UDC 004.93:004.415.2 DOI: 10.20998/2413-3000.2025.11.9

O. CHEREDNICHENKO, V. MALIARENKO

THE DISPATCHER: BRIDGING THE PROBABILISTIC GAP IN AUTOMATED DECISION MODELING

In the contemporary landscape of Software Engineering and Business Process Management (BPM), the integration of generative artificial intelligence has precipitated a paradigm shift from manual, deterministic specification to automated, probabilistic generation. While offering scalability, this transition introduces a fundamental volatility known as the "Probabilistic Gap"—the chasm between the fluid, high-variance output of Large Language Models (LLMs) and the strict, zero-tolerance syntactic requirements of execution engines like DMN (Decision Model and Notation). This paper addresses the "Struc-Bench Paradox," highlighting the limitations of transformer architectures in generating complex structured data without rigid orchestration. The study formally defines and implements the "Dispatcher," a pivotal control plane component designed to function as an intelligent resource arbiter and quality gatekeeper within a neuro-symbolic architecture. The theoretical framework shifts the economic focus from Baumol's Cost Disease, which addresses production speed, to Boehm's Law of Software Economics, which emphasizes the exponential cost of defects propagated to production. To operationalize this, the Dispatcher represents a discrete deterministic process modeled using Cost-Colored Petri Nets rather than Finite State Machines (FSMs). The Petri Net formalism allows for precise modeling of concurrency, state accumulation, and the strict enforcement of "Retry Budgets," thereby mathematically guaranteeing system termination and preventing infinite loops of costly regeneration. The architectural implementation utilizes a "Test-First" generation philosophy: the system first synthesizes validation criteria (JSON test cases) utilizing Schema Injection and RAG, and subsequently grounds the generation of DMN logic (XML) in these pre-validated scenarios. Experimental analysis was conducted using a controlled set of 200 generation cycles to evaluate two distinct error-recovery strategies: Strategy A (Independent regeneration of DMN tables only) and Strategy B (Joint/Dynamic regeneration of both DMN and Test Cases). Quantitative results demonstrate that Strategy B is economically superior, achieving a 6.06% reduction in total cost and an 8.44% reduction in token consumption compared to the independent patching approach. The findings indicate that simultaneous regeneration empowers the LLM to resolve semantic incoherence and hallucinations more effectively than iterative repairs, prioritizing logical consistency over partial code retention. The study concludes that the Dispatcher effectively bridges the neuro-symbolic divide by transforming validation from a post-production manual review into a pre-production automated cycle. By enforcing a "Stop-Loss" mechanism driven by economic constraints, the framework minimizes the Total Cost of Ownership and serves as a critical "Trust Proxy," mitigating automation bias and ensuring that AI-generated artifacts meet the rigorous reliability standards required for enterprise deployment.

Keywords: Large Language Models; Automated Decision Modeling; Petri Nets; Validation and Verification; Business Process Management

О. Ю. ЧЕРЕДНІЧЕНКО, В. В. МАЛЯРЕНКО

ДИСПАТЧЕР: ПОДОЛАННЯ ІМОВІРНІСНОГО РОЗРИВУ В АВТОМАТИЗОВАНОМУ МОДЕЛЮВАННІ РІШЕНЬ

У сучасному ландшафті управління бізнес-процесами (BPM) та програмної інженерії інтеграція генеративного штучного інтелекту зумовила фундаментальний зсув парадигми від ручної детермінованої специфікації до автоматизованої імовірнісної генерації. Цей перехід, забезпечуючи безпрецедентну масштабованість, створює критичний «імовірнісний розрив» між стохастичною природою великих мовних моделей (LLM) та суворими синтаксичними вимогами середовищ виконання рішень, таких як DMN (Decision Model and Notation). Дослідження фокусується на вирішенні проблеми «Struc-Bench Paradox», яка демонструє нездатність стандартних трансформерних архітектур надійно генерувати складні структуровані дані без зовнішнього керування. Центральним елементом запропонованого рішення є «Dispatcher» (Диспатчер) – архітектурний компонент, що виконує роль інтелектуального арбітра ресурсів та шлюзу якості в нейро-символічній системі. Методологічною основою роботи є перехід від економічної теорії «хвороби витрат Баумоля», яка пріоритезує швидкість виробництва, до «закону економіки ПЗ Боема», що встановлює логарифмічну залежність між часом виявлення дефекту та вартістю його виправлення. Для формалізації дискретних детермінованих процесів Диспатчера у роботі застосовано математичний апарат Мереж Петрі (Cost-Colored Petri Nets) замість класичних скінченних автоматів (FSM). Це дозволило ефективно моделювати стан системи, управляти конкурентністю процесів та суворо контролювати бюджет повторних спроб (retry budget), уникаючи ризиків нескінченних циклів регенерації та «спіралі смерті» витрат токенів. Реалізація системи базується на патерні «Test-First Generation», де процес валідації відокремлено від генерації бізнес-логіки: спочатку формуються тестові кейси (JSON), і лише на їх основі генерується DMN-модель, яка негайно перевіряється вбудованим рушієм Camunda. Емпірична частина дослідження включає аналіз ефективності двох стратегій відновлення після помилок на вибірці з 200 циклів генерації: Стратегії А (незалежна регенерація тільки таблиці DMN) та Стратегії В (спільна регенерація DMN та тестових кейсів). Результати експерименту виявили контрінтуїтивну перевагу Стратегії В, яка продемонструвала зниження загальної вартості генерації на 6,06% та скорочення споживання токенів на 8,44%, при досягненні 100% успішності валідації. Доведено, що повна регенерація контексту дозволяє LLM усунути логічні галюцинації ефективніше, ніж ітеративне виправлення окремих фрагментів коду. Отримані результати підтверджують, що Диспатчер виступає економічним щитом підприємства, забезпечуючи принцип «fail fast and fix cheap» (швидка помилка – дешеве виправлення). Впровадження запропонованого фреймворку трансформує процес створення моделей рішень з ризикованого експерименту у надійну інженерну дисципліну, де символічний валідатор виступає гарантом істини для нейронного генератора, забезпечуючи довіру користувачів та стабільність бізнес-систем.

