MASTER'S THESIS
Master's Degree in Artificial Intelligence

Automatic Generation of BPMN Processes Using Artificial Intelligence and a DSL

Student: Ignacio González Peris

Supervisors: Vicente Octavio Herrera García

Diego Canales Aguilera

Seville, June 2026.

Abstract

The recent rise of Large Language Models (LLMs), such as ChatGPT, Claude, and Gemini, has encouraged the automated generation of business-process models from natural language. However, converting a description directly into a Business Process Model and Notation (BPMN) 2.0 representation entails syntactic, structural, and semantic difficulties, especially when the output requires verbose formats with many gateways and deep nesting.

The hypothesis is stated operationally: in the evaluated configurations, a *pipeline* based on a Domain-Specific Language (DSL) should achieve a higher end-to-end success rate and higher failure-penalized quality than direct BPMN-XML generation. To test it, a system is designed and implemented in which the LLM generates a simplified textual DSL and a deterministic compiler parses this representation, validates its constraints, and produces the final BPMN 2.0 model. The main evaluation examines robustness, five structural indicators, automatic coverage of expected constructions, and stability over fifteen synthetic processes and three models, using a balanced matrix of 15 processes $\times$ 3 replicates per model.

The results show high internal-analysis success (`semantic_success`, indicating compilability and internal validation rather than semantic equivalence to the description) and rendering success. Quality declines in the strata defined as more difficult, while the main omissions concern collaborations, *pools*, and message flows. GPT-5.5 achieves the highest descriptive mean; however, the process-paired analysis detects no differences among the three models (Friedman, $p = 0.516$).

As an experimental extension, GPT-5.5 is evaluated with two complete configurations: DSL v3.1 plus compiler and direct BPMN-XML generation using a short zero-shot *prompt*. XML achieves higher conditional quality (0.927) and coverage (0.983), whereas DSL achieves a higher success rate (1.00 versus 0.87) and failure-penalized quality (0.875 versus 0.804). Because the *prompts* and post-processing are not equivalent, the result provides partial support for the robustness hypothesis but does not isolate the causal effect of the representation or demonstrate complete semantic fidelity.

Keywords:

- Business Process Model and Notation (BPMN)
- Large Language Model (LLM)
- Domain-Specific Language (DSL)
- Prompt engineering
- Process generation
- Automated evaluation
- Natural language
- Artificial Intelligence
- BPMN Sketch Miner
- BEF4LLM

Contents

1 Introduction **1**
- 1.1 Problem context . . . 1
- 1.2 Motivation . . . 1
- 1.3 Basic BPMN concepts . . . 2
- 1.4 Problem formulation . . . 4
- 1.5 Contributions . . . 4
- 1.6 Document structure . . . 5

2 Context and State of the Art **7**
- 2.1 Introduction to conversational process modelling . . . 7
- 2.2 Technological evolution: from traditional approaches to LLMs . . . 8
- 2.3 The challenge of direct generation and XML verbosity . . . 8
- 2.4 Intermediate representations and Domain-Specific Languages (DSLs) . . . 10
- 2.5 Advanced prompt-engineering strategies . . . 11
- 2.6 Multi-agent orchestration . . . 11
- 2.7 Quality evaluation: the 3C system and symbolic verification . . . 11
- 2.8 Conclusions and future of AI-assisted modelling . . . 12

3 Methodology **13**
- 3.1 Model selection . . . 14
- 3.2 Access mode: web interface versus programmatic access . . . 14
- 3.3 Compensation: multiple replicates . . . 15
- 3.4 *System prompt* as a controlled variable . . . 15
- 3.5 Minimal post-processing . . . 15
- 3.6 Absence of a closed correction loop . . . 16

4 System Development **17**
- 4.1 Overview: a monorepo with a shared engine . . . 17
- 4.2 The `bpmn-core` engine: from textual description to diagram . . . 18
- 4.3 The DSL as a readable intermediate representation . . . 19
  - 4.3.1 Lexical and syntactic analysis . . . 22
  - 4.3.2 Semantic analysis . . . 22
- 4.4 BPMN-XML emission . . . 23

4.5 Layout and rendering subsystem . . . 23
4.6 Validation and diagnostics . . . 24
4.7 Evolution of the *system prompt* and *pool–lane* mapping . . . 24
4.8 Applications built on the engine . . . 25
4.8.1 `apps/tfm-lab`: research laboratory . . . 25
4.8.2 `apps/company-web`: demonstration application . . . 25
4.9 Formative feedback from users and Metadev . . . 26
4.10 Evaluation pipeline and test suite . . . 26
4.11 Synthesis . . . 27

5 Evaluation **28**
5.1 Evaluation objective . . . 28
5.2 Experimental design . . . 29
5.3 Metrics used . . . 30
5.3.1 Relation to BEF4LLM . . . 30
5.3.2 Robustness . . . 33
5.3.3 Structural quality . . . 33
5.3.4 Automatic construction coverage . . . 34
5.3.5 Estimated token consumption . . . 34
5.3.6 Engine diagnostics . . . 34
5.4 Dataset overview . . . 35
5.5 RQ1: Robustness of the DSL approach . . . 36
5.6 RQ2: Comparison of quality across models . . . 36
5.7 RQ3: Effect of process difficulty . . . 38
5.8 RQ4: Presence of expected constructions . . . 40
5.8.1 Coverage of expected constructions . . . 40
5.8.2 Diagnostic taxonomy . . . 41
5.9 RQ5: Comparison of v3.1, v4, and v5 configurations . . . 41
5.10 RQ6: Description length (unevaluated question) . . . 43
5.11 RQ7: Stability across runs . . . 43
5.12 RQ8: *Zero-shot* comparison with direct BPMN-XML generation . . . 44
5.13 Formative observations and applicability limits . . . 47
5.14 Statistical significance . . . 47
5.15 Threats to validity . . . 48
5.15.1 Internal validity . . . 48
5.15.2 Construct validity . . . 48
5.15.3 External validity . . . 48
5.15.4 Statistical validity . . . 48
5.16 Reproducibility . . . 49
5.17 Evaluation conclusions . . . 50

6 Conclusions and Future Work **51**
6.1 Conclusions . . . 51
6.2 Future work . . . 52
6.2.1 Improving collaboration between participants . . . 52
6.2.2 Refining the *system prompt* . . . 52
6.2.3 Closed-loop self-repair . . . 52
6.2.4 Extending the *zero-shot* comparison . . . 52
6.2.5 Reference-based semantic evaluation . . . 53
6.2.6 Symbolic verification and orchestration . . . 53
6.2.7 Extending external validity . . . 53
6.2.8 API access and controlled hyperparameters . . . 53
6.2.9 Expanding the model set . . . 53

7 Repository Organization **55**
7.1 Consolidation in a monorepo . . . 55
7.2 Integration of the evaluation package . . . 55
7.3 Documentation and automation . . . 55

List of Acronyms **57**

Bibliography **58**

A Additional BPMN Material **63**
A.1 DSL and *system prompt* specification . . . 63
A.2 Complete results and experimental traceability . . . 64
A.3 End-to-end example . . . 64
A.4 Gallery of *canon* diagrams . . . 66
A.5 Diagnostic-code catalogue . . . 66
A.6 Evolution of the *system prompt* . . . 68
A.7 Synthetic process set . . . 69

List of Figures

1.1 Legend of the basic elements and symbols used in the BPMN 2.0 standard notation. 3
1.2 Example of a BPMN 2.0 diagram for a new-employee onboarding process. It shows the sequential structure, decision gateways, and organization into department-assigned lanes. . . . 3
1.3 User interface of `CompanyWeb`, a demonstration web application developed to support conversational modelling and process documentation with AI assistance (tested at SGS Productivity). . . . 5

4.1 System architecture: from natural language to a renderable BPMN 2.0 model. The external LLM generates the intermediate textual DSL, which a deterministic compiler transforms through `parseDsl`, `emitBpmnXml`, `renderSemanticXml`, and `validateBpmnModel`. Generation is *single-shot*: the engine does not feed diagnostics back to the model. . . . 19
4.2 BPMN 2.0 diagram generated from the *canon-1* example. . . . 22

5.1 Robustness by model . . . 36
5.2 Composite mean of structural proxies by configuration . . . 37
5.3 Distribution of task and gateway counts by model. The three models have similar verbosity, with some outliers in complex processes. . . . 37
5.4 Relationship between total node count and control-flow complexity of generated diagrams, by model and difficulty level. . . . 38
5.5 Mean number of tasks and gateways by model and difficulty level. Diagram size increases markedly in the stress level. . . . 39
5.6 Undermodeling in complex processes . . . 39
5.7 Construction coverage by v3.1, v4, and v5 configuration . . . 42
5.8 Robustness, conditional quality, and effective quality in the global comparison . . 44
5.9 DSL and direct XML internal-analysis success rate by difficulty . . . 45
5.10 Conditional and penalized quality by difficulty . . . 45
5.11 Input–output token asymmetry by representation . . . 46

7.1 Relationship between raw data, code, derived artifacts, and the thesis. The chain must record the version and hashes at each transition. . . . 56

A.1 BPMN model of the waste-treatment plant. . . . 65

List of Tables

2.1 Synthetic comparison of related proposals and the scope of this work. 7
2.2 Technical comparison of fine-tuning and prompt engineering. 9

4.1 Main constructions of the intermediate DSL. 21
4.2 Differences between the inspiring basis and the implemented extensions. 21

5.1 Research questions of the evaluation 29
5.2 Difficulty levels of the synthetic corpus 30
5.3 Operational metric subset inspired by BEF4LLM 32
5.4 Pipeline robustness metrics 33
5.5 Compilation-state distribution of the DSL v3.1 subset 35
5.6 Models with complete experimental coverage 35
5.7 Internal-analysis and rendering success rate by configuration. 36
5.8 Composite mean of five structural proxies by configuration (mean $\pm$ standard deviation over 45 runs) 37
5.9 Effect of process difficulty 39
5.10 Mean coverage of expected features 40
5.11 Most frequently omitted expected features 41
5.12 Diagnostic taxonomy by model 42
5.13 Mean feature coverage by *system prompt* version 42
5.14 Stability across repeated runs 43
5.15 GPT-5.5 robustness, quality, and coverage by configuration 44
5.16 DSL v3.1 versus direct XML by difficulty 45
5.17 Textual token estimate per GPT-5.5 generation 46
5.18 Global statistical-significance test 47
5.19 Process-paired comparisons using Wilcoxon and Holm correction 47
5.20 Evaluation reproducibility artifacts 49

A.1 Relationship among the experimental sets used. 64
A.2 Canonical diagrams used as regression tests. 66
A.4 Model and *layout* subsystem diagnostics. 67
A.3 Lexical and syntactic analyzer diagnostics. 68
A.5 Functional evolution of the *system prompt*. 68

A.6 Corpus processes, difficulty, and language. 70

Chapter 1

Introduction

1.1 Problem context: the evolution of process modelling

Business-process modelling turns operational descriptions into representations that support analysis, standardization, and automation. The reference notation used in this work is *Business Process Model and Notation* (BPMN), a graphical standard for representing processes and collaborations. Producing a model generally requires both domain knowledge and command of the notation's rules; initial capture and model review therefore still depend substantially on specialized analysts [1].

Large Language Models (LLMs), including the GPT, Claude, and Gemini families, can transform natural-language instructions into structured outputs. In BPM, however, quality depends on the task and the evaluation criterion: a model may produce well-formed XML while omitting participants, decisions, or process exceptions [2]. The literature documents syntax errors, inconsistent references, and incomplete structures in direct BPMN-XML generation [1, 3]. XML verbosity and nesting increase the syntactic burden; referential consistency and behavioral correctness pose additional problems that cannot be solved simply by shortening the output.

1.2 Motivation: democratization and efficiency in process design

This research addresses the measurable gap between a business description and a BPMN artifact that can be imported, rendered, and evaluated without manual repair. Conversational interfaces can reduce the need to know BPMN-XML syntax and facilitate an initial model, although they do not eliminate human review. BPMN-Chatbot++ exemplifies this line of work through a natural-language interface for creating collaboration diagrams [4]. Recent studies compare generative-chatbot-assisted modelling with human modelling and propose incremental conversational processes, thereby providing a clearer account of their capabilities and limitations [5, 6].

From an architectural perspective, in-context learning through instructions and examples makes it possible to adapt a general model without changing its parameters. Compared with fine-tuning, it has advantages and costs that depend on the model, instruction length, number

of inferences, and degree of reuse [2]. In this work, generative capability is combined with a deterministic symbolic component: the LLM generates an intermediate representation based on a Domain-Specific Language (DSL), which the engine parses, validates, and transforms into BPMN 2.0. A DSL is a language with a deliberately restricted vocabulary and set of rules, designed to describe one type of content—here, business processes—and its simplicity makes it possible to check automatically whether a text complies with it. The division of roles is therefore explicit: the LLM is the non-deterministic part of the *pipeline*, since its output may vary between runs, whereas the compiler is deterministic and always produces the same result for the same input. The implementation includes neither a multi-agent architecture nor complete formal soundness verification, where soundness is understood as a behavioral property requiring, among other aspects, proper termination and the absence of dead tasks. The scope is limited to studying the extent to which the DSL, versioned instructions, and compiler enable reproducible and automatically evaluable results.

1.3 Basic BPMN concepts: notation and diagram reading

To support readers unfamiliar with BPMN, this section summarizes its basic elements according to the OMG BPMN 2.0.2 specification [7]. The standard defines the semantics and shape of symbols; color and other styling depend on the tool.

Informally, a BPMN diagram can be read as a standardized flowchart: rectangles represent tasks, diamonds represent decisions, and circles mark where the process starts and ends, all connected by arrows subject to precise rules.

The elements are grouped into different categories:

- **Flow objects:** events, represented by circles, indicate start, intermediate, or end occurrences and may include timer, message, error, or other definitions; activities, represented by rounded rectangles, describe work; and gateways, represented by diamonds, control branching and merging.
- **Connecting objects:** a sequence flow orders activities within the same process and cannot cross a pool boundary. A message flow represents communication between different participants and does not replace the sequence flow within a participant.
- **Participants and responsibilities:** a *pool* delimits a participant; a *lane* subdivides that participant to assign responsibilities to roles, departments, or systems.
- **Data and artifacts:** data objects and stores, annotations, and associations provide complementary information without being confused with the control flow.

An exclusive (XOR) gateway selects one alternative among two or more branches according to their conditions and may define a default route; it is not limited to binary yes/no questions. A parallel (AND) gateway activates or synchronizes concurrent branches.

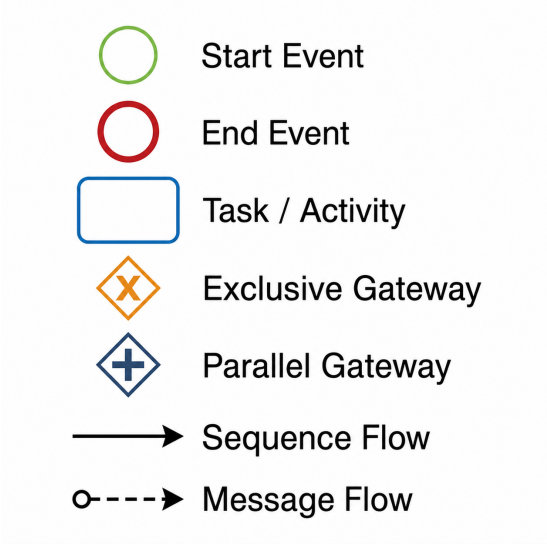

Figure 1.1: Legend of the basic elements and symbols used in the BPMN 2.0 standard notation.

Figure 1.1 presents a visual legend of the classic symbols used as a reference for interpreting the structure of any process represented in this notation.

Diagrams may be organized into *pools* (participants) and *lanes*. For example, Customer and Company would be represented as two *pools* connected by message flows; within the Company participant, Sales and Logistics *lanes* would distribute responsibilities without becoming independent participants. Thus, each activity is interpreted both through its position in the flow and through the party responsible for executing it.

To illustrate how these components interact in a real scenario, Figure 1.2 models the onboarding process for a new employee. The diagram shows the process start, the coordination of tasks distributed across lanes associated with different roles or departments (such as Human Resources and the direct supervisor), the use of gateways to validate approvals, and successful flow completion.

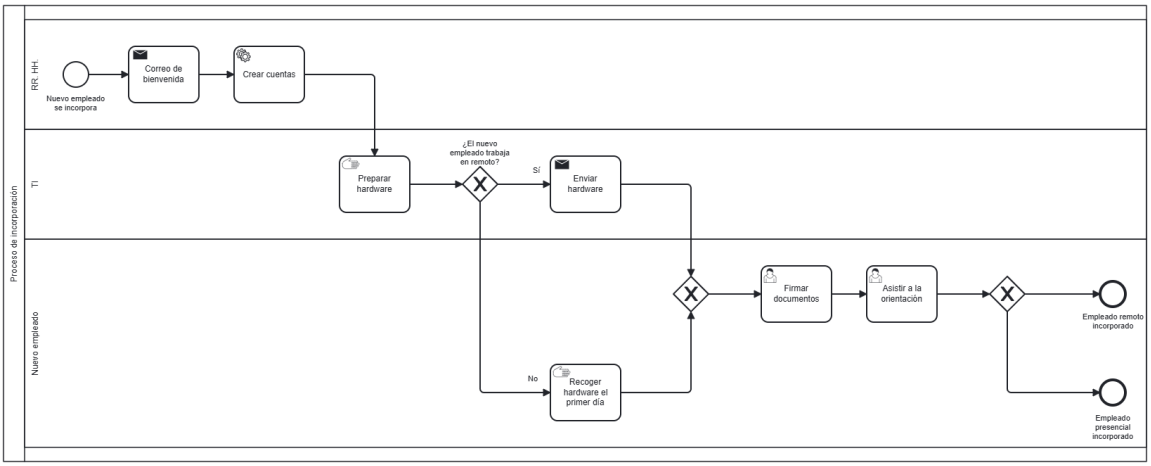

Figure 1.2: Example of a BPMN 2.0 diagram for a new-employee onboarding process. It shows the sequential structure, decision gateways, and organization into department-assigned lanes.

1.4 Problem formulation: the gap between natural language and BPMN syntax

The problem studied is the difficulty of preserving process content and formal-notation constraints simultaneously. Direct XML generation may produce syntax errors, inconsistent references, and incomplete flow structures [3]. The operational hypothesis is that, in the evaluated configurations, the DSL-plus-compiler *pipeline* will achieve a higher end-to-end success rate and higher failure-penalized quality (the mean quality score that assigns zero to failed runs) than direct BPMN-XML generation. Structural quality is defined in Chapter 5 as the mean of five automatic indicators (M2, M3, M4, M5, and M8), not as a semantic-fidelity or soundness test.

The hypothesis is broken down into questions concerning robustness, model–platform differences, difficulty, construction coverage, the joint evolution of instructions and system, stability, and DSL–XML comparison. The process, rather than each isolated replicate, is the primary unit for comparisons: the three replicates of a process are aggregated before statistical analysis.

