenario ranges while considering the expected outcome: values within the range with a successful result are treated as positive instances, values outside the range with a failure result are treated as negative instances, and any contradictions are excluded as out-of-scope. This approach ensures that only semantically and contextually consistent test cases are linked, faithfully capturing cascading-effect dependencies.

Algorithm 1 aligns test cases with scenarios considering cascading effects. It initializes an empty scenario-to-test suite mapping (Line 1) and, for each scenario and test case chain, retrieves the related requirements and test cases (Lines 2-5). A key-value pair matches if: (i) for background constraints, the test key exactly matches the domain-specific term or is semantically similar (Lines 10-12); (ii) for numerical constraints, the key is for the same attribute, the context matches, the value falls within the required interval, and the expected result is consistent (Lines 13-15). Only fully matched test cases are recorded (Lines 16-17). If all test cases correspond to requirements, or form a prefix with the last failing, the chain is assigned to the scenario and added to its test suite (Lines 18-19). The algorithm outputs the scenario-to-test suite mapping (Line 20).

By systematically matching test cases to scenarios, the algorithm aggregates all relevant cases into scenario-specific *test suites*. This mapping not only preserves the logical order and dependencies within test case chains but also establishes a clear link from rules through scenarios to test cases.

4.4 Phase II: Cascading Effect-Aware Reuse and Update

We use the rule changes in Fig. 1 to illustrate the steps in this phase, and the schematic diagram is shown in Fig. 4. This phase focuses on handling the cascading effects of regulatory changes on test suites, enabling precise detection of impacted scenarios and efficient reuse or update of test cases.

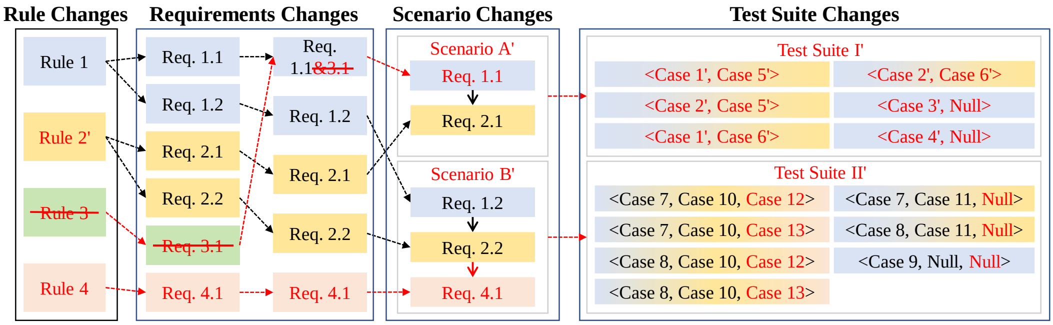


Fig. 4. The change identification and impact scope establishment process for rule changes in Fig. 1. Elements highlighted in red indicate the modified rules, along with the affected requirements and test cases.

4.4.1 Regulatory Change Identification. To accurately detect impactful changes, we first use the structuring model from Phase I to generate independent requirements for the rules before and after updates. Comparing these requirements allows us to distinguish changes that affect test scenarios from those that do not, overcoming the limitations of relying solely on natural language comparison. For example, although the wording of *Rule 2'* differs from *Rule 2*, its structured requirement remains the same, so *Rule 2'* triggers no changes. Conversely, if a structured requirement changes even in a single attribute, we consider it a change. For instance, the deletion of *Rule 3* also removes its structured requirement *Req. 3.1*, causing a change.

4.4.2 Cascading Impact Scope Analysis. This step leverages the *cascading effect* to determine which scenarios and test cases are impacted and to identify the level of test suite reuse. If a rule change does not affect the scenario, the test suite can be **fully reused**. For example, in Fig. 4, updating *Rule 2* to *Rule 2'* does not affect *Scenario A*, so *Test Suite I* can be fully reused. If a rule change affects a scenario, two situations arise:

(i) If the changed requirement appears later in the scenario, the test suite can be **partially reused**, as earlier test cases remain valid. For example, the newly added *Rule 4* is appended after *Req. 2.2* due to a data dependency, updating Scenario B to *Scenario B'*. Test cases for *Req. 1.2* and *Req. 2.2* are reused, and new tests for *Req. 4.1* are added, forming *Test Suite I'*.

(ii) If the changed requirement is at the front, all subsequent requirements may be affected. The test suite falls into the category of **no reuse** and must be fully regenerated. For example, deleting *Rule 3* modifies *Scenario A* by replacing *Req. 1.1&3.1* with *Req. 1.1*. Since *Req. 1.1* is at the head, all related test cases are invalid and must be regenerated.

4.4.3 Test Suite Reuse and Update. Depending on the reuse level, different strategies are applied. For full reuse, existing test suites are retained. For partial reuse, only the new requirements and subsequent ones are processed, with generated test cases integrated with reusable ones. For no reuse, all related requirements are regenerated. The test case repository is updated accordingly.

Similar to [57], new test cases are generated by starting from the scenario and combining its numerical constraints with strategies such as boundary value analysis. For partial reuse, the impact of reused parts on new requirements is considered to control test data generation. By explicitly modeling cascading impacts, this phase ensures that test suites remain accurate, traceable, and aligned with evolving regulations.

To facilitate understanding of the overall process, we present Algorithm 2 as a concrete procedure. It first identifies rule changes through structured requirements, then analyzes their impact scope across related scenarios, and finally updates the corresponding test suites by determining

Algorithm 2: Test Suite Reuse and Update under Cascading Regulatory Changes

```
Input  : $Rule_{old}$: rules before change; $Rule_{new}$: rules after change; $G$: relation graph between rules, scenarios,
         and test cases; $TS$: initial test suites.
Output : $TS'$: updated test suites.
$TS' \leftarrow TS$ ;                                   // Initialize updated test suites as a copy of the original ones
$R_{old} \leftarrow ExtractRequirements(G, Rule_{old})$ ;   // Extract the structured requirements linked to $Rule_{old}$
$R_{new} \leftarrow ExtractRequirements(G, Rule_{new})$ ;   // Extract the structured requirements linked to $Rule_{new}$
$R_{diff} \leftarrow Compare(R_{old}, R_{new})$ ;            // Identify if the requirement changes
$S_{diff} \leftarrow Array()$ ;                         // Initialize a list of scenarios that are impacted by the detected changes
for each pair of changed requirement $r_{old}, r_{new}$ in $R_{diff}$ do
    $s \leftarrow GetScenario(G, r_{old})$ ;             // Retrieve the scenario associated with $r_{old}$
    $chg \leftarrow DetectImpact(s, r_{old}, r_{new})$ ;   // Determine whether the requirement change influences s
    if $chg! = None$ then
        if $s \notin S_{diff}$ then
            $S_{diff} \leftarrow Append(S_{diff}, (s, chg))$ ;   // If S not yet recorded, add it along with chg
        else
            $S_{diff} \leftarrow Update(S_{diff}, (s, chg))$ ;   // If recorded, update chg to be the more forward pos
for each changed scenario s and change impact chg in $S_{diff}$ do
    if $chg.pos == Head$ then
        $sce'_s \leftarrow GenerateScenarios(G, s, r_{new})$ ;   // Regenerate the whole scenario s with $r_{new}$
        $suite_{new} \leftarrow GenerateTests(sce'_s)$ ;         // Generate completely new test suites for $sce'_s$
        $TS' \leftarrow UpdateTS(TS', G, s, suite_{new})$ ;     // Replace the old suite in the repository with $suite_{new}$
    else
        $sce'_s \leftarrow UpdateScenarios(G, s, chg, R_{new})$ ;   // Regenerate only the relevant part of the scenario
        $suite_{reuse}, suite_{new} \leftarrow ParcialGenerateTests(TS, G, s, sce'_s)$ ;   // Return and generate test suites
        $suite_{updated} \leftarrow Merge(suite_{reuse}, suite_{new})$
        $TS' \leftarrow Append(TS', G, s, suite_{updated})$ ;   // Replace the old suite in the repository with $suite_{updated}$
return $TS'$
```

whether they can be fully reused, partially reused, or need complete regeneration, highlighting the importance of cascading effects in updating and reusing test suites under regulatory changes.