Ключові слова: великі мовні моделі, автоматизоване моделювання рішень, мережі Петрі, валідація та верифікація, управління бізнес-процесами.

1. Introduction. In the contemporary landscape of software engineering and Business Process Management, the integration of generative artificial intelligence has precipitated a paradigm shift. We are moving from an era of manual, deterministic specification to one of automated, probabilistic generation. This transition, while offering unprecedented scalability, introduces a fundamental volatility into the heart of enterprise architecture. The central challenge is no longer the generation of code or logic – Large Language Models have demonstrated sufficiency in this regard – but rather the rigid orchestration of these non-deterministic outputs into reliable, executable artifacts.

This report focuses on the Dispatcher, the pivotal component of a proposed framework designed to automate the creation of Decision Model and Notation artifacts. The

Dispatcher is not merely a message router; it is an intelligent control system, a resource arbiter, and a quality gatekeeper. It exists to bridge the "Probabilistic Gap" – the chasm between the fluid, high-variance output of an LLM and the strict, zero-tolerance syntactic requirements of a DMN execution engine like Camunda.

Historically, the automation of decision logic has been viewed through the lens of Baumol's Cost Disease, which posits that labor-intensive sectors, like business analysis, suffer from stagnant productivity compared to manufacturing [1]. While valid, this economic theory addresses only the speed of production. In the context of AI-generated code, the more pressing economic principle is Boehm's Law of Software Economics [2]. It is often cited in quality engineering contexts as Bem's Law of the cost of defects.

Boehm's Law establishes a logarithmic relationship between the time a defect remains in a system and the cost to repair it [3]. An error in a decision table caught during the drafting phase costs strictly the time required to rewrite a line of text. The same error, if propagated to a production runtime environment, incurs costs related to system downtime, regulatory fines, customer service remediation, and reputational damage. The cost differential can arguably exceed 100:1.

Therefore, the primary economic mandate of the Dispatcher is not simply to "generate DMN fast" (addressing Baumol) but to "fail fast and fix cheap" (addressing Boehm). By encapsulating the generation process within a rigorous, iterative validation loop, the Dispatcher spends cheap computational resources, like tokens, to prevent expensive operational failures. It transforms the validation process from a post-production manual review into a pre-production automated cycle.

This report aims to:

1. Formally define the Dispatcher abstraction and its role in the wider framework.

2. Provide a comparative mathematical modeling of the Dispatcher's behavior, specifically analyzing the trade-offs between Finite State Machines and Petri Nets [9] in representing its discrete deterministic processes.

3. Detail the concrete implementation of the Dispatcher as a backend service.

4. Analyze the efficiency of different validation strategies using empirical data, focusing on token consumption and cost optimization.

5. Derive conclusions regarding the critical importance of the Dispatcher in enabling safe, autonomous BPM systems.

2. Related Works. The DMN has emerged as the industry standard for bridging the communication gap between business analysts and technical developers. DMN provides a standardized XML-based syntax for defining decision tables and FEEL logic, allowing organizations to externalize decision rules from application code. This externalization is crucial for agility, compliance, and maintainability.

The core of DMN is the Decision Table, which maps a set of inputs to a set of outputs based on a list of rules. Crucial to the deterministic nature of DMN are Hit Policies (e.g., Unique, Any, Priority, First), which dictate how the engine resolves scenarios where multiple rules match the input data. A violation of the Hit Policy (e.g., overlapping rules in a 'Unique' table) renders the model invalid and halts execution. The complexity of DMN lies in its combination of structural rigidity (strict XML schema validation) and expressive power (FEEL functions for string manipulation, date calculations, and list filtering).

Recent academic discourse highlights the "Complexity Wall" in manual DMN creation. As the number of input variables and rules increases, the state space grows combinatorially [4]. This "Combinatorial Explosion" exceeds the cognitive capacity of human analysts, leading to errors of omission or contradiction that are difficult to detect manually. This limitation underscores the necessity for automated generation, yet simultaneously raises the bar for the quality of that automation.

Recent research into the capabilities of Large Language Models has identified a phenomenon known as the "Struc-Bench Paradox". Study [5] indicates that while LLMs excel at generating coherent natural language, they struggle significantly with complex structured data generation. The probabilistic nature of the Transformer architecture is optimized for semantic flow and narrative coherence, not for the rigid syntactic constraints of formal languages like XML or JSON.

LLMs frequently lose track of long-range dependencies in XML structures (e.g., closing tags opened hundreds of tokens prior) or fail to maintain type consistency across a decision table. Experiments using GPT-3 for DMN generation have shown "underwhelming" results when used in a zero-shot, unconstrained manner, often failing to understand critical logical concepts such as mutual exclusivity and completeness [6]. The models may generate a table that appears correct to a human reader but fails validation due to subtle schema violations or "hallucinated" FEEL functions that do not exist in the standard specification. This necessitates a shift from simple "Prompt Engineering" to robust "Neuro-Symbolic Architectures," where the LLM is treated as a stochastic component within a larger, deterministic system.

The proposed solution aligns with the emerging field of Neuro-Symbolic AI [7, 13], which seeks to combine the learning and generative capabilities of neural networks with the reasoning and guarantees of symbolic logic. In this context, the LLM provides the "Neuro" component – the ability to translate fuzzy, unstructured intent into draft code – while the Dispatcher and the DMN Engine provide the "Symbolic" component – the verification of syntax, logic, and execution.

Research [8, 12] indicates that RAG significantly improves the factual accuracy of LLMs by grounding generation in retrieved documents, such as domain-specific policy manuals or schema definitions. Furthermore, the concept of "Self-Correction" or "Self-Healing" loops has gained traction. However, as noted in "Struc-Bench," self-correction is only effective if the external signal (the error message) is precise and the system has a strategy to utilize it effectively [15]. This report explores specifically how to orchestrate that correction loop efficiently, moving beyond simple retries to strategic regeneration.

