Explicating Tacit Regulatory Knowledge from LLMs to Auto-Formalize Requirements for Compliance Test Case Generation

Anonymous ACL submission

Abstract

Compliance testing in highly regulated domains is crucial but largely manual, requiring domain experts to translate complex regulations into executable test cases. While large language models (LLMs) show promise for automation, their susceptibility to hallucinations limits reliable application. Existing hybrid approaches mitigate this issue by constraining LLMs with formal models, but still rely on costly manual modeling. To solve this problem, this paper proposes RAFT, a framework for Requirements Auto-Formalization and compliance Test generation via explicating tacit regulatory knowledge from multiple LLMs. RAFT employs an Adaptive Purification-Aggregation strategy to explicate tacit regulatory knowledge from multiple LLMs and integrate it into three artifacts: a domain meta-model, a formal requirements representation, and testability constraints. These artifacts are then dynamically injected into prompts to guide high-precision requirement formalization and automated test generation. Experiments across financial, automotive, and power domains show that RAFT achieves expert-level performance, substantially outperforms state-of-the-art (SOTA) methods while reducing overall generation and review time.

1 Introduction

In highly regulated domains such as finance, automotive, and energy, software compliance testing is crucial to ensure adherence to regulations, industry standards, and internal policies (Badrzadeh and Halley, 2014; Tabani et al., 2019; Xue et al., 2024b; Zhang et al., 2025). Failures in compliance can lead to severe safety incidents, regulatory penalties, and substantial financial losses (Wikipedia contributors, 2014; Sivakumar et al., 2024). Despite its importance, compliance testing in practice remains predominantly manual. Domain experts are required to interpret lengthy and complex regulatory texts and translate them into executable test cases. This process is labor-intensive and error-prone, making its automation a key focus of industry development (Castellanos Ardila et al., 2022; Jain et al., 2025).

Existing automated solutions face a trade-off between modeling cost and generation reliability. Traditional model-based approaches rely on explicitly constructed models, graphs, or formal representations for test generation (Wang et al., 2020; Stefani et al., 2025; Zyberaj et al., 2025; Yang et al., 2025). While precise and controllable, these approaches require substantial upfront expert effort, limiting their scalability to other regulatory domains. In contrast, LLM-based methods reduce manual effort by generating tests directly from regulatory texts or via multi-agent workflows (Castellanos Ardila et al., 2022; Boukhlif et al., 2024; Korraprolu et al., 2025; Hasan et al., 2025; Masuda et al., 2025). However, their probabilistic nature leads to hallucinations and low interpretability, hindering their adoption in regulation-critical settings.

To reconcile this tension, hybrid test generation frameworks have combined model-driven techniques with LLM-based generation (Xue et al., 2024b; Chen et al., 2024; Sun et al., 2025; Liu et al., 2025; Shrestha et al., 2025). In these approaches, LLMs translate natural language regulations into predefined formal requirements, which constrain generation and mitigate hallucinations. However, the representation of these requirements still requires substantial manual definition and maintenance by domain experts; when the domain shifts (e.g., from finance to automation), they must be re-engineered, resulting in high maintenance costs.

Automating requirements formalization itself is the most direct way to break this deadlock, but it demands regulatory knowledge on demand. Hence, the paradigm must move from "manual modeling" to "automatic knowledge explication". Modern LLMs, having undergone large-scale pre-training, already encode a wealth of tacit regulatory knowledge, yet it remains implicit and unstructured. Only

by unlocking and formalizing this knowledge into explicit, verifiable artifacts can we enable test-case generation that is both reliable and scalable.

In this paper, we propose RAFT, a novel framework for Requirement Auto-Formalization and compliance Test generation via explicating tacit regulatory knowledge from multiple LLMs. RAFT introduces a dedicated *Regulatory Knowledge Explication* phase, in which an Adaptive Purification-Aggregation strategy is employed to extract and formalize three knowledge artifacts from LLMs: a domain meta-model, a formal requirements representation, and testability constraints. Together, these artifacts establish a coherent conceptual and logical foundation prior to processing individual regulatory texts. Building upon these knowledge artifacts, RAFT performs knowledge-injected prompting during the *Prompt and Test Case Generation* phase. The extracted artifacts are dynamically incorporated into prompts as structural and semantic constraints, guiding the LLM to formalize regulations into precise, interpretable requirements that can be transformed into high-quality test cases.

RAFT promotes LLMs from "Text Translators" to "Knowledge Architects". By explicitly extracting and injecting tacit regulatory knowledge, the framework dramatically boosts the controllability, interpretability, and reliability of LLM-driven requirements formalization and compliance testing, delivering a fully automated and scalable solution.

We conducted a comprehensive evaluation on RAFT. In the financial domain, it achieved expert-level performance with an average 91.7% F1 and 86.5% business scenario coverage, outperforming SOTA methods by up to 34.7%, while reducing the total generation and review time from 5.7 hours or even 2 weeks to just 2.5 hours. Ablation studies confirm that each explicated knowledge artifact, as well as the proposed Adaptive Purification-Aggregation strategy, is essential for high-quality test generation. Furthermore, RAFT demonstrates strong cross-domain generalizability, achieving average F1 scores of 92.7% and 87.2% in the automotive and power domains, respectively. Our code and data are available at https://github.com/1767675261/RAFT.

Our contributions are summarized as follows:

(1) **Knowledge Enhanced Requirements formalization and Test Generation:** We propose RAFT, a framework that automatically explicates regulatory knowledge from LLMs and injects it into the test generation, shifting compliance testing from expert-driven modeling to knowledge-driven automation.

(2) **Regulatory Knowledge Explication:** We define and extract three reusable, explicit artifacts, including domain meta-model, formal requirements representation, and testability constraints, turning originally implicit and fragmented tacit regulatory knowledge into verifiable and cross-domain portable assets.

(3) **Knowledge-Constrained Prompting:** We devise a prompting scheme that dynamically embeds the three artifacts as structured constraints in the prompt, enabling LLMs to produce precise, regulation-compliant outputs without retraining, balancing accuracy and generality.

2 Related Work

This section reviews prior work on automated test case generation, focusing on how regulatory knowledge is represented and utilized.

Model-Based Test Generation. Model-based testing introduces explicit intermediate graphs or formal models for testing. Representative examples include control-flow graphs for path coverage (Yang et al., 2025), System Graphs converted into executable tests (Zyberaj et al., 2025), and formal ODD or rule models for safety-critical systems (Stefani et al., 2025; Xue et al., 2024b). While controllable, these methods depend heavily on manually constructed models that are complex and labor-intensive, limiting scalability (Xue et al., 2024a).

LLM-Based Test Generation. Recent studies leverage LLMs or multi-agent systems to generate test cases. Prompt-based methods produce test inputs or structured test steps (Wang et al., 2025; Korraprolu et al., 2025), while multi-agent pipelines emulate human testing workflows (Milchevski et al., 2025) or extract testing strategies prior to generation (Masuda et al., 2025). Domain-specific fine-tuning has also been explored (Boukhlif et al., 2024; Hasan et al., 2025). While these approaches reduce manual effort, their reliance on probabilistic outputs leads to hallucinations and instability, limiting their applicability in highly regulated domains (Zhang et al., 2024; Korraprolu et al., 2025).

Hybrid Frameworks. Hybrid approaches combine formal representations with LLM-based generation to balance controllability and flexibility. Examples include integrating LLMs with structured test scenarios in financial software (Xue et al., 2024b; Liu et al., 2025), combining ontologies and

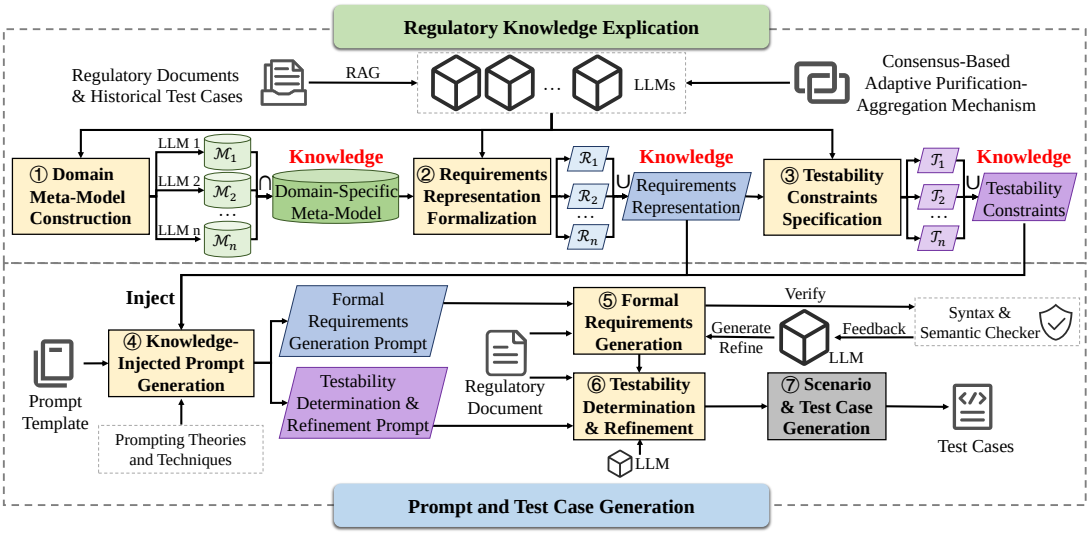

Figure 1: The regulatory knowledge explication and injection-driven framework for fully automated test generation.

learning-based methods for BIM compliance checking (Chen et al., 2024), and guiding LLMs with retrieval-augmented generation or predefined requirement templates (Sun et al., 2025; Shrestha et al., 2025). Although these approaches mitigate hallucinations by constraining the generation space, they still rely on hand-crafted modeling, leaving regulatory knowledge statically engineered and thus limiting scalability and adaptability.