4.5 LLM Fine-Tuning

4.5.1 Corpus Construction. The Structuring Model is tasked with extracting key elements from natural-language regulatory rules and generating structured requirements. Since our experiments showed that directly applying in-context learning approaches to general LLMs such as GPT-5 [40] performs poorly in this task, we adopt a fine-tuned LLM instead. During fine-tuning, we employ instruction tuning: each sample consists of a prompt and an answer. The prompt consists of a brief task description, precautions, and the rule to be processed, and the answer is the corresponding structured requirements without any explanation, as illustrated in Example 5. We deliberately keep the prompt concise as our ablation study shows that, under fine-tuning, prompt complexity has minimal impact on generation quality. To build the training corpus, we spent two weeks annotating and proofreading 639 samples drawn from 20 financial regulations. All annotations were cross-checked by co-authors with financial-domain expertise to ensure accuracy and reliability.

Example 5: Annotation for Structured Requirement Generation

{

"prompt": "Given a rule, please extract its requirement-related elements and labels and combine them into a structured format. If the rule specifies multiple distinct conditions or results, divide it into

> multiple independent requirements.
> Rule: Only UTP Securities are eligible to trade in the Early Trading Session. The Early Trading Session will begin at 7:00 a.m. and conclude at 9:30 a.m.",
> "answer": "
>
> ```
> Req. 1
> if Trading Variety is "UTP␣Securities" and Trading Session is "Early␣Trading␣Session"
> then Time is "7:00␣a.m.␣-␣9:30␣a.m."
> ```
>
> "}

It is worth noting that the structured requirements adopt the standardized "if-then" representation proposed in [57]. In this format, the "if" clause specifies the preconditions under which the rule applies, while the "then" clause describes the expected outcome or constraint. Both clauses may include multiple constraint items connected with logical operators such as "and" or "or", allowing the expression of composite conditions. This structure provides a clear, machine-readable form that facilitates subsequent automated analysis and test generation, while remaining sufficiently expressive to capture the complexity of financial regulatory rules. In addition, the key attribute labels such as *Trading Variety*, *Quantity*, and *Time*, are also consistent with the definitions in [57], ensuring its applicability across different rules and scenarios.

4.5.2 Structuring Model Training and Selection. We investigated and tried seven advanced generative LLMs of different architectures, as shown in Table 1, to serve as the base model for training. By assessing their efficacy in this task, we aimed to determine the LLM with the best performance to conduct the structured requirement generation task.

The constructed corpus was initially divided 9:1 into training (575 samples) and validation sets (64 samples). Due to the small initial training set size that potentially harms generalization, we augmented the training set using the augmentation tool from [1], employing techniques such as random insertion, replacement, deletion, and swapping of elements and labels. This resulted in 6,673 training samples. We considered using the Low-Rank Adaptation (LoRA) [28] approach, which is achieved by changing the parameters of the added low-rank matrix with lower resource demands and time. Besides efficiency, another reason we choose LoRA over full-parameter fine-tuning is that LoRA updates only a small fraction of parameters, which helps preserve LLM's original comprehension ability, reduce overfitting, and enhance generalization. We used cross-entropy loss [45] with the AdamW optimizer [34]. The learning rate was set to $1e-5$ with 200 warm-up steps, $lora_r$ was set to 8, and $lora_\alpha$ was configured with 16.

After training, we evaluate the fine-tuned LLMs on the validation dataset using a comprehensive set of metrics. Specifically, we assess the quality of the extracted "label-is-element" triples in our structured requirements through Precision, Recall, and F1 Score. Additionally, we adopt the widely-used BLEU-3 [42] and ROUGE-F1 [33] metrics for machine translation to further evaluate the generated text. As a comparative baseline, we also employ GPT-5 with in-context learning (ICL) to generate structured requirements and evaluate on the same set of metrics. The evaluation results are summarized in Table 1. The prompt for GPT-5 is given in Appendix A.1.

The experimental results indicate an overall increasing trend in the metrics as the number of parameters increases. For example, within the Llama series, the 3B model outperforms the 1B model across all metrics, while the 8B model achieves even higher scores. Besides, fine-tuned models with over 3B parameters demonstrate performance surpassing that of GPT-5. Among the LLMs we fine-tuned, GLM4-9B excels in all metrics and surpasses other models, demonstrating its robust capability and superior performance. Consequently, we chose GLM4-9B as our structuring model to generate structured requirements for the subsequent steps.

Table 1. The Precision, Recall, F1 Score, BLEU-3, and ROUGE-F1 of our fine-tuned Structuring Model on the evaluation dataset. GPT-5 with in-context learning (ICL) is employed as a comparative baseline.

LLM	Precision (%)	Recall (%)	F1 Score	BLEU-3	ROUGE-F1
GPT-5 (ICL) [40]	23.9	23.5	23.7	36.0	63.6
Llama3.2-1B [25]	13.8	14.1	14.0	25.6	54.2
Llama3.2-3B [25]	65.9	65.3	65.6	68.3	84.4
Qwen1.5-4B [4]	70.4	70.1	70.2	71.5	86.1
Chatglm3-6B [24]	17.7	34.7	23.5	25.5	53.1
Qwen2-7B [58]	83.3	82.7	83.0	83.6	92.1
Llama3-8B [25]	90.3	89.0	89.6	90.5	95.7
GLM4-9B [24]	**95.2**	**94.6**	**94.9**	**95.6**	**98.1**

Table 2. Characteristics of the six constructed evaluation datasets. #X means the number of X here.

Dataset	Initial Document	#Rule	#Test Scenario	#Test Case	Updated Document	#Updated Rule	#Affected Scenario	#Affected Case
1	New York Stock Exchange Auction Market Bids and Offers Rules (2015) [11]	24	406	1587	New York Stock Exchange Auction Market Bids and Offers Rules (2017) [13]	12	319	1313
2	New York Stock Exchange Equities Trading Rules (2016) [12]	10	92	338	New York Stock Exchange Equities Trading Rules (2019) [14]	8	86	320
3	Nasdaq Stock Market Options Trading Rules (2021) [35]	28	103	130	Nasdaq Stock Market Options Trading Rules (2025) [36]	8	91	112
4	Shanghai Stock Exchange Trading Rules (2nd Ed., 2020) [15]	34	386	648	Shanghai Stock Exchange Trading Rules (2023) [17]	40	236	420
5	Shenzhen Stock Exchange Trading Rules (2021.3) [16]	27	224	470	Shenzhen Stock Exchange Trading Rules (2023) [18]	27	131	254
6	Tokyo Stock Exchange Bond Trading Regulations (2015) [19]	12	104	562	Tokyo Stock Exchange Bond Trading Regulations (2025) [20]	3	44	414

5 Evaluation

To evaluate the performance of our approach, we implemented a prototype named **ReTool** and conducted a series of experiments on it. We aim to answer the following research questions:

(1) How effectively can our approach identify regulatory changes and propagate their cascading effects to generate accurate and comprehensive test cases across diverse regulatory frameworks?
(2) To what extent can our approach identify and reuse existing test cases and test suites under regulatory changes, and how does such reuse impact overall testing efficiency?
(3) How does the choice and capacity of the LLM affect the quality of generated and reused test suites in our approach?