The implementation of the Dispatcher also addresses critical human factors in AI adoption. Trust in AI systems is heavily influenced by Transparency and Reliability [10]. The "Black Box" nature of LLMs reduces trust, as users cannot see the internal reasoning process. By wrapping the LLM in a Dispatcher that validates output against explicit test cases, the system provides a powerful transparency mechanism – users can see why a model was accepted, because it passed specific tests, or why it was rejected.

Moreover, the literature suggests that user Mental Models play a crucial role in the successful deployment of AI. Users with inaccurate mental models of AI capabilities may over-rely on the system, accepting incorrect outputs without scrutiny, or under-rely on it, rejecting valid outputs due to skepticism. The Dispatcher mitigates over-reliance by acting as a hard quality gate; it simply refuses to present an invalid model to the user, thereby enforcing a baseline of reliability that fosters appropriate trust. The "Feeling of Rightness" and "Feeling of Error" are metacognitive signals that influence user reliance [11]; the Dispatcher's explicit validation reports align these feelings with the actual technical reality of the generated artifact.

3 Formalizing the Dispatcher. In the framework, the Dispatcher represents the Control Plane. The architecture separates the Request, the Generator, and the Validator. The Dispatcher sits at the intersection of these three vectors.

It functions as a feedback transducer. Current LLMs lack internal feedback loops; they generate token t+1 based on token t without the ability to pause, test, and retract. The Dispatcher imposes this missing loop externally, acting as the analytic layer over the generative layer.

To engineer a robust Dispatcher, we must move beyond ad-hoc scripting and strictly define its behavior using mathematical formalisms. This allows us to prove properties about the system, such as termination, resource boundedness, and deadlock freedom. The two primary candidates for modeling discrete event systems like the Dispatcher are Finite State Machines and Petri Nets.

3.1 Finite State Machines.

A Finite State Machine is a computational model consisting of a finite number of states, transitions between those states, and inputs. Formally, a deterministic finite automaton is a 5-tuple:

$$M = (Q, \Sigma, \delta, q_0, F)$$

Where:

Q is a finite set of states (e.g., Idle, Generating, Validating, Success, Failure).

Σ is a finite set of input symbols (e.g., ReceiveRequest, GenSuccess, GenFail, ValSuccess, ValFail).

δ is a transition function QQ.

q_0 is the start state.

F is the set of accepted states.

An FSM can model the basic lifecycle of a request:

1. Start in Idle.
2. On “Receive Request”, transition to Generating.
3. On “GenSuccess”, transition to Validating.
4. On “ValSuccess”, transition to Success (Final).
5. On “ValFail”, transition back to Generating.

The limitation of the FSM becomes apparent when we introduce resources, specifically the "Retry Limit." An FSM has no internal memory of how many times it has visited a state. To model a retry limit of 5, an FSM essentially requires distinct states for each attempt. Figure 1 demonstrates a potential infinite loop in FSM representation of Dispatcher.

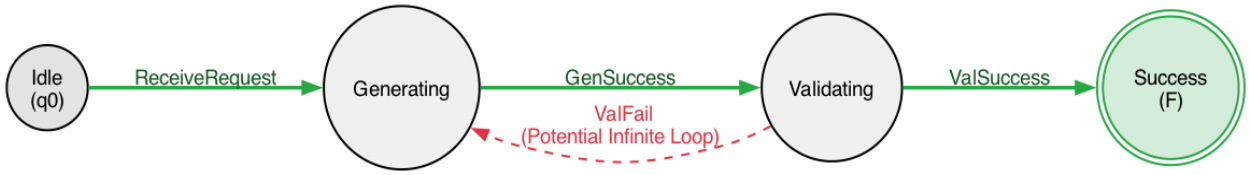

Fig. 1. Potential issue with FSM Dispatcher

The complexity of the diagram grows linearly with the magnitude of the counter. Furthermore, FSMs are inherently sequential. They cannot easily model a scenario where the Dispatcher generates the DMN and the Test Cases in parallel threads and waits for both to complete before validating.

3.2 Petri Nets.

A Petri Net is a mathematical modeling language for the description of distributed systems [9]. It is a directed bipartite graph. Formally, a Petri Net is a tuple:

$$PN = (P, T, F, W, M_0)$$

Where:

P is a finite set of Places (represented by circles). Places hold Tokens.

T is a finite set of Transitions (represented by bars/rectangles).

$F \subseteq (P \times T) \cup (T \times P)$ is a set of flow relations (arcs) connecting Places to Transitions and Transitions to Places.

W is a weight function (how many tokens are consumed/produced).

M_0 is the initial marking (distribution of tokens).

The Dispatcher is best modeled as a High-Level Petri Net or a Colored Petri Net, where tokens utilize values (attributes).

Places (P)

- P_{in} – Incoming Requests.
- P_{budget} – Available Retries.
- P_{gen} – The system is generating.
- P_{val} – The system is validating.
- P_{out} – final Output.

Transitions (T)

- T_{start} moves token from P_{in} to P_{gen}.

- T_{retry} consumes 1 token from P_{budget}. If Pbudget is empty, this transition cannot fire. This natively enforces the logic: "If retries > 0, then regenerate."

- T_{abort} fires only when P_{budget} is empty and validation fails.

The fundamental constraint of the Dispatcher is the "Retry Limit" (driven by cost). Petri Nets model this naturally through the initial marking $M_0(P_{budget}) = 5$. The availability of a token in the Pbudget place is a hard requirement for the T_{retry} transition. If the place is empty, the logic physically prevents the Dispatcher from retrying, enforcing the "Stop-Loss" strictly without the need for external counters or state explosion.

Petri Nets natively handle concurrency. If we decide to optimize speed by generating DMN and Test Cases simultaneously, a Petri Net handles this with a "Fork" transition (one input token produces two output tokens into parallel places (e.g., P_{genDMN} and $P_{genTest}$) and a "Join" transition (which waits for tokens in both parallel places before firing). FSMs cannot model this without extreme complexity.