An adapted subset of BEF4LLM is used [3]. The selected validity and structural metrics are reference-free, whereas the original semantic metrics compare against human or reference models; in this thesis they are replaced by automatic detectors for expected constructions, with the limitations discussed in Section 5.3.4. POWL, MAO, the 3C system, and symbolic verifiers are analyzed as prior work but are not part of the implemented architecture [1, 8, 9]. The system uses a single LLM delivery, an internal DSL, a deterministic compiler, and automatic metrics.

1.5 Contributions of the work

This work makes the following original contributions to the field of AI-assisted automated business-process modelling:

1. **Evolution of a DSL derived from BPMN Sketch Miner.** The first version used the textual syntax documented by BPMN Sketch Miner as a reference [10]; Table 4.2 distinguishes that basis from the extensions introduced here. In v4, the instructions were extended with readability criteria and event handling. In v5, the instructions, syntax, parser support, and engine support changed simultaneously: explicit *pool* and *lane* declarations were added to represent independent participants and their interactions. The evolution is therefore not exclusively a prompt evolution, nor is the representation presented as created from scratch.
2. **Design and implementation of a complete generation and validation *pipeline*.** The developed chain consists of: (1) an LLM guided by a structured *system prompt*, (2) a deterministic parser that converts the DSL into an internal representation, (3) syntactic and semantic validations performed by the engine, and (4) a BPMN 2.0 emitter and renderer. Although these validations detect specific structural defects and warnings about possible bad practices, they are not equivalent to a complete formal soundness proof.

3. **Traceable and extensible evaluation protocol.** The evaluation compares three model–platform configurations over fifteen synthetic processes and three replicates per process. It includes comparisons among configurations v3.1, v4, and v5 and a comparison with direct XML generation.

4. **Prototype oriented towards professional use.** `CompanyWeb` (Figure 1.3) integrates the engine into a demonstration application that SGS Productivity colleagues tested exploratorily. The activity collected observations about usability and layout, but because participants, tasks, instrument, and results were not documented, it is presented as formative feedback rather than empirical validation of usefulness. Case studies of LLMs and process modelling in organizations exist [11], although evaluation with non-expert users remains limited.

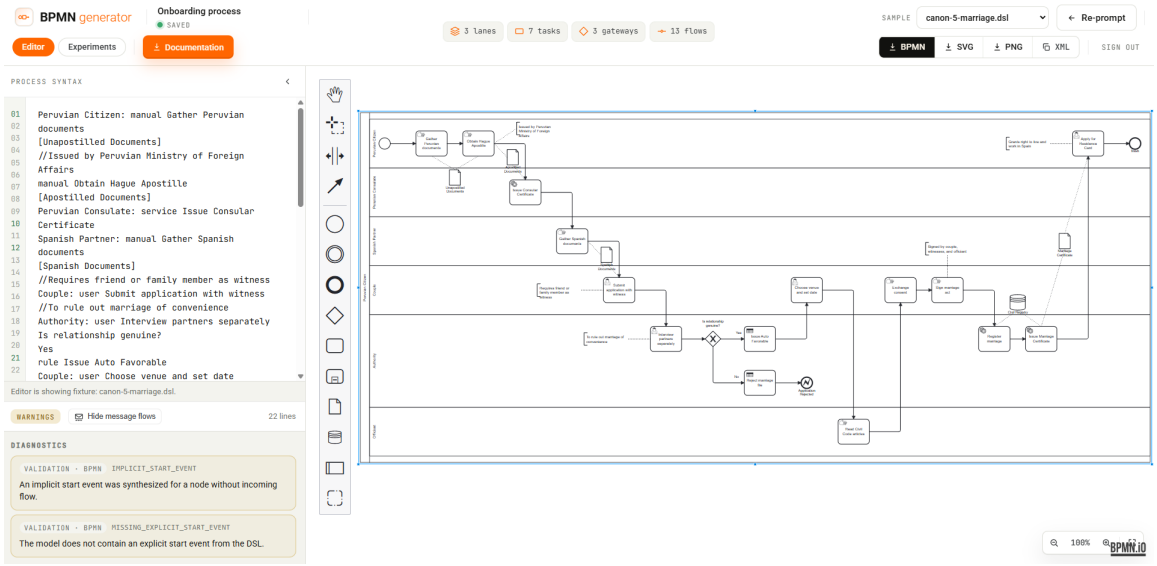

Figure 1.3: User interface of `CompanyWeb`, a demonstration web application developed to support conversational modelling and process documentation with AI assistance (tested at SGS Productivity).

The collaboration with Metadev [12] provided technical mentoring during DSL specification and design. This contribution is acknowledged as formative expert feedback: by itself, it does not constitute scientific validation, which would require a documented protocol, criteria, participants, and results.

1.6 Document structure: a roadmap of the research

To facilitate tracking the thesis's logical progression, the document is structured as follows:

1. **State of the Art**: a critical analysis of LLMs in BPM and prompt-engineering techniques in response to the scarcity of data in the sector.

2. **Methodology**: the non-deterministic stage of the *pipeline*—DSL generation by the LLM—and its experimental controls: compared models, replicates, a fixed *system prompt*, and minimal post-processing.
3. **System Development**: the monorepo architecture, the `bpmn-core` engine for DSL→BPMN transformation, and the layout, validation, and diagnostic subsystems; that is, the deterministic stage of the *pipeline*.
4. **Evaluation and Results**: robustness tests, structural proxies, automatic construction coverage, and stability, partly drawing on the BEF4LLM framework [3].
5. **Conclusions and Future Work**: a synthesis of the findings and new directions for applying generative AI to business processes.
6. **Repository Organization**: the traceability between code, raw data, and derived artifacts, without presenting the mutable repository as an immutable snapshot of the results.

This structure leads to a review of the technical literature, which is necessary to understand the theoretical and technological foundation supporting the proposal.

Chapter 2

Context and State of the Art

This chapter reviews the state of the art on automated BPMN-model generation using Large Language Models (LLMs). The analysis ranges from conversational modelling to quality-evaluation frameworks that contextualize the proposal developed in this work.

Table 2.1 summarizes the relationship among the closest proposals and avoids treating them as a catalogue of equivalent tools.

Proposal	**Representation and iteration**	**Validation and relevant limitation**
ProMoAI/POWL [1]	Intermediate POWL and conversational modelling.	Guarantees within the language's scope; expressiveness constrained by POWL.
BPMN-Chatbot++ [4]	JSON/BPMN with guided refinement.	Specialized services; greater infrastructure and system-dependent evaluation.
MAO [8]	Multi-agent generation with iterative review.	Review and testing; adds orchestration calls and decisions.
BEF4LLM [3]	Candidate BPMN and reference models.	39 metrics; semantic component depends on references.
This thesis	Derived DSL plus compiler, without repair.	XSD and automatic proxies; no human reference or formal *soundness*.

Table 2.1: Synthetic comparison of related proposals and the scope of this work.

2.1 Introduction to conversational process modelling

Conversational modelling uses natural language to create or modify process representations. It does not necessarily replace human editing and validation: it may support initial generation, iterative refinement, or information extraction, tasks that require different metrics. Recent studies show that LLMs can address several of these tasks without task-specific fine-tuning, although results depend on the model, corpus, and dimension evaluated [13, 14]. This line of work includes *Large Process Models*, which combine language models with structured process knowledge, and systems that generate BPMN or check style guidelines [15, 16].

ProMoAI [1] and BPMN-Chatbot++ [4] present alternatives for generating processes and

multi-participant collaborations. The line continues with BPMN Assistant and approaches for creating actionable models conversationally [17, 18]. Benefits motivating these systems, still requiring empirical validation in each context, include:

- **Accessibility for domain experts:** process owners can create models without prior technical training in BPMN-XML syntax.
- **Potential operational efficiency:** an initial version can be obtained without manually writing the complete BPMN-XML; the actual saving depends on subsequent repair.
- **Iterative refinement:** models can be modified through conversation when the system implements a review loop.

2.2 Technological evolution: from traditional approaches to LLMs

Process-extraction techniques have evolved from Natural Language Processing (NLP) methods based on rules and traditional text mining to transformer architectures. This evolution has enabled a shift from purely syntactic treatment to semantic understanding of business context. Early systems for generating models from text relied on linguistic rules and syntactic analysis, offering limited coverage and depending heavily on wording [19]. This difficulty prompted the creation of specialized benchmarks for extracting processes from text [20]. The gap has gradually narrowed with later NLP methods [21] and, more recently, systems that automate end-to-end generation from textual requirements [22].

According to Busch et al. [2], the current paradigm revolves around two alternatives: fine-tuning and in-context learning. Within specialization-oriented work, some recent studies fine-tune LLMs for process modelling through reinforcement learning with verifiable rewards [23]. Table 2.2 summarizes the main technical differences between the alternatives:

2.3 The challenge of direct generation and XML verbosity

One barrier to automating the complete workflow is the failure rate observed when LLMs generate BPMN-XML directly. Lauer et al. [3] document structural errors even with explicit instructions. Hörner et al. [24] systematize these challenges and propose an evaluation framework, while other studies use multi-stage architectures [25].

XML verbosity and nesting increase output length and the opportunities for improperly closed tags or inconsistent references. Several failure levels must nevertheless be distinguished: malformed XML, XSD non-compliance (*XML Schema Definition*, the official schema specifying which elements and attributes a valid BPMN-XML file may contain), broken references, missing BPMN DI, import failure, incomplete structure, and incorrect behavior. Well-formed XML may contain deadlocks or dead tasks; verbosity alone therefore does not explain behavioral errors.

Feature	Fine-tuning	Prompt Engineering (In-context Learning)
Approach	Parameters are updated using labelled data.	The model remains “frozen” and is guided through instructions.
Computational cost	Requires an additional training phase; its cost is amortized according to the number of uses.	Avoids retraining, but long and repeated instructions increase inference cost.
Data requirements	Depend on the technique, model, and desired degree of specialization.	May work with zero or few examples, although quality depends on the instructions.
Flexibility	Can specialize behavior and reduce instruction length.	Makes it possible to change the task by changing the context without altering parameters.
Control and reproducibility	Fixes a trained artifact when the checkpoint and environment are preserved.	Depends on the served model, prompt, platform, and possible provider changes.
Amortized cost	May be favorable when training is reused at scale.	May be favorable for prototypes or small volumes; input, output, and caching must be included.

Table 2.2: Technical comparison of fine-tuning and prompt engineering.

The following fragment illustrates this type of failure through defective BPMN-XML characteristic of direct generation. It contains an unclosed tag and a sequence flow referring to a nonexistent node, making the resulting model invalid or disconnected.

```
<bpmn:task id="Task_1" name="Validate order">
<!-- unclosed <task> tag -->
<bpmn:sequenceFlow id="Flow_1"
    sourceRef="Task_1" targetRef="Task_99"/>
<!-- Task_99 does not exist: dangling reference -->
```

The recurrence of these structural errors, also systematically documented by Lauer et al. [3], is precisely what motivates the use of a more restricted intermediate representation.

2.4 Intermediate representations and Domain-Specific Languages (DSLs)

Before presenting specific proposals, it is useful to define a Domain-Specific Language (DSL). Unlike a general-purpose language, a DSL is designed for a delimited domain and uses a reduced syntax and vocabulary adapted to that domain's concepts. This specialization offers a more concise and readable representation than a generic format such as XML. It also makes it possible to restrict the allowed constructions in advance, simplifying both LLM generation and automatic validation. A familiar example is a spreadsheet formula: it cannot express any program, but it expresses calculations compactly and in an easily verifiable form.

To reduce problems arising from XML verbosity, current research proposes DSLs and structured intermediate formats:

- **Structured JSON:** BPMN-Chatbot++ uses it to manage complexity associated with pools, lanes, and message flows in inter-organizational scenarios [4].
- **POWL:** This language, used by ProMoAI, provides soundness guarantees within its semantics and the subset of models it can express. Its "parallelism by default" logic treats tasks as concurrent unless explicit dependencies are established [26].

Advantages and costs of DSLs compared with XML: a restricted representation may reduce output length and enable deterministic validation of the constructions it accepts. These properties do not by themselves guarantee soundness or semantic fidelity. In return, a DSL limits expressiveness, requires the grammar and compiler to be maintained, and may transfer the engine's design biases to the final artifact. Recent comparative studies show that textual representations can be competitive with XML on certain metrics without demonstrating universal superiority [27, 28].

2.5 Advanced prompt-engineering strategies

Automated-modelling quality depends on instruction design, but adding rules or examples does not necessarily yield a monotonic improvement [1, 29]. This work distinguishes:

1. **Role Prompting:** assigning the LLM the role of a senior BPM expert and process owner to establish the appropriate semantic context.
2. **Knowledge Injection:** incorporating the DSL's technical grammar (such as POWL or JSON schemas) directly into the prompt to constrain the output to the required format.
3. **Few-shot Learning:** providing robust examples, such as the bicycle-manufacturing process [1], so that the model learns the relation between textual description and logical structure.
4. **Negative Prompting:** using preventive instructions to avoid frequent errors, such as irreflexivity violations in dependencies or local choices between activities instead of choices among complete process routes.

In addition to these specific techniques, general-purpose prompting strategies have been proposed to extract process-model information from text using different LLMs [30].

2.6 Multi-agent orchestration

Multi-agent orchestration proposals have emerged in process modelling, including MAO (*Multi-Agent Orchestration*) [8]. MAO distributes generation, refinement, review, and testing among specialized agents. This separation allows control and repair cycles to be introduced, but adds calls, orchestration decisions, and potential sources of variation that must be recorded to reproduce the result.

This architecture is included as context and as a possible development path. The thesis prototype does not implement specialized agents or self-correction; it uses a single DSL delivery followed by deterministic compilation and validation.

2.7 Quality evaluation: the 3C system and symbolic verification

Validating AI-generated models requires evaluation frameworks capable of integrating the business perspective with technical rigor:

- **BEF4LLM:** proposes a multidimensional evaluation of LLM-generated BPMN models through file-validity, syntactic-quality, pragmatic-quality, and semantic-quality metrics [3]. Several measures are close to those applied in this work, particularly those concerning BPMN-XML validity, connectivity, labelling, complexity, and element structure. The

evaluation chapter describes the framework's specific adaptation and the differences arising from the absence of a reference diagram.

- **3C dimensions:** Drakopoulos et al. [9] organize evaluation into Clarity (visual understanding), Correctness (syntactic rules), and Completeness (coverage of the original text).
- **Symbolic AI and *model checking*:** independent services can verify formal properties within defined scopes [4]. The cited tools include:
  - **S³ (Soundness Checker):** verifies safeness, soundness, and message-relaxed soundness within the supported BPMN subset and formal semantics.
  - **vPAV (viadee Process Application Validator):** analyzes data flows and identifies anomalies such as variables that are written but never read.

Lin et al. [8] report improvements of 89%, 61%, 52%, and 75% on four specific datasets and metrics. These figures contextualize orchestration's potential but do not amount to general technical superiority over human modellers.

S³ and vPAV verifiers, POWL, and the 3C system are not included in the evaluated implementation. They require additional infrastructure or human evaluation. The BEF4LLM adaptation used here retains automatic reference-free indicators but does not reproduce its semantic metrics based on comparison with human models. The developed validations check connectivity, degrees, labels, gateway pairing, construction presence, and diagnostics; they do not establish that deadlocks are absent from every possible trace.

2.8 Conclusions and future of AI-assisted modelling

The literature reflects a shift towards hybrid systems integrating generation, intermediate representations, validation, and, in some cases, iterative repair. In this context, the thesis analyzes one specific component: the feasibility of a textual DSL generated in a single delivery and transformed by a deterministic compiler. POWL, MAO, 3C, and formal verifiers represent broader alternatives, whereas the experimental work focuses on robustness and structural indicators of the implemented approach.

Chapter 3

Methodology

This chapter defines a *pipeline*—a sequence of stages executed in a fixed order—that starts from a natural-language prompt and attempts to produce a BPMN 2.0 diagram through an LLM and a deterministic compiler. The underlying idea is straightforward. Rather than generating BPMN-XML directly, the model produces an intermediate representation in a textual DSL; the compiler parses and validates that representation, serializes the resolved model as BPMN-XML, and adds the graphical information required to render the diagram. Figure 4.1 (Chapter 4) provides an overview of the flow. The following sections describe its stages and the methodological constraints governing them.

It is useful to clarify the scope of this chapter. Of the two parts of the *pipeline*, it addresses the non-deterministic one: how the LLM is prompted to generate the DSL and which protocol decisions—model selection, replicates, a fixed *system prompt*, minimal post-processing, and the absence of a correction loop—make it possible to study it under controlled conditions. The deterministic part, namely the compiler that parses, validates, and transforms the DSL, is described in Chapter 4; the measurement protocol and its metrics are presented in Chapter 5.

In the first stage of the *pipeline*, an informal process specification is converted into a document conforming to an internal DSL derived from the textual syntax documented by BPMN Sketch Miner [10]. This wording indicates provenance, not full compatibility: later versions add project-specific syntax and compiler behavior. This conversion is performed by an LLM conditioned by a fixed *system prompt*. Version v3.1 specifies the DSL rules and limitations, anti-patterns, several complete examples, and a final checklist; Appendix A reproduces its full content. This artifact is versioned in the project repository at `apps/tfm-lab/prompts/system/bpmn_sketch_miner_system_prompt_v3_1.md` [31]. Its content builds on the textual syntax of BPMN Sketch Miner [10], extended with expert knowledge during the development of this work. This type of conditioning without retraining falls within the *in-context learning* paradigm, introduced at scale by Brown et al. [32] and subsequently systematized by Liu et al. [33]. The development sequence recorded for the controlled prompt is v1 $\rightarrow$ v2 $\rightarrow$ v3 $\rightarrow$ v3.1; no v1.2 is assumed. Versions v1 and v2 were development iterations rather than stable experimental baselines, and v2 still exhibited fragment-boundary failures. Version v3 made trace-based notation the default and introduced the anchor-boundary rule and the AP-6/AP-7 anti-patterns. Version v3.1 then re-audited the evidence: AP-6 remained an observed failure, whereas AP-

7 was classified as a predicted but, at that point, unobserved risk because the alleged real case was a misdiagnosed return start event. Later experimental AP-7 diagnostics are reported separately in Section 5.8.2. Thus, v3.1 is the first sufficiently validated version to serve as a controlled baseline, not the first version capable of producing any valid diagram.

3.1 Model selection

The main comparison includes three configurations available to end users: GPT-5.5 Thinking, Gemini 3.5 Flash Thinking, and Claude Opus 4.7 [34, 35, 36]. They were selected because they were available in 2026, belong to different providers, and offer reasoning modes. Each configuration has 45 runs: fifteen processes with three replicates each. The comparison jointly characterizes model and platform; it does not allow differences to be attributed exclusively to model weights. Opus 4.8 Medium has a partial v3.1 matrix and is retained as an exploratory result rather than included in the main comparison.