3 The RAFT Approach

3.1 Regulatory Knowledge and Overview

To achieve fully automated test case generation based on a hybrid framework, it is necessary to automate the conversion of regulatory documents into formal requirements. Accomplishing this goal requires satisfying two prerequisites: first, clarifying how requirements should be expressed; second, confirming that the requirements are testable.

Regarding the question of "how to represent requirements", we focus on two aspects: the composition of requirement elements and the form of their expression. Accordingly, the representation process is divided into two stages: first, extracting key requirement elements to construct a requirement metamodel; and then completing the formal requirements representation based on this model. In parallel, we define testability constraints to identify testable requirements. Ultimately, we derive three core types of regulatory knowledge: the domain meta-model $\mathcal{M}$, the requirements representation $\mathcal{R}$, and the testability constraints $\mathcal{T}$.

After the regulatory knowledge is formed, it can be systematically embedded into a prompt-based framework to guide LLMs in two key tasks: converting natural-language clauses into formal requirements and automatically generating executable test cases. To this end, we propose RAFT (Figure 1), a two-phase approach that ensures semantic consistency and testability by combining knowledge explicitation with prompt engineering:

Phase I: Regulatory Knowledge Explicitation. The domain meta-model $\mathcal{M}$, the requirements representation $\mathcal{R}$, and the testability constraints $\mathcal{T}$ from regulatory documents and existing test cases to establish a structured knowledge system.

Phase II: Prompt and Test Case Generation. Inject the explicated knowledge into prompt templates to steer the LLMs through sequential requirement formalization and test case generation, resulting in an end-to-end automated workflow.

3.2 Regulatory Knowledge Explication

Challenges in explicating regulatory knowledge from LLMs. The core objective of this phase is to obtain the three knowledge assets: the domain meta-model $\mathcal{M}$, the requirements representation $\mathcal{R}$, and the testability constraints $\mathcal{T}$. We decompose the process into three extraction steps, all implemented using LLMs. To assess the feasibility of automated regulatory knowledge explication, we conducted preliminary experiments using SOTA LLMs to construct these assets with direct instructions. The results revealed two recurring challenges:

(1) **Granularity and Structural Inconsistency.** LLMs often fail to identify an appropriate level of abstraction, leading to missing elements or

inconsistent structural representations.

(2) **Hallucination and Output Variability.** Generated structures may be ungrounded or unstable across runs, hindering the construction of reliable regulatory knowledge.

To address these issues, we propose two corresponding solutions: a Chain-of-Thought (CoT) (Wei et al., 2022) enhanced with Retrieval Augmented Generation (RAG) (Lewis et al., 2020) prompting method, and a multi-LLM-based Adaptive Purification-Aggregation strategy.

CoT & RAG Prompting. This mechanism constrains LLM reasoning along four dimensions: (1) *Task Orientation*, framing the task as explicit reconstruction of tacit regulatory knowledge; (2) *Hierarchical Decomposition*, enforcing a structured whole-to-part reasoning process tailored to each knowledge type; (3) *Granularity Constraint*, restricting the level of detail to what is necessary for the target artifact; and (4) *Retrieval Grounding*, incorporating regulatory documents and historical test cases to reduce hallucinations. This prompt framework is applied to *Steps* ①, ②, and ③, with task-specific adaptations.

Adaptive Purification-Aggregation. The key idea is to apply reconciliation mechanisms tailored to different knowledge types, reflecting their distinct structural and semantic characteristics. Specifically, In *Step* ①, the domain meta-model requires both structural integrity and parsimony (Evans, 2004). We therefore introduce a K*-Purification* operation, which treats individual LLM outputs as noisy observations and extracts a stable meta-model core via majority consensus:

Definition: K-Purification

Let $\mathbf{M} = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_N\}$ be the meta-models generated by N independent LLMs. Given a consensus threshold K ($1 < K \leq N$), the final meta-model $\mathcal{M}$ is defined as:

$$\mathcal{M} = \{e \in \bigcup_{i=1}^{N} \mathcal{M}_i \mid \sum_{i=1}^{N} \mathbb{I}(e \in \mathcal{M}_i) \geq K\} \quad (1)$$

where e denotes meta-model elements and $\mathbb{I}(\cdot)$ is the indicator function.

We set $N = 3$ and $K = 2$ in our implementation, retaining majority-supported elements to suppress stochastic hallucinations from individual LLMs, while grouping low-frequency elements as "Others" to preserve coverage of regulatory semantics.

In *Steps* ② and ③, the primary risk shifts from structural noise to information omission (Chomsky, 2002). Since missing regulatory logic or testability constraints can undermine downstream correctness, we adopt an *Aggregation* strategy that prioritizes logical completeness over frequency:

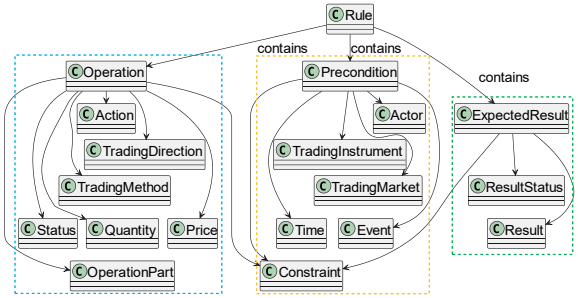

Figure 2: Meta-model for the finance domain.

Definition: Aggregation

Given formal requirements representations $\{\mathcal{R}_i\}_{i=1}^{N}$ and testability constraints $\{\mathcal{T}_j\}_{j=1}^{N}$ from N LLMs, aggregation is performed via:

$$\mathcal{R} = \bigcup_{i=1}^{N} \mathcal{R}_i, \quad \mathcal{T} = \bigcup_{j=1}^{N} \mathcal{T}_j \quad (2)$$

With $N = 3$, this strategy minimizes omission risk by aggregating all inferred regulatory logic and constraints. Overall, the Adaptive Purification-Aggregation strategy mitigates hallucination and divergence in single LLMs by enforcing structural rigor in meta-model construction and logical completeness in requirements representation formalization and testability constraints specification.

Example of Explicated Regulatory Knowledge. We illustrate the explicated regulatory knowledge using the finance domain, employing GPT-5 (OpenAI, 2025), Grok-4 (xAI, 2025), and DeepSeek-R1 (Guo et al., 2025). The financial meta-model $\mathcal{M}_f$ is expressed as PlantUML, consisting of 14 core elements (e.g., *TradingVariety*, *Price*) organized into a three-layer hierarchy in PlantUML (Team, 2025), as shown in Figure 2. By explicitly separating *Preconditions*, *Operations*, and *Expected Results*, the meta-model provides a verifiable causal structure and consolidates vague regulatory concepts into representative domain entities.

Grounded in $\mathcal{M}_f$, the finance requirements representation $\mathcal{R}_f$ defines a library of domain symbols and a BNF-based (Backus et al., 1960) formal syntax. As exemplified in Figure 3, requirements follow a structured *IF–THEN* logic composed of *KEY–OP–VALUE* clauses connected by logical operators, where *KEY* is strictly aligned with entities in $\mathcal{M}_f$ to ensure semantic consistency, *VALUE*

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.
Rule 2: A NYSE Bonds GTD Order is eligible for execution in the Core Bond Trading Session only. A designation for any other trading session but the Core Bond Trading Session will be cancelled.

Requirement 1.1
IF Action = Trade AND TradingMethod = Block Trading AND TradingVariety = Security AND Quantity = 10,000 shares or more OR Price = $200,000 or more
THEN Result = Succeed
Requirement 2.1
IF TradingMarkey = NYSE AND TradingVariety = Bond AND OperationPart = GTD Order AND Action = execute AND Time = Core Bond Trading Session
THEN Result = Succeed
Requirement 2.2
IF TradingMarkey = NYSE AND TradingVariety = Bond AND OperationPart = GTD Order AND Action = execute AND Time != Core Bond Trading Session
THEN Result = Failure

Figure 3: An example of 2 regulatory rules from NYSE (Exchange, 2025d) and their requirement representation.

specifies the corresponding instance, and *OP* denotes the operator that defines their relations.