Petri Nets allow us to visualize the flow of the process and the accumulation of state (tokens) simultaneously.

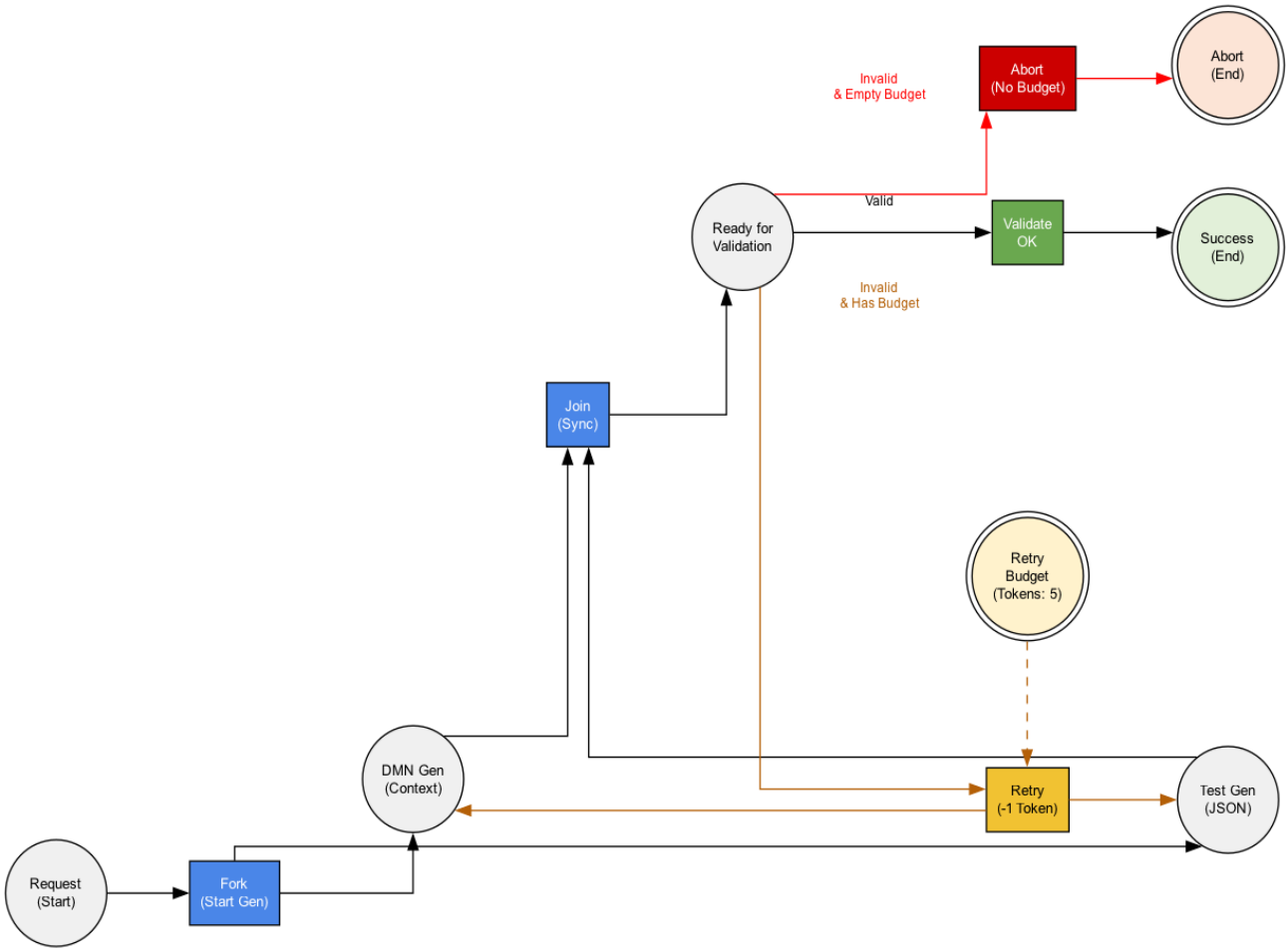

Fig. 2. Petri Net Visualisation

We can extend this to a Cost-Colored Petri Net where each transition t is associated with a cost function $C(t)$. The total cost of a trace is the summation of costs of all fired transitions. This maps directly to the economic analysis of token consumption, allowing for precise modeling of operational expenditure.

For the purpose of this framework, we define the Dispatcher as a Discrete Deterministic Process modeled by a Petri Net. This choice allows us to strictly define the state space while explicitly managing the economic constraints, represented by the budget tokens, that govern the system's operation. It provides the mathematical rigor necessary to prove that the system will eventually terminate either in Success or Exhaustion and will never enter an infinite loop of costly regeneration.

4 Dispatcher Implementation. Having defined the abstraction and the mathematical model, we now describe the Dispatcher as it exists in the current technological implementation. The architecture is designed around a "Micro-Kernel" pattern where the Dispatcher orchestrates specialized modules, ensuring separation of concerns and scalability.

4.1 Component Architecture.

The system consists of the following core modules, orchestrated by the Dispatcher shown on Figure 3.

The Dispatcher accepts the RuleSet, manages the RetryBudget, and maintains the state of the transaction. It acts as the "Single Source of Truth" for the modeling process.

Schema Injector takes the target JSON schema and injects the schema definition directly into the prompt context. This constrains the LLM's search space, preventing it from hallucinating variable names which would cause immediate validation failures.

Prompt Composer is responsible for assembling all necessary input elements into a structured prompt. It combines:

- System Instructions – syntax rules like "Return only XML" or "Do not use markdown formatting".
- Schema Context – the output from the Schema Injector.
- RAG Content external knowledge retrieved from the knowledge base (e.g., domain policies, previous successful models).
- User Rule Set the natural language description of the logic.
- Retry Context if in a retry loop, it appends the previous error log and the failed XML to guide the correction.