3.2 Access mode: web interface versus programmatic access

All models were invoked through their respective public web interfaces instead of an application programming interface (API). The analysis could have been conducted through either the web interface or an API, and the former was chosen. This was a deliberate decision grounded in ecological validity in the classical sense of Shadish, Cook, and Campbell [37]. Ecological validity expresses the extent to which a study's conditions correspond to the real-world contexts to which its results are intended to generalize. The greater the similarity between the experimental conditions and the real setting, the greater this validity. In many process-consulting scenarios, a typical profile is that of a functional analyst interacting with an LLM through its *chat*, without integrating it through an API, although in other contexts, such as Metadev's integration into its own tool, access is provided through an API. Studying the web interface therefore captures one of the most representative workflows. A systematic comparison with API access is left for future work.

Adopting this mode explicitly entails giving up several experimental controls available through an API. Specifically:

1. **There is no control over sampling hyperparameters.** The provider internally sets the effective values of *temperature*, *top-p*, and *random seed*, which the user cannot inspect.

2. **The model identifier is a commercial name, not a weight checkpoint.** Labels used in the study, such as "GPT-5.5 Thinking" or "Gemini 3.5 Flash Thinking", may correspond to different weight versions at different times without notifying the user. To mitigate this effect, each experiment records the execution date and the exact label displayed by the interface. Both data items form part of the experiment's unique identifier (`EXP-<process>-<sysprompt>-<model>-<version>-R<n>`).

3. **There is no batching API.** Each generation requires a manual session. This limits the total corpus size, but also ensures that no run contaminates subsequent runs.

4. **There is a risk of context leakage between sessions.** To avoid it, each (process, model) pair is run in a new conversation without reusing the history.

3.3 Compensation: multiple replicates

Without control over the random seed, three operationally isolated replicates are run for each cell (model $\times$ process), R01, R02, and R03. This number allows an initial observation of variability, but does not stabilize its estimate or follow a prior power calculation. The observed variation combines internal sampling, infrastructure, and possible service changes. Accordingly, the process is the primary unit: replicates are aggregated within each process–model pair before comparative tests. This decision is consistent with the sensitivity of LLMs to instruction design described by Sclar et al. [38], without treating three replicates as independent process observations.

3.4 *System prompt* as a controlled variable

In the main matrix, the *system prompt* is fixed so that its changes are not confounded with the comparison between model–platform configurations. The artifact used is `bpmn_sketch_miner_system_prompt_v3_1.md`; 135 balanced runs are filtered with the label `SYSV31`. Keeping the text constant does not control for platform, wrapper, or reasoning-mode differences; therefore, the observed associations are not interpreted as a pure causal effect of the model. Configurations v4 and v5 are analyzed separately in RQ5 for GPT-5.5 and Gemini 3.5. V5 changes not only the instructions but also the DSL's expressive capacity and the engine's support; consequently, the comparison is described as a configuration comparison rather than a prompt-only ablation.

3.5 Minimal post-processing

The LLM's raw output is preserved and normalized through a closed list of operations: trimming outer whitespace and removing a Markdown block delimited by ```, when it wraps the complete response. Raw and normalized outputs are stored separately. Labels, references, and DSL constructs are not corrected. This delimitation makes it possible to audit which part of the artifact comes from the model and which part comes from the normalization performed before parsing.

3.6 Absence of a closed correction loop

The LLM receives no feedback from the parser or renderer. The main matrix measures a single visible delivery without automatic repair. Syntax errors are counted as failures. This design describes the performance of the observed configuration; it is not necessarily a lower bound, since a repair loop could also introduce regressions. When the DSL violates the AP-6 or AP-7 fragment-boundary rule, the error is recorded without reinjecting it into the model. The details of both anti-patterns and their diagnostics are moved to Appendix A.5. Adding a *self-repair* loop, along the lines of Self-Refine [39] or PICARD [40], is left for future work.

Chapter 4

System Development

This chapter presents the software system built during the project. The starting point is a natural-language process description; from it, the LLM produces an intermediate representation in a textual DSL. That output is not transformed directly into a diagram. Instead, it passes through a deterministic compiler—deterministic in the sense that the same input always produces the same output, unlike the LLM—which parses and validates it, generates the corresponding BPMN 2.0, applies automatic layout, and emits quality diagnostics. The same set of functions—DSL analysis, BPMN emission, layout, and validation—is used in three different contexts: a demonstration application for real users, the experimental laboratory, and the evaluation *pipeline*. To prevent divergence between these surfaces, the project was organized as a *monorepo*. The following sections explain this decision and describe the system's main components. Figure 4.1 summarizes their relationship visually.

4.1 Overview: a monorepo with a shared engine

The development result is not a one-off *script*, but a small ecosystem of tools that depend on the same core. For this reason, the code was consolidated as a *monorepo* once configuration v4 reached development stability, as detailed in Chapter 7. A *monorepo* is a single repository grouping related packages and applications so that they can share code and versions. Managed through *pnpm workspaces*, it groups the engine package, the applications that consume it, the evaluation system, and auxiliary documentation and orchestration resources:

```
packages/
  bpmn-core/      Shared engine: DSL -> BPMN
apps/
  tfm-lab/        Laboratory, experiments, and headless mode
  company-web/    Demonstration web application
TFM-eval/         Python evaluation pipeline
docs/             Documentation and design decisions
scripts/          Rendering and evaluation orchestration
```

The complete repository is publicly available at https://github.com/ignagp34/BPMN-DSL-Monorepo [31] and serves as the project's source of truth. The paths cited in this chapter should always be understood relative to that repository.

The practical advantage of this organization is that engine improvements reach every surface without copying logic or maintaining parallel variants. The applications consume only the public API exposed by packages/bpmn-core/src/index.ts. The remaining modules, dsl/, bpmn/, render/, and validation/, are encapsulated as implementation details.

This decision also has an important methodological consequence. Within a pinned repository revision, the diagram seen by a user in the application and the diagram evaluated in the experiments come from the same shared engine. This reduces the risk of divergence between duplicated implementations, although historical reproducibility still requires the exact *commit*, environment, and artifact hashes.

4.2 The bpmn-core engine: from textual description to diagram

The central component is the @text-to-bpmn/core engine. Its task is to take a description written in a derived and extended DSL, designed to be more concise than complete BPMN-XML, and convert it into an importable, laid-out BPMN 2.0 diagram. This formulation does not imply behavioral *soundness* or complete fidelity to the description. To keep the process controlled, the transformation is divided into four stages:

1. **dsl/:** lexical and syntactic analysis, construction of the abstract syntax tree (AST) — the tree structure through which the program represents interpreted text—, and semantic resolution of roles, implicit events, and participants.
2. **bpmn/:** emission of BPMN-XML from the resolved semantic model.
3. **render/:** node placement, orthogonal routing, edge distribution, label positioning, and organization of *pools* and *lanes*.
4. **validation/:** XSD, structural, and graphical checks, together with diagnostic generation; it does not formally verify behavior.

Dividing the engine in this way makes failures easier to interpret. If the problem appears in dsl/, the model output could not be interpreted correctly; if it appears in render/, the conflict belongs to layout and should not be attributed to the LLM's ability to describe the process. This distinction underlies the separation between semantic_success and render_success used in Chapter 5.

The engine API reflects this division. Each phase can be invoked independently, and the end-to-end method generateDiagram conceptually chains the complete flow:

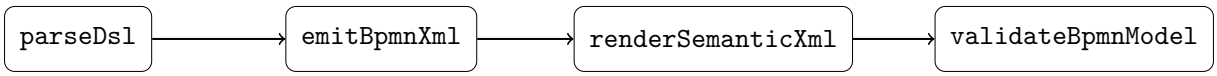

Figure 4.1 shows this end-to-end architecture. The process starts with the natural-language description and the manual transfer of the LLM output. There is no API access or automatic correction loop. From that output, the deterministic `@text-to-bpmn/core` compiler generates a renderable BPMN 2.0 model.

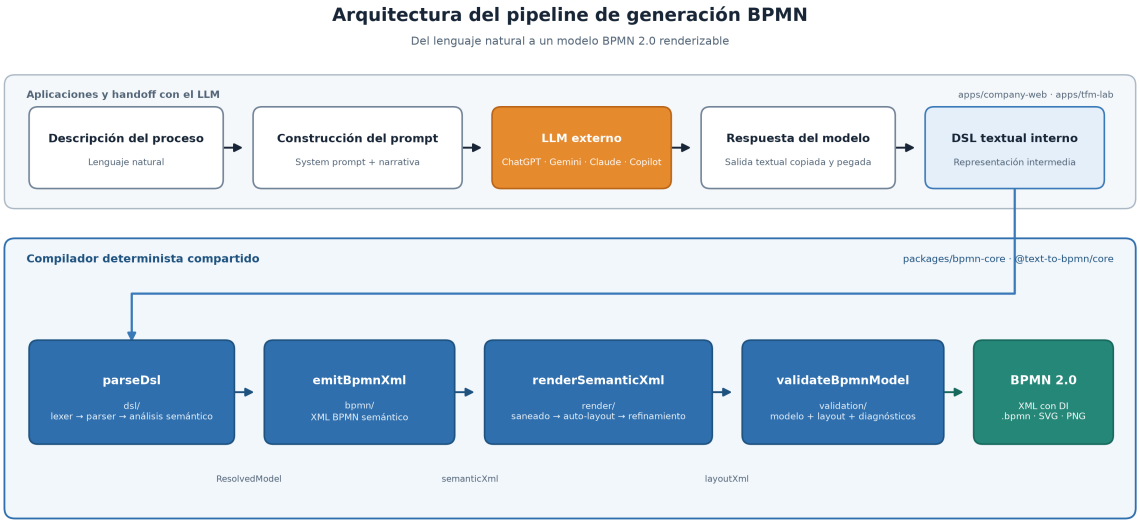

Figure 4.1: System architecture: from natural language to a renderable BPMN 2.0 model. The external LLM generates the intermediate textual DSL, which a deterministic compiler transforms through `parseDsl`, `emitBpmnXml`, `renderSemanticXml`, and `validateBpmnModel`. Generation is *single-shot*: the engine does not feed diagnostics back to the model.

4.3 The DSL as a readable intermediate representation

Choosing an intermediate DSL is one of the decisions that most strongly shapes the approach. It is not merely a technical layer between the LLM and final BPMN: because it is textual, readable, and deterministically verifiable, it makes it possible to inspect the model output before it reaches the diagram. This distinguishes two aspects that would be mixed if BPMN-XML were generated directly: what the LLM described correctly and what the engine manages to lay out. This objective checking capability, anticipated in the introduction and state of the art, supports the automatic measurement protocol presented in Chapter 5.

Instead of asking the model to write BPMN-XML, the system requests a textual, hierarchical representation. Each line describes a step, and that line's position within a trace marks the sequence flow. The grammar appears in the *system prompt* and is implemented in the lexer. Table 4.1 lists the main constructions of the intermediate DSL.

BPMN Sketch Miner treats the textual DSL as a set of traces rather than as a notation in which gateways must be written explicitly. The following example shows two complete traces with a shared prefix and alternative branches:

```
(start New employee joins)
HR: send Welcome email
HR: service Create accounts
IT: manual Prepare hardware
Is the new hire remote?
Yes
IT: send Ship hardware
New Hire: user Sign documents
New Hire: user Attend orientation
(finish Remote employee onboarded)

(start New employee joins)
HR: send Welcome email
HR: service Create accounts
IT: manual Prepare hardware
Is the new hire remote?
No
New Hire: manual Pick up hardware on day one
New Hire: user Sign documents
New Hire: user Attend orientation
(finish On-site employee onboarded)
```

Each separated block is a complete trace. The shared prefix is merged; the identical question followed by *Yes*/*No* infers an XOR split; the distinct tasks form the branches; and the shared activities converge before the two distinct end states. The `HR`, `IT`, and `New Hire` prefixes assign responsibilities, while `send`, `service`, `manual`, and `user` determine task types. These inferences are properties of the trace-based notation and its parser; the example does not require explicit gateway declarations in the input.

Table 4.2 defines the representation's provenance. V3.1 retains the textual basis inspired by BPMN Sketch Miner; v5 does not merely add instructions, but also extends the grammar and engine.

The *canon-1* example, stored at `packages/bpmn-core/tests/fixtures/canon-1-cheeseburger.dsl`, illustrates the intended input: a few readable lines containing enough information to express roles, parallel workstations, a timer event, and explicit start and end events. Figure 4.2 shows the BPMN 2.0 diagram generated by the engine from this DSL input.

The internal representation relies on the AST and resolved model defined in `packages/bpmn-core/src/dsl/ast.ts`. It distinguishes steps such as `Task`, `Event`, `Data`, `Question`,

Construction	Example syntax	BPMN meaning
Task	`Do something`	Activity.
Role-based task	`Cashier: Take order`	Activity assigned to a *pool* or *lane*.
Task type	`service Validate payment`	User, service, rule, manual, receive, send, or *script* task.
Event	`(timer rings)`	Intermediate timer, message, signal, or error event.
Start and end	`(start Start)`, `(finish)`	Start and end events.
Decision	`Question?` and its branches	Exclusive *gateway* with labelled branches.
Parallelism	`A | B | C`	Concurrent branches through a parallel *gateway*.
Data	`[Invoice]`, `[db Orders]`	Data object or data store.
Fragment	`...`	Trace continuity through anchors.
Pool map	`== pools ==`	Explicit declaration of *pools* and *lanes* in v5.

Table 4.1: Main constructions of the intermediate DSL.

Layer	Inherited or inspiring basis	Extension introduced in this work
Textual syntax	Traces, roles, decisions, and parallelism from BPMN Sketch Miner.	Extended events and data, anti-patterns, fragments, and the `== pools ==` block in v5.
Parser and model	Textual conventions of the source notation.	AST, semantic resolution, `DeclaredPool`, normalizations, and project-specific diagnostics.
Emission and layout	The source tool's engine is not reused.	BPMN-XML emitter, layout, validation, and shared monorepo API.
Instructions	Syntax documented by the original work.	Versioned examples, anti-patterns, checklist, and modelling criteria.

Table 4.2: Differences between the inspiring basis and the implemented extensions.

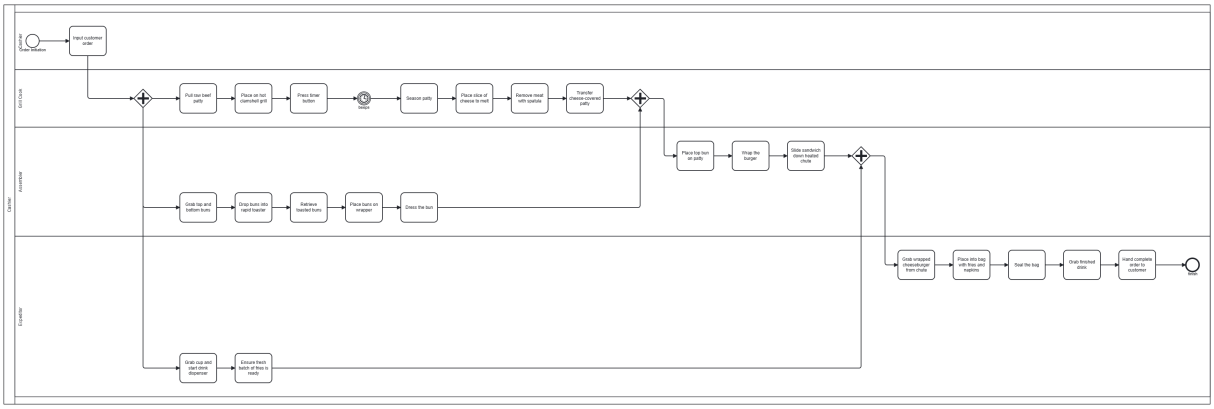

Figure 4.2: BPMN 2.0 diagram rendered by the engine from the *canon-1* example (`canon-1-cheeseburger.dsl`). It illustrates roles, parallel workstations, a timer event, and explicit start and end events.

`Parallel`, `PoolScope`, `Annotation`, and `FragmentMarker`. These elements are later converted into BPMN nodes, sequence flows, message flows, and resolved participants.

4.3.1 Lexical and syntactic analysis

Lexical and syntactic analysis is implemented with Chevrotain, mainly in `lexer.ts` and `parser.ts`. The first part works line by line: it identifies whether each input is a task, event, data item, question, parallel construction, annotation, fragment, or *pool*-map header. The parser then groups those steps into traces and builds the AST.

Deviations are reported through stable codes grouped by origin:

- basic analysis: `LEX-1`, `PARSE-1`, `AP-6`, and `AP-7`;
- fragment anchoring: `ANCHOR-MISSING`, `FRAGMENT-OPEN`, and `FRAGMENT-DUP`; and
- format and structure: `INLINE-COMMENT` and `DANGLING-QUESTION`.

Stable codes prevent errors from being reduced to textual messages that are difficult to compare. The same taxonomy is used to locate problems during development and to reuse diagnostics during evaluation.

4.3.2 Semantic analysis

Once the AST has been built, the semantic phase converts it into a `ResolvedModel`. Implemented in `semantic.ts`, this logic assigns unique identifiers, resolves roles in *pools* and *lanes*, connects flows according to trace order, pairs anchors, and synthesizes required implicit elements.

Here, *semantic analysis* means resolving the DSL into the engine's internal process model and checking its internal constraints. It does not compare that model with the original natural-language description and therefore does not establish semantic fidelity.

A common case is the absence of explicit start or end events. In such situations, the engine can generate them, but records the normalization through the diagnostics `IMPLICIT_START_EVENT` and `IMPLICIT_END_EVENT`. The correction is therefore not hidden: it remains traceable for later analysis.

4.4 BPMN-XML emission

BPMN-XML generation is concentrated in the `bpmn/` module, mainly comprising `emit.ts` and `xml.ts`. This stage serializes the resolved internal model as BPMN 2.0 XML containing processes, participants, flow nodes, sequence and message flows, data objects and stores, and boundary events. At this point, the XML represents the process structure; the later layout stage adds BPMN Diagram Interchange (BPMN DI) geometry so that a renderer can draw it.

The distinction between semantics and graphical representation is especially useful here. First, the system determines what the process contains; in a later phase, it determines where each element is drawn. This separation allows layout to be treated as an independent and replaceable stage, and allows the model to be validated before a specific geometry is assigned.

4.5 Layout and rendering subsystem

Layout required a substantial part of the engineering effort. Even when BPMN-XML is correct, the result loses value if the diagram cannot be read clearly. Rendering was therefore organized as a *pipeline* of small steps, each with a specific responsibility:

1. sanitizing semantic XML for `bpmn-auto-layout`;
2. applying a base *auto-layout*;
3. laying out processes that do not yet contain graphical information;
4. placing *pools* and *lanes* and preparing message flows;
5. orthogonalizing the control flow;
6. distributing parallel edges across channels to reduce overlaps;
7. placing labels in free spaces near their elements;
8. extending outer *lanes*; and
9. placing data objects and annotations.