Datasets. Since no publicly available datasets or benchmarks exist for updating and reusing test cases under regulatory changes, we constructed six datasets from five exchanges for this experiment. Each dataset consists of an initial document, an updated document, and two corresponding sets of test cases. The regulatory documents are authentic financial rules collected from official exchange websites, capturing real modifications in regulations. Datasets 1 and 2 contain regulatory rules from the New York Stock Exchange, and Dataset 3 is from the Nasdaq Stock Market. Their corresponding test cases are *authentic cases provided by our industry customers and used in practice*. Datasets 4-6 are non-English corpora, where Dataset 4 is from the Shanghai Stock Exchange, Dataset 5 from the Shenzhen Stock Exchange, and Dataset 6 from the Japan Exchange Group. Their test cases were initially generated using the automated tool LLM4Fin [57] and subsequently refined manually and validated by our industry customers. This design ensures that the datasets encompass diverse languages, regulatory frameworks, and market contexts, thereby providing a comprehensive and realistic evaluation setting. The characteristics of the datasets are summarized in Table 2.

Competitors. Since no prior work can directly identify document changes and update test cases, we compare our approach with state-of-the-art end-to-end LLMs, including Grok-4 [55] and GPT-5 [40], using the chain-of-thought prompt detailed in Appendix A.2. GPT-5 is selected for its established reputation and consistently strong performance across diverse tasks, while Grok-4 represents a frontier model with a distinct training paradigm that emphasizes reasoning and knowledge-intensive tasks. Together, they provide an end-to-end baseline of applying general LLMs to identify rule changes and generate updated test cases. In addition, we include LLM4Fin [57], a domain-specific automatic test case generation tool, to serve as a baseline for assessing test case quality and execution efficiency when reuse is not supported.

Experimental Setup. All the experiments are conducted on a workstation equipped with a 32-core AMD Ryzen Threadripper PRO 5975WX CPU, 256GB RAM, and an NVIDIA RTX 3090 Ti GPU running Ubuntu 22.04.

5.1 Experiment I: Effectiveness of Tracing Cascading Effect and Updating Test Cases

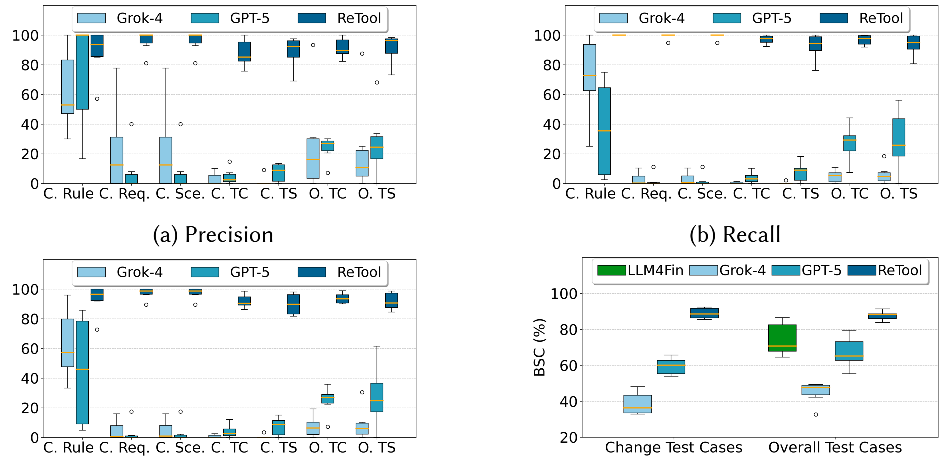
5.1.1 Experiment Preparation. We first introduce the evaluation metrics and then describe how their golden standards are constructed.
Metrics. We evaluate our approach from three complementary perspectives:

(1) **Implementation Level.** The correctness of generated test cases and test suites is assessed using Precision, Recall, and F1 Score. These metrics quantify how accurately the generated artifacts align with the cases that are actually adopted in practice, thus approximating their practical utility in detecting defects when executed.
(2) **Requirement Level.** Business Scenario Coverage (BSC) [57] measures how well the generated test cases cover the underlying requirements. It models regulatory rules as hierarchical trees of constraints and outcomes and computes the ratio of triggered constraints to the total. Higher values indicate broader coverage and a stronger guarantee of requirement-level adequacy.
(3) **Cascading-Effect Level.** To further evaluate whether regulatory changes are correctly propagated, we compute Precision, Recall, and F1 for intermediate artifacts, including the changed rules, updated requirements, and updated scenarios. This reflects the approach's ability to maintain consistency across the entire chain from regulation to executable test cases.

The ground truth for these metrics is prepared as follows: changed rules are manually identified, changed requirements and scenarios are automatically derived and then validated by domain experts, and test cases and suites are drawn from customer-provided cases used in real systems.

Due to strict confidentiality restrictions in the financial domain, we cannot access source code or defect repositories, making conventional metrics such as code coverage or fault detection inapplicable. Instead, we use customer-provided test cases as reference standards, which are manually designed and validated in practice, and compute Precision, Recall, and F1 of generated cases against them. This design indirectly approximates defect detection capability and code coverage under our constraints, consistent with prior studies [21, 51].

5.1.2 Experiment Procedure. For LLM4Fin, the updated document is directly used to generate test cases. Grok-4 and GPT-5 receive the initial document, initial test cases, and updated document simultaneously, with the chain-of-thought prompt that guides step-by-step reasoning to produce changed rules, updated requirements, scenarios, and test cases. For ReTool, the initial document and test cases are first used to construct the rule-scenario-test case graph and organize the test suites. The updated document is then processed via change impact analysis to identify the changed rules and update the corresponding test cases. Finally, the three levels of evaluation metrics are computed automatically using a script.



(a) Precision

(b) Recall

(c) F1 Score

(d) Business Scenario Coverage

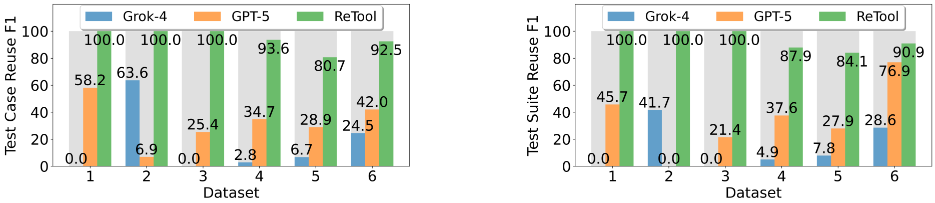
Fig. 5. Comparison of the precision, recall, and F1 score of the changed rules (C. Rule), requirements (C. Req.), scenarios (C. Sce.), test cases(C. TC), test suites (C. TS), and overall test cases (O. TC) and test suites (O. TS), as well as the business scenario coverage (BSC) of the changed and overall test cases, that identified and generated by LLM4Fin, Grok-4, GPT-5, and ReTool on six evaluation datasets. The precision, recall, and F1 scores for change-related items do not apply to LLM4Fin, as it cannot identify changes. Therefore, we only report its BSC and use it as a baseline for the quality of generated test cases.

5.1.3 Experimental Result. The experimental results are presented in Figure 5. At the implementation level where test cases and suites are measured by precision, recall, and F1, Grok-4 and GPT-5 perform poorly, with the three metrics mostly below 20% for changed test assets and 40% for overall test assets across the six datasets. In contrast, our approach consistently achieves high accuracy, with a median of the three metrics generally above 80% for the changed and overall test assets, producing reliable test cases and coherent test suites under regulations change. Regarding BSC from the requirement-level perspective, our approach consistently achieves the highest coverage, followed by LLM4Fin, while Grok-4 and GPT-5 perform the worst. This indicates that our approach better translates requirement changes into broadly covering test cases, whereas Grok-4 and GPT-5 cannot ensure comprehensive coverage. At the cascading effect level, our approach captures dependencies among changed rules, requirements, and scenarios with precision, recall, and F1 mostly above 90% across the six datasets, ensuring accurate propagation to test cases. In contrast, Grok-4 and GPT-5 fail such modeling, with F1 scores below 20%, resulting in low-quality test cases and suites. This highlights the importance of modeling cascading effects in regulatory changes.