Generation Module is an interface to the probabilistic agent (e.g., GPT-4 or Claude 3.5 Sonnet). It handles API communication, token limits, and temperature settings. For code generation tasks, the temperature is typically set low (T=0 to 0.3) to minimize variance and maximize determinism.

Validator Module is a deterministic verification engine. It performs a multi-stage check:

- Syntax Checker validates against the DMN XML Schema (XSD).
- Semantic Checker parses FEEL expressions for type safety.
- Logic Checker runs the generated DMN against generated Test Cases using an embedded Camunda DMN Engine.

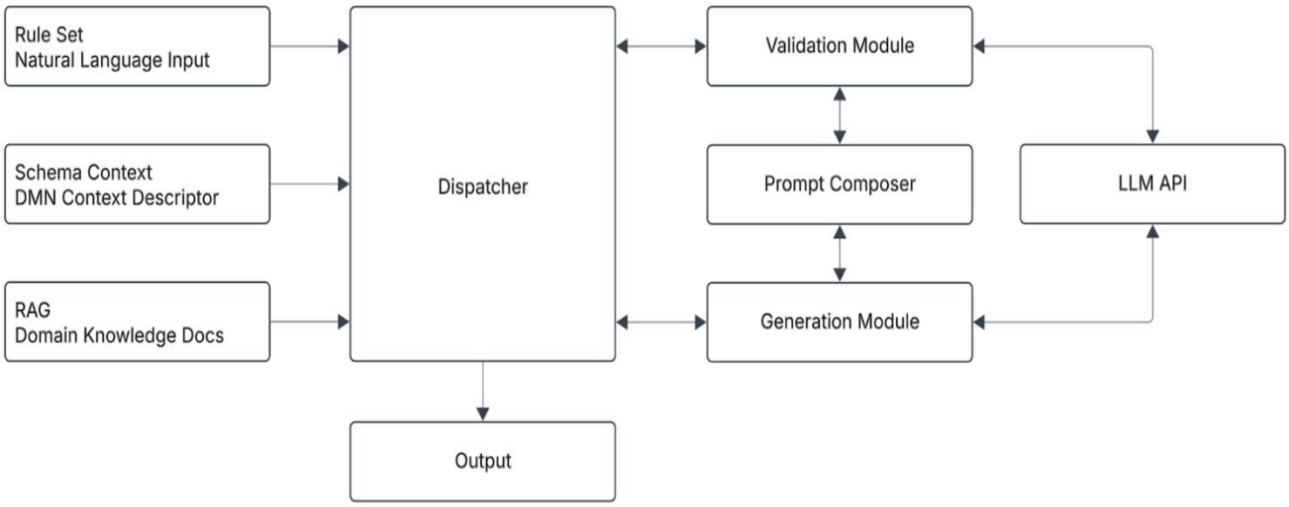

Fig. 3. Framework Structure

4.2 The Execution Flow.

The Dispatcher's execution flow orchestrates a sophisticated, multi-stage process designed to enforce reliability through a "Test-First" generation philosophy. Unlike rigorous single-shot approaches, the architecture decouples the creation of validation criteria from the generation of business logic. The process initiates when the Dispatcher receives a request and loads the relevant schema definitions. In the first phase, the Prompt Composer constructs a targeted prompt dedicated solely to generating a comprehensive set of Test Cases in JSON format. This utilizes the injected schema context and retrieved domain knowledge to ensure the tests cover edge cases before any decision logic is written.

Upon successfully receiving the structured Test Cases, the Dispatcher transitions to the second phase: DMN generation. The Prompt Composer constructs a new, distinct prompt that incorporates the user's natural language rules alongside the specific Test Cases generated in the previous step. This architectural pattern grounds the Large Language Model, effectively instructing it to write XML logic that satisfies the concrete data scenarios already defined. This significantly reduces the hallucination of variables, as the model is constrained by the strict structure of the pre-generated test data.

Once the DMN XML is generated, the Dispatcher does not merely store the artifact but subjects it to immediate, execution-based validation. The system spins up an instance of the embedded Camunda DMN engine and executes the newly created logic against the Test Cases. This internal loop acts as the process's immune system, comparing the engine's actual computed outputs against the expected results. If the validation passes, the artifact is stamped as valid and returned to the user.

However, if a discrepancy arises – whether a syntax violation or a logical mismatch where the actual output differs from the expected – the Dispatcher triggers a convergence loop. The exact error context is captured and fed back into the subsequent generation prompt, instructing the model to debug its previous output. This iterative refinement continues until the system converges on a solution that passes all tests or until the operational Retry Budget is exhausted, ensuring the system fails fast and fixes cheaply according to the economic principles defined earlier.