This logic is distributed among `sanitize.ts`, `layout-missing.ts`, `pools.ts`, `orthogonal.ts`, `edge-channels.ts`, `labels.ts`, and `artifacts.ts`. The division is not merely an internal organizational choice: it makes it possible to adjust one heuristic without rewriting the rest of the renderer.

Robustness to boundary events without a continuation. One example of this type of problem involved boundary events without a continuation in the trace. In that case, `bpmn-auto-layout` attempted to process a nonexistent list of outgoing edges and rendering stopped completely.

The correction was not implemented as a patch at the end of rendering, but at the source of the model. The AST records a `boundaryEdgeOverride` that correctly reroutes the flow, and the sanitizer handles the degenerate case before invoking the library. The behavior is covered by the regression test `tests/render/boundary-completion.test.ts`. This case summarizes the project's strategy: locate the failure, correct its cause, and add a test that makes recurrence less likely.

4.6 Validation and diagnostics

After rendering, the system examines the result again. The `validation/bpmnValidation.ts` module performs structural checks on the model and layout. Each finding is classified by severity—`error`, `warning`, or `info`—and by origin—`model` or `layout`.

Model diagnostics include:

- `IMPLICIT_START_EVENT`, `IMPLICIT_END_EVENT`, and `IMPLICIT_BOUNDARY_END_EVENT`;
- `MISSING_EXPLICIT_START_EVENT` and `MISSING_EXPLICIT_END_EVENT`;
- reference and endpoint errors in sequence and message flows;
- `MESSAGE_FLOW_WITHIN_POOL` and `POOL_BOUNDARY_CONTROL_FLOW`; and
- issues in the explicit map, including `POOL_MAP_DUPLICATE_LANE`, `POOL_MAP_LANE_UNASSIGNED`, `POOL_MAP_NAME_COLLISION`, and `POOL_MAP_UNKNOWN_LANE`.

At the graphical level, diagnostics include `LAYOUT_ARTIFACT_TOO_FAR`, `LAYOUT_ELEMENT_OVERLAP`, `LAYOUT_UNREADABLE_LABEL`, and `LAYOUT_XML_PARSE_ERROR`. Validation also collects control measures such as the number of activities, *gateways*, events, isolated nodes, and nodes without outgoing edges.

4.7 Evolution of the *system prompt* and *pool–lane* mapping

The *system prompt* was developed within this thesis from the textual syntax documented by BPMN Sketch Miner [10]. Its early sequence is v1 $\rightarrow$ v2 $\rightarrow$ v3 $\rightarrow$ v3.1. Versions v1 and v2 were development iterations, with fragment-boundary failures still present in v2. Version v3 made trace-based notation the default and added the anchor-boundary rule and AP-6/AP-7. Version v3.1 re-examined the supporting cases: AP-6 remained documented as observed, whereas

AP-7 was retained as a preventive rule because its supposed observed case was actually a misdiagnosed return start event. This historical classification does not contradict the AP-7 diagnostics later found in the experimental matrix (Section 5.8.2); it explains the evidence available when v3.1 was fixed. V3.1 was therefore selected as the first controlled baseline, not as the first version able to generate a valid diagram.

After that baseline, v4 expanded readability and event-handling instructions; v4.1 refined the prompt using canonical cases but is not included in the 420 consolidated rows; and v5 added both instructions and a syntactic construction, together with parser and engine support for explicit mapping between *pools* and *lanes*. The contribution lies in these versioned and evaluated extensions; the source syntax is not presented as an original creation.

The final `== pools ==` block was introduced to declare participant grouping and *swimlanes* without relying solely on connectivity-based inference. Its design is specified in version 5 of the *system prompt* (`apps/tfm-lab/prompts/system/bpmn_sketch_miner_system_prompt_v5.md`) and is represented in the AST by the `DeclaredPool` type. This addresses, in a more controlled way, a particularly sensitive semantic issue: distinguishing independent participants and representing their communication through message flows.

This technical evolution should be separated from the main experimental protocol. The evaluation matrix deliberately keeps the *system prompt* at v3.1 to isolate the model effect. Versions v4–v5 form part of system development, but their systematic comparison is treated as an ablation rather than as the basis for conclusions drawn from the v3.1 matrix.

4.8 Applications built on the engine

4.8.1 `apps/tfm-lab`: research laboratory

`apps/tfm-lab` is the thesis's experimental tool. It combines an editor that displays the DSL and diagram in real time, a tab dedicated to experiments, and a *headless* mode for rendering batches automatically.

It also stores the *prompts*, synthetic processes `SYN001`–`SYN015`, and the results of each run. Folders follow a traceable naming scheme and contain the input, raw and normalized output, BPMN, SVG, and PNG artifacts, metadata, diagnostics, and input and output *hashes*.

4.8.2 `apps/company-web`: demonstration application

`apps/company-web` is aimed at end users. It provides a guided workflow for describing a process, generating the DSL, editing the diagram, and documenting it. The **Document** function exports a Word file; the repository includes the example `docs/New_Hire_Onboarding_Process.docx`.

The tool was made available to SGS Productivity colleagues, who tested it informally with cases from their work; the scope and limitations of that test are detailed in Section 4.9.

For technical reasons related to branding and licensing, the application presents the notation as an *internal BPMN textual DSL*. It also avoids using third-party marks in text sent to models or published by the tool.

4.9 Formative feedback from users and Metadev

In addition to internal testing, development included two forms of external validation that provide methodological context for the work.

Formative user test. SGS Productivity colleagues involved in *Lean* consulting and process improvement used the application with spontaneous descriptions. The activity helped identify adjustments to labels, lanes, and layout. Because the number of participants, tasks, instrument, consent, and per-case results were not recorded, it is not included in the experimental set and is not used to claim usefulness or comprehensibility.

Collaboration with Metadev. The company provided technical mentoring during grammar specification and design. This feedback guided development decisions, but it is not treated as independent scientific validation because no rubric was applied and evaluators, disagreements, and results were not documented.

4.10 Evaluation pipeline and test suite

The metric package is located in `TFM-eval/` and is implemented in Python. It processes artifacts and computes metrics without a reference diagram. Rendering is not performed directly by this package: `scripts/run-evaluation.ps1` orchestrates the TypeScript engine through Playwright/Chromium and then the Python evaluation. Metrics include XSD validity, connectivity, degree, branch labels, name completeness, gateway pairing, size, complexity, and cyclicity.

The package incorporates the BPMN 2.0 XSD schemas and provides command-line interfaces at `TFM-eval/src/bpmn_eval/cli.py` and `TFM-eval/src/bpmn_eval/cli_experiments.py`. It generates `results.csv` and `results.json`. The notebook `TFM-eval/notebooks/analysis.ipynb` is used for exploration, while `docs/analisis-derivado/build_report.py` non-interactively generates the derived analyses and final figures identified in the evaluation chapter.

System verifiability is further supported by a test suite covering:

- the DSL parser and semantic analysis;
- BPMN emission and *pool* mapping;
- orthogonal routing, boundary events, artifacts, and participant layout;
- structural validation and diagnostics; and
- *canon* diagrams as end-to-end regressions.

4.11 Synthesis

The chosen architecture responds to the work's dual nature. On the one hand, it supports reproducible research through the engine, laboratory, and evaluation *pipeline*. On the other, it provides an applied tool tested in a professional context. Both use the same engine, avoiding an artificial separation between the experimental and demonstration components.

Incremental development, stable diagnostics, separation between semantics and graphical representation, and *canon* diagrams make it possible to address a complex visual problem with a system whose quality can be checked automatically.

Chapter 5

Evaluation

5.1 Evaluation objective

The evaluation studies the behavior of several model–platform configurations when generating BPMN 2.0 from textual descriptions. Automatic analysis does not require a reference diagram, which supports scale and traceability but limits any claim of complete semantic equivalence.

The evaluated application follows a two-stage architecture (Figure 4.1). First, the model produces an intermediate representation inspired by BPMN Sketch Miner [10]. A deterministic compiler then transforms the DSL into renderable BPMN 2.0. The separation locates the stage at which a failure is observed, but does not automatically establish causality: the result depends on the interaction among model output, grammar, normalizations, compiler, and rendering.

The evaluation design takes BEF4LLM, proposed by Lauer et al. [3], as its starting point. This framework organizes the evaluation of LLM-generated BPMN models into four dimensions: syntactic quality, pragmatic quality, semantic quality, and BPMN-XML file validity. Although BEF4LLM was introduced briefly in earlier chapters, this chapter develops its adaptation to the present work in greater detail (Section 5.3.1). The adaptation is necessary for two reasons. First, BEF4LLM includes 39 metrics, several of which assume the existence of a reference BPMN. Second, in this thesis the final XML does not come directly from the model, but from a deterministic compiler. An operational subset of metrics that can be computed reliably from the available artifacts is therefore used.

The 3C system is also considered for assessing Clarity, Correctness, and Completeness [9]. It is not adopted as an instrument because it requires human evaluation or an *LLM-as-a-judge* approach. The automatic indicators in this work are not presented as equivalent to 3C: robustness, file validity, five structural proxies (indirect indicators that approximate a property difficult to measure directly), construction presence, and diagnostics are reported separately. Visual clarity and completeness with respect to the text fall outside the quantitative core.

Table 5.1 lists the research questions guiding the evaluation. The numbering preserves the study's original design: RQ6, concerning the effect of description length, was formulated but not evaluated, so it is absent from the table; Section 5.10 justifies this decision:

Code	Research question	Evaluation approach
RQ1	Does the DSL-based approach generate valid and renderable BPMN 2.0?	Syntactic, semantic, and rendering robustness.
RQ2	Which model produces higher-quality diagrams with the same prompt?	Model comparison using structural and semantic metrics.
RQ3	How does quality change as process difficulty increases?	Comparison by difficulty: easy, medium, difficult, and stress.
RQ4	Which expected BPMN constructions does automatic evaluation detect?	Construction presence and diagnostic analysis.
RQ5	How does the result change across configurations v3.1, v4, and v5?	Joint comparison of instructions and, in v5, syntax and engine support.
RQ7	Are the models stable across repeated runs?	Intra-model and intra-process variability across R01, R02, and R03.
RQ8	How do the evaluated DSL and direct BPMN-XML configurations differ?	Paired comparison of two complete configurations with different instruction richness.

Table 5.1: Research questions of the evaluation

5.2 Experimental design

The evaluation relies on a controlled set of synthetic processes identified as `SYN001`–`SYN015`. These processes were designed to cover different difficulty levels and several BPMN constructions. Each model runs every process several times, making it possible to analyze not only mean quality but also stability across runs.

The corpus was created specifically for this work and is written in English. It covers sequences, gateways, events, data, *lanes*, and collaborations. Each process includes an author-assigned difficulty and a list of expected constructions. There were no external annotators or independent quantitative rubric; the strata are also unbalanced (3 easy, 9 medium, 2 difficult, and 1 stress process). Difficulty is therefore interpreted as a descriptive stratification partially confounded with construction type. Appendix A.7 details each case and warns of possible thematic overlap between *prompt* examples and some processes.

The main corpus is organized into four difficulty levels (Table 5.2):

The configurations with a complete v3.1 matrix are `GPT_5_5_THINKING`, `OPUS_4_7`, and `GEMINI_3_5_FLASH_THINKING`. Each contributes 45 runs (15 processes $\times$ 3 replicates), so the main comparison contains 135 rows. The consolidated set also includes v4/v5 comparisons, exploratory configurations, and 45 direct BPMN-XML runs, which are kept separate.

The evaluation is implemented as a chain of distinct stages:

1. **Non-deterministic generation:** obtaining and preserving the model's raw response.
2. **Normalization:** trimming and removing the surrounding Markdown block without repairing its content.

Level	Description	Purpose
Easy	Linear processes or processes with simple branching.	Check whether the system correctly handles basic cases.
Medium	Processes with decisions, parallelism, data, or lanes.	Evaluate structures common in real business processes.
Difficult	Processes with events, exceptions, or more complex collaboration.	Measure the ability to model advanced BPMN situations.
Stress	Dense processes or processes with multiple simultaneous requirements.	Identify the practical limits of the approach.

Table 5.2: Difficulty levels of the synthetic corpus

3. **Deterministic compilation and rendering:** DSL analysis, BPMN emission, and layout.
4. **Metrics and aggregation:** computation over artifacts and writing of `results.csv` and `results.json`, followed by analysis using versioned scripts.

An important decision is that metrics are not computed directly in the *notebook*. They are all implemented and tested within the evaluation package, avoiding dependence on *ad hoc* or difficult-to-reproduce analysis code.

5.3 Metrics used

The evaluation combines robustness, structural, semantic, and stability metrics.

5.3.1 Relation to BEF4LLM

BEF4LLM provides a broad framework for evaluating LLM-generated BPMN and groups results by quality dimensions [3]. It distinguishes one BPMN-XML validity measure, 16 syntactic-quality metrics, 15 pragmatic-quality metrics, and 7 semantic-quality metrics. Validity is checked against the BPMN 2.0 XSD schema. Syntactic quality includes, among other aspects, the presence of start and end events, flow-connection rules, task labelling, and the in-degree and out-degree of events, tasks, and gateways. Pragmatic quality approximates readability through size, density, connectivity, partitionability, concurrency, and other complexity properties. Finally, semantic quality compares the candidate model with a reference using label similarity, graph structure, and behavior.

Not directly using distance metrics against human models does not mean discarding BEF4LLM. Rather, it acknowledges a limitation relevant to this context. BPMN does not always have one uniquely correct representation: two diagrams with structural or visual differences may express equivalent business semantics. The converse may also occur: two diagrams with similar labels or superficial structure may differ in important semantic aspects, such as participant separation or message-flow use. A metric based on distance from a human

reference diagram could penalize valid alternatives or overvalue formally similar but incomplete diagrams. For this reason, the thesis evaluates whether the generated diagram preserves the BPMN properties required by the description and maintains structural consistency.

This thesis adopts BEF4LLM's objective and multidimensional approach, although it does not include all of its metrics. The reasons are as follows:

1. **No reference BPMN for each case.** BEF4LLM's semantic metrics compare the candidate diagram with a reference. Since this work uses no manual reference, those metrics are replaced by expected-feature coverage.
2. **Multiplicity of valid diagrams.** A process admits different BPMN representations without them necessarily being lower quality. Distance from a single human diagram is therefore not considered sufficient to assess semantic quality.
3. **DSL-plus-compiler architecture.** XML validity is strongly conditioned by the deterministic compiler. Consequently, M1 is presented as a basic validity check, but is not used to rank models when it reaches saturation—that is, when nearly all runs receive the maximum score and the metric ceases to discriminate between models.
4. **Avoiding penalties for engine decisions.** Some strict BPMN-XML rules depend on how the compiler materializes the DSL. For example, requiring condition expressions on every branch would penalize an implementation decision rather than necessarily an LLM error.
5. **Reproducibility and traceability.** Only metrics that can be computed automatically, stably, and audibly over current *pipeline* outputs are included.

The package computes ten metrics, identified as M1–M10. M1–M5 and M8 act as quality or control indicators. M6, M7, M9, and M10 instead describe size and complexity and are therefore not included in `scored_mean`. Table 5.3 summarizes the operational subset of metrics.

The identifier `scored_mean` is retained for artifact compatibility, but the text calls it the *composite mean of five structural proxies*. Its validity is exploratory. In particular, M2 may retain a high score despite broken references, and M4/M8 may take the maximum value when the evaluated construction is absent. Since the repository is not modified and metrics are not regenerated, these defects are disclosed and the mean is not used as evidence of complete fidelity or formal correctness.

M6, M7, M9, and M10 should not be interpreted as quality scores bounded in $[0, 1]$. They are descriptive quantities concerning graph size and complexity. In particular, control-flow complexity (M7, `cfc_total`) follows Cardoso's *Control-Flow Complexity* metric [41]. It is a dimensionless integer calculated as the sum of divergent-gateway contributions: an XOR gateway with n branches contributes n; a parallel (AND) gateway contributes 1; and an inclusive (OR) gateway with n branches contributes $2^n - 1$. A high `cfc_total` means that a process reader must consider more decision states. It is therefore used as a structural-difficulty indicator, not as a correctness measure.

Metric	Interpretation	Use in this thesis
M1	XSD validity of generated BPMN 2.0.	Basic validity check; excluded from `scored_mean` because of saturation.
M2	Reachability from the start.	Checks reachable nodes; broken references affect `passed` but do not necessarily reduce its *score*.
M3	Element degree.	Penalizes anomalous incoming or outgoing edges in tasks, events, and gateways.
M4	Decision labelling.	Assesses whether XOR branches are labelled; it does not test exclusivity, logical completeness, or executability.
M5	Label completeness.	Assesses whether relevant elements have informative names.
M6	Total model size (`tnn`).	Descriptor: total number of diagram nodes.
M7	Control-flow complexity (`cfc_total`).	Descriptor: complexity introduced by gateways and their branches.
M8	Gateway pairing.	Measures a strict structural pattern; it is not equivalent to *soundness*.
M9	Network connectivity coefficient (`cnc_value`).	Descriptor of the relation between sequence flows and nodes.
M10	Cyclicity (`has_cycle`).	Binary descriptor indicating whether the graph contains cycles.

Table 5.3: Operational metric subset inspired by BEF4LLM

5.3.2 Robustness

Robustness is summarized by two success rates (Table 5.4):

Metric	Definition	What it measures
`semantic_success`	DSL: `parserValid` ∧ `semanticValid`; direct XML: `bpmnXmlValid` ∧ `bpmnImportValid`	Internal-analysis/compilability success; it does not measure equivalence to the description.
`render_success`	`renderValid`	The renderer finished; it does not measure readability, crossings, or comprehensibility.

Table 5.4: Pipeline robustness metrics

The separation between `semantic_success` and `render_success` identifies the stage at which the *pipeline* fails; it does not by itself attribute the failure exclusively to the model or engine. In direct BPMN-XML experiments, M1–M10 and coverage are computed over the raw file. If BPMN DI is missing, `bpmn-auto-layout` is used only for visualization. The six XML outputs that fail the success criterion violate XSD even though they may be importable, so "importable" and "XSD-valid" cannot be treated as synonyms.

5.3.3 Structural quality

The composite mean of five structural proxies, stored as `scored_mean`, ranges from 0 to 1 and includes M2, M3, M4, M5, and M8 with equal weights. Equal weighting has not been validated through sensitivity analysis. M1 is reported separately: its saturation reflects that the compiler usually produces conforming XML, an architectural result that discriminates little between models.