Beyond accuracy and coverage, our method demonstrates strong generality across the six datasets, achieving the best performance across all datasets. LLM4Fin shows competitive performance on Chinese datasets but degrades on others, while Grok-4 and GPT-5 exhibit unstable performance with high variance. Our approach consistently handles diverse types of rule change, demonstrating robust applicability across languages and regulatory frameworks.

In this experiment, all metrics are computed against real test cases provided by industry collaborators. The generated test cases achieve code-coverage and fault-detection capability comparable to the ground-truth cases that are actually in use, confirming their practical value and reliability.

Conclusion: Effectively capturing cascading effects is crucial in this task. Our method excels at handling these dependencies, consistently generating high-quality test cases and demonstrating strong generality across diverse regulatory frameworks.



(a) Test Case Reuse F1 Score

(b) Test Suite Reuse F1 Score

Fig. 6. Comparison of the F1 score of the correctly reused test cases and test suites generated by Grok-4, GPT-5, and ReTool. The x-axis corresponds to the benchmark datasets, while the y-axis reports the reuse F1 score for each approach and dataset.

Table 3. Comparison of test efficiency between ReTool (with and without test case reuse) and LLM4Fin across six evaluation datasets, including Generation Time (G. Time, in seconds) and Execution Time (E. Time, in hours). Impr. represents the efficiency improvement achieved by reducing the total time (G. Time + E. Time).

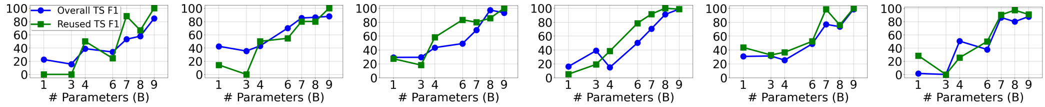
Dataset	**ReTool**				**LLM4Fin**		
	G. Time (s)	**E. Time (h, with reuse)**	**E. Time (h, no reuse)**	**Impr. (%)**	**G. Time (s)**	**E. Time (h)**	**Impr. (%)**
1	1882.5	4.3	5.4	20.4 ↑	19.5	23	79.0 ↑
2	1348.7	1.8	3.5	48.6 ↑	13.9	6.6	67.1 ↑
3	1128.3	1.3	2.7	51.9 ↑	11.5	3.9	58.7 ↑
4	1017.9	7.0	14.6	52.1 ↑	5.7	13.4	45.7 ↑
5	1712.9	1.7	7.2	76.4 ↑	11.5	19.1	88.6 ↑
6	1250.1	1.3	1.4	7.1 ↑	14.2	7.6	78.3 ↑

5.2 Experiment II: Effectiveness of Reusing Test Suites and Efficiency Gains

5.2.1 Experiment Preparation. This experiment evaluates the reuse rate of each approach and its impact on overall testing efficiency. We use the same datasets and competitors as in Experiment I. **Metrics.** Reuse rate (reuse effectiveness) is measured by the F1 score of both test cases and test suites. It balances the proportion of correctly reused test assets among all candidates (precision) and the proportion of correctly reused test assets among all reusable ones (recall), providing a comprehensive assessment that penalizes both false positives and false negatives. Efficiency is measured as the total time, which is defined as the sum of Generation Time for updating test cases and Execution Time for running the newly generated, non-reusable test cases in a real-world trading system. Execution times are recorded by our industry partners for Datasets 1-3, and are estimated for Datasets 4-6 due to the lack of relevant software. Shorter total time indicates higher efficiency, demonstrating the practical benefit of reuse in adapting to regulatory changes. The ground truth for evaluation is constructed by selecting previously reused test cases and assembling them into test suites, ensuring that metrics reflect real-world reuse potential.

5.2.2 Experiment Procedure. For reuse effectiveness, we collect the reused test cases from prior outputs, assemble them into test suites, and compute the F1 score of the correctly reused test cases and test suites. Results are reported for Grok-4, GPT-5, and our approach, while LLM4Fin is excluded as it does not support reuse. For efficiency, we measure the time to generate updated test cases and execute non-reusable cases on a real stock trading system. Since Grok-4 and GPT-5 achieved low F1 in the previous experiments, their efficiency is omitted. The total runtime of LLM4Fin serves as a baseline for non-reuse approaches, and our method is evaluated both with and without reuse.

5.2.3 Experimental Result. The experimental results are presented in Figure 6. At the test case level, ReTool consistently outperforms Grok-4 and GPT-5. Across all datasets, the test case reuse F1



(a) Dataset 1 (b) Dataset 2 (c) Dataset 3 (d) Dataset 4 (e) Dataset 5 (f) Dataset 6

Fig. 7. Comparison of the accuracy and reuse rate of the updated test cases generated by ReTool using fine-tuned Llama3.2-1B, Llama3.2-3B, Qwen1.5-4B, ChatGLM3-6B, Qwen2-7B, Llama3-8B, and GLM4-9B as the structuring model on the five evaluation datasets. The x-axis represents the number of LLM parameters.

exceeds 80%, reaching 100% in three datasets, whereas Grok-4 and GPT-5 remain below 25% and 42%, respectively. This indicates that our approach can reliably identify reusable test cases even amid complex regulatory changes. At the test suite level, ReTool again shows clear superiority, with F1 consistently exceeding 84%, while Grok-4 and GPT-5 mostly stay below 40%. The lower F1 at the suite level, even when individual test cases are correctly reused, arises from structural and sequential dependencies among cases within suites, which require careful alignment to maintain correctness. Overall, these results demonstrate the robustness of ReTool in maximizing reuse at both the case and suite levels, highlighting its ability to preserve the integrity and coherence of test suites while efficiently leveraging historical test assets.

In terms of efficiency, Table 3 shows that enabling reuse reduces total testing time by 20.4%-76.4% compared to ReTool without reuse, minimizing redundant effort by reusing unchanged test suites. Compared to LLM4Fin, ReTool achieves up to 8.8 times speedup on Dataset 5. Although the initial generation, which models cascading effects between rules and test cases, is slightly slower, execution savings dominate, confirming the practical efficiency benefits of our reuse-centric design.

Conclusion: ReTool achieves near-optimal test case and test suite reuse rate under regulatory changes, significantly reducing testing time and improving overall efficiency.

5.3 Experiment III: Impact of LLM Capacity on Reusing and Updating Test Suites

5.3.1 Experiment Procedure. This experiment investigates how the choice and capacity of LLMs influence performance, aiming to identify a practical lower bound on LLM capability required to generate high-quality test cases, thereby guiding both model selection and method optimization. To isolate this effect, we replace the original structuring model (GLM4-9B) in ReTool with the seven different LLMs fine-tuned and evaluated in Section 4.5, whose capacities increase with the number of parameters, as shown in Table 1. All other components, including prompts and downstream algorithms, are kept constant to ensure a fair comparison. We compute F1 scores for both overall and reused test suites as metrics.

5.3.2 Experimental Result. The results are shown in Fig. 7. In terms of the F1 score of the overall test suites, the performance of our approach gradually improves with the increase in model parameter scale. Particularly, LLMs with 7B parameters or more consistently achieve high F1 scores across all datasets, with an average exceeding 80%. Larger LLMs are better able to identify changes and track cascading dependencies among rules, requirements, and scenarios, thus producing high-quality test cases. A similar trend is observed for the reused test suites. F1 scores increase from LLaMA3.2-1B to Qwen2-7B, highlighting the importance of adequate LLM capacity to update and reuse test suites.

While model size clearly impacts performance, it is not the sole determinant. For example, ChatGLM3-6B and Qwen2-7B differ on Datasets 1, 5, and 6, indicating that architecture, pretraining data, and design choices also significantly influence how well an LLM captures changes, propagates dependencies, and generates reliable test suites.

Conclusion: LLMs with 7B or more parameters can effectively support high-quality test case generation and reuse by accurately capturing changes and cascading dependencies. However, differences among models of similar size underscore the importance of careful LLM selection beyond just parameter count.