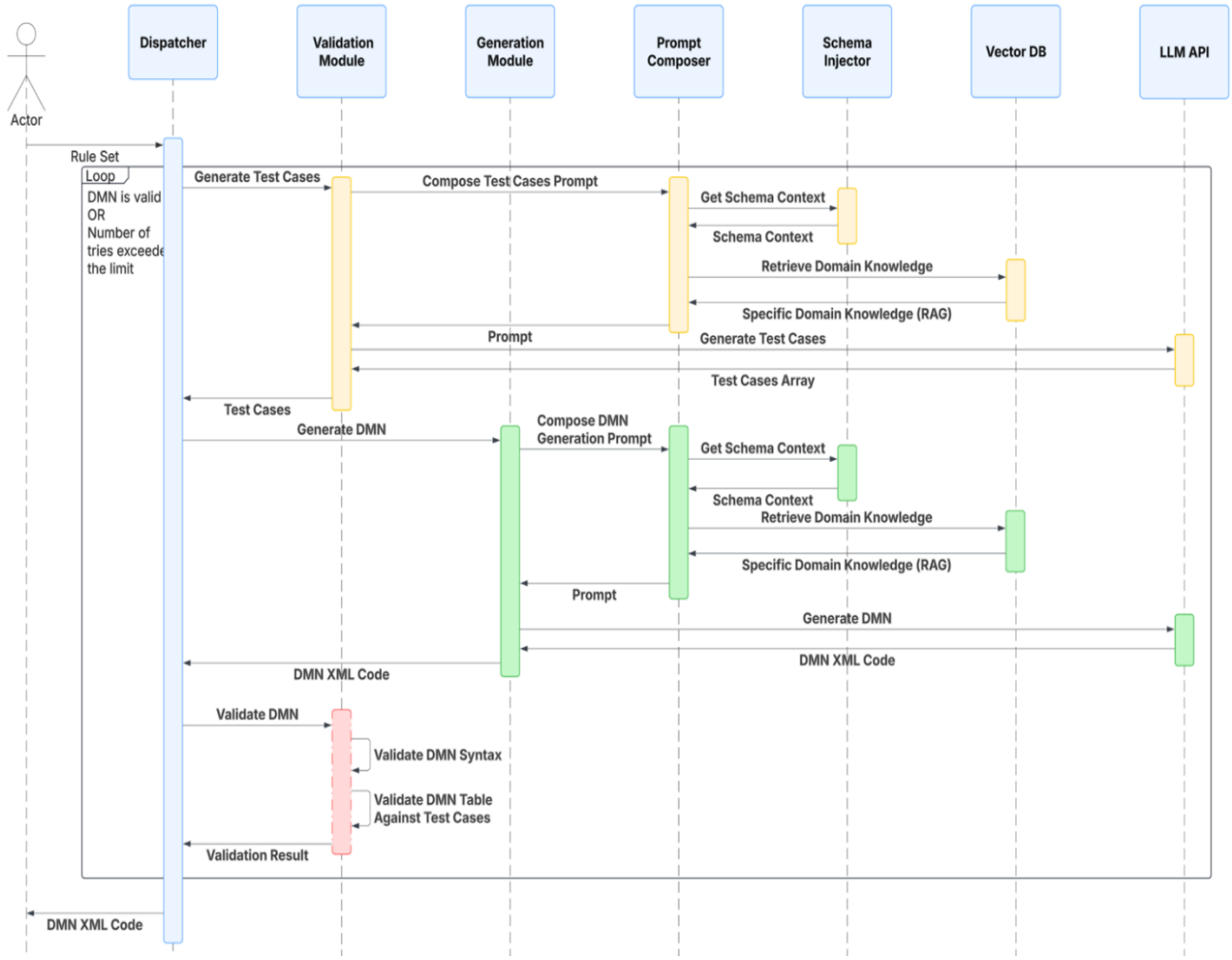

Fig. 4. Execution Sequence Diagram

4.3 Parameters and Configuration.

The Dispatcher's operation is defined by a set of organization-configured parameters. Parameters, which represent system constraints, include Max_Retries, a hard limit on the processing budget, the Model_Temperature, which determines the variance and determinism of the underlying AI model and Validation_Strategy taht allows the Dispatcher to select between a fast, low-confidence "Syntax Only" check and the more rigorous, expensive, and slow "Full Semantic Execution." The current system implementation is configured to use the "Full Semantic Execution" strategy.

5 Efficiency Optimization.

The introduction of the Dispatcher effectively replaces the human labor cost with a computational resource cost. While computing is cheap, it is not free. To optimize this, we must rigorously define the cost function governed by Boehm's Law.

5.1 The Cost Function of Determinism.

We can define the cost of a single validated DMN model (C_{model}) as:

$$C_{model} = C_{gen} + \sum_{k=1}^{N} (C_{eval} + C_{fix})$$

Where:

C_{gen} – is the initial generation cost (Tokens In + Tokens Out).

N – is the number of retries required.

C_{eval} – is the computational cost of running the validation engine (negligible in cloud terms, but non-zero in time).

$C_{fix}$is the cost of the regeneration call.

Crucially, C_{fix}tends to be higher than C_{gen}because the context window grows. The "Fix" prompt must contain: (Original Rules + Original Prompt + Bad XML + Error Log + Fix Instruction). Thus, the cost of retries accelerates.

5.2 Boehm's Law and the "Stop-Loss".

Boehm's Law states that the cost of a defect grows exponentially with the phase of detection. Let $Cost_{design} = 1X$. Let $Cost_{production} = 100X$.

The Dispatcher's goal is to minimize the Total Cost of Ownership:

$TCO = C_{model} + (P_{failure} \times Cost_{production})$

Where $P_{failure}$– is the probability that a defective model slips through the Dispatcher.

By implementing the Validation Loop, the Dispatcher drives $P_{failure}$toward zero. Even if the Dispatcher spends 5x the generation cost on retries, it is strictly economically superior to releasing a bug that costs 10.