Based on $\mathcal{R}_f$, finance testability constraints $\mathcal{T}_f$ are specified as OCL constraints (Group, 2014) covering five aspects, including *Structural Completeness*, *Element Determinism*, *Action Executability*, *Result Observability*, and *Rule Non-Conflict*. For example, *Rule 1* in Figure 3 satisfies all constraints and is therefore testable, whereas *Rule 2* is deemed untestable due to the non-deterministic element "core bond trading session".

3.3 Prompt and Test Case Generation

This phase focuses on transforming regulatory rules into executable test cases through formalizing requirements. There are four steps: Prompt Generation, Requirements Formalization, Testability Determination & Refinement, and finally Test Case Generation. The prompts used and formalized regulatory knowledge artifacts in this section are provided in Appendices A.1 and A.2, respectively.

Knowledge-Injected Prompt Generation. In *Step* ④, we design two specialized prompts that dynamically inject the explicated knowledge artifacts into the generation process, as shown in Figure 4:

(1) **Formal Requirement Generation Prompt ($P_{\mathcal{R}}$).** This prompt embeds the domain symbol library and the requirements representation syntax from $\mathcal{R}$ into a zero-shot template, constraining the LLM to generate formal *IF–THEN* requirements. By enforcing these grammatical constraints, all generated symbols and relations are strictly grounded in $\mathcal{R}$.

(2) **Testability Determination & Refinement Prompt ($P_{\mathcal{T}}$).** This prompt incorporates constraints from $\mathcal{T}$ as fine-grained evaluation criteria. Each testability constraint is assessed independently across all requirements, enabling the detection of subtle logical gaps that holistic judgments may miss due to long context.

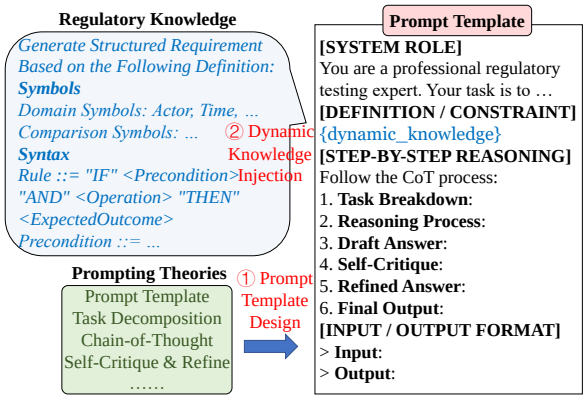

Figure 4: Dynamic knowledge injection-based prompt generation for Formal Requirement Generation.

Formal Requirement Generation and Testability Determination & Refinement. In *Step* ⑤, $P_{\mathcal{R}}$ is used to guide an LLM to generate formal requirements from regulatory documents within a *Verification-Feedback-Refinement* loop. The BNF syntax of $\mathcal{R}$ is first converted by an LLM into Xtext-compliant grammar, which Xtext then uses to automatically construct a checker. All generated requirements are iteratively validated against this checker and corrected by the LLM.

In *Step* ⑥, the generated formal requirements are assessed against all testability constraints by instructing an LLM with $P_{\mathcal{T}}$. Violations trigger targeted refinement attempts by the LLM, including completing missing elements, concretizing abstract expressions, and so on. Requirements that remain logically inconsistent, unrefinable, or purely descriptive are excluded for expert review.

We evaluated this process on 19 financial regulatory documents using GPT-5, Grok-4, and DeepSeek-R1, with expert feedback establishing ground truth. The performance of our approach was high, achieving token-wise and word-wise Precision/Recall/F1 of 91.6%/87.7%/89.0% and 89.8%/85.3%/86.8%, respectively. Testability assessment showed that 77.6% of requirements are automatable, with fewer than 5% requiring manual intervention, indicating that it can effectively identify testable rules and minimize human effort.

Scenario and Test Case Generation. In *Step* ⑦, we integrate testable formal requirements into a hybrid test generation framework, LLM4Fin (Xue et al., 2024b), which is most compatible with our requirements representation, to perform the final Scenario and Test Case Generation.

Test scenarios are built by identifying inter-rule

dependencies, mapping behavioral elements (*Action*, *Status*) to activity diagrams and state machines to form end-to-end business flows. For each scenario, concrete test cases are generated using strategies such as Equivalence Partitioning and Boundary Value Analysis (Graham et al., 2006). Numerical inputs produce positive, negative, and boundary values, while finite non-numerical inputs are exhaustively enumerated, yielding a compact yet comprehensive set of test cases.

4 Experimental Evaluation

To evaluate the performance of RAFT, we conducted a comprehensive experimental study to address the following research questions:

RQ1: How does our approach perform compared with SOTA baselines in terms of the effectiveness and efficiency of generated test cases?

RQ2: What is the contribution of the proposed Adaptive Purification-Aggregation strategy and each explicated regulatory knowledge artifact to the quality of generated test cases?

RQ3: How well does RAFT generalize across different regulated domains?

4.1 Experiment I: Effectiveness and Efficiency of Test Cases Generation

This experiment evaluates the overall effectiveness and efficiency of RAFT in the financial domain.

Competitors. We compare RAFT against three representative categories of baselines: (1) *Domain Experts*, consisting of three senior testing engineers with 3-5 years of financial experience; (2) *LLM4Fin* (Xue et al., 2024b), a SOTA hybrid regulatory test generation framework; and (3) *End-to-End (E2E) LLMs*, represented by GPT-5, Grok-4, and DeepSeek-R1, which directly generate test cases from regulations using prompt engineering.

Datasets. Due to the limited availability of public datasets with sufficient number and coverage, we constructed six financial datasets based on official regulatory documents from financial exchanges, paired with anonymized real-world test cases provided by industry partners. The datasets span multiple regulatory frameworks and languages. An overview is given in Table 1. Notably, *Datasets 4* and *5* follow the evaluation setup of LLM4Fin, but use complete regulatory documents instead of 10–20 manually selected rules, resulting in a more realistic and challenging evaluation setting.

Metrics. Due to strict security constraints in financial institutions, executable system code is unavailable. Following prior work (Tufano et al., 2020; Fatima et al., 2022), we use anonymized real-world test cases adopted in practice as ground truth. Effectiveness is evaluated using *Precision*, *Recall*, and *F1* with the adopted test cases, as well as *Business Scenario Coverage (BSC)* (Xue et al., 2024b) to assess regulatory requirement coverage. Efficiency is measured by *Time Consumption* and *Token Usage*. We report the time required for regulatory analysis (if applicable), test generation, and expert review to achieve 100% correctness. For LLM-based methods, we additionally report prompt and completion token usage and the corresponding monetary cost.

Table 1: Details of the six financial evaluation datasets for Exp. I. #X means the number of X, Req. means testable requirement, and TC means test case.

Dataset	Document	# Rule	# Req.	# TC
1	New York Stock Exchange Auction Market Bids and Offers Rules (Exchange, 2025c)	142	114	927
2	The Nasdaq Stock Market Options Trading Rules (Exchange, 2025b)	289	276	1465
3	Hong Kong Stock Exchange Trading Rules (Exchange, 2025a)	129	292	2128
4	Shanghai Stock Exchange Trading Rules (Exchange, 2023)	212	195	682
5	Shenzhen Stock Exchange Bond Trading Rules (Exchange, 2022)	190	167	1389
6	Tokyo Stock Exchange Buying and Selling Rules (Exchange, 2025e)	104	198	1020

Experimental Procedure. Each of the six datasets was provided to the three domain experts, who manually authored compliance test cases. For E2E LLMs, prompts were designed following SOTA prompt engineering practices (see Appendix A.3). LLM4Fin was executed fully automatically using its default configuration. For RAFT, test cases were generated following the workflow described in Section 3. Reported results for domain experts are averaged across the three individuals, while results for E2E LLMs and RAFT are averaged over multiple runs using GPT-5, Grok-4, and DeepSeek-R1.

Experimental Results. The effectiveness results are summarized in Table 2. Domain experts achieve the highest performance, while RAFT attains comparable results and substantially outperforms all automated baselines. Across the six datasets, RAFT balances precision and recall, achieving an average F1 of 91.7% and BSC of 86.5%, improving over LLM4Fin by 34.7% and 18.2%, respectively. LLM4Fin underperforms due to its reliance on smaller BERT-based models with limited domain knowledge, which hinders its ability to capture complex and implicit regulatory logic. In contrast,

Table 2: Comparison of Precision (%), Recall (%), F1 (%), and Business Scenario Coverage (BSC, %) of test cases generated by Experts, LLM4Fin, End-to-end LLMs, and RAFT on the six evaluation datasets.