6 Threats to Validity

Cascading Effect Misidentification. An internal validity threat is cascading effect misidentification, where real cascading effects fail to be captured and spurious ones are incorrectly identified as valid. Such misidentification may lead to incomplete test suites that miss critical failure paths, or redundant test suites that inflate cost and obscure true system risks. Misidentification might occur due to (i) LLMs are not perfectly accurate and can misinterpret requirements, and (ii) linking scenarios to test cases is inherently complex, as they are at different levels of granularity. To address LLM inaccuracies, we currently rely on post-processing and selective human verification, and future improvements could leverage domain-specific fine-tuning to raise accuracy. For the complexity of scenario-test case, we mitigate it with structured representations and semantic alignment, and future work could employ domain ontologies to standardize terminology and improve consistency.

Applicability to Other Domains. Our framework has so far been evaluated only in the financial domain, which raises external validity threats when applied to other areas such as automotive or medical regulations. Two main risks exist: (i) domain-specific languages (DSLs) may be required to formally capture regulatory requirements with sufficient precision and traceability, and (ii) the LLM has been fine-tuned only on financial requirements and may not generalize to other domains.

To address these risks, DSLs can be carefully designed with domain experts to ensure expressiveness and traceability. For cross-domain LLM adaptation, new domain-specific corpora can be constructed based on the DSL, followed by targeted fine-tuning. This process is expected to take roughly two weeks, the same effort as the initial financial tuning. This approach allows the framework to maintain its cascade-aware capabilities while extending to new regulatory environments.

7 Conclusion

This paper proposes a novel cascading effect-aware and LLM-driven framework centered around regulatory rules, designed to address the critical challenge of test suite reuse in frequently evolving systems. Through explicit cascading dependency modeling, the framework can precisely characterize and track fine-grained cascading effects triggered by changes. Leveraging precise change impact analysis and an incremental test generation mechanism, it transforms dynamically changing regulatory rules into reusable and self-adaptive test assets. Notably, each closed-loop process of "rule change-scenario impact-test adaptation" produces a traceable evidence chain, achieving end-to-end interpretability and robustly meeting compliance requirements for regulation and auditing. This establishes a trustworthy and continuously evolvable testing paradigm for frequently evolving software, ensuring both long-term adaptability and regulatory confidence.

In our future work, we will focus on two major directions: cross-domain generalization and large-scale industrial deployment. First, we will extend the current test case reuse framework to other strongly-regulated domains with high-frequency regulation changes, such as automotive electronics and medical software, to validate the method's generalizability and cross-domain adaptability. Furthermore, we will conduct long-term and systematic validation in more large-scale productions, evaluating its practical benefits across multiple dimensions, including cost reduction, defect detection rate, and efficiency improvement. We also plan to explore integrating AI techniques, such as large language models, to further enhance automated reasoning over regulatory requirements. Finally, we aim to develop guidelines and best practices to facilitate adoption of the framework in industry, ensuring its scalability and maintainability in complex environments.

8 Data Availability

An anonymized replication package, including code, the constructed regulatory corpus, and the technical report, is available at https://github.com/AnonymousAuthorsForFSE2026/ReTool. The fine-tuned LLMs are available at https://huggingface.co/AnonymousAuthorsForFSE2026/ReTool. Upon acceptance, we will release the replication package to ensure transparency and reproducibility.

References

[1] 425776024. 2020. NLPCDA. https://github.com/425776024/nlpcda. Accessed: 2025-05.

[2] Suriya Priya R Asaithambi and Stan Jarzabek. 2013. Towards test case reuse: A study of redundancies in android platform test libraries. In *Safe and Secure Software Reuse: 13th International Conference on Software Reuse*. Springer, 49–64.

[3] Suriya Priya R Asaithambi and Stan Jarzabek. 2015. Pragmatic Approach to Test Case Reuse-A Case Study in Android OS BiDiTests Library. In *International Conference on Software Reuse*. Springer, 122–138.

[4] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[5] Lizhi Cai, Weiqin Tong, Zhenyu Liu, and Juan Zhang. 2009. Test case reuse based on ontology. In *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE, 103–108.

[6] Xiaomeng Chen, Jinbo Wang, Shan Zhou, Panpan Xue, and Jiao Jia. 2021. Test Case Reuse based on ESIM Model. In *8th International Conference on Dependable Systems and Their Applications*. IEEE, 700–705.

[7] Nicolas Devos, Christophe Ponsard, Jean-Christophe Deprez, Renaud Bauvin, Benedicte Moriau, and Guy Anckaerts. 2012. Efficient reuse of domain-specific test knowledge: An industrial case in the smart card domain. In *2012 34th International Conference on Software Engineering*. IEEE, 1123–1132.

[8] André Di Thommazo, Kamilla Camargo, Elis Montoro Hernandes, Gislaine Gonçalves, Jefferson Pedro, Anderson Belgamo, and Sandra CPF Fabbri. 2015. Using the Dependence Level Among Requirements to Priorize the Regression Testing Set and Characterize the Complexity of Requirements Change.. In *ICEIS (2)*. 231–241.

[9] Parisa Elahidoost, Daniel Mendez, Michael Unterkalmsteiner, Jannik Fischbach, Christian Feiler, and Jonathan Streit. 2024. Practices, Challenges, and Opportunities When Inferring Requirements From Regulations in the FinTech Sector-An Industrial Study. In *2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW)*. IEEE, 137–145.

[10] Farah Elkourdi, Chenhao Wei, LU Xiao, Zhongyuan YU, and Onur Asan. 2024. Exploring current practices and challenges of HIPAA compliance in software engineering: scoping review. *IEEE Open Journal of Systems Engineering* 2 (2024), 94–104.

[11] New York Stock Exchange. 2015. New York Stock Exchange Auction Market Bids and Offers Rules (2015). "https://history.srorules.com/app/#/2025-08-23/tax/Dealings-and-Settlements+Dealings-and-Settlements-Rules-45-299C-+Auction-Market-Bids-and-Offers-Rules-63-80B-/document/Rule-64.-Bonds-Rights-and-100-Share-Unit-Stocks.html?uniqueId=0WiJe5P4EshEbN5I_3sADw==&corpus=nyse".

[12] New York Stock Exchange. 2016. New York Stock Exchange Equities Trading Rules (2016). "https://history.srorules.com/app/#/2025-08-23/tax/Rules+Rules+Rule-7P-EQUITIES-TRADING/document/Rule-7P-EQUITIES-TRADING.html?uniqueId=zHrE1fKB9Dk4x_Fl4pvfhg==&corpus=nyse".

[13] New York Stock Exchange. 2017. New York Stock Exchange Auction Market Bids and Offers Rules (2017). "https://nyseguide.srorules.com/rules/09013e2c8553e79b".

[14] New York Stock Exchange. 2019. New York Stock Exchange Equities Trading Rules (2019). "https://nyseguide.srorules.com/rules/adabe2b07d5e100088ed005056881d2301".

[15] Shanghai Stock Exchange. 2020. Shanghai Stock Exchange Trading Rules (Second Revision in 2020). "https://www.sse.com.cn/lawandrules/sselawsrules/repeal/rules/c/10118619/files/b39b8db2daa74eac916d8b3d6907af51.docx".

[16] Shenzhen Stock Exchange. 2021. Shenzhen Stock Exchange Trading Rules (Revised in March 2021). "https://docs.static.szse.cn/www/lawrules/rule/trade/current/W020210331716400647184.pdf".

[17] Shanghai Stock Exchange. 2023. Shanghai Stock Exchange Trading Rules (Revised in 2023). "https://www.sse.com.cn/lawandrules/sselawsrules/fund/trading/c/10118521/files/bc17a0dcf97e4b39accffdcfbae3eb91.docx".

[18] Shenzhen Stock Exchange. 2023. Shenzhen Stock Exchange Trading Rules (Revised in 2023). "https://docs.static.szse.cn/www/lawrules/rule/stock/W020230217564423808793.pdf".