Two aggregations are used to compare configurations. **Conditional quality** is the mean of `scored_mean` over runs with successful internal analysis. **Failure-penalized quality**, `scored_mean_penalized`, includes all runs and assigns 0 to failures:

$$q_i^* = \begin{cases} \texttt{scored\_mean}_i, & \text{if the run passes internal analysis,} \\ 0, & \text{otherwise,} \end{cases} \qquad \overline{Q}_{\text{pen}} = \frac{1}{N} \sum_{i=1}^{N} q_i^*.$$

This quantity equals the product of internal-analysis success rate and conditional quality. Intuitively, conditional quality answers "How good are the diagrams that are successfully generated?", whereas penalized quality answers "What mean quality does a system user obtain when failed attempts are also counted?". It is reported alongside both dimensions to make survival bias visible: the distortion that arises when only cases passing a filter are averaged and failures are ignored.

The definition of M4 is adapted from a stricter BPMN reading based on condition expressions and default flows. In this compiler, that reading would penalize an engine implementation decision rather than necessarily a model error. M4 is therefore interpreted as decision completeness: for each divergent XOR gateway, the proportion of outgoing branches with a non-empty label is computed.

5.3.4 Automatic construction coverage

Semantic evaluation is performed without a reference diagram. Instead of comparing each result with an ideal BPMN model, a list of expected features is defined for each process. These include XOR gateways, parallel gateways, lanes, pools, *message flows*, data objects, timer events, and exception events.

Automatic detectors then check whether each expected construction appears in the diagram. If a process declares E constructions and D are detected, coverage is D/E. The mean gives each process equal weight, regardless of whether it expects four or six constructions. Several detectors check existence only: `merge` and `fork_join` are associated with gateways, `send_receive` with a message flow, and `cancellation` with a boundary event. They do not verify the relation required by the description, and shared detectors may count the same evidence several times. The metric is therefore called *automatic coverage of expected constructions*, not complete semantic coverage.

Coverage is also grouped by `system_prompt_version`. This dimension is required for RQ5. The main model comparison is restricted to v3.1, whereas the ablation contrasts v3.1, v4, and v5 without merging their results under a single `representation=dsl` label. If a *prompt* version cannot express certain collaboration features, those omissions should be read as limitations of that configuration, not necessarily as model failures.

5.3.5 Estimated token consumption

The *pipeline* estimates text length offline using `o200k_base`. Instructions, description, input, output, and total are stored separately. This homogeneous rule does not necessarily reproduce Claude or Gemini tokenizers, nor does it measure hidden reasoning, wrappers, caching, latency, or billing. The results are consequently described as text-length estimates rather than economic costs.

5.3.6 Engine diagnostics

In addition to numerical metrics, diagnostic codes classify errors, warnings, information, and normalizations, including implicit events, unreachable-ending branches, invalid references, anchoring errors, duplicate objects, and automatic boundary-event completion.

The counts indicate occurrences, not affected diagrams. A single diagram may emit the same code several times, so the counts are not interpreted as prevalence or compared as rates without normalization by run or node count.

5.4 Dataset overview

The consolidated set contains 420 traceable experiments: 177 use the DSL with *system prompt* v3.1, 99 use v4, 99 use v5, and 45 use direct *zero-shot* BPMN-XML generation (that is, instructions without solved examples). The main model comparison is limited to the 135 balanced v3.1 runs. Table 5.5 shows the state distribution of the DSL v3.1 subset:

State	Cases	Interpretation
`success_with_warnings`	151	The diagram compiles, although the engine applies normalizations or emits warnings.
`success`	15	The diagram compiles without warnings.
`semantic_error`	11	The DSL fails semantic validation.
`render_error`	0	No rendering errors are observed in the final set.

Table 5.5: Compilation-state distribution of the DSL v3.1 subset

The presence of warnings does not by itself imply that a diagram is incorrect. In many cases they result from benign normalizations, such as implicit creation of start or end events when the generated DSL does not explicitly declare them.

Of the 177 v3.1 observations, 135 form the balanced matrix of three models, fifteen processes, and three repetitions. The remaining 42 are exploratory: 24 from `OPUS_4_8_MEDIUM`, 12 from `GPT_5_4_THINKING`, 5 from `3_1_PRO`, and 1 from `OPUS_ADAPTATIVE`. The v4 and v5 matrices are reserved for RQ5, while the 45 `ZSXML` outputs are studied in RQ8. Keeping them separate prevents different *prompt* versions or representations from being mixed in one aggregate mean.

Table 5.6 shows the models with complete coverage and therefore direct comparability:

Model	Runs	Use in the analysis
`GPT_5_5_THINKING`	45	Complete model, included in the main ranking.
`OPUS_4_7`	45	Complete model, included in the main ranking.
`GEMINI_3_5_FLASH_THINKING`	45	Complete model, included in the main ranking.

Table 5.6: Models with complete experimental coverage

The remaining models are retained as exploratory results because they do not cover the same *grid* in every configuration. Opus 4.8 Medium, for example, combines 24 v3.1 runs with two subsets of 9 v4 and v5 runs focused on processes of different difficulty. The means of these versions are not comparable, so they are excluded from rankings until a balanced matrix is available.

5.5 RQ1: Robustness of the DSL approach

The first research question concerns the approach's basic robustness: whether the DSL-based *pipeline* systematically produces valid and renderable BPMN diagrams.

Model	n	Semantic success	Render success
GPT_5_5_THINKING	45	**100 %**	**100 %**
OPUS_4_7	45	96 %	96 %
GEMINI_3_5_FLASH_THINKING	45	96 %	96 %

Table 5.7: Internal-analysis and rendering success rate by configuration.

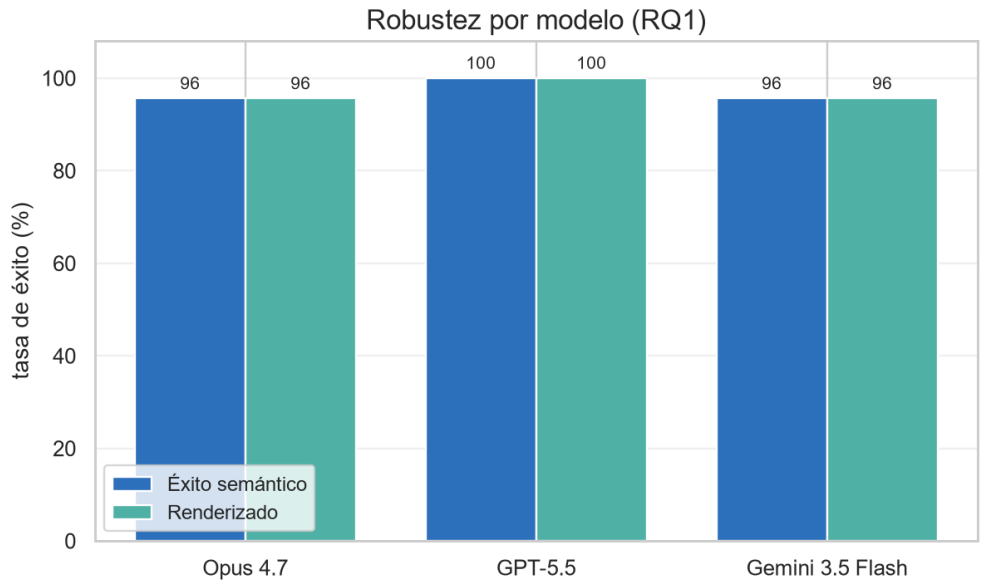

Figure 5.1: Robustness by model

The data show high robustness. GPT_5_5_THINKING completes 45/45 runs; OPUS_4_7 and GEMINI_3_5_FLASH_THINKING complete 43/45. The four failures are concentrated in SYN015, the only process in the stress stratum, so they reflect case dependence rather than dispersed failures.

The agreement between the two rates indicates that no additional rendering failures occurred among outputs that passed internal analysis; it does not demonstrate universal visual robustness.

5.6 RQ2: Comparison of quality across models

The second question compares the mean quality of diagrams generated by models with a complete matrix. The *prompt* and process set are held constant to keep the comparison clean.

The highest mean belongs to GPT_5_5_THINKING, followed by OPUS_4_7 and GEMINI_3_5_FLASH_THINKING. The value after $\pm$ is the standard deviation of the 45 rows, not a confidence

Model	n	scored_mean	M1	M2	M3	M4	M5	M8
GPT_5_5_THINKING	45	**0,875 ± 0,128**	1,00	0,93	1,00	0,86	0,90	0,70
OPUS_4_7	45	0,833 ± 0,215	0,96	0,92	0,95	0,90	0,80	0,59
GEMINI_3_5_FLASH_THINKING	45	0,804 ± 0,223	0,96	0,92	0,95	0,82	0,81	0,51

Table 5.8: Composite mean of five structural proxies by configuration (mean ± standard deviation over 45 runs)

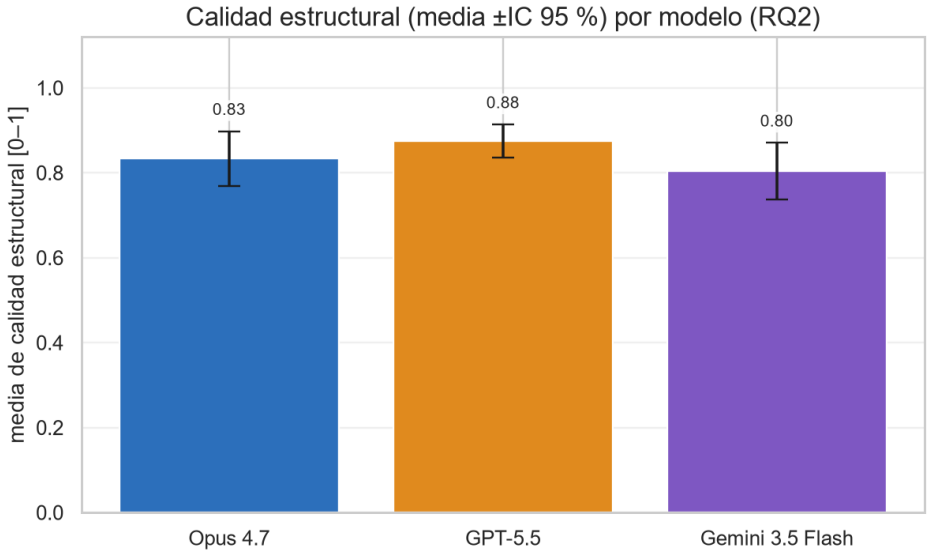

Figure 5.2: Composite mean of structural proxies by configuration

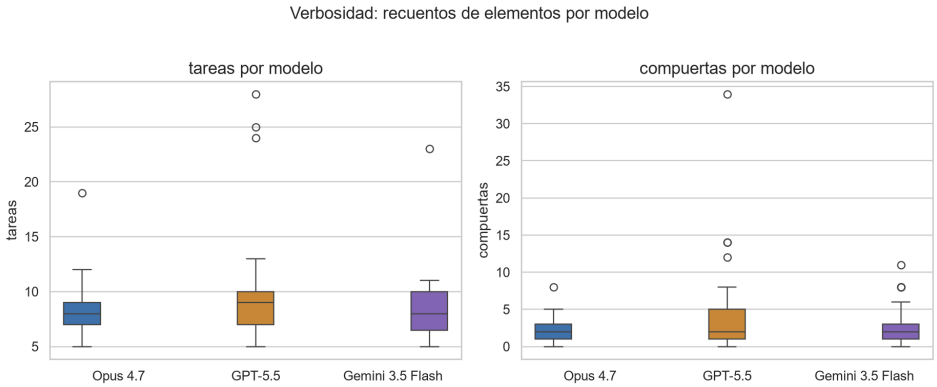

Figure 5.3: Distribution of task and gateway counts by model. The three models have similar verbosity, with some outliers in complex processes.

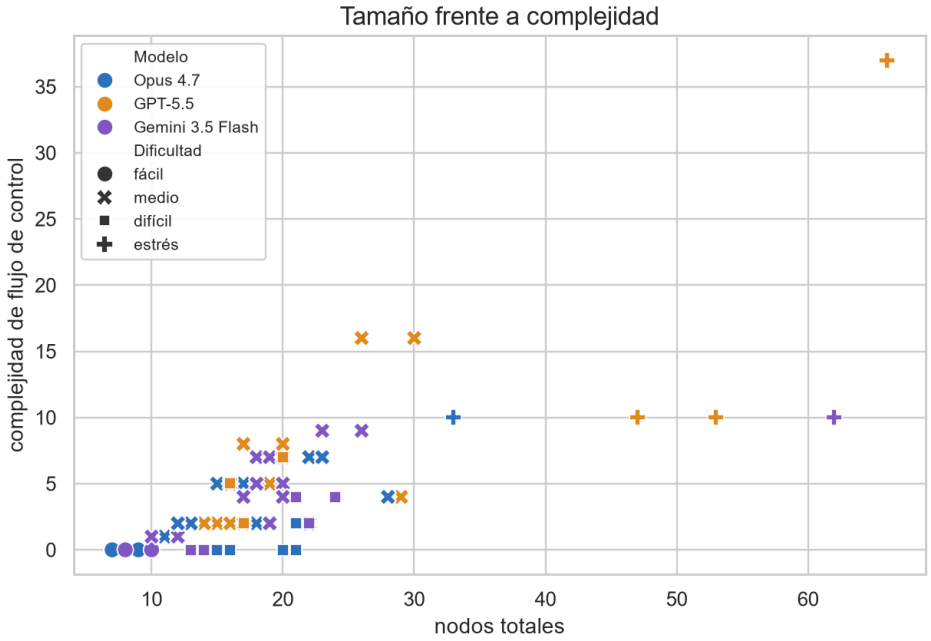

Figure 5.4: Relationship between total node count and control-flow complexity of generated diagrams, by model and difficulty level.

interval. Rows are not independent because they repeat the same processes; the paired test in Section 5.14 detects no differences among configurations.

M1, M2, and M3 are close to saturation. Once the model produces a valid DSL, the compiler usually generates well-connected and formally consistent BPMN diagrams. Differences appear mainly in M4 and M8. M4 reflects both gateway creation and explicit decision logic. M8 measures whether branching and synchronization structures are correctly balanced, which is especially relevant in processes with parallelism, exceptions, or alternative flows.

The size and complexity figures are descriptive. Without a per-process test or size normalization, they cannot establish that verbosity profiles are equivalent.

Overall, the three complete configurations achieve high descriptive means on the five structural proxies, but the paired analysis does not establish a global ranking. Their error profiles still differ, so any deployment-oriented comparison should also consider the processes to be modelled and the cost of each failure type.

5.7 RQ3: Effect of process difficulty

The third question analyzes how quality changes as the difficulty of the described process increases.

Difficulty stratification shows the most visible descriptive pattern. The mean decreases monotonically from easy cases to the stress case, although the strata combine different numbers of processes and construction types.

Difficulty	n	Int. success	Render success	scored_mean	Coverage
Easy	27	100 %	100 %	**0,984**	1,00
Medium	81	100 %	100 %	0,841	0,84
Difficult	18	100 %	100 %	0,788	0,76
Stress	9	56 %	56 %	**0,463**	0,46

Table 5.9: Effect of process difficulty

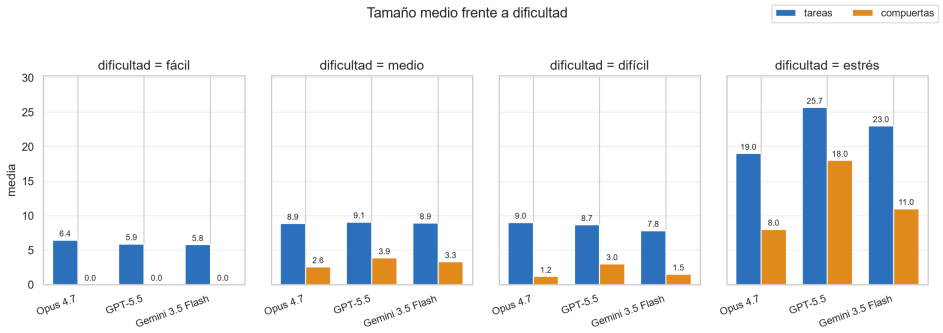

Figure 5.5: Mean number of tasks and gateways by model and difficulty level. Diagram size increases markedly in the stress level.

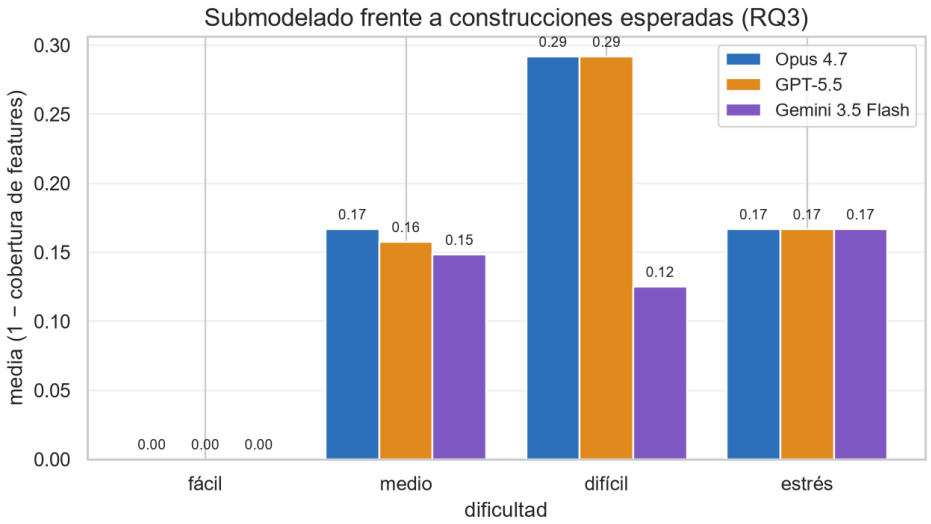

Figure 5.6: Undermodeling in complex processes

In this corpus, the DSL configurations achieve higher proxy values and success rates in the easy and medium strata than in the difficult and stress strata. These automatic results do not by themselves establish practical usefulness. The lowest values occur when many BPMN requirements accumulate simultaneously: the single stress process combines participant collaboration, exceptions, several branches, events, and modelling requirements that must remain coherent at the same time.

The increase in diagram size with difficulty (Figure 5.5) and the under-modeling observed in the most complex processes (Figure 5.6) illustrate this pattern graphically.

The decline coincides with greater size and construction accumulation, but cannot be attributed causally to “semantic complexity”: difficulty was assigned by the author and is confounded with construction type. Moreover, the stress stratum contains one process and nine runs, so it functions as a boundary case rather than a generalizable estimate.

5.8 RQ4: Presence of expected constructions

The fourth question studies whether detectors locate BPMN constructions declared as expected. It does not check whether they are used in the correct position or semantic relation.

5.8.1 Coverage of expected constructions

Table 5.10 reports mean *feature* coverage for the three complete models:

Model	Mean coverage
GPT_5_5_THINKING	0,86
GEMINI_3_5_FLASH_THINKING	0,85
OPUS_4_7	0,81

Table 5.10: Mean coverage of expected features

The most frequently omitted *features* in the 135-run balanced matrix are listed in Table 5.11. Each count is a failed expected-feature check; the denominator varies by feature because only processes that declare that feature contribute opportunities:

The main detected omission concerns collaboration between participants. Processes requiring several *pools* and message flows are most often simplified into a single participant. This interpretation was confirmed by case inspection, but the automatic rate reflects presence only, not correct use.