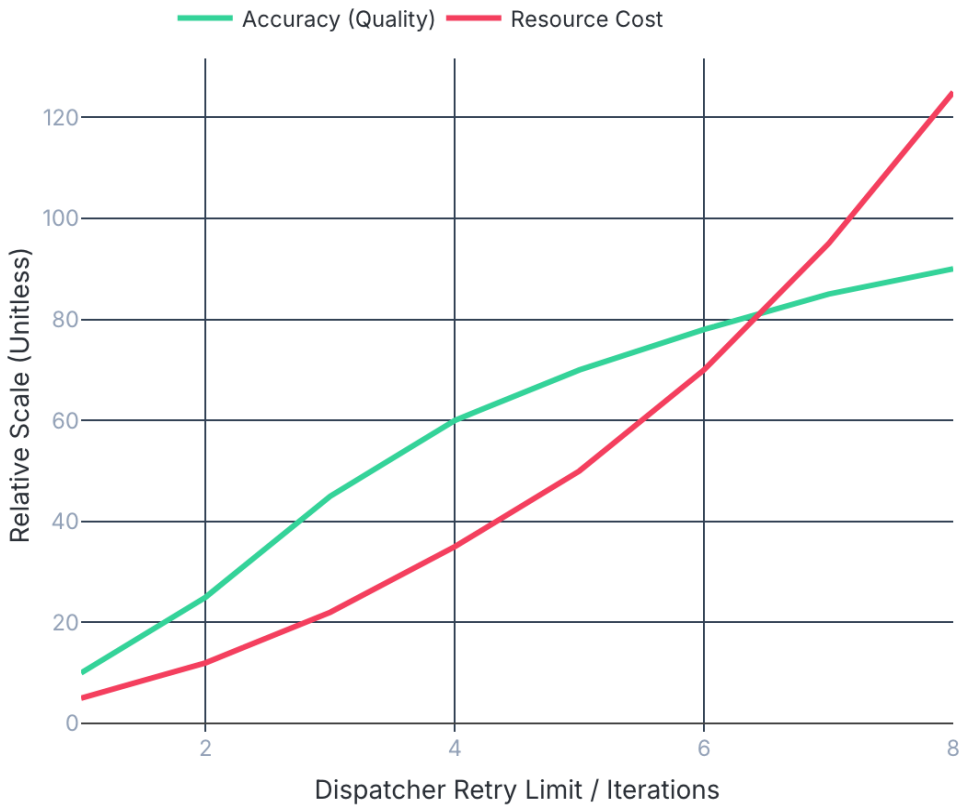

Fig. 5. The theoretical intersection of cost and quality managed by the Dispatcher

However, there is a limit. If the model fails 5 times, the probability of success on the 6th try drops significantly (diminishing returns), while the cost accumulates. The Retry Limit serves as the economic "Stop-Loss." It prevents the "Death Spiral" where the AI consumes infinite tokens trying to solve an impossible or poorly defined prompt. At N=5, the Dispatcher admits defeat and escalates to a human, preserving the remaining budget.

The Dispatcher tracks dependencies between resources spent and value obtained.

- Input. Time and Money (Tokens).
- Output. Accuracy (Valid DMNs) and Automation (Removal of Human).

The experimental data allows us to quantify this relationship.

6 Experimental Analysis.

We analyze the performance of the Dispatcher based on a controlled experiment comprising 200 generation cycles. The experiment compares two distinct error-recovery strategies utilized by the Dispatcher.

6.1 Experiment Setup

The modeling and experimental phase for Dispatcher optimization involved running 200 requests with a maximum limit of 5 retries. The core validation method was to execute the generated Test Cases against the corresponding DMN table. Two distinct strategies were tested for handling validation failures.

Strategy A, the "Independent" approach, mandated that upon a validation error, the Dispatcher would request a regeneration of only the DMN Table, treating the original Test Cases as the definitive "Ground Truth." In contrast, Strategy B, the "Joint/Dynamic" approach, allowed for more flexibility by instructing the Dispatcher to request a simultaneous regeneration of both the DMN Table and the Test Cases when a validation failure occurred. These strategies aimed to compare the efficiency and effectiveness of selectively regenerating components versus regenerating them together to correct discrepancies.

6.2 Quantitative Results

The user provided summary statistics for the two strategies.

Table 1 –Strategies Comparison

Metric	Strategy A (DMN Only)	Strategy B (Joint Regen)	Improvement
Total Cost (USD)	$21.96	**$20.63**	**6.06% Savings**
Total Tokens In	1,378,730	**1,262,336**	**8.44% Reduction**
Total Tokens Out	1,188,312	1,189,275	~0% (Neutral)
Overall Success Rate	95.5% (191/200)	100% (200/200)	4.5% Improvement

6.3 Interpretation of Results.

The experimental findings reveal a counter-intuitive outcome critical to the Dispatcher's architectural design. Initially, one might hypothesize that Strategy A would prove more economical due to the reduced text generation (only the DMN) during retry cycles. However, the data indicates a higher total cost for this approach.

Analysis of the experiment logs suggests that DMN failures frequently stem from inherent ambiguities or hallucinations in the initial generation. When the Dispatcher mandates the Language Model to rectify the DMN to align with the originally generated – and potentially flawed – test cases, the LLM encounters significant difficulty. This situation precipitates a "conflict state" wherein the logical reconciliation is unattainable. The consequence is an increased number of retries (reaching 3, 4, or 5 attempts), which inflates the consumption of Input Tokens (history) and, consequently, the overall operational cost.

In contrast, Strategy B (Joint Regeneration) provides the Dispatcher with the capability to declare: "The current logic is fundamentally inconsistent. The entire artifact must be discarded and regenerated." By executing a simultaneous regeneration of both the DMN and the Tests, the LLM is empowered to construct a new, internally coherent semantic structure. This process resolves the initial ambiguity by initiating a clean slate. The data shows that Strategy B achieves valid convergence faster (fewer retries).

Even though each retry generates more tokens (DMN + JSON), the total number of retries drops significantly enough to reduce the overall Token usage by 8.44%. The experiment logs confirm the robustness of the Dispatcher in Strategy B.

7 Discussion.

The analysis of the Dispatcher reveals broader implications for the future of AI in BPM. The Dispatcher effectively acts as an economic shield for the enterprise. By strictly enforcing Boehm's Law – catching errors when they cost $0.10 at one retry rather than $10,000 at one production incident. The cost of roughly $0.10 per validated model is orders of magnitude lower than the human equivalent, which would likely exceed hundreds of dollars in billable hours for analysis and testing.

The superiority of Strategy B suggests a best practice for Neuro-Symbolic systems: Coherence over Patching. When a probabilistic model fails to produce a consistent logical structure, it is often cheaper to discard the artifact and regenerate it than to attempt iterative repairs. Strategy B effectively implements "Test-Driven Development" for AI. The model is forced to align its semantic understanding of the problem across two different modalities (JSON data and XML logic), filtering out hallucinations that would appear in only one.

From a human factors perspective, the Dispatcher serves as a "Trust Proxy." As noted in the literature, users are prone to "Automation Bias" or "Algorithm Aversion" based on their mental models of the AI's reliability. Inaccurate mental models can lead to dangerous over-reliance. The Dispatcher's strict validation regime ensures that the system output is never a hallucination; it is either a valid model or an error message.