Datasets	Experts				LLM4Fin				End-to-end LLMs				RAFT			
	Pre.	Rec.	F1	BSC	Pre.	Rec.	F1	BSC	Pre.	Rec.	F1	BSC	Pre.	Rec.	F1	BSC
1	83.7	**96.3**	89.6	90.0	80.3	70.2	74.9	78.1	7.1	41.0	11.8	32.2	**95.0**	95.8	**95.4**	**90.2**
2	**90.8**	**91.5**	**91.1**	85.3	52.2	72.8	60.8	76.6	8.2	21.1	11.7	29.8	85.0	84.4	84.7	**88.4**
3	95.1	**98.9**	**96.9**	**92.3**	74.3	77.0	75.6	79.4	11.9	33.6	16.8	42.2	**96.9**	93.4	95.0	91.1
4	**93.5**	93.7	**93.6**	81.1	59.9	82.1	69.3	72.8	27.0	46.7	33.6	40.0	79.1	**98.7**	87.8	**83.1**
5	**97.3**	96.0	**96.6**	89.1	61.2	87.7	72.1	83.7	25.4	49.8	33.2	38.0	95.4	**97.4**	96.4	**89.5**
6	**94.3**	**97.1**	**95.7**	**86.1**	83.7	41.8	55.7	48.6	19.1	36.5	24.6	31.0	92.1	89.7	90.9	76.9
Average	92.4	95.6	93.9	87.3	68.6	71.9	68.1	73.2	16.4	38.1	21.9	35.5	90.6	93.2	91.7	86.5
Variance	18.9	5.6	7.7	13.2	132.8	215.2	54.6	131.7	62.2	88.3	83.8	22.4	41.4	24.0	18.6	25.0

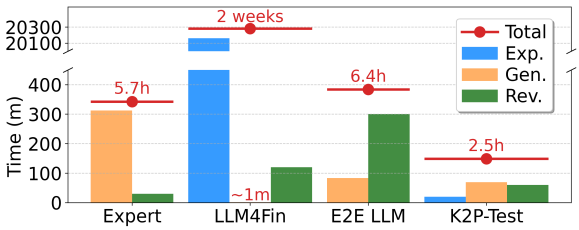

Figure 5: Comparison of the time consumption on knowledge explication (Exp.), test generation (Gen.), and expert review (Rev.) on processing the six datasets.

E2E LLMs suffer from hallucinations, reasoning errors, and output instability, resulting in lower accuracy and coverage. Although RAFT exhibits slightly higher variance than human experts due to its reliance on probabilistic LLM outputs, its variance is markedly lower than that of E2E LLMs and LLM4Fin, demonstrating that the proposed multi-LLM Adaptive Purification-Aggregation strategy effectively mitigates hallucination and randomness.

Efficiency results, shown in Figure 5 and Table 3, further highlight the practical advantages of RAFT. Test generation by domain experts and LLM4Fin requires substantial manual effort, ranging from 5.7 hours to 2 weeks. In contrast, RAFT significantly reduces end-to-end time, achieving a 2.3× speedup over domain experts and more than a 134× speedup over LLM4Fin. Even compared to E2E LLMs, RAFT provides a 156% speedup, primarily because its high-accuracy test cases reduce expert review time from 2–4 hours to approximately 1 hour. From a cost perspective, RAFT is also more economical, as it emphasizes prompt tokens over expensive completion tokens. Although LLM4Fin minimizes token usage through local deployment, this advantage comes at the expense of reduced test quality due to limited model capacity.

Conclusion: RAFT achieves expert-level effectiveness, significantly improving efficiency and reducing cost, demonstrating its practicality for real-world regulatory compliance testing.

Table 3: Comparison of the average token usage and monetary cost for each LLM-based approach.

Method	Prompt Tokens	Completion Tokens	Cost ($)
LLM4Fin	227.3K	114.8K	0
E2E LLM	115.8K	1328.6K	12.34
RAFT	1047.0K	128.3K	2.84

4.2 Experiment II: Ablation Study

Experimental Design. We evaluate the necessity of explicated regulatory knowledge by comparing the full RAFT, including meta-model $\mathcal{M}$, requirements representation $\mathcal{R}$, and testability constraints $\mathcal{T}$, against three degraded variants: (1) w/o $\mathcal{T}$, which generates test cases without filtering non-testable requirements; (2) w/o $\mathcal{M}$ & $\mathcal{R}$, which generates test cases directly from rules after testability assessment; and (3) w/o all: which relies on E2E LLMs without explicit regulatory knowledge.

To assess the proposed Adaptive Purification-Aggregation strategy, we further compare the full RAFT using GPT-5, Grok-4, and DeepSeek-R1 with variants that perform knowledge explication using them separately. All datasets, metrics, and procedures follow Experiment I.

Experimental Result. As shown in Figure 6, removing $\mathcal{T}$ causes a notable precision drop of 33.6% due to noise introduced by non-testable requirements. Removing both $\mathcal{M}$ and $\mathcal{R}$ leads to a more severe degradation, with F1 decreasing by 66.5%, indicating that LLMs struggle to capture regulatory semantics and implicit logic without explicit guidance. The poorest performance of the no-knowledge variant confirms that integrating $\mathcal{M}$, $\mathcal{R}$, and $\mathcal{T}$ is essential for high-quality test generation.

Figure 7 shows that the Adaptive Purification-Aggregation strategy consistently outperforms the single-LLM variants. While individual LLMs are prone to hallucination, resulting in incomplete knowledge explication and degraded test quality, the consensus-based strategy aggregates complementary insights to produce more precise and robust regulatory knowledge, thereby improving the

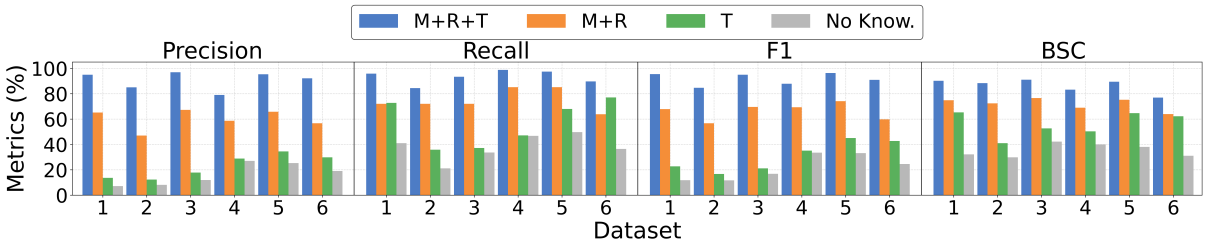

Figure 6: Impact of regulatory knowledge components ($\mathcal{M}, \mathcal{R}, \mathcal{T}$) on the quality of generated test cases.

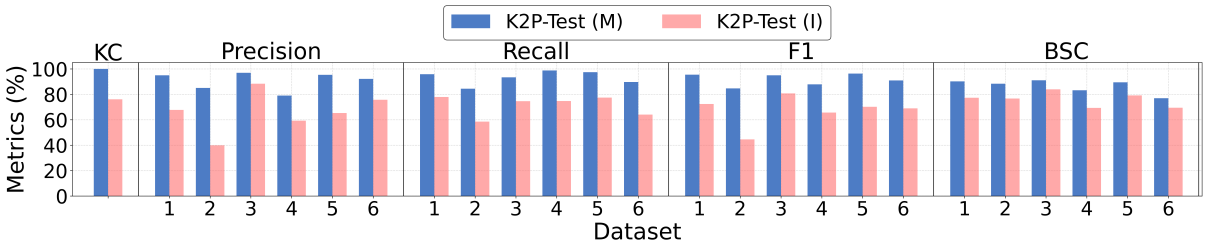

Figure 7: Comparison between our Adaptive Purification-Aggregation strategy (RAFT (M)) and individual LLMs (RAFT (I)) on knowledge coverage (KC) and metrics of generated test cases.

Table 4: Details of the five evaluation datasets on automotive and power domains for Exp. III.

Dataset	Domain	Document	# Rule	# Req.	# TC
7	Automotive	2025 New Jersey Driver Manual (Commission, 2025)	47	427	1570
8		Road Traffic Safety Law of PRC (China, 2021)	171	613	2124
9		Road Traffic Law of Japanese (Japan, 2018)	93	808	3132
10	Power	EN 50549-1:2019 (Standardization, 2019)	75	362	703
11		GB/T 19964-2024 (China, 2024)	189	195	1311

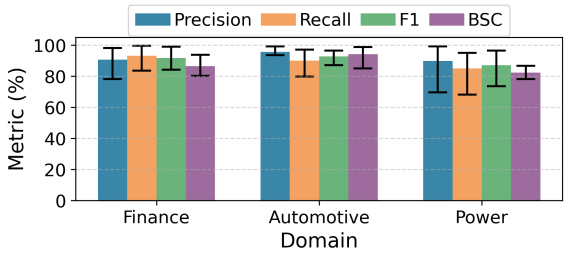

Figure 8: Average performance of test cases generated by our approach across three industrial domains.

quality of generated test cases.