[19] Tokyo Stock Exchange. 2015. Tokyo Stock Exchange Business Regulations (2015). "https://www.jpx.co.jp/rules-participants/rules/revise/00-archives-10.html".

[20] Tokyo Stock Exchange. 2025. Tokyo Stock Exchange Business Regulations (2025). "https://resource.lexis-asone.jp/jpx/rule/tosho_regu_201305070003001.html".

[21] Sakina Fatima, Taher A Ghaleb, and Lionel Briand. 2022. Flakify: A black-box, language model-based predictor for flaky tests. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1912–1927.

[22] Gatekeeper. 2024. The Impact of Regulatory Changes on the Financial Services Industry. https://www.gatekeeperhq.com/blog/the-impact-of-regulatory-changes-in-the-financial-services-industry.

[23] Monica Giffin, Olivier de Weck, Gergana Bounova, Rene Keller, Claudia Eckert, and P. John Clarkson. 2009. Change Propagation Analysis in Complex Technical Systems. *Journal of Mechanical Design* 131, 8 (2009), 081001.

[24] Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, et al. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793* (2024).

[25] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[26] Siqi Gu, Zhibiao Liu, Qiaoqing Liang, Zhimin Deng, and Xiangyu Xiao. 2023. Test Case Reuse Method Based on Deep Semantic Matching. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion*. IEEE, 593–599.

[27] Hui Guan, Hang Xu, and Lie Cai. 2024. Requirement dependency extraction based on improved stacking ensemble machine learning. *Mathematics* 12, 9 (2024), 1272.

[28] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *The Tenth International Conference on Learning Representations* 1, 2 (2022), 3.

[29] Pilsu Jung, Seonah Lee, and Uicheon Lee. 2024. Automated code-based test case reuse for software product line testing. *Information and Software Technology* 166 (2024), 107372.

[30] Sungwon Kang, Haeun Baek, Jungmin Kim, and Jihyun Lee. 2015. Systematic software product line test case derivation for test data reuse. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, Vol. 3. IEEE, 433–440.

[31] Tobias Landsberg, Christian Dietrich, and Daniel Lohmann. 2022. TASTING: Reuse Test-case Execution by Global AST Hashing.. In *International Conference on Software Technologies*. 33–45.

[32] Wenwu Li and Miyi Duan. 2014. A study on the software test case reuse model of feature oriented. In *2014 IEEE 3rd International Conference on Cloud Computing and Intelligence Systems*. IEEE, 241–246.

[33] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[34] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[35] Nasdaq Stock Market. 2021. Nasdaq Stock Market Options Trading Rules (2021). "https://listingcenter.nasdaq.com/RulebookVersionHistory.aspx?docId=8522&version=16".

[36] Nasdaq Stock Market. 2025. Nasdaq Stock Market Options Trading Rules (2025). "https://listingcenter.nasdaq.com/RuleBook/Nasdaq/rules/Nasdaq%20Options%203".

[37] Thomas O Meservy, Chen Zhang, Euntae T Lee, and Jasbir Dhaliwal. 2011. The business rules approach and its effect on software testing. *IEEE software* 29, 4 (2011), 60–66.

[38] Mukelabai Mukelabai, Christoph Derks, Jacob Krüger, and Thorsten Berger. 2023. To share, or not to share: Exploring test-case reusability in fork ecosystems. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 837–849.

[39] Zhang Nan, Chai Haiyan, and Han Xinyu. 2018. Research on the reuse of test case for warship equipment software. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion*. IEEE, 64–69.

[40] OpenAI. 2025. ChatGPT. "https://chatgpt.com/".

[41] Kamalendu Pal. 2020. Framework for Reusable Test Case Generation in Software Systems Testing. In *Software Engineering for Agile Application Development*. IGI Global, 212–229.

[42] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[43] Sachin Patel and Ramesh Kumar Kollana. 2014. Test Case Reuse in Enterprise Software Implementation–An Experience Report. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 99–102.

[44] PwC. 2025. Our Take: financial services regulatory update. https://www.pwc.com/us/en/industries/financial-services/library/our-take/01-24-25.html.

[45] Reuven Rubinstein. 1999. The cross-entropy method for combinatorial and continuous optimization. *Methodology and computing in applied probability* 1 (1999), 127–190.

[46] Ya-Qing Shi, Song Huang, and Jin-Yong Wan. 2022. A Reuse-oriented Clustering Method for Test Cases. In *2022 9th International Conference on Dependable Systems and Their Applications*. IEEE, 153–162.

[47] Luis Alvaro de Lima Silva, Lori RF Machado, and Leonardo Emmendorfer. 2024. A Case and Cluster-Based Framework for Reuse and Prioritization in Software Testing. In *Proceedings of the 20th Brazilian Symposium on Information Systems*. 1–10.

[48] Baradwaj Bandi Sudakara. 2025. Leading Cross-Functional QA in Healthcare: A Playbook for Automation and Compliance at Scale. *Journal of Computer Science and Technology Studies* 7, 8 (2025), 937–945.

[49] Hamid Tabani, Leonidas Kosmidis, Jaume Abella, Francisco J Cazorla, and Guillem Bernat. 2019. Assessing the adherence of an industrial autonomous driving framework to iso 26262 software guidelines. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[50] Saeid Tizpaz-Niari, Verya Monjezi, Morgan Wagner, Shiva Darian, Krystia Reed, and Ashutosh Trivedi. 2023. Metamorphic testing and debugging of tax preparation software. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Society*. IEEE, 138–149.

[51] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).

[52] Bangchao Wang, Rong Peng, Yuanbang Li, Han Lai, and Zhuo Wang. 2018. Requirements traceability technologies and technology transfer decision support: A systematic review. *Journal of Systems and Software* 146 (2018), 59–79.

[53] Wentao Wang, Faryn Dumont, Nan Niu, and Glen Horton. 2020. Detecting software security vulnerabilities via requirements dependency analysis. *IEEE Transactions on Software Engineering* 48, 5 (2020), 1665–1675.

[54] David Willmor and Suzanne M Embury. 2006. Testing the implementation of business rules using intensional database tests. In *Testing: Academic & Industrial Conference-Practice and Research Techniques*. IEEE, 115–126.

[55] xAI. 2025. Grok 4. "https://grok.com/".

[56] Xinyue Xu, Sinong Chen, Zhonghao Guo, and Xiangxian Chen. 2025. PSTR: A Test Case Reuse Method Based on Path Similarity (September 2024). *IEEE Access* (2025).

[57] Zhiyi Xue, Liangguo Li, Senyue Tian, Xiaohong Chen, Pingping Li, Liangyu Chen, Tingting Jiang, and Min Zhang. 2024. LLM4Fin: Fully Automating LLM-Powered Test Case Generation for FinTech Software Acceptance Testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1643–1655.

[58] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115* (2024).

[59] Wansheng Yang, Fei Deng, Siyou Ma, Linbo Wu, Zhe Sun, and Chi Hu. 2021. Test case reuse based on software testing knowledge graph and collaborative filtering recommendation algorithm. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion*. IEEE, 67–76.

[60] He Zhang, Juan Li, Liming Zhu, Ross Jeffery, Yan Liu, Qing Wang, and Mingshu Li. 2014. Investigating dependencies in software requirements for change propagation analysis. *Information and Software Technology* 56, 1 (2014), 40–53.

[61] Shunran Zhang, Zhengmin Jiang, Ming Sang, Yunjie Han, Huiyun Li, et al. 2025. A Comprehensive Survey on Driving Compliance Assessment Methodologies for Autonomous Vehicles. *IEEE Intelligent Transportation Systems Magazine* (2025).