This distinction matters because, in BPMN, the difference between a *sequence flow* and a *message flow* is not merely visual. A *sequence flow* expresses continuity within the same process or participant; a *message flow* represents interaction between independent participants. If the model groups several participants into one pool, the diagram may remain understandable but loses semantic fidelity to the real process.

Omitted feature	Omissions	Interpretation
`message_flow`	27	The model does not correctly represent communication between participants.
`multiple_pools`	18	The process is collapsed into one pool when several participants should exist.
`send_receive`	18	Explicit send or receive events or tasks are omitted.
`cancellation` / `exception_flow`	5 / 5	Exceptions and cancellations are modelled incompletely.
`annotations`, `multiple_lanes`, `xor_gateway`, `parallel_gateway`, `data_objects`, `timer_or_exception_event`	4 each	Scattered omissions in more difficult processes.

Table 5.11: Most frequently omitted expected features

The weakness in modelling multiple *pools* is largely explained by a limitation inherited from the source syntax. BPMN Sketch Miner [10] did not implement *pools* fully explicitly in the DSL. Grouping lanes within *pools* was performed automatically and implicitly from control-flow connectivity. Because these boundaries could not be controlled directly, independent participants could be incorrectly integrated into one pool when the LLM omitted message-flow connections. As detailed in Section 5.9, version 5 of the *system prompt* addresses this limitation by adding the explicit declaration block `== pools ==`, which allows organizational boundaries to be delimited and named directly in the DSL.

5.8.2 Diagnostic taxonomy

Engine diagnostics complement the numerical score with a classification of specific problems (Table 5.12).

The values in Table 5.12 are occurrences without a denominator, not numbers of diagrams. The most frequent codes concern start and end normalizations; one diagram may contribute several occurrences. Less common codes locate specific defects, but are likewise not compared as unnormalized rates.

`OPUS_4_7` and `GEMINI_3_5_FLASH_THINKING` record some `AP-7` cases associated with fragment-anchoring problems. `GPT_5_5_THINKING` does not show this error, although it has some branches that do not reach an end event. This shows that models can fail in different ways even with similar aggregate means.

5.9 RQ5: Comparison of v3.1, v4, and v5 configurations

The fifth question compares configurations labelled by `system_prompt_version`. It is not a pure ablation: v4 changes the instructions, while v5 also modifies the available syntax and parser

Diagnostic code	OPUS	GPT	GEMINI	Interpretation
IMPLICIT_END_EVENT	104	51	93	Frequent normalization: the engine adds implicit ends.
IMPLICIT_START_EVENT	33	29	39	Frequent normalization: the engine adds implicit starts.
MISSING_EXPLICIT_END_EVENT	34	20	36	The DSL omits explicit ends.
MISSING_EXPLICIT_START_EVENT	23	25	15	The DSL omits explicit starts.
DUPLICATED_DATA_OBJECT	3	1	3	Data objects are repeated.
AP-7	3	0	3	Fragment-anchoring rule violation.
BRANCH_WITHOUT_END_REACHABILITY	0	3	0	Branches that do not reach an end event.
ACTIVITY_WITHOUT_INCOMING	1	0	1	Activity without an incoming flow.
SEQUENCE_FLOW_INVALID_REFERENCE	1	0	1	Invalid reference in a sequence flow.
IMPLICIT_BOUNDARY_END_EVENT	4	3	5	Boundary event without an outgoing flow completed by the engine.

Table 5.12: Diagnostic taxonomy by model

and engine support. Complete matrices of 45 runs per configuration are compared for GPT-5.5 and Gemini 3.5.

Model	v3.1	v4	v5
GPT-5.5	0,856	0,828	**0,957**
Gemini 3.5	0,846	0,830	**0,911**

Table 5.13: Mean feature coverage by *system prompt* version

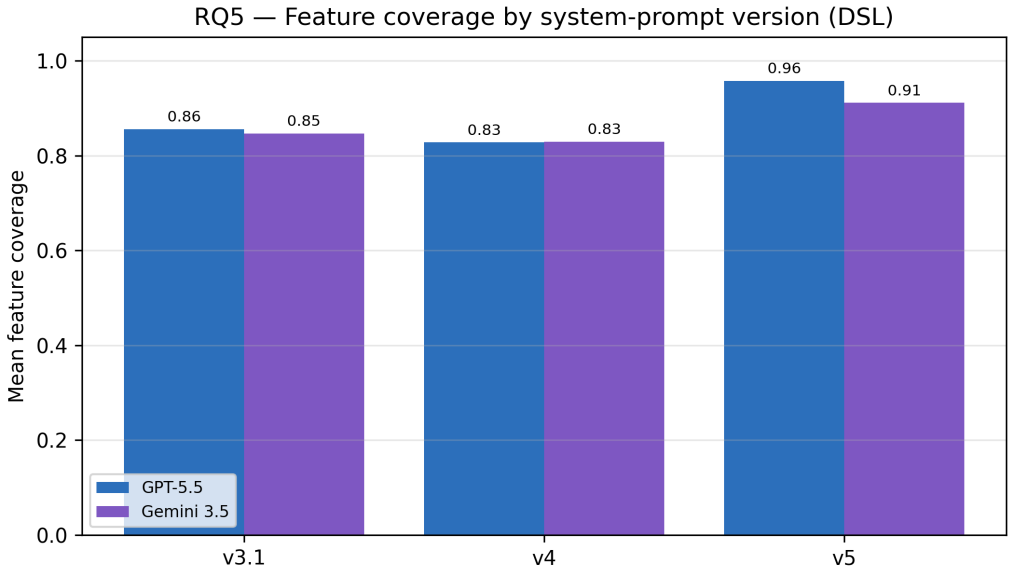

Figure 5.7: Construction coverage by v3.1, v4, and v5 configuration

v4 does not improve on v3.1 coverage. v5 raises it to 0.957 for GPT-5.5 and 0.911 for Gemini

3.5. The difference is concentrated in `multiple_pools`, `message_flow`, and `send_receive`. The result shows that the complete v5 configuration detects more collaboration constructions; it does not allow the difference to be attributed solely to better instructions.

With v5, GPT-5.5 achieves observed coverage of 0.957 versus 0.983 for direct XML and retains 100% internal-analysis success. Descriptive proximity is not evidence of equivalence. Opus 4.8 Medium is excluded because its versions cover subsets with different difficulty.

5.10 RQ6: Description length (unevaluated question)

The study does not include short, medium, and long versions of the same process; therefore, it does not answer the question of description length. Growth of the *system prompt* between v3.1 and v5 concerns another variable and is not used as a substitute. This question is left for future work through a design that keeps semantic content constant and varies only its level of detail.

5.11 RQ7: Stability across runs

The seventh question concerns stability: what happens when the same model runs the same process several times with the same *prompt*. It is measured by computing the mean standard deviation of `scored_mean` within each `(model, process)` cell across repetitions R01, R02, and R03.

Model	Within-cell standard deviation
GPT_5_5_THINKING	**0,055**
GEMINI_3_5_FLASH_THINKING	0,064
OPUS_4_7	0,077

Table 5.14: Stability across repeated runs

The observed values are small, with `GPT_5_5_THINKING` below `GEMINI_3_5_FLASH_THINKING` and `OPUS_4_7`. Without a prespecified threshold, they are not labelled “reasonable” or interpreted as an inferential ranking.

Each standard deviation is calculated from only three observations, and its mean may be dominated by `SYN015`. Lower dispersion does not rule out systematic bias. These results describe local repeatability and require more replicates or a variance-components model to support claims of predictability.

5.12 RQ8: *Zero-shot* comparison with direct BPMN-XML generation

This question compares two end-to-end configurations for GPT-5.5: DSL v3.1 plus compiler and direct BPMN-XML with a short *zero-shot prompt*. Both use the same fifteen processes and three replicates. The representation effect is not isolated: the DSL *prompt* contains grammar, examples, anti-patterns, and a checklist, whereas XML receives much shorter instructions. Raw XML is preserved unchanged; automatic layout is used only to visualize it.

Configuration	Int. success	Cond. quality	Pen. quality	Coverage
DSL v3.1	**1,00**	0,875	**0,875**	0,856
DSL v5	**1,00**	0,859	**0,859**	0,957
Direct XML	0,87	**0,927**	0,804	**0,983**

Table 5.15: GPT-5.5 robustness, quality, and coverage by configuration

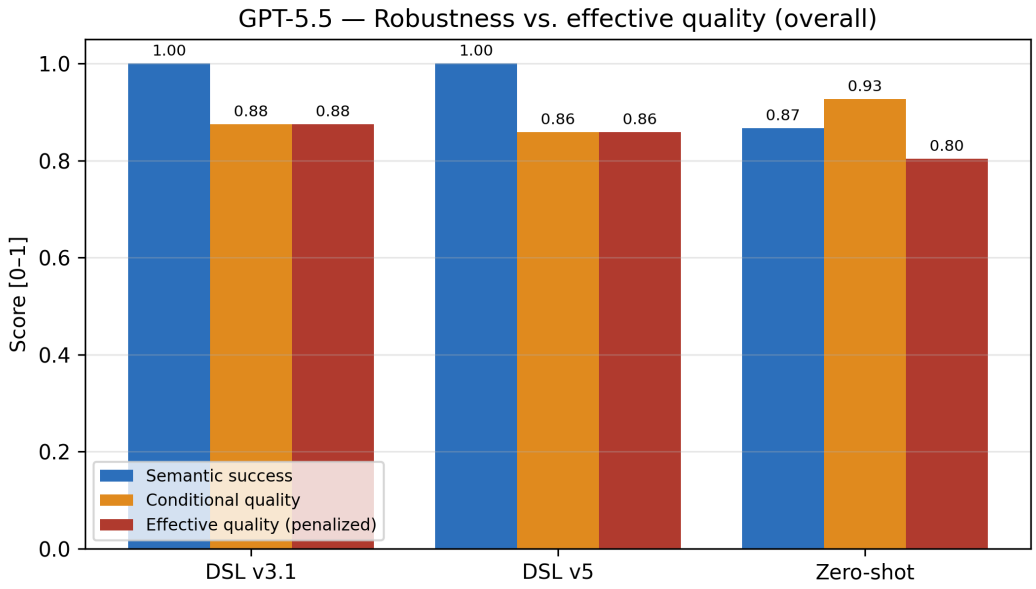

Figure 5.8: Robustness, conditional quality, and effective quality in the global comparison

Among outputs passing the internal criterion, XML reaches 0.927, but this figure excludes 13% of failures. Penalized quality is 0.804, compared with 0.875 for DSL v3.1. A paired Wilcoxon test over the 45 rows gives $p \approx 0{,}694$, so this metric provides no evidence of a statistical difference. Conditional quality favors XML, although it is affected by survivor selection.

Difficulty	DSL: success	DSL: quality	XML: success	XML: cond. / pen.
Easy	1,00	1,000	1,00	1,000 / 1,000
Medium	1,00	0,851	0,85	0,925 / 0,788
Difficult	1,00	0,812	1,00	0,847 / 0,847
Stress	1,00	**0,846**	0,33	0,813 / **0,271**

Table 5.16: DSL v3.1 versus direct XML by difficulty

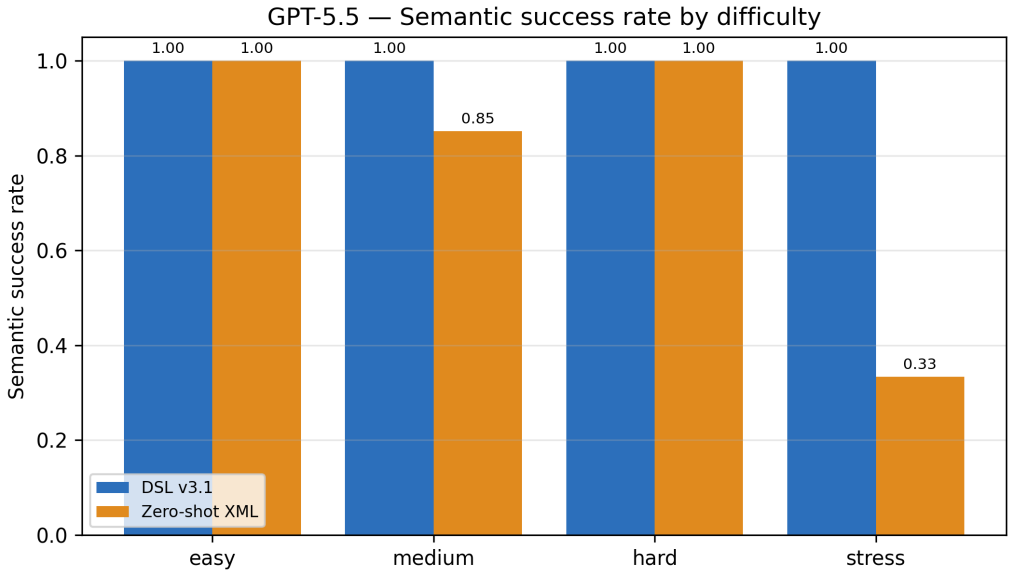

Figure 5.9: DSL and direct XML internal-analysis success rate by difficulty

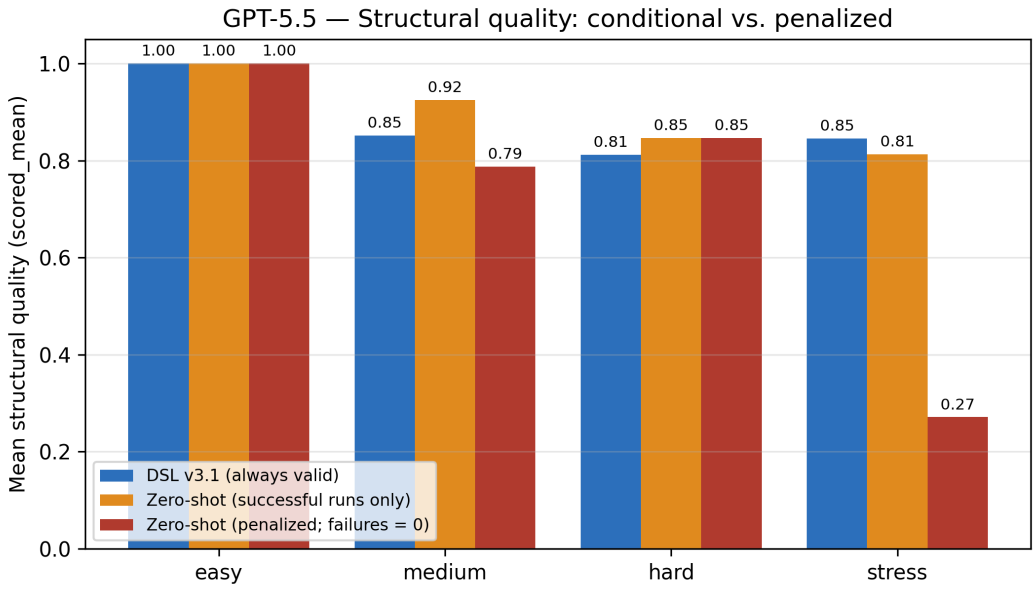

Figure 5.10: Conditional and penalized quality by difficulty

Stratification shows where the contrast appears. In the stress level, conditional quality of

0.813 comes from one valid output out of three; with two zeros, it falls to 0.271. In the nine observed easy cases, both configurations achieve 1.00, describing equality in this sample rather than statistical equivalence. The 33% stress success rate means 1/3 and is highly uncertain. The clearest evidence favors DSL in success rate: six XML outputs violate XSD compared with none for GPT-5.5 DSL.

Approach	Input tokens	Output tokens
DSL v3.1	11.186	197
DSL v4	13.519	252
DSL v5	15.160	263
Direct XML	393	2.979

Table 5.17: Textual token estimate per GPT-5.5 generation

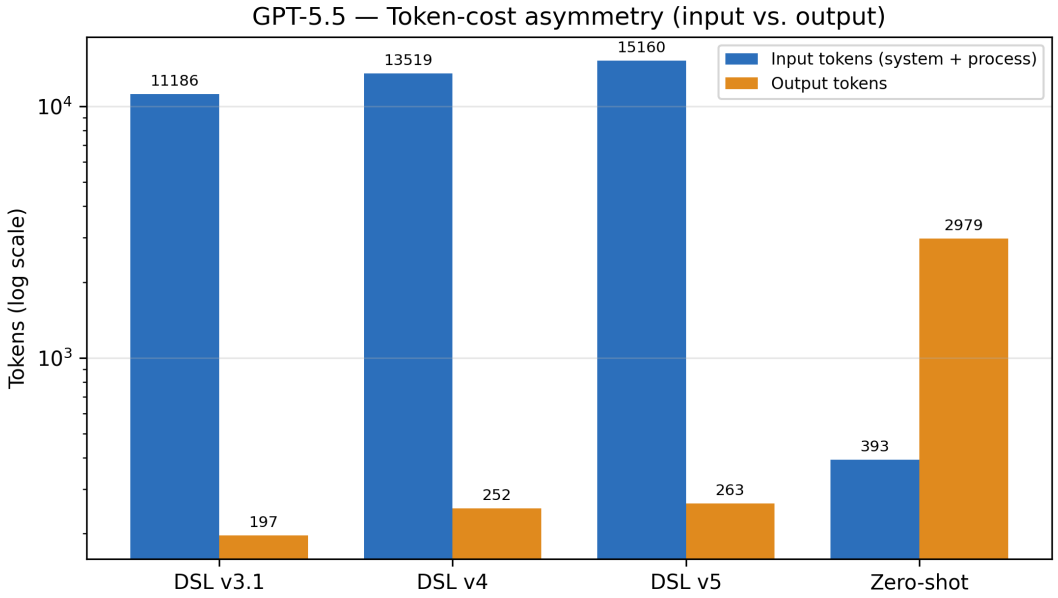

Figure 5.11: Input–output token asymmetry by representation

The estimated text-length distribution is almost inverse: DSL concentrates text in instructions and XML in the output. The figures use `o200k_base`; they do not represent all providers' actual tokenizers, caching, latency, prices, or hidden reasoning. They therefore do not support an economic conclusion.

Comparison with direct image generation and MCP servers is left for future work. These modalities do not necessarily generate BPMN-XML with the same degree of formality. Their evaluation should use common dimensions such as success, time, tokens, and requirement coverage, without applying irrelevant structural metrics.

5.13 Formative observations and applicability limits

During development, informal observations were received from SGS Productivity colleagues concerning Spanish-language cases (Section 4.9), without records of participants, tasks, criteria, or reproducible results. Nor are any rows in the consolidated set used in this chapter identified as real processes. Consequently, these observations are not part of the evaluation and do not support conclusions about usefulness, comprehensibility, or external validity. Applicability to real processes, non-expert users, and Spanish inputs remains a limitation and future-work topic; any subsequent study should document anonymization, privacy, consent, protocol, and results.