Conclusion: Explicit regulatory knowledge ($\mathcal{M}$, $\mathcal{R}$, $\mathcal{T}$) is essential for effective compliance test generation, and the proposed Adaptive Purification-Aggregation strategy further enhances it through a more comprehensive knowledge explication.

4.3 Experiment III: Method Generalizability

Experimental Design. We evaluate the generalizability of RAFT in the automotive and power domains. Five datasets were constructed, covering three traffic regulations and two power grid standards, each paired with real-world industrial test cases, as summarized in Table 4. Together with the financial datasets from Experiment I, this evaluation spans three regulated domains.

Experimental Results. The results are shown in Figure 8. RAFT demonstrates consistently strong performance across all 11 datasets. In the automotive and power domains, it achieves average F1/BSC scores of 92.7%/87.2% and 94.3%/82.3%, respectively, comparable to those in the financial domain. Its performance remains stable across domains, with variance below 20 in most cases. These results indicate that the proposed regulatory knowledge explication and test generation approach generalizes well across diverse regulatory contexts.

Conclusion: Beyond the financial domain, RAFT maintains high performance in both automotive and power domains, demonstrating its strong cross-domain generalizability.

5 Conclusion

In this paper, we propose RAFT, a framework that automatically explicates tacit regulatory knowledge from LLMs into structured artifacts to drive high-quality test case generation. Experimental results across financial, automotive, and power domains demonstrate that RAFT achieves expert-level effectiveness, high efficiency, and strong cross-domain generalizability. By shifting from manual modeling to knowledge-driven automation, our approach provides a scalable and reliable solution for compliance testing in highly regulated domains.

Limitations

Dependence on the Quality and Coverage of External Knowledge Sources. Although the proposed framework supplies knowledge for LLMs by incorporating regulatory documents and historical test cases through RAG, its performance still depends on the quality, completeness, and representativeness of these external knowledge sources. In domains where regulations are ambiguous, rapidly evolving, or insufficiently documented, the extracted domain models and requirements representations may be incomplete or biased, thereby affecting downstream test generation. Addressing this limitation requires continuous maintenance and supplementation of the knowledge base, and mechanisms for detecting and resolving gaps or inconsistencies in regulatory knowledge.

Limited Formal Guarantees on Model Soundness and Completeness. While the collective wisdom strategy and model-based generation improve controllability compared to end-to-end LLM approaches, the automatically constructed metamodels and requirements representations lack formal guarantees of soundness or completeness with respect to the original regulations. Subtle semantic nuances, implicit assumptions, or exceptional regulatory clauses may still be partially or inaccurately captured. As a result, human expert review remains necessary, especially for safety-critical or legally sensitive scenarios. Future work could explore integrating formal verification techniques or regulation-specific logical formalisms to further strengthen the theoretical guarantees of the generated models and test cases.

References

John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, and 1 others. 1960. Report on the algorithmic language algol 60. *Communications of the ACM*, 3(5):299–311.

Babak Badrzadeh and Andrew Halley. 2014. Challenges associated with assessment and testing of fault ride-through compliance of variable power generation in australian national electricity market. *IEEE Transactions on Sustainable Energy*, 6(3):1160–1168.

Mohamed Boukhlif, Nassim Kharmoum, Mohamed Hanine, Mohcine Kodad, and Souad Najoua Lagmiri. 2024. Towards an intelligent test case generation framework using llms and prompt engineering. In *International Conference on Smart Medical, IoT & Artificial Intelligence*, pages 24–31. Springer.

Julieth Patricia Castellanos Ardila, Barbara Gallina, and Faiz Ul Muram. 2022. Compliance checking of software processes: A systematic literature review. *Journal of Software: Evolution and Process*, 34(5):e2440.

Nanjiang Chen, Xuhui Lin, Hai Jiang, and Yi An. 2024. Automated building information modeling compliance check through a large language model combined with deep learning and ontology. *Buildings*, 14(7):1983.

National Standardization Management Committee of China. 2024. Gb/t 19964-2024 technical requirements for connecting photovoltaic power station to power system. https://openstd.samr.gov.cn/bzgk/std/newGbInfo?hcno=40D8691DFD7EC3CBA423CCBA65D262F3.

Standing Committee of the National People's Congress of China. 2021. Road traffic safety law of the people's republic of china. https://flk.npc.gov.cn/detail?id=ff8081817ab231eb017abd617ef70519.

Noam Chomsky. 2002. *Syntactic structures*. Walter de Gruyter.

New Jersey Motor Vehicle Commission. 2025. 2025 new jersey driver manual. https://www.nj.gov/mvc/pdf/license/drivermanual.pdf.

Eric Evans. 2004. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

Hong Kong Stock Exchange. 2025a. Hong kong stock exchange trading rules. https://www.hkex.com.hk/-/media/HKEX-Market/Services/Rules-and-Forms-and-Fees/Rules/SEHK/Whole_SEHK_c.pdf.

Nasdaq Stock Exchange. 2025b. The nasdaq stock market options trading rules. https://listingcenter.nasdaq.com/RuleBook/Nasdaq/rules/Nasdaq%20Options%203.

New York Stock Exchange. 2025c. New york stock exchange auction market bids and offers rules. https://nyseguide.srorules.com/rules/09013e2c8553e79b.

New York Stock Exchange. 2025d. Nyse rules. https://nyseguide.srorules.com/rules.

Shanghai Stock Exchange. 2023. Shanghai stock exchange trading rules. https://www.sse.com.cn/lawandrules/sselawsrules/stocks/exchange/c/10118395/files/ae39775817cd4a7183159cd0f1547e26.docx.

Shenzhen Stock Exchange. 2022. Shenzhen stock exchange bond trading rules. https://docs.static.szse.cn/www/lawrules/rule/bond/bonds/trade/W020220127631859244722.pdf.

Tokyo Stock Exchange. 2025e. Tokyo stock exchange buying and selling rules. https://resource.lexis-asone.jp/jpx/rule/tosho_regu_201305070003001.html.

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):1912–1927.

Dorothy Graham, Erik van Veenendaal, Isabel Evans, and Rex Black. 2006. *Foundations of software testing: ISTQB certification*. Cengage Learning.

Object Management Group. 2014. Object constraint language (ocl), version 2.4. http://www.omg.org/spec/OCL/2.4.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Navid Bin Hasan, Md Ashraful Islam, Junaed Younus Khan, Sanjida Senjik, and Anindya Iqbal. 2025. Automatic high-level test case generation using large language models. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pages 674–685. IEEE.

Jivitesh Jain, Nivedhitha Dhanasekaran, and Mona Diab. 2025. From complexity to clarity: Ai/nlp's role in regulatory compliance. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 26629–26641.

Ministry of Justice of Japan. 2018. Japanese road traffic law. https://www.moj.go.jp/content/001226848.pdf.

Brahma Reddy Korraprolu, Pavitra Pinninti, and Y Raghu Reddy. 2025. Test case generation for requirements in natural language-an llm comparison study. In *Proceedings of the 18th Innovations in Software Engineering Conference*, pages 1–5.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, and 1 others. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474.

Kaibo Liu, Zhenpeng Chen, Yiyang Liu, Jie M Zhang, Mark Harman, Yudong Han, Yun Ma, Yihong Dong, Ge Li, and Gang Huang. 2025. Llm-powered test case generation for detecting bugs in plausible programs. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 430–440.

Satoshi Masuda, Satoshi Kouzawa, Kyousuke Sezai, Hidetoshi Suhara, Yasuaki Hiruta, and Kunihiro Kudou. 2025. Generating high-level test cases from requirements using llm: An industry study. *arXiv preprint arXiv:2510.03641*.

Dragan Milchevski, Gordon Frank, Anna Hätty, Bingqing Wang, Xiaowei Zhou, and Zhe Feng. 2025. Multi-step generation of test specifications using large language models for system-level requirements. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, pages 132–146.

OpenAI. 2025. Chatgpt. https://chatgpt.com/.

Abhishek Shrestha, Bernd-Holger Schlingloff, and Jürgen Großmann. 2025. Less is more: Guiding llms for formal requirement and test case generation. In *International Conference on Cybersecurity and Artificial Intelligence Strategies*, pages 366–383. Springer.

Mithila Sivakumar, Alvine B Belle, Kimya Khakzad Shahandashti, Oluwafemi Odu, Hadi Hemmati, Segla Kpodjedo, Song Wang, and Opeyemi O Adesina. 2024. I came, i saw, i certified: some perspectives on the safety assurance of cyber-physical systems. *arXiv preprint arXiv:2401.16633*.