A Appendix

A.1 Prompt Design for Structured Conversion of Financial Business Rules Using GPT-5

This section presents the carefully engineered prompt design employed to guide GPT-5 in transforming natural language financial business rules into structured rule representations. The prompt framework strategically combines task-driven and example-driven components to enable precise semantic parsing and formalization of complex business logic. Key elements of the prompt design include:

1. **Task-Driven Instructions**: Clearly specify the objective of converting natural language financial rules into rigorously defined if-then logical formats, grounded in a predefined label system. Emphasis is placed on preserving the complete semantics of original rules while ensuring machine interpretability, thus supporting downstream applications such as requirements analysis, automated code generation, and test case creation.
2. **Label Taxonomy and Rule Decomposition Strategies**: Define a detailed set of predefined semantic labels alongside flexible extension rules to accommodate novel concepts. Specific instructions on splitting complex or conjunctive conditions into multiple rules enhance the precision and usability of the structured outputs.
3. **Output Format and Semantic Fidelity Constraints**: Standardize the triple-based representation format, impose strict semantic fidelity requirements, and address special cases such as negations and range values to ensure generated rules remain consistent, clear, and faithful to business intent.
4. **Example-Driven Guidance**: Provide comprehensive input and output examples illustrating typical business rules, demonstrating rule decomposition, multi-condition expressions, label selection, and extension principles. These exemplars facilitate the model's accurate understanding of intricate domain logic and expression nuances.

The specific prompts used are shown below. It reflects a deep comprehension of financial domain language characteristics and leverages large language model capabilities effectively, providing a robust foundation for automating the structured formalization of business rules.

```
You are an expert in the field of financial technology. Your task is to perform structured transformation
of financial business rules.

## **Task Description**
This task requires converting natural language business rules in the financial domain into structured
rules. The structured rules must follow a strict **if-then** logic format and use a predefined labeling
system to precisely express conditions and constraints. The transformation must **preserve the complete
semantics** of the original rule while ensuring **machine interpretability**, making it suitable for
scenarios such as **requirement analysis**, **automatic code generation**, and **automatic test case
generation**.

## **Input**
- **Type**: One or more financial business rules described in natural language.
- **Example**:
  ```
  For bond transactions using the matched trade method, the declared quantity must be in face value units
of 100,000 yuan or multiples thereof.
  ```
- **Requirements**:
  - Input must be complete sentences in Chinese.
  - May include financial terminology related to time conditions, state descriptions, and operational
constraints.
  - Can contain conjunctions (e.g., "and", "or") or nested logic.

## **Output Format**
### **1. Basic Structure**
```

~~~
```
rule [number]\nif [condition clause]\nthen [constraint clause]
```
- **Condition Clause (if)**: Describes the trigger condition for the rule, formed by `key is value`
triplets connected with `and`.
- **Constraint Clause (then)**: Describes the restriction once the condition is met, also using the same
triplet structure.
- **Multi-rule splitting**: When the natural language rule includes parallel logic or complex semantics,
split into multiple rules (e.g., `rule 1`, `rule 2`).

### **2. Triplet Structure**
```
key is value
```
- **key**: Must be selected from the predefined label set (see below), or flexibly extended based on
context.
- **value**: Specific values extracted from the natural language input (redundant modifiers should be
removed).

### **3. Predefined Label Set**
| Label     | Meaning            | Example Values     |
|-----------|-------------------|--------------------|
| Operator  | Who acts           | Investor, Broker   |
| Action    | What is done       | Buy, Sell, Cancel  |
| Target    | Acted-on object    | Security, Account  |
|Target Part|Part of the target  | Principal, Interest|
| Business  | Business type      | Margin, Options    |
| Product   | Product category   | Stock, Bond, ETF   |
| Method    | Execution method   |Market, Limit, After|
|Trade Type | Trade category     | Spot, Repo, Deriv. |
| Time      | Time condition     | T+1, 15:00 on day  |
| Quantity  |Quantity constraint| >= 100 shares      |
| Price     | Price constraint   | >= 10%, +-2%       |
| State     | Current state      | Holding, Suspended |
| Event     | Trigger event      |Ex-dividend, Default|

### **4. Principles for Flexible Label Extension**
When the predefined labels are insufficient, follow the priority rules below:
1. **Semantic Reduction**: Map new terms to the closest existing label (e.g., "block trade" -> Method).
2. **Contextual Derivation**: Create labels based on financial domain knowledge (e.g., "volatility" ->
Price attribute).
3. **Generic Label**: Use the "Constraint" label when no suitable match is available.

### **5. Multi-rule Splitting Scenarios**
- **Parallel Conditions**: Split when natural language contains "and"/"or".
  ```
  Natural language:
  When the price increase exceeds 5% or the trading volume exceeds 10,000 lots, the circuit breaker is
triggered.
  ```
  ```
  Structured:
  rule 1\nif Price is Increase > 5%\nthen Event is Trigger Circuit Breaker
  rule 2\nif Quantity is Volume > 10,000 lots\nthen Event is Trigger Circuit Breaker
  ```
- **Nested Logic**: Complex conditions should be broken into independent rule chains.

## **Notes**
1. **Semantic Fidelity**:
   - Do not add or remove core constraints from the original rule (e.g., "not allowed" must be rendered as
 `Action is Not Allowed`).
   - Normalize time expressions (e.g., "before market close" -> "before 15:00 on the day").
2. **Label Selection Guidelines**:
   - Prefer predefined labels; avoid excessive expansion.
   - Do not assign the same value to multiple different labels (e.g., "after-hours pricing" should only
map to Method).
3. **Special Value Handling**:
   - Negations: Express in the value (e.g., `Action is Not Allowed`), do not introduce a new label.
   - Ranges: Retain original comparison symbols (e.g., `Price is >= 10 yuan`).
~~~

~~~
4. **Financial Terminology Norms**:
   - Use standard terminology (e.g., "suspension" instead of "pause trading").
   - Use full names for institutions (e.g., "China Securities Depository and Clearing Corporation" instead
 of abbreviation).
5. **Output Validation Checklist**:
   - Each rule must contain at least one `if` triplet and one `then` triplet.
   - Within a single rule, the same key must not appear multiple times.
   - All conditions in the original rule must be covered by the resulting rules.

## **Complete Example**
**Input**:
```
Unless otherwise specified by the exchange, the following quantity rules apply to bond trading: (1) For
matched trades, declared quantity of spot bonds must be in face value units of 100,000 yuan or multiples
thereof. For sell orders of amounts less than 100,000 yuan face value, declaration must be one-time. For
general pledged repo of bonds, the quantity must be in face value units of 1,000 yuan or multiples thereof
; (2) For click-trades, the declared quantity must be in face value units of 100,000 yuan or multiples
thereof; (3) For inquiry or auction trades, the declared quantity must be no less than 100,000 yuan and be
 in face value units of 1,000 yuan; (4) For negotiated trades, declared quantity of spot bonds must be no
less than 1,000 yuan and be in face value units of 100 yuan; for general pledged repo, it must be in face
value units of 1,000 yuan or multiples thereof.
```
**Output**:
```
rule 1
if Operator is Exchange and Method is Matched Trade and Product is Spot Bond
then Quantity is 100,000 yuan or multiples

rule 2
if Operator is Exchange and Method is Matched Trade and Product is Spot Bond and Action is Sell and
Remaining Quantity is Less Than 100,000 yuan
then Action is One-time Declaration

rule 3
if Operator is Exchange and Method is Matched Trade and Product is General Pledged Repo
then Quantity is 1,000 yuan or multiples

rule 4
if Operator is Exchange and Product is Bond and Method is Click Trade
then Quantity is 100,000 yuan or multiples

rule 5
if Operator is Exchange and Product is Bond and Method is Inquiry Trade
then Quantity is >= 100,000 yuan and Quantity is 1,000 yuan multiples

rule 6
if Operator is Exchange and Product is Bond and Method is Auction Trade
then Quantity is >= 100,000 yuan and Quantity is 1,000 yuan multiples

rule 7
if Operator is Exchange and Method is Negotiated Trade and Product is Spot Bond
then Quantity is >= 1,000 yuan and Quantity is 100 yuan multiples

rule 8
if Operator is Exchange and Method is Negotiated Trade and Product is General Pledged Repo
then Quantity is 1,000 yuan or multiples
```
~~~