5.14 Statistical significance

The three models solve the same fifteen processes, and each process has three replicates. To respect this structure, replicate means are first computed for each process–model pair, producing fifteen paired blocks. The global comparison uses Friedman [42]; pairs are tested with the Wilcoxon signed-rank test [43], and the three p values are adjusted using Holm [44]. All three are non-parametric tests, suitable for small samples where a normal distribution cannot be assumed: Friedman tests for any global difference among the three configurations solving the same processes; Wilcoxon compares configurations pairwise; and Holm correction accounts for multiple simultaneous comparisons, reducing the risk of treating a chance difference as significant.

Test	Statistic	p-value	Interpretation
Friedman	$\chi^2 = 1{,}322$	$p = 0{,}516$	No global configuration effect is detected.

Table 5.18: Global statistical-significance test

Pairwise comparisons are shown in Table 5.19:

Comparison	p Wilcoxon	p Holm
`GPT_5_5_THINKING` vs. `OPUS_4_7`	0,470	0,941
`GPT_5_5_THINKING` vs. `GEMINI_3_5_FLASH_THINKING`	0,088	0,265
`OPUS_4_7` vs. `GEMINI_3_5_FLASH_THINKING`	0,804	0,941

Table 5.19: Process-paired comparisons using Wilcoxon and Holm correction

Although `GPT_5_5_THINKING` achieves the highest descriptive mean, the design detects no differences at the 0.05 level. This result does not demonstrate equality or equivalence; it indicates that, with fifteen processes and these proxies, there is insufficient evidence to rank configurations by global quality.

The failure profile provides a more useful reading than the aggregate mean. Some models tend to omit collaboration between pools; others have fragment-anchoring problems; others generate branches that do not reach an explicit end. For system improvement, knowing this error pattern is more informative than a statistically non-significant mean difference.

5.15 Threats to validity

5.15.1 Internal validity

The separation between `semantic_success` and `render_success` locates stages but does not fully separate responsibilities: grammar, normalizations, and compiler interact with model output. The corpus, difficulty, and expected constructions were defined by the author without independent annotation. Some *prompt* examples share motifs with `SYN009`/`SYN010`, creating a risk of overlap. In addition, v5 changes instructions, grammar, and engine simultaneously.

The DSL–XML comparison contrasts two configurations with different instruction richness and therefore does not identify the pure effect of representation. Conditional quality suffers from survival bias and is therefore presented alongside success rate and penalized quality. Runs were performed at different times using closed services subject to updates.

5.15.2 Construct validity

The absence of references and human evaluation prevents claims of semantic equivalence, clarity, or completeness. The multiplicity of valid diagrams justifies not relying on a single reference, but does not rule out multiple references, declarative constraints, trace equivalence, or expert consensus.

The `scored_mean` combines equally weighted proxies. M2 does not penalize every broken reference in its value; M4/M8 may be perfect when gateways are absent; and detectors check presence more than relations. These defects may favor incomplete diagrams. Values are retained for traceability, but conclusions are limited to what each indicator observes.

5.15.3 External validity

The English synthetic corpus does not represent the variety of real domains, languages, and styles. Formative observations in Spanish lack a protocol and are not incorporated as validation. Results generalize only to the fifteen processes and configurations studied.

5.15.4 Statistical validity

The paired analysis avoids treating the 45 rows per model as independent processes, but fifteen blocks provide limited power and do not allow $p > 0{,}05$ to be interpreted as equivalence.

Exploratory models have incomplete samples and are not compared with the main matrix.

The study's most pronounced result requires an additional qualification. The stress difficulty level is represented by one synthetic process with three repetitions per model. The marked reduction in quality and success observed at that level, in both RQ3 and RQ8, should be read as an indication of the approach's breaking point rather than as a generalizable estimate. Reliable characterization of behavior on highly complex processes would require several additional stress processes.

5.16 Reproducibility

The evaluation was designed to regenerate results from repository artifacts. However, a mutable repository does not replace an immutable *release*: the thesis should be associated with a *commit* or tag, a manifest of the 420 rows, and hashes of consolidated files. In the absence of these data in the current version, reproducibility is considered partial. Table 5.20 lists the available elements:

Artifact	Function
`TFM-eval/results/experiments/results.csv`	Main table with metrics per experiment.
`TFM-eval/results/experiments/results.json`	Detailed results, aggregates, and diagnostics.
`TFM-eval/results/experiments/figures`	Comparative figures generated by the evaluation. This thesis incorporates the final figures available in `imagenes/`.
`docs/analisis-derivado/build_report.py`	Recalculates penalized quality, version ablation, and comparative figures.
`TFM-eval/src/bpmn_eval/usage.py`	Estimates textual token consumption and documents its limits.
`docs/zero-shot-bpmn-xml-harness.md`	Specifies the reproducible protocol for direct BPMN-XML.
`TFM-eval/notebooks/analysis.ipynb`	Analysis and visualization notebook.
`TFM-eval/README.md`	Methodological design and evaluation decisions.

Table 5.20: Evaluation reproducibility artifacts

A complete replication should record Node, pnpm, Python, Playwright/Chromium, and dependency versions; the exact command; overwritten files; model configuration; actual timestamp; seed when available; and the reason for excluding folders. Closed models may be updated, so this documentation supports repeating the *pipeline* but does not guarantee recovering the same generation.

5.17 Evaluation conclusions

The evaluation answers the research questions in four main blocks:

1. **Technical robustness (RQ1).** The DSL configurations achieve high internal-analysis success in the corpus, and no additional rendering failures are observed among compilable outputs. These results establish technical feasibility, while the partial metrics do not demonstrate *soundness*, clarity, or complete fidelity to the description.

2. **Model comparison, difficulty, and stability (RQ2, RQ3, and RQ7).** The statistical tests do not identify a winning model–platform configuration or prove equivalence. The decline in more complex strata and the dispersion across replicates are descriptive patterns, constrained by three repetitions and by a stress stratum containing one process.

3. **Construction coverage and prompt evolution (RQ4 and RQ5).** Collaboration features are the most frequent omissions detected. The higher coverage of v5 belongs to a configuration that changes instructions, syntax, and engine together, so it cannot be attributed to the *prompt* alone.

4. **Description length (RQ6).** This question remains unanswered because the available descriptions were not designed as semantically equivalent length variants. Answering it requires a controlled dataset rather than a post-hoc correlation with token count.

5. **DSL–XML comparison (RQ8).** Direct XML provides higher conditional quality and coverage for GPT-5.5, whereas DSL v3.1 provides higher success and observed failure-penalized quality. The difference in penalized quality is not significant, and the two treatments do not isolate the effect of representation.

Overall, the evaluation supports the technical feasibility of the DSL-plus-compiler *pipeline* in the corpus studied, but it does not establish global semantic superiority or external validity. The available artifacts allow the results to be recalculated; complete replication would additionally require an immutable *release*, environment, and manifest. Chapter 6 relates these findings to the thesis contributions and outlines the resulting priorities.

Chapter 6

Conclusions and Future Work

This chapter interprets the findings in relation to the thesis objective, summarizes the contributions, and identifies the limitations that define the next steps. The hypothesis receives partial support: in the configurations studied, the DSL-plus-compiler is more robust than direct XML with a short *prompt*, but it does not achieve higher quality in every dimension, nor has full semantic fidelity been demonstrated.

6.1 Conclusions

The thesis makes four contributions that connect the implemented system with the empirical evidence:

1. **Technical contribution.** A natural-language description can be processed through a restricted textual DSL, a deterministic compiler, automatic layout, and validation within a shared engine. Two applications and the evaluation *pipeline* reuse this foundation, making the system functional, traceable, and extensible rather than a collection of isolated prototypes.

2. **Methodological contribution.** The evaluation protocol separates internal-analysis success, rendering, conditional quality, failure-penalized quality, construction coverage, and diagnostics. This separation prevents file validity or a saturated proxy from being presented as complete semantic correctness and makes failed generations visible in aggregate comparisons.

3. **Empirical contribution and hypothesis assessment.** The balanced comparison shows that the DSL-plus-compiler approach is technically viable and, for GPT-5.5, more robust than direct BPMN-XML under the tested instructions. However, conditional quality and coverage reveal a trade-off, the statistical analysis does not rank the model–platform configurations, and the evidence does not establish that the DSL is globally superior or semantically equivalent to the source description.

4. **Boundary of the contribution.** The study identifies collaboration and complex cases as the main areas requiring improvement. Its evidence comes from synthetic processes,

three replicates per process, partially validated proxies, and changing closed platforms; formative feedback from colleagues and expert mentoring does not constitute an external-validity study. The results should therefore be interpreted as evidence of feasibility within the evaluated corpus, not as a general estimate of real-world performance.

These conclusions draw primarily on the balanced matrix of 135 experiments. The additional observations in the consolidated set support specific prompt-version and representation analyses, but are not merged into a single comparison across incompatible configurations.

Taken together, the work moves the problem from merely producing valid BPMN files to evaluating whether the generated models preserve the intended process. The main open issue is therefore fidelity, especially for collaboration and complexity. Section 5.17 provides the detailed answers to the evaluation questions; the following directions derive from those answers without treating the observed differences as causal effects of format alone.

6.2 Future work

6.2.1 Improving collaboration between participants

The priority is to reinforce participant separation and the use of message flows. Three avenues are proposed: refining the *system prompt*, exploiting v5's explicit *pool–lane* map, and extending validation to warn when a process with several actors is concentrated in a single participant.

6.2.2 Refining the *system prompt*

The main matrix uses v3.1 as the controlled variable, whereas the configuration comparison shows higher coverage with v5. Isolating causes will require varying instructions, grammar, and compiler separately, identifying which blocks explain each change, and studying possible side effects such as longer inputs, unnecessarily large diagrams, or new normalizations.

6.2.3 Closed-loop self-repair

In the current implementation, the LLM does not receive the diagnostics generated by the *parser* or renderer. Returning these errors to the model together with a correction request could recover some cases that currently fail. This extension could integrate *self-repair* techniques with deterministic validation while keeping an explicit record of every attempt.

6.2.4 Extending the *zero-shot* comparison

Direct BPMN-XML comparison has already been performed for GPT-5.5. Its extension should include other models and modalities based both on images and on MCP servers. For the latter,

visual quality, requirement coverage, time, tokens, iterations, tool errors, and symbol richness should be recorded, distinguishing common metrics from those requiring structured BPMN-XML.

6.2.5 Reference-based semantic evaluation

The evaluation used does not require a reference diagram, which prevents claims of complete semantic equivalence. Blind annotation of a subset by several experts, using a rubric, agreement measurement, and disagreement resolution, would extend the qualitative dimension. An *LLM-as-a-judge* should only be added after calibration against those human judgments and analysis of representation-related bias.

6.2.6 Symbolic verification and orchestration

Introducing soundness analysis through Petri nets should first define the supported BPMN subset and translation semantics before identifying deadlocks, branches without an end, and synchronization problems. An architecture with specialized agents and quality gates for planning, generating, verifying, and repairing complex processes could also be studied.

6.2.7 Extending external validity

Applicability should be studied with documented participants, tasks, criteria, instruments, consent, privacy, and results across several domains and description styles. Analyzing input length requires semantically equivalent versions with short, medium, and long levels of detail.

6.2.8 API access and controlled hyperparameters

The experimental phase used public web interfaces. This decision favors ecological validity but limits control over *temperature*, *top-p*, seeds, and exact versions. An API replication would allow the sample size to be increased, configurations to be fixed, and small differences to be analyzed with greater statistical power.

6.2.9 Expanding the model set

Before adding models, the number of processes per stratum should be increased and levels balanced, because the primary unit is the process and the stress stratum contains one case. The Opus 4.8 Medium v3.1 matrix could then be completed and the paired analysis recalculated with four configurations.

In conclusion, the work culminates in a functional, traceable, and evaluable system accompanied by clearly delimited directions for development. The challenge is no longer merely to

produce valid BPMN, but also to represent collaboration and complexity faithfully. To this end, the *prompt*, DSL, compiler, and evaluation protocol can be refined independently on a common foundation.

Chapter 7

Repository Organization

This chapter does not repeat the engine's internal architecture, already described in Chapter 4. Its purpose is to explain the traceability between code, raw data, and derived artifacts.

The code, experiments, and documentation are available at `https://github.com/ignagp34/BPMN-DSL-Monorepo` [31]. The repository facilitates auditing, but its active branch is mutable and does not replace the snapshot used for the PDF; a replication must pin the *commit*, *release*, environment, and result hashes.

7.1 Consolidation in a monorepo

The code was consolidated in a *monorepo* once configuration v4 reached development stability. At that point, the informal test with SGS Productivity colleagues described in Section 4.9 took place. Consolidation brought together the application, laboratory, and shared engine so that the demonstration and experiments would not evolve from separate implementations.

7.2 Integration of the evaluation package

The second organizational decision was to integrate `TFM-eval`, which had initially been independent. This integration improved traceability by linking the engine, experiments, and metric calculation in a common history. It does not by itself guarantee historical reproducibility: a later modification may regenerate different artifacts unless the exact version is preserved.

7.3 Documentation and automation

The documentation and automation folders complete the structure. `docs` contains decisions and derived artifacts, while `scripts` contains rendering and evaluation orchestration. Taken together, the repository documents a tool with formative use and a partially auditable experimental environment.

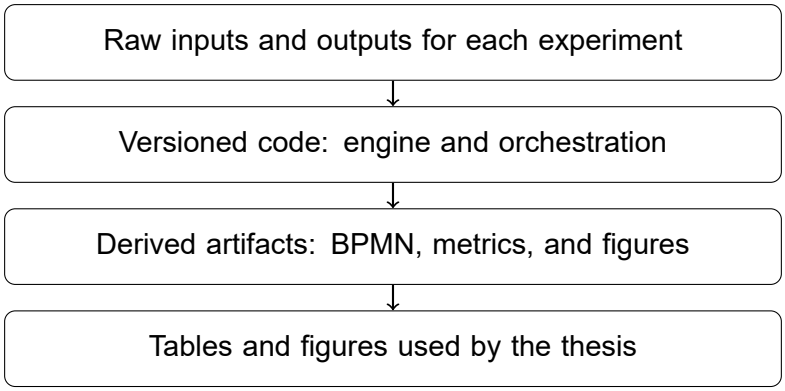

Figure 7.1: Relationship between raw data, code, derived artifacts, and the thesis. The chain must record the version and hashes at each transition.

List of Acronyms

AI Artificial Intelligence.

API Application Programming Interface.

AST Abstract Syntax Tree.

BPM Business Process Management.

BPMN Business Process Model and Notation.

DAG Directed Acyclic Graph.

DSL Domain-Specific Language.

FSM Finite State Machine.

LLM Large Language Model.

MAO Multi-Agent Orchestration.

MCP Model Context Protocol.

NLP Natural Language Processing.

POWL Partially Ordered Workflow Language.

TFM Master's Thesis.

XML eXtensible Markup Language.

XSD XML Schema Definition.

Bibliography

[1] H. Kourani, A. Berti, D. Schuster, and W. M. van der Aalst, “Promoai: Process modeling with generative ai,” in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, 2024, pp. 8708–8712.

[2] K. Busch, A. Rochlitzer, D. Sola, and H. Leopold, “Just tell me: Prompt engineering in business process management,” *arXiv preprint arXiv:2304.07183*, 2023.

[3] C. Lauer, P. Pfeiffer, A. Rombach, and N. Mehdiyev, “Assessing the business process modeling competences of large language models,” *arXiv preprint arXiv:2601.21787*, 2026. [Online]. Available: https://arxiv.org/abs/2601.21787

[4] A. Safan and J. Köpke, “BPMN-Chatbot++: LLM-based modeling of collaboration diagrams with data,” in *Business Process Management (BPM 2025)*. Springer, 2025.

[5] N. Klievtsova, J.-V. Benzin, T. Kampik, J. Mangler, and S. Rinderle-Ma, “Process modeler vs. chatbot: Is generative ai taking over process modeling?” in *Business Process Management Workshops*. Springer, 2025, pp. 637–649.

[6] L. F. Hörner, “Towards an llm-based conversational framework for business process modeling: Research approach and preliminary results,” in *Advanced Information Systems Engineering*. Springer, 2025, pp. 286–293.

[7] Object Management Group, “Business Process Model and Notation (BPMN), Version 2.0.2,” Object Management Group, Tech. Rep. formal/2013-12-09, Jan. 2014. [Online]. Available: https://www.omg.org/spec/BPMN/2.0.2/PDF

[8] L. Lin, Y. Jin, Y. Zhou, W. Chen, and C. Qian, “MAO: A framework for process model generation with multi-agent orchestration,” *arXiv preprint arXiv:2408.01916*, 2024.

[9] P. Drakopoulos, P. Malousoudis, N. Nousias, G. Tsakalidis, and K. Vergidis, “Do LLMs speak BPMN? an evaluation of their process modeling capabilities based on quality measures,” *Computation*, 2026.

[10] A. Ivanchikj, S. Serbout, and C. Pautasso, “Live process modeling with the BPMN Sketch Miner,” *Software and Systems Modeling*, vol. 21, no. 5, pp. 1877–1906, 2022.

[11] C. Ziche and G. Apruzzese, “Llm4pm: A case study on using large language models for process modeling in enterprise organizations,” in *Lecture Notes in Business Information Processing 527 LNBIP*. Springer, 2024, pp. 472–483.

[12] Metadev, “Metadev,” 2026, accessed June 19, 2026. [Online]. Available: https://metadev.pro/

[13] M. Grohs, L. Abb, N. Elsayed, and J.-R. Rehse, “Large language models can accomplish business process management tasks,” in *Business Process Management Workshops*. Springer, 2024, pp. 453–465.

[14] A. Rebmann, F. D. Schmidt, G. Glavaš, and H. Van Der Aa, “Evaluating the ability of llms to solve semantics-aware process mining tasks,” in *2024 6th International Conference on Process Mining (ICPM)*. IEEE, 2024, pp. 9–16.

[15] T. Kampik, C. Warmuth, A. Rebmann, R. Agam, L. N. Egger, A. Gerber, J. Hoffart, J. Kolk, P. Herzig, G. Decker *et al.*, “Large process models: Business process management in the age of generative ai,” *arXiv preprint arXiv:2309.00900*, 2023.

[16] S. Wenger, M. Spahic-Bogdanovic, and A. Martin, “Large language models for democratizing business process modeling: Bpmn model generation and style guide adherence,” in *Artificial Intelligence Research*. Springer, 2025, pp. 363–379.

[17] J. T. Licardo, N. Tanković, and D. Etinger, “BPMN assistant: An LLM-based approach to business process modeling,” *Applied Sciences*, 2026.

[18] N. Klievtsova, T. Kampik, J. Mangler, and S. Rinderle-Ma, “Conversationally actionable process model creation,” in *Cooperative Information Systems*. Springer, 2025, pp. 39–55.