European Electrotechnical Committee for Standardization. 2019. En 50549-1:2019 requirements for generating plants to be connected in parallel with distribution networks - part 1: Connection to a lv distribution network - generating plants up to and including type b. https://standards.iteh.ai/catalog/standards/clc/948bd277-da91-40c7-b7e0-bbf561035638/en-50549-1-2019.

Thomas Stefani, Johann Maximilian Christensen, Akshay Anilkumar Girija, Siddhartha Gupta, Umut Durak, Frank Köster, Thomas Krüger, and Sven Hallerbach. 2025. Automated scenario generation from operational design domain model for testing ai-based systems in aviation. *CEAS Aeronautical Journal*, 16(1):197–212.

Jingyun Sun, Zhongze Luo, and Yang Li. 2025. A compliance checking framework based on retrieval augmented generation. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 2603–2615.

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*, pages 1–6.

PlantUML Team. 2025. Plantuml. https://plantuml.com.

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*.

Chunhui Wang, Fabrizio Pastore, Arda Goknil, and Lionel C Briand. 2020. Automatic generation of acceptance test cases from use case specifications: an nlp-based approach. *IEEE Transactions on Software Engineering*, 48(2):585–616.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2025. Testeval: Benchmarking large language models for test case generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3547–3562.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Wikipedia contributors. 2014. General motors ignition switch recalls. https://en.wikipedia.org/wiki/General_Motors_ignition_switch_recalls.

xAI. 2025. Grok. https://grok.com.

Zhiyi Xue, Liangguo Li, Senyue Tian, Xiaohong Chen, Pingping Li, Liangyu Chen, Tingting Jiang, and Min Zhang. 2024a. Domain knowledge is all you need: A field deployment of llm-powered test case generation in fintech domain. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 314–315.

Zhiyi Xue, Liangguo Li, Senyue Tian, Xiaohong Chen, Pingping Li, Liangyu Chen, Tingting Jiang, and Min Zhang. 2024b. 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*, pages 1643–1655.

Zhenzhen Yang, Chenhui Cui, Tao Li, Rubing Huang, Nan Niu, Dave Towey, and Guo Shikai. 2025. Llmcfg-tgen: Using llm-generated control flow graphs to automatically create test cases from use cases. *arXiv preprint arXiv:2512.06401*.

Quanjun Zhang, Ye Shang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. 2024. Testbench: Evaluating class-level test case generation capability of large language models. *arXiv preprint arXiv:2409.17561*.

Shunran Zhang, Zhengmin Jiang, Ming Sang, Yunjie Han, Huiyun Li, and 1 others. 2025. A comprehensive survey on driving compliance assessment methodologies for autonomous vehicles. *IEEE Intelligent Transportation Systems Magazine*.

Denesa Zyberaj, Lukasz Mazur, Nenad Petrovic, Pankhuri Verma, Pascal Hirmer, Dirk Slama, Xiangwei Cheng, and Alois Knoll. 2025. Genai-based test case generation and execution in sdv platform. *arXiv preprint arXiv:2509.05112*.

A Appendix

A.1 Prompt Design in Our Framework

A.1.1 Meta-model Construction

The prompt design theories and techniques used for constructing financial meta-models from regulatory texts and test cases have been detailed in Section 3.2. It integrates role-based instructions, structural constraints, and staged outputs to support the abstraction of domain knowledge into a formal and test-oriented representation. External financial regulations and test cases are provided as supplementary knowledge to enhance semantic completeness.

The specific prompts applied are shown below. The proposed prompt design has been empirically validated across multiple domains, including finance, automotive, and energy, and evaluated on diverse LLMs such as GPT, Grok, and DeepSeek, demonstrating its domain-agnostic nature and independence from specific model implementations.

```
You are a software requirements modeling and testing
expert. I will provide two types of inputs:
1. A domain rule document;
2. Corresponding partial test cases.

Your task is to construct a **domain meta-model** for
the rules of the domain. The meta-model should
describe the syntax of rules in a class diagram-style
tree structure. The construction of the meta-model
must be completed in two steps: first, extract and
confirm rule elements (15-20 leaf elements); second,
based on these elements, build a three-layer
meta-model and establish relationships between these
elements and Precondition, Operation, and
ExpectedResult.
Please strictly follow the following specifications
and output format: strict, complete, and
machine-readable:

### Meta-Model Construction Specifications (Must be
followed)
1. **Overall Structure**
   * The meta-model should be a tree structure in
class diagram style (classes and relationships only,
without attributes or methods).
   * Must adopt a three-layer structure: top layer (1
node) -> middle layer (fixed 3 nodes) -> bottom layer
(n leaf nodes).
   * The top-layer node is fixed as `Rule` (single
class).
   * The middle layer is fixed as three classes: `
Precondition`, `Operation`, `ExpectedResult`.
   * The bottom layer consists of minimal rule
elements (leaf nodes).

2. **Leaf Node (Rule Element) Requirements**
   * Must be testable, quantifiable, and concrete rule
 elements.
   * The total number of elements must be around 15.
High-frequency/core elements are listed individually;
semantically overlapping elements are merged;
scattered low-frequency elements are merged into a
general leaf `Others`. Note: Precondition, Operation,
and ExpectedResult should all include `Others`.
   * Leaf nodes should be attributes (keys), and must
not include specific values or examples (e.g., "
interest rate" is correct, "annual interest rate 5%"
is incorrect).
```

```
   * Preferably extracted from test cases and
supplemented by rule documents. Elements presented in
test cases as ``RuleElement1/RuleElement2'' should be
merged as a single element.
   * Elements should be reusable within the domain and
 cover common elements across different domain rule
documents.

3. **Relationships**
   * Relationship types and applicable scenarios:
     * `contains` - composition relationship of
middle-layer nodes to bottom-layer elements
     * `constrains` - constraint relationship of
elements to nodes/other elements
     * `dependsOn` - prerequisite relationship of
elements at runtime
     * `triggers` - operation triggering expected
result
     * Other relationships
   * Every relationship must be labeled with its name
in PlantUML.
   * Bottom-layer elements can only establish
relationships with one of the three middle-layer nodes
, and the relationship type must match the element's
semantics. Carefully consider whether each rule
element belongs to Precondition, Operation, or
ExpectedResult.

4. **Two-Step Output Requirements**
   * **Step 1 - Rule Element List**: Output a list of
approximately 15 leaf classes (class names in
UpperCamelCase), each with a semantic description
within 10 words.
   * **Step 2 - Meta-Model**: Construct a complete
three-layer class diagram in PlantUML:
     * Top layer `Rule`;
     * Middle layer `Precondition`, `Operation`, `
ExpectedResult`;
     * Bottom layer directly referencing the leaf
classes from Step 1, without renaming.
   * In Step 2, clearly indicate the direction and
type of all relationships, ensuring a tree hierarchy
and logical consistency.

5. **Naming and Style**
   * All class names use UpperCamelCase (e.g., `
AccountStatus` rather than `account_status`).
   * Class bodies remain empty (class name only).
   * Relationship names use lowercase verbs (e.g., `
contains`, `triggers`).

### Output (Mandatory)
* The rule element list should be output as a numbered
 list; the meta-model should be output as PlantUML
code.
* The number of leaf nodes must be strictly around 15;
 the leaf class names in Step 1 and Step 2 must match
exactly (including capitalization).
* If you are unsure of which elements to include,
refer to the keys in the test cases. Elements must be
concise, representative, and high-frequency. For
example, only one Actor should exist as a primary
entity; having X Actor, Y Actor, etc., would be too
repetitive.
```

A.1.2 Requirements Representation Formalization

Requirement Representation Formalization defines a formal, machine-readable requirement representation that captures the essential semantics of domain rules while enabling consistent parsing and downstream analysis. Given natural-language rules, example test cases, and a domain meta-model, this step formalizes informal requirements into a formal representation that bridges free-text specifications and executable artifacts, without yet enforcing testability constraints.