A.2 Prompt Design for Updating and Reusing Test Suites When Regulations Change

This section outlines the prompt design for guiding LLMs (Grok-4 and GPT-5) in updating and reusing JSON-format test case repositories under financial requirement changes. Since the overall principles of task-driven instruction, semantic label design, and output format control have been detailed in the previous section, here we briefly highlight the adaptations specific to the test case reuse task:

(1) **Task-Oriented Guidance**: The prompt explicitly instructs LLMs to analyze differences between original and revised requirement documents, identify added/modified/deleted business rules, and accordingly update the JSON test case repository. The objective is defined as maximizing test case reuse while maintaining full alignment with the latest requirements.
(2) **Mapping and Update Strategies**: Building upon the label taxonomy principles discussed earlier, the prompt provides rules for mapping unchanged business rules directly to existing test cases, regenerating test cases for modified rules, and discarding those linked to deleted rules. New rules must be covered by freshly generated test cases that reflect positive, negative, and boundary conditions.
(3) **Format Consistency and Traceability**: As in the prior prompt design, strict constraints are imposed on the output format. Test cases must preserve the JSON schema, ensure rule fields maintain one-to-one traceability to business rules, and assign new unique identifiers to regenerated cases.
(4) **Illustrative Examples**: A minimal set of input-output examples is included to demonstrate the expected update logic. Compared to the previous section's detailed semantic parsing examples, here the focus is on showing how changes in requirements propagate to test case transformations.

This adapted prompt design inherits the general principles of precision, fidelity, and clarity from the earlier framework, while tailoring them to the practical objective of intelligent test case reuse under evolving financial requirements.

```
You are an expert in financial software testing and quality assurance. Your task is to intelligently
update and reuse an existing JSON-format test case repository when financial business requirements change.

## **Task Description**
This task requires intelligently updating a pre-existing JSON-format test case repository based on changes
 in financial business requirements. The system must analyze the differences between the original
requirement document and the revised requirement document, and accurately identify newly added, modified,
or deleted business rules.
- For business rules that remain unchanged (identified by the `rule` field), their associated test cases
must be directly reused.
- For modified business rules, new test cases must be generated to accurately validate the updated rules,
replacing the old ones.
- For deleted rules, their associated test cases should be removed.
The overall goal is to **maximize reuse of testing assets** while ensuring that the test case repository
remains fully aligned with the latest business requirements, thereby guaranteeing the accuracy and
stability of the financial trading system.

## **Input**
- **Type**: A data package containing three core elements:
  1. **Original Requirement Document**: A document describing the original financial business rules (e.g.,
 business rule document, Software Requirement Specification SRS).
  2. **New Requirement Document**: A document describing revised or newly added financial business rules.
  3. **Original Test Case Repository**: A JSON array, where each object represents a test case. Each test
case object contains multiple key-value pairs describing the test scenario's input, operations, and
expected outcomes. Key fields include:
     - `rule` (ID of the associated business rule)
     - `testid` (unique test case identifier)
     - other fields describing business scenarios and results (e.g., "Actor", "Exchange Market", "Result
Status", etc.).

## **Output**
- **Type**: An updated JSON-format test case repository.
  - The output must be a JSON array with the same structure as the input repository.
  - For test cases associated with unchanged rules (identified via `rule`), the cases must be preserved
exactly, including all key-value pairs.
```

~~~
  - For test cases associated with changed, added, or deleted rules:
    - **Modified/Added Rules**: Generate new test case objects. The `rule` field should point to the
updated or new rule ID, `testid` must be regenerated as a new unique identifier, and other field values
must be set according to the new rule to accurately reflect the updated business logic and expected
outcomes.
    - **Deleted Rules**: Test cases associated with deleted rules should be excluded from the output.
  - The output repository must fully cover all valid business rules in the new requirement document.

## **Requirements**
- **Accuracy**: Newly generated test cases must precisely reflect the details of the new requirements,
with all key-value pairs strictly matching the updated business rules.
- **Traceability**: Each test case's `rule` field must accurately point to its corresponding requirement/
rule ID, ensuring bidirectional traceability between requirements and tests.
- **Efficiency**: Unaffected test cases must be reused without modification. Only test cases linked to
changed rules should be replaced with newly generated ones, with updated `testid`s to indicate distinction
.
- **Format Consistency**: The updated test case repository must strictly follow the same JSON structure as
 the input (outer array and inner object keys) to ensure compatibility with existing test automation
frameworks or test management systems.
- **Completeness**: For modified rules, sufficient test cases must be generated to cover positive,
negative, and boundary conditions, ensuring thorough testing.

## **Example**
### Input Example

1. Original Requirement Document (excerpt):
  - Rule 3.2.9.1.1.1: A client placing a **security** buy order at the Shanghai Stock Exchange, when in "
Not Submitted" status, may perform a "Cancel" operation. After execution, the order status becomes "
Cancelled", and the operation succeeds.
2. New Requirement Document (excerpt):
  - Rule 3.2.9.1.1.1 (Modified): A client placing a **fund** buy order at the Shanghai Stock Exchange,
when in "Not Submitted" status, may perform a "Cancel" operation. After execution, the order status
becomes "Cancelled", and the operation succeeds.
  - Rule 3.2.9.1.1.2 (New): A client placing a fund sell order at the Shenzhen Stock Exchange, when in "
Partially Filled" status, may perform a "Cancel" operation. After execution, the unfilled portion of the
order becomes "Cancelled", and the operation succeeds.
3. Original Test Case Repository (JSON excerpt):
```json
[
  {
    "rule": "3.2.9.1.1.1",
    "testid": "3.2.9.1.1.11",
    "actor": "client",
    "exchange": "Shanghai Stock Exchange",
    "direction": "buy",
    "instrument": "security",
    "status": "not_submitted",
    "action": "cancel",
    "action_part": "unfilled portion",
    "final_status": "cancelled",
    "result": "success"
  },
  {
    "rule": "3.2.9.1.1.1",
    "testid": "3.2.9.1.1.12",
    "actor": "client",
    "exchange": "Shanghai Stock Exchange",
    "direction": "buy",
    "instrument": "security",
    "status": "not_submitted",
    "action": "non_cancel",
    "final_status": "not_submitted",
    "result": "failure"
  }
]
```

### Output Example
~~~

Based on the above input, the system should detect that Rule `3.2.9.1.1.1` has been modified ("Security" -> "Fund"), and a new Rule `3.2.9.1.1.2` has been added. New test cases must therefore be generated for the modified rule, and brand-new test cases must be created for the new rule.

```json
[
  {
    "rule": "3.2.9.1.1.1",
    "testid": "3.2.9.1.1.13",
    "actor": "client",
    "exchange": "Shanghai Stock Exchange",
    "direction": "buy",
    "instrument": "fund",
    "status": "not_submitted",
    "action": "cancel",
    "action_part": "unfilled portion",
    "final_status": "cancelled",
    "result": "success"
  },
  {
    "rule": "3.2.9.1.1.1",
    "testid": "3.2.9.1.1.14",
    "actor": "client",
    "exchange": "Shanghai Stock Exchange",
    "direction": "buy",
    "instrument": "fund",
    "status": "not_submitted",
    "action": "non_cancel",
    "final_status": "not_submitted",
    "result": "failure"
  },
  {
    "rule": "3.2.9.1.1.2",
    "testid": "3.2.9.1.1.21",
    "actor": "client",
    "exchange": "Shenzhen Stock Exchange",
    "direction": "sell",
    "instrument": "fund",
    "status": "partially_filled",
    "action": "cancel",
    "action_part": "unfilled portion",
    "final_status": "cancelled",
    "result": "success"
  }
]
```