This work contributes to the broader field of Neuro-Symbolic AI by demonstrating a practical implementation of the Neuro → Symbolic → Neuro Architecture. The symbolic validator acts as the ground truth that guides the neural generator. This overcomes the "Struc-Bench" limitations not by making the LLM "smarter", which is expensive and uncertain, but by placing it in a system that makes it "safer." Future work should explore the integration of formal verification methods (e.g., SMT solvers) into the Validator module to provide even stronger guarantees than test-based execution.

8 Conclusion.

The Dispatcher is the keystone of the automated decision modeling framework. It transforms the integration of Large Language Models into Business Process Management from a risky experiment into a viable engineering discipline.

By abstracting the non-deterministic interactions of the AI into a discrete, deterministic process modeled by Petri Nets, the Dispatcher ensures operational stability. It manages the inherent trade-off between the cost of generation and the accuracy of the result, leveraging the logic of Boehm's Law to minimize the Total Cost of Ownership.

The experimental evidence unequivocally supports the implementation of a Dispatcher that utilizes a Joint Regeneration strategy. This approach, which prioritizes the creation of internally consistent logic-validation pairs over iterative patching, demonstrated a 6.06% reduction in cost and an 8.44% reduction in token consumption compared to traditional methods.

In conclusion, the Dispatcher validates the premise that while AI can generate the logic, it is the deterministic orchestration – the rigid framework of checks, balances, and economic limits – that creates the value. It solves the "Modeling Bottleneck" not just by working faster than a human, but by validating cheaper than a human.

References

1. Baumol, W. J. (1967). Macroeconomics of Unbalanced Growth: The Anatomy of Urban Crisis. *The American Economic Review*, 57(3), 415–426. DOI: 10.2478/ie-2023-0066.
2. Boehm, B. W. (1981). *Software Engineering Economics*. Prentice-Hall. DOI: 10.1109/TSE.1984.5010193.
3. Boehm, B. W., & Basili, V. R. (2001). Software Defect Reduction Top 10 List. *Computer*, 34(1), 135–137. DOI: 10.1109/2.962984.
4. Hasic, F., & Vanthienen, J. (2019). Complexity metrics for DMN decision models. *Computer Standards & Interfaces*, 65, 15–37. DOI: 10.1016/j.csi.2019.01.001.
5. Tang, X., Zong, Y., Phang, J., Zhao, Y., Zhou, W., Cohan, A., & Gerstein, M. (2024). Struc-Bench: Are Large Language Models Good at Generating Complex Structured Tabular Data? *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 12–34. DOI: 10.18653/v1/2024.naacl-short.2.
6. Goossens, A., Vandevelde, S., Vanthienen, J., & Vennekens, J. (2023). GPT-3 for Decision Logic Modeling. *Proceedings of the 17th International Rule Challenge @ RuleML+RR* 2023, CEUR Workshop Proceedings, Vol-3485.
7. Bhuyan, B. P., Ramdane-Cherif, A., Tomar, R., & Singh, T. P. (2024). Neuro-symbolic artificial intelligence: a survey. *Neural Computing and Applications*, 36, 12809–12844. DOI: 10.1007/s00521-024-09960-z.

8. Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., & Bi, Y. (2023). Retrieval-Augmented Generation for Large Language Models: A Survey. *arXiv preprint*. DOI: 10.48550/arXiv.2312.10997.
9. Weske, M. (2019). *Business Process Management: Concepts, Languages, Architectures* (3rd ed.). Springer. DOI: 10.1007/978-3-662-59432-2.
10. Kaplan, A. D., Kessler, T. T., Brill, J. C., & Hancock, P. A. (2023). Trust in Artificial Intelligence: Meta-Analysis. *Human Factors*. DOI: 10.1177/00187208211013986.
11. Kaplan, A. D., Kessler, T. T., Brill, J. C., & Hancock, P. A. (2023). Trust in Artificial Intelligence: Meta-Analysis. *Human Factors*, 65(2), 337–365. DOI: 10.1177/00187208211013988.
12. Etikala, V., Van Veldhoven, Z., & Vanthienen, J. (2020). Text2Dec: Extracting Decision Dependencies from Natural Language Text for Automated DMN Decision Modelling. *Business Process Management Workshops (BPM 2020)*. Lecture Notes in Business Information Processing, 397. DOI: 10.1007/978-3-030-66498-5_27.
13. Goossens, A., De Smedt, J., & Vanthienen, J. (2023). Extracting Decision Model and Notation models from text using deep learning techniques. *Expert Systems with Applications*, 211, 118667. DOI: 10.1016/j.eswa.2022.118667.
14. Bork, D., Ali, S. J., & Dinev, G. M. (2023). AI-Enhanced Hybrid Decision Management. *Business & Information Systems Engineering*, 65(2), 179-199. DOI: 10.1007/s12599-023-00790-2.
15. Abedi, S., & Jalali, A. DMN-Guided Prompting: A Low-Code Framework for Controlling LLM Behavior. 2025. *arXiv preprint* arXiv:2505.11701. DOI: 10.48550/arXiv.2510.16062

Received (надійшла) 14.11.2025

Відомості про авторів / About the Authors

Чередніченко Ольга Юріївна (Cherednichenko Olga) – доктор технічних наук, доцент, професор кафедри програмної інженерії та інтелектуальних технологій управління, Національний технічний університет «Харківський політехнічний інститут», м. Харків, Україна; e-mail: Olga.Cherednichenko@khpi.edu.ua; ORCID: http://orcid.org/0000-0002-9391-5220.

Маляренко Владислав Вікторович (Maliarenko Vladyslav) – аспірант кафедри управління проєктами в інформаційних технологіях, Національний технічний університет «Харківський політехнічний інститут», м. Харків, Україна; e-mail: Vladyslav.Maliarenko@cs.khpi.edu.ua; ORCID: http://orcid.org/0009-0009-6064-061X.