The formalized requirement representation consists of two core components: Symbols and Syntax. Symbols define the vocabulary and semantic roles of elements used in requirement expressions, while Syntax specifies the hierarchical structure and composition rules for combining these elements into well-formed requirements. This representation is domain-agnostic and independent of any specific modeling language or testing framework, enabling reuse across different regulatory domains and LLM backends.

```
You are a software requirements modeling expert.

Your task is to define a formal requirement
representation language for describing domain rules in
 a structured and machine-readable form.

This representation aims to formalize natural language
 rules or requirement documents so that they can be
consistently parsed, analyzed, and used as a basis for
 further validation and testing.

### Input
You will receive the following three types of input:
1. **Rule/Requirement Document** - natural language
descriptions of domain rules or system behavior;
2. **Test Cases** - example test cases that illustrate
 the intended behavior of some rules;
3. **Meta-Model Syntax Structure** - defining the core
 elements of rules and their hierarchical
relationships.

### Task Objective
Based on the above inputs, define a complete formal
Requirement Representation.
The language definition includes two parts:
**Symbols** and **Syntax**.

### **Symbol System**
* Based on the given meta-model, define the types of
symbols used in the language and their scope,
including:
  * Logical Symbols
  * Comparison Symbols
  * Domain Symbols
* Clearly specify the syntactic position and semantic
role of each type of symbol.

### **Syntax**
* Based on the given meta-model, define the core
components of the language and their hierarchical
composition rules;
* The value of each element should be a string or
number, not an enum;
* Use formal BNF to define the language structure;
* The syntax should be simple and explicit, while
supporting complex rule expressions (e.g., AND/OR/NOT,
 nested conditions, composite actions);
* Condition, Action, and ExpectedOutcome should follow
 consistent structural patterns.

### **Output Requirements**
Please output:
1. The formal definition of Symbols and Syntax;
2. A complete definition of the requirement
representation;
3. **An example** showing how a natural language rule
is transformed into an instance of this representation.


### Meta-Model (PlantUML Representation)
{}
```

A.1.3 Testability Constraints Specification

Testability Constraint Specification defines a set of explicit constraints over the formalized requirement representation to determine whether a requirement is testable. This step focuses on identifying structural, semantic, and data-related conditions that must be satisfied for a requirement to support automated test generation and verification.

Testability constraints are specified independently of requirement syntax and are applied as a validation layer over the formal requirement representation. These constraints capture necessary conditions such as observability, determinism, input completeness, and verifiable outcomes, and are formalized using OCL to enable automated checking.

```
You are a software testing and requirements validation
 expert.

Your task is to define a set of formal testability
constraints for a given formal requirement
representation.

These constraints are intended to determine whether a
requirement instance is testable, i.e., whether it can
 be unambiguously verified and used for automated test
 generation.

### Input
You will receive:
1. **Formal Requirement Representation Definition** -
including Symbols and Syntax.

### Task Objective
Based on the above inputs, define a comprehensive set
of **Testability Constraints**.

### **Testability Constraints**
* Identify the necessary and sufficient conditions for
 a requirement to be testable, considering:
  * Completeness of conditions and actions;
  * Observability and verifiability of expected
outcomes;
  * Absence of ambiguity or underspecification;
  * Compatibility with available test inputs and
outputs.
* Formalize these constraints using **OCL** over the
requirement representation model.

### **Output Requirements**
Please output:
1. A complete set of testability constraints expressed
 in OCL;
2. A clear explanation of each constraint and its
rationale;
3. **An example** demonstrating how a formalized
requirement is checked against these constraints and
classified as testable or non-testable.


### Formal Requirements Representation
{}
```

A.1.4 Formal Requirement Generation

This section presents formal requirement generation, which transforms natural language rules into formal, testable requirements based on the testable requirements representation. The task is guided by prompts specifying the expert role, task objective, inputs, output format, and language constraints. While earlier methods required model training (Xue et al., 2024b), modern large language models can achieve strong results using prompt engineering with few-shot examples alone. An example prompt in the financial domain is provided below.

```
You are a software requirements modeling and testing
expert, proficient in requirements engineering, formal
 modeling, and test case generation. You are familiar
with testable requirements representation, which can
accurately represent rules described in natural
language as formal requirements that are executable,
verifiable, and testable.

### Task Description
I will provide one or more rules described in natural
language, and your task is to convert them into the
Testable Requirement Language.

### Input
1. Definition of the Testable Requirement Language (
provided below).
2. Natural language rules to be converted, provided
one at a time in multiple interactions.

### Output
Formal Requirements

### Generation Requirements
1. The output testable requirements must conform to
the elements and syntax of the representation, and
must not exceed the defined scope.
2. If a rule can be interpreted in multiple ways,
provide the best expression.
3. TRL must accurately express the meaning of the rule
 and fully include all elements of the rule; nothing
should be omitted.
4. Only output the testable requirement; do not output
 extra content, and do not evaluate testability at
this step (this will be handled by other tools later).

### Testable Requirement Language Definition
#### Symbol Definition
{}

#### Syntax Definition
{}

#### Other Requirements
1. Generally, strictly follow the defined symbols and
syntax when writing testable requirements.
2. Except for cases requiring modulo calculations, all
 expressions must follow the key-op-value triple
format; binary, quadruple, or quintuple formats are
prohibited.
3. Must accurately express the rule meaning and fully
include all elements (subject, verb, object, events,
and domain-specific elements); redundancy is allowed,
omission is not.
4. Do not overuse parentheses; use only when necessary
 (e.g., with or, not).
5. Remain faithful to the original text; only words
present in the original text may appear. No invented
words are allowed.

### Examples
* Example 1
  * Input: Unless otherwise specified by the exchange,
 the bond trading submission quantity shall comply
with the following requirements: (1) for matching
execution, the bond cash submission quantity shall be
100,000 yuan or multiples thereof; for sales of less
than 100,000 yuan, submit in a single order; for
general pledged repo of bonds, the submission quantity
 shall be 1,000 yuan or multiples thereof; (2) for
```

```
click execution, the submission quantity shall be
100,000 yuan or multiples thereof; (3) for inquiry or
competitive purchase execution, the submission
quantity shall be no less than 100,000 yuan and in
multiples of 1,000 yuan; (4) for negotiated execution,
 the bond cash submission quantity shall be no less
than 1,000 yuan and in multiples of 100 yuan; general
pledged repo submission quantity shall be 1,000 yuan
or multiples thereof.
  * Output:
```
rule 1
if TradingInstrument = Bond and TradingMethod =
MatchingExecution and BondType = CashBond and Quantity
 % 100000 == 0
then Result = Success
rule 2
if TradingInstrument = Bond and TradingMethod =
MatchingExecution and BondType = CashBond and Action =
 Sell and Balance < 100000 and Action =
SingleSubmission
then Result = Success
rule 3
if TradingInstrument = Bond and TradingMethod =
MatchingExecution and BondType = GeneralPledgedRepo
and Quantity % 1000 == 0
then Result = Success
rule 4
if TradingInstrument = Bond and TradingMethod =
ClickExecution and Quantity % 100000 == 0
then Result = Success
rule 5
if TradingInstrument = Bond and (TradingMethod =
InquiryExecution or TradingMethod =
CompetitivePurchase) and Quantity >= 100000 and
Quantity % 1000 == 0
then Result = Success
rule 6
if TradingInstrument = Bond and TradingMethod =
NegotiatedExecution and BondType = CashBond and
Quantity >= 1000 and Quantity % 100 == 0
then Result = Success
rule 7
if TradingInstrument = Bond and TradingMethod =
NegotiatedExecution and BondType = GeneralPledgedRepo
and Quantity % 1000 == 0
then Result = Success
```

* Example 2
  * Input: If both parties agree to a manual method,
and the client wishes to continue participating in the
 next period on the repo maturity date, they shall
re-issue the initial order to the securities company.
  * Output:
```
rule 1
if Actor = BothParties and Action = Agree and
Constraint = ManualMethod and Actor = Client and Time
= RepoMaturityDate and Action = WishToContinue and
OperationPart = NextPeriodTrade and OperationTarget =
SecuritiesCompany and Action = Issue and OperationPart
 = InitialOrder
then Result = Success
```

### A.1.5 Testability Determination

This section covers testability determination, which evaluates whether a requirement satisfies specified testability constraints. Using few-shot prompts, the model receives a natural language rule, its formal requirement, and a single constraint per evaluation. The process checks constraints iteratively: if the constraint is satisfied, it proceeds to the next; if not, it stops. This leverages large language models' reasoning without additional training.

```
You are a software requirements modeling and testing
expert, specialized in evaluating whether a natural
language rule and its formalized requirement
constitute a testable requirement.
Now, you need to determine whether a given rule is a
Testable Requirement.
The rule inputs include:
1. Natural Language Rule
2. Formalized Rule
You need to assess whether the rule meets both
requirement validity and testability criteria, and
output:
* `True` (testable requirement) or `False` (
non-testable or non-requirement)
* If `False`, provide a brief explanation.

Input Format
For example, a rule in natural language:
The trading system shall complete matching within 5
seconds after receiving a valid order.
Its corresponding formalized expression:
```
rule 1
if Actor = TradingSystem and Event = ReceivedOrder and
 Time = within5s and Action = CompleteMatching
then Result = Success
```


Evaluation Criteria
{}

Output Format
Output `true` or `false` to indicate whether the rule
is a testable requirement.
If the rule is not testable, provide a one-sentence
explanation.

Reasoning Strategy (Thinking Sequence)
1. First, determine if it is a requirement (exclude
background, definitions, etc.).
2. Then, check each of the five testability
constraints one by one.
3. If all are satisfied, output `True`; otherwise,
output `False`.
4. The explanation should be concise and focused on
verifiability.