[19] F. Friedrich, J. Mendling, and F. Puhlmann, “Process model generation from natural language text,” in *Advanced Information Systems Engineering*. Springer, 2011, pp. 482–496.

[20] P. Bellan, M. Dragoni, C. Ghidini, H. van der Aa, and S. P. Ponzetto, “Process extraction from text: Benchmarking the state of the art and paving the way for future challenges,” *arXiv preprint arXiv:2110.03754*, 2023.

[21] J. T. Licardo, N. Tanković, and D. Etinger, “A method for extracting bpmn models from textual descriptions using natural language processing,” *Procedia Computer Science*, vol. 239, pp. 483–490, 2024.

[22] Q. Nivon and G. Salaün, “Automated generation of bpmn processes from textual requirements,” in *Service-Oriented Computing*. Springer, 2025, pp. 185–201.

[23] A. Berti, X. Wang, H. Kourani, and W. M. P. van der Aalst, “Specializing large language models for process modeling via reinforcement learning with verifiable and universal rewards,” *Process Science*, 2025.

[24] L. F. Hörner, M. Möller, and M. Reichert, “Automatically generating BPMN 2.0 process models from natural language process descriptions: Challenges, framework, quality assessment,” *Business & Information Systems Engineering*, 2026.

[25] P. K. Menaka Sekar, I. Matei, M. Zhenirovskyy, H. Y. Wong, S. Kohmura, S. Hotta, and A. Inomata, “Automated BPMN model generation from textual process descriptions: A multi-stage LLM-driven approach,” in *2026 IEEE International Systems Conference (SysCon)*. IEEE, 2026.

[26] H. Kourani and S. J. van Zelst, “POWL: partially ordered workflow language,” in *Business Process Management: 21st International Conference (BPM 2023)*. Springer, 2023, pp. 92–108.

[27] A. Brissard, F. Cuppens, and A. Zouaq, “What is the best process model representation? a comparative analysis for process modeling with large language models,” in *AI4BPM Workshop at BPM 2025*, 2025, preprint arXiv:2507.11356.

[28] N. Ayoughi, D. Dewar, S. Nejati, and M. Sabetzadeh, “DSL or code? evaluating the quality of LLM-generated algebraic specifications: A case study in optimization at kinaxis,” in *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP ’26)*, 2026.

[29] H. Kourani, A. Berti, D. Schuster, and W. M. van der Aalst, “Process modeling with large language models,” in *Enterprise, Business-Process and Information Systems Modeling*. Springer, 2024, pp. 229–244.

[30] J. Neuberger, L. Ackermann, H. van der Aa, and S. Jablonski, “A universal prompting strategy for extracting process model information from natural language text using large language models,” in *Conceptual Modeling*. Springer, 2024, pp. 38–55.

[31] I. González Peris, “BPMN-DSL-Monorepo: engine, experimental laboratory, and evaluation pipeline for generating BPMN 2.0 through an intermediate DSL,” https://github.com/ignagp34/BPMN-DSL-Monorepo, 2026, project software repository. A commit or release must be pinned for historical reproducibility. Last accessed June 2026.

[32] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, 2020, pp. 1877–1901.

[33] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, “Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing,” *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[34] OpenAI, “Introducing GPT-5.5,” Apr. 2026, accessed July 11, 2026. [Online]. Available: https://openai.com/index/introducing-gpt-5-5/

[35] Google, “Gemini 3.5 flash,” 2026, accessed July 11, 2026. [Online]. Available: https://ai.google.dev/gemini-api/docs/models/gemini-3.5-flash

[36] Anthropic, “Claude opus 4.7,” 2026, accessed July 11, 2026. [Online]. Available: https://www.anthropic.com/claude/opus

[37] W. R. Shadish, T. D. Cook, and D. T. Campbell, *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin, 2002.

[38] M. Sclar, Y. Choi, Y. Tsvetkov, and A. Suhr, “Quantifying language models’ sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting,” in *International Conference on Learning Representations (ICLR)*, 2024.

[39] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang *et al.*, “Self-refine: Iterative refinement with self-feedback,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[40] T. Scholak, N. Schucher, and D. Bahdanau, “PICARD: Parsing incrementally for constrained auto-regressive decoding from language models,” in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

[41] J. Cardoso, “Control-flow complexity measurement of processes and weyuker’s properties,” in *Proceedings of World Academy of Science, Engineering and Technology*, vol. 8, 2005, pp. 213–218.

[42] M. Friedman, “The use of ranks to avoid the assumption of normality implicit in the analysis of variance,” *Journal of the American Statistical Association*, vol. 32, no. 200, pp. 675–701, 1937.

[43] F. Wilcoxon, “Individual comparisons by ranking methods,” *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[44] S. Holm, “A simple sequentially rejective multiple test procedure,” *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65–70, 1979.

Appendices

Appendix A

Additional Material and BPMN Reproducibility Artifacts

A.1 DSL and *system prompt* specification

The complete grammar of the textual DSL is preserved in the following file:

`apps/tfm-lab/prompts/system/bpmn_sketch_miner_system_prompt_v3_1.md`

This document has a dual function: it specifies the language and provides the model's *few-shot* conditioning. In the experiments corresponding to the main matrix, version v3.1 is used as the controlled variable.

The artifact consists of 1,201 lines and has the following SHA-256 hash (computed over the file as preserved in the repository):

`5A574F36BE54EBD8B6E22F65D747974265616B15A547B893170365995CCC247F`

Its main blocks document:

- the output format and fundamental DSL rules;
- tasks, roles, *pools*, *lanes*, gateways, events, and data elements;
- parallelism, decisions, and trace-based modelling;
- fragments and anchoring constraints, including anti-patterns AP-6 and AP-7;
- complete *few-shot* examples; and
- a final checklist to complete before returning the response.

References in the methodology to internal *prompt* sections, anti-patterns, and their examples point to this versioned electronic artifact. Although later versions are located in `apps/tfm-lab/prompts/system`, they do not form part of the results obtained with the v3.1 matrix.

A.2 Complete results and experimental traceability

The consolidated results are provided in `TFM-eval/results/experiments/results.csv` and `TFM-eval/results/experiments/results.json`. The first file facilitates tabular analysis, while the second preserves the complete structure of metrics and diagnostics. In addition, the experimental folder for each run preserves the input, normalized output, diagram, metadata, and validation result.

Set	N	Use in the thesis
Main v3.1 matrix	135	Three configurations, fifteen processes, and three replicates; supports the paired analysis.
Exploratory v3.1	42	Incomplete configurations; 177 v3.1 rows minus the 135 main rows.
Consolidated set	420	177 v3.1 rows, 99 v4, 99 v5, and 45 direct BPMN-XML.
Raw inventory consulted	441	Folders; 21 do not appear in `results.csv` and require an identifier-based exclusion reason.

Table A.1: Relationship among the experimental sets used.

The inclusion relation is 441 located folders $\rightarrow$ 420 consolidated rows $\rightarrow$ 135 main-matrix rows. The 21 exclusions require an identifier-based manifest to be fully auditable. Tables can be regenerated using `scripts/run-evaluation.ps1`. To pin a run, the engine version, date, model, process, replicate, and configuration must be recorded. Metrics include `semantic_success`, `render_success`, `scored_mean`, M1–M5, M8, and construction coverage.

A.3 End-to-end example

The process of a waste-treatment plant is included as an illustrative example. Figure A.1 shows the available export, but this version does not identify a `.bpm` package, DSL, BPMN-XML file, or exact path that would allow the complete chain to be audited. The case is therefore not presented as reproducible end-to-end evidence.

The case visually illustrates the final part of the chain and the difficulty of preserving participants, *lanes*, events, decisions, and branches. To turn it into an auditable example, the description, raw output, normalization, DSL, BPMN, diagnostics, and metrics must be published together with identifiers and hashes.

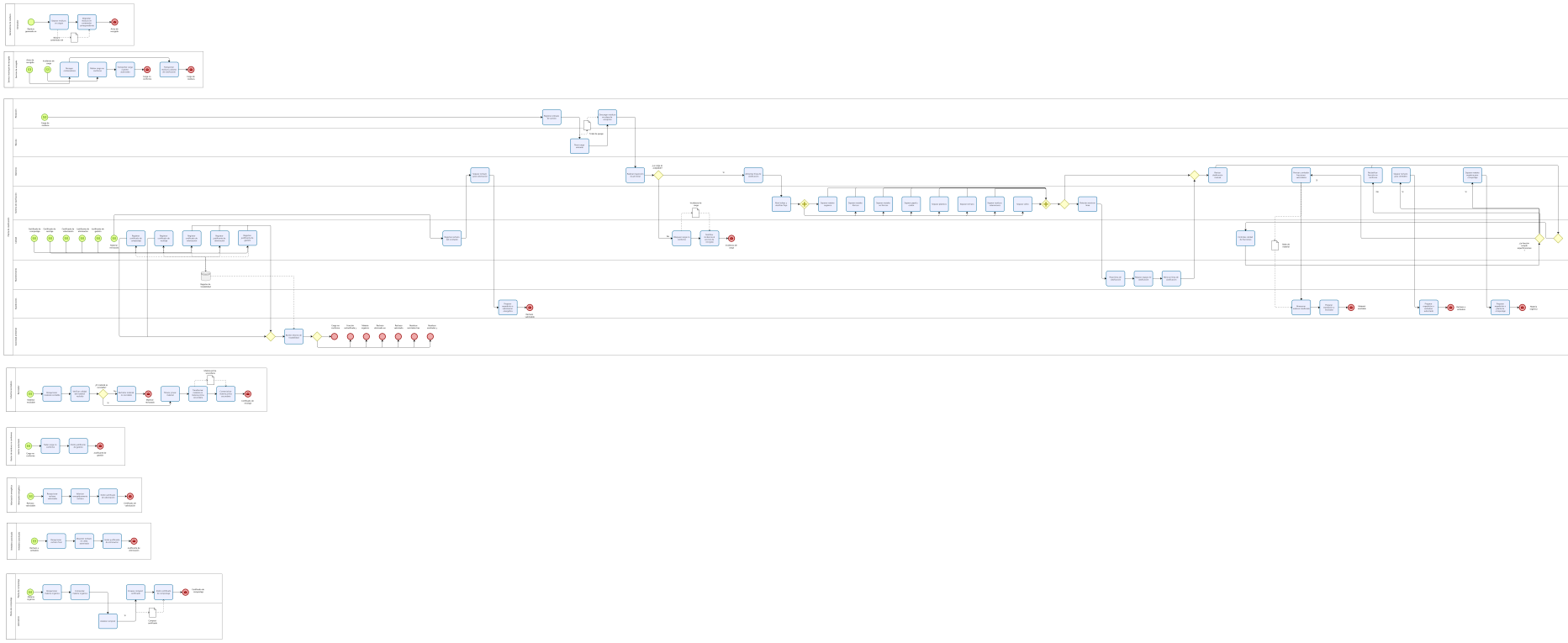

Figure A.1: BPMN model of the waste-treatment plant.

A.4 Gallery of *canon* diagrams

The *canon* diagrams are used as end-to-end regression cases. Their sources are available in `packages/bpmn-core/tests/fixtures`, and any engine modification must preserve the visual and structural properties represented by these cases.

Case	Process	Checked aspects
canon-1	Cheeseburger preparation	Roles, parallel stations, timer, and explicit start and end events.
canon-2	Incoming flight	Participants, events, and flow continuity between related activities.
canon-3	Conveyor belt	Orthogonal routing and separation between branches and process elements.
canon-4	Composting	Decisions, joins, and readability in processes with several alternatives.
canon-5	Wedding	*Lanes*, collaboration, branches, and join placement in a larger process.

Table A.2: Canonical diagrams used as regression tests.

The automatic suite checks that sources are parsed and artifacts generated without encoded regressions. Visual inspection follows a separate checklist: (1) no overlap between nodes; (2) arrows do not cross activities; (3) readable labels; (4) separation of branches and joins; and (5) coherent *pool*/*lane* boundaries. The thesis does not preserve evaluator, date, or result by case, so this inspection is considered a development control rather than an experimental metric.

In the project's electronic version, each DSL entry is preserved together with its generated rendering. This correspondence makes it possible to compare the textual source with the graphical result without manually interpreting the change history.

A.5 Diagnostic-code catalogue

Diagnostics make it possible to determine whether a failure originates in DSL analysis, the BPMN model, or the *layout*. Table A.3 presents input errors, and Table A.4 summarizes findings from later stages.

Code or family	Origin	Interpretation
`IMPLICIT_START_EVENT`, `IMPLICIT_END_EVENT`, `IMPLICIT_BOUNDARY_END_EVENT`	Model	Elements synthesized by the engine to complete a trace.

Code or family	Origin	Interpretation
MISSING_EXPLICIT_START_EVENT, MISSING_EXPLICIT_END_EVENT	Model	The DSL does not explicitly declare the start or end.
SEQUENCE_FLOW_INVALID_REFERENCE	Model	A sequence flow points to a nonexistent element.
MESSAGE_FLOW_INVALID_REFERENCE, MESSAGE_FLOW_INVALID_ENDPOINT	Model	Invalid reference or endpoint in a message flow.
MESSAGE_FLOW_WITHIN_POOL	Model	A message flow is used within the same participant.
POOL_BOUNDARY_CONTROL_FLOW	Model	A control flow improperly crosses a participant boundary.
POOL_MAP_DUPLICATE_LANE, POOL_MAP_LANE_UNASSIGNED	Model	A *lane* is duplicated or unassigned in the explicit map.
POOL_MAP_NAME_COLLISION, POOL_MAP_UNKNOWN_LANE	Model	Name collision or reference to an unknown *lane*.
LAYOUT_ARTIFACT_TOO_FAR	*Layout*	An artifact is excessively far from its associated element.
LAYOUT_ELEMENT_OVERLAP	*Layout*	Two graphical elements overlap.
LAYOUT_UNREADABLE_LABEL	*Layout*	A label has no readable placement.
LAYOUT_XML_PARSE_ERROR	*Layout*	XML cannot be parsed during the graphical phase.

Table A.4: Model and *layout* subsystem diagnostics.

Lexical/syntactic analysis codes and invalid references may prevent `semantic_success`; `LAYOUT_*` codes may affect `render_success`; `IMPLICIT_*` normalizations and `MISSING_*` warnings do not necessarily reduce `scored_mean`. Thus, a run with warnings may receive a high score even though the engine completed elements omitted by the LLM. Exact severity should be consulted in each run's JSON result.

Code	Interpretation
`LEX-1`, `PARSE-1`	The text cannot be tokenized or grouped according to the grammar.
`AP-6`, `AP-7`	Violation of construction or fragment-boundary rules.
`ANCHOR-MISSING`	The anchor required to continue a trace cannot be found.
`INLINE-COMMENT`	A comment is detected in a disallowed position.
`FRAGMENT-OPEN`	A fragment remains open at the end of analysis.
`FRAGMENT-DUP`	The same fragment or anchor is declared more than once.
`DANGLING-QUESTION`	A decision has no valid associated branches.

Table A.3: Lexical and syntactic analyzer diagnostics.

A.6 Evolution of the *system prompt*

Version	Use	Main changes
v1	Development only	Initial iteration; it was not used as a controlled experimental baseline.
v2	Development only	Development iteration in which fragment-boundary failures remained.
v3	Development only	Made trace-based notation the default and introduced the anchor-boundary rule and AP-6/AP-7.
v3.1	Main experimental matrix	Re-audited the evidence: AP-6 remained observed; AP-7 was retained as a predicted risk because the alleged observed case was a misdiagnosed return start event. It was the first sufficiently validated controlled baseline, not the first version capable of producing a valid diagram.
v4	Subsequent comparison	Additional readability advice and improvements in end-event generation.
v4.1	Refinement excluded from the 420 rows	Adjustments derived from tests on canonical cases.
v5	Subsequent configuration included in RQ5/RQ8	Instructions, syntax, and engine support for explicit *pool–lane* mapping.
ZSXML	Direct BPMN-XML comparison	Brief *zero-shot prompt*, without examples equivalent to v3.1.

Table A.5: Functional evolution of the *system prompt*.

The early changelog sequence used in this thesis is v1 $\rightarrow$ v2 $\rightarrow$ v3 $\rightarrow$ v3.1; it does not contain a v1.2. The status of AP-7 in the v3.1 row describes the evidence available when the baseline was fixed. The AP-7 occurrences later recorded for Opus and Gemini in Table 5.12 are experimental observations and do not alter that historical account.

The artifacts stored in `apps/tfm-lab/prompts/system` have the following sizes and SHA-256 hashes (computed over the files as preserved in the repository):

v3.1 1,201 lines: `5A574F36BE54EBD8B6E22F65D747974265616B15A547B893170365995CCC247F`

v4 1,347 lines: `FF470D282CD40DCBC31E9B313388B2438B3823C19C7526362631ADA4CDD40E50`

v4.1 Artifact referenced in development; its line count and SHA-256 were not incorporated into this snapshot.

v5 1,408 lines: `70FCF2DB9276A3E8B59E2125779D6767DCD25C0177F5F4F74FA7B62E9A82BC16`

ZSXML Direct-comparison *prompt*; its SHA-256 was not incorporated into this snapshot.

The main matrix uses v3.1 only. The consolidated set also includes v4, v5, and ZSXML for their specific analyses; v4.1 documents the evolution but is not part of the 420 rows.

A.7 Synthetic process set

The experimental corpus consists of fifteen processes identified as `SYN001`–`SYN015`. The specification of each one is preserved in `apps/tfm-lab/prompts/processes` and indicates both its difficulty level and expected BPMN features.

Process	Difficulty	Lang.	Design observation
`SYN001`	Easy	English	Basic case; overlap with examples not audited.
`SYN002`	Easy	English	Basic case; overlap with examples not audited.
`SYN003`	Easy	English	Basic case; overlap with examples not audited.
`SYN004`	Medium	English	Intermediate constructions; overlap not audited.
`SYN005`	Medium	English	Intermediate constructions; overlap not audited.
`SYN006`	Medium	English	Intermediate constructions; overlap not audited.
`SYN007`	Medium	English	Intermediate constructions; overlap not audited.
`SYN008`	Medium	English	Intermediate constructions; overlap not audited.

Process	Difficulty	Lang.	Design observation
SYN009	Medium	English	Possible thematic proximity to the v3.1 request/interview example.
SYN010	Medium	English	Possible thematic proximity to the v3.1 order/shipping example.
SYN011	Medium	English	Intermediate constructions; overlap not audited.
SYN012	Medium	English	Intermediate constructions; overlap not audited.
SYN013	Difficult	English	Events/exceptions; difficulty assigned by the author.
SYN014	Difficult	English	Advanced collaboration; difficulty assigned by the author.
SYN015	Stress	English	Only stress case; combines multiple requirements.

Table A.6: Corpus processes, difficulty, and language.

Word and *token* lengths and the complete relational list of requirements must be regenerated from the input files and fixed in the *release* manifest; they were not incorporated into the thesis snapshot. Difficulty was assigned by the author and was not independently reviewed.