Some Intuitive Guidelines
1. Statements like ``may'' are considered acceptable,
as doing or not doing does not affect testability
determination.
2. All elements of a testable requirement must be
deterministic; expressions such as ``other'', ``unless
 otherwise specified'', or ``submission time (instead
of a specific 9:00-10:00)'' are not allowed.
3. Testable requirements cannot contain references (e.
g., Article A, previous chapter, or earlier rules).
4. The formalized rule has already undergone some
testability processing and is more testable than the
original text. Evaluation mainly focuses on whether
the formalized rule meets testability requirements;
the natural language text serves as a reference.
Note: All testability determinations use strict OCL
constraint patterns. No contextual assumptions are
made; the judgment is based on pure textual-level
testability (must be independent and directly
verifiable).

Example
Input:
When the system receives a user payment instruction,
it should immediately deduct the corresponding account
 balance and return the transaction result status.
```
rule 1
if Actor = System and Event =
ReceivedUserPaymentInstruction and Action = Deduct and
 OperationPart = AccountBalance and Action = Return
and OperationPart = TransactionResultStatus
then Result = Success and ResultStatus =
TransactionSuccess
```
Output:
```
```

```
true
```

A.2 Extracted Regulatory Knowledge for the Financial Domain

This chapter presents the domain meta-model and the testable requirements representation for the financial domain.

A.2.1 Meta-Model

```
@startuml
skinparam defaultFontSize 20
skinparam ranksep 20
skinparam nodesep 20

class TradingRule

class Condition
class Operator
class Indicator
class Parameter
class Action
class Signal
class RuleSet
class PositionSizingRule
class RiskManagementRule
class EntryRule
class ExitRule

TradingRule *-- Condition : conditions
TradingRule *-- Action : actions
TradingRule -- RuleSet : part of
RuleSet *-- TradingRule : contains

Condition *-- Operator
Condition *-- Indicator
Condition *-- Parameter

Action -- Signal

EntryRule --|> TradingRule
ExitRule --|> TradingRule
PositionSizingRule --|> TradingRule
RiskManagementRule --|> TradingRule
@enduml
```

A.2.2 Formal Requirements Representation

```
### Symbols:
* Logical Symbols:
  * and, or, not
  * implicate
* Comparison Symbols
  * =, !=, >, >=, <, <=, in
* Domain Symbols
  * Precondition: Actor, TradingInstrument,
TradingMarket, Time, Constraint, Event
  * Action: Action, TradingDirection, TradingMethod,
Quantity, Price, OperationPart, Status
  * ExpectedResult: ResultStatus, Result

### Syntax:
```BNF
Rule ::= "IF" <Precondition> "AND" <Operation> "THEN"
<ExpectedOutcome>

Precondition ::= <AtomicPrecondition> | <
CompoundPrecondition>
AtomicPrecondition ::= <PreconditionElement> <
Comparator> <Value>
CompoundPrecondition ::= "(" <Precondition> ")"
                    | <Precondition> "AND" <
Precondition>
                    | <Precondition> "OR" <
Precondition>
                    | "NOT" <Precondition>

Operation ::= <AtomicOperation> | <CompoundOperation>
AtomicOperation ::= <OperationElement> <Comparator> <
Value>
CompoundOperation ::= "(" <Operation> ")"
                 | <Operation> "AND" <Operation>
                 | <Operation> "OR" <Operation>
                 | "NOT" <Operation>

ExpectedOutcome ::= <AtomicOutcome> | <CompoundOutcome
>
AtomicOutcome ::= <ResultElement> <Comparator> <Value>
CompoundOutcome ::= "(" <ExpectedOutcome> ")"
                  | <ExpectedOutcome> "AND" <
ExpectedOutcome>

PreconditionElement ::= "Actor" | "TradingInstrument"
| "TradingMarket"
                | "Time" | "Event" | "Constraint"

OperationElement ::= "Action" | "TradingDirection" | "
TradingMethod"
                    | "Quantity" | "Price" | "
OperationPart" | "Status" | "Constraint"

ResultElement ::= "ResultStatus" | "Result" | "
Constraint"

Comparator ::= "=" | "!=" | ">" | "<" | ">=" | "<=" |
"in"

Value ::= <StringLiteral> | <NumberLiteral> | <
BooleanLiteral> | <TimeLiteral> | <TimeRangeSet>
StringLiteral ::= "\"" [^"]* "\""
NumberLiteral ::= [0-9]+ ("." [0-9]+)?
BooleanLiteral ::= "true" | "false"

TimeLiteral ::= "\"" [0-9]{2} ":" [0-9]{2} (":" [0-9
]{2})? "\""  // Support minutes and seconds
TimeRange ::= <TimeLiteral> "-" <TimeLiteral>
TimeRangeSet ::= "[" <TimeRange> ("," <TimeRange>)*
"]"  // Example: [10:00-12:00,13:00-14:00]
```
```

A.2.3 Testability Constraints

```
1. Structural Completeness: Condition, Action, and
Result must be non-empty
context Rule
```OCL
inv StructuralCompleteness:
    not self.Precondition.oclIsUndefined() and
    not self.Operation.oclIsUndefined() and
    not self.ExpectedResult.oclIsUndefined()
```
2. Deterministic Rule Elements: Precondition,
Operation, and ExpectedResult elements must be
concrete and deterministic
```OCL
context Rule
inv RuleElementDeterministic:
    self.Precondition->forAll(e | e.concrete() and e.
deterministic())
    self.Operation->forAll(e | e.concrete() and e.
deterministic())
    self.ExpectedResult->forAll(e | e.concrete() and e.
deterministic())
```
3. Action Executability: Actions must be executable
```OCL
context Rule
inv ActionExecutable:
    self.Operation.Action.notEmpty() and
    self.Operation.Action.executable()
```
4. Expected Result Observability: Results must be
observable
```OCL
context Rule
inv ExpectedResultObservable:
    self.ExpectedResult.Result.notEmpty() and
```
```

```
    self.ExpectedResult.Result.observable()
```
5. Unambiguity Outcome: The same precondition and operation should not yield conflicting expected results
```OCL
context Rule
inv DeterministicOutcome:
    Rule.allInstances()->forAll(r2 |
        if r2 <> self and r2.Precondition = self.Precondition and r2.Operation = self.Operation
        then r2.ExpectedResult.ResultStatus = self.ExpectedResult.ResultStatus
        else true
        endif)
```

A.3 Prompt Used for compared E2E LLMs

In the experiments, we compare our approach with end-to-end LLMs in generating test cases from regulatory documents. Representative models, including GPT-5, Grok-4, and DeepSeek-R1, are selected as baselines. Task-specific prompts are carefully designed for them to elicit the best performance in test case generation. The prompts used in Experiment I are presented below; for Experiment II, the composition is the same, except for the application domain and the running example.

```
You are a Financial Requirement Modeling & Testing Expert, specializing in software requirements engineering, financial business rule analysis, and test case generation.
Your task is to generate test cases for the given financial-domain rules.

### Task Objective
Test Case Generation (only when the rule is testable)
* Based on the natural language rule, generate a set of high-quality test cases.
* Generation requirements:
  1. Content requirements
     * Each test case must be a JSON object (map).
     * Each test case should include multiple key elements, such as: actor, time, trading method, trading instrument, action, operation target, quantity, price, result, etc. (to be flexibly selected according to the rule content).
  2. Quality requirements
     * Test cases must cover both positive (success) and negative (failure) scenarios.
     * Test cases should reflect rule boundaries and constraint validation.
     * Test cases should not be duplicated and must be representative and distinguishable.
     * The output must be strictly in JSON array format.
* If the rule is not testable and test cases cannot be generated, do not generate any output.

### Input Format
Input consists of one natural language financial-domain rule.

### Output Format
Output test cases in JSON format:
[
  {
    "id": 1,
    "TradingInstrument": ...,
    "Time": ...,
    ...
  },
  {
    "id": 2,
    ...
  },
  ...
]

### Example
#### Input:
An investor who buys bond ETF shares through auction trading on the same day may sell them on the same day.
#### Output:
[
  {
    "id": 1,
    "Actor": "Investor",
    "Time": "Same day",
    "TradingMethod": "Auction trading",
    "Action": "Buy",
    "TradingInstrument": "Bond ETF shares",
    "Time2": "Same day",
    "Action2": "Sell",
    "Result": "Success"
  },
  {
    "id": 1,
    "Actor": "Investor",
    "Time": "Not the same day",
    "TradingMethod": "Auction trading",
    "Action": "Buy",
    "TradingInstrument": "Bond ETF shares",
    "Time2": "Same day",
    "Action2": "Sell",
    "Result": "Failure"
  },
  ...
]
```