



University of
Zurich^{UZH}

LLM-Assisted Generation of Validated Test Invoices in the Healthcare Insurance Domain

Jamo Sharif
Zürich, Switzerland
Student ID: 11-453-388

Supervisor: Chao Feng & Weijie Niu
Date of Submission: August 13, 2026

Kurzfassung

Krankenversicherer prüfen täglich grosse Mengen medizinischer Rechnungen automatisiert gegen Schemata und tarifliche Regeln. Damit diese Prüfsysteme zuverlässig arbeiten, braucht es realistische und vielfältige Testrechnungen, insbesondere solche, die gezielt eine einzelne Regel verletzen und dabei strukturell gültig bleiben. Heute erstellen Fachpersonen solche Rechnungen von Hand. Das ist zeitraubend und fehleranfällig, setzt vertieftes Fachwissen für Tarife voraus und deckt seltene Randfälle kaum ab.

Diese Arbeit untersucht, ob ein grosses Sprachmodell (LLM) diese Aufgabe übernehmen kann, wenn es mit verlässlichem Domänenwissen versorgt und in eine deterministische Validierung eingebettet wird. Der entwickelte Prototyp erzeugt aus einer Beschreibung in natürlicher Sprache eine XML-Testrechnung. Er kombiniert eine hybride Suche über strukturierte Tarifdaten und unstrukturierte Richtlinien, eine deterministische Transformation auf das Sumex-Format und eine Generieren-Validieren-Reparieren-Schleife gegen das XSD-Schema.

Die Evaluation verläuft zweigleisig. Quantitativ, auf einem eingefrorenen Referenzdatensatz aus echten Produktivrechnungen, erreicht der Prototyp 99,3% XSD-Konformität (100% nach Reparatur) und eine starke Verankerung der Tarifcodes (96,9%), die über acht Sprachmodelle hinweg nahezu unverändert bleibt. Qualitativ, in einer Studie mit vierzehn Fachpersonen, steigert der Prototyp die Abschlussquote von 69% auf 100% und wird als sehr benutzerfreundlich bewertet (SUS 87).

Getragen wird die Qualität von der Architektur, nicht vom einzelnen Modell. Der Prototyp ist genau dort vertrauenswürdig, wo seine Ausgaben verankert sind: Die Tarifcodes sind gut verankert, während die übrigen Geschäftsfelder (Preise, Taxpunkte und Partnerkennungen) noch nicht verankert sind und weiterhin manuell korrigiert werden müssen. Diese Felder zu verankern ist der wichtigste nächste Schritt. Ist dieser Schritt getan, eröffnet dasselbe architekturgetragene Design den Weg zu einer breiteren und verlässlicheren Testdatengenerierung im gesamten Bereich der Krankenversicherung.

Abstract

Health insurers automatically check large volumes of medical invoices against schemas and tariff rules every day. For these checking systems to work reliably, they need realistic and diverse test invoices, in particular ones that deliberately violate a single rule while remaining structurally valid. Today, domain practitioners create such invoices by hand. This is time-consuming and error-prone, requires deep tariff knowledge, and barely covers rare edge cases.

This thesis investigates whether a large language model (LLM) can take over this task when it is supplied with authoritative domain knowledge and embedded in deterministic validation. The developed prototype generates an XML test invoice from a natural language description. It combines a hybrid search over structured tariff data and unstructured guidelines, a deterministic transformation into the Sumex format, and a generate-validate-repair loop against the XSD schema.

The evaluation follows two tracks. Quantitatively, on a frozen reference dataset drawn from real production invoices, the system reaches 99.3% XSD conformance (100% after repair) and a strong grounding of the tariff codes (96.9%), which remains nearly unchanged across eight language models. Qualitatively, in a study with fourteen practitioners, the tool raises the completion rate from 69% to 100% and is rated highly usable (SUS 87).

The quality is carried by the architecture, not by the individual model. The tool is trustworthy exactly where its outputs are grounded: the tariff codes are well grounded, whereas the surrounding business fields (prices, tax points, and partner identifiers) are not yet, and therefore still require manual correction. Grounding those fields is the main next step. Once taken, the same architecture-led design points the way to broader and more reliable test-data generation across the healthcare insurance domain.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor, Chao Feng, for his continuous support, expert feedback, and engaging discussions throughout this thesis. I would also like to thank my co-supervisor, Weijie Niu, for his valuable guidance and inputs.

I am grateful to Prof. Dr. Burkhard Stiller for the opportunity to complete my Master's thesis at the Communication Systems Group (CSG) of the University of Zurich. My thanks also go to all of the ELCA/Sumex employees who helped me realise this project. In particular, I would like to thank Ayberk Bicici, Jonas Binder, Jonas Scheiwiller and Raphael Schober for their help and expertise.

Finally, I would like to thank my friends and family for their constant support, especially my brother-in-law Wais Barlas and my mother, Ghotei Sharif.

Contents

Kurzfassung	i
Abstract	iii
Acknowledgments	v
1 Introduction	1
1.1 Motivation	1
1.2 Problem Statement	2
1.3 Research Questions and Contributions	2
1.4 Description of Work	3
1.5 Thesis Outline	4
2 Background	5
2.1 Domain and Data Foundations	5
2.1.1 Healthcare Invoice Processing	5
2.1.2 Structured Data and XML	6
2.1.3 XML Schema Definition (XSD)	6
2.1.4 Semantic Validation and Business Rules	6
2.2 Test Data and Test Case Generation	7
2.2.1 Test Data in Software Validation	7
2.2.2 Challenges in Manual Test Data Creation	7
2.2.3 Synthetic Test Data	8

2.3	LLMs	8
2.3.1	Overview of LLMs	8
2.3.2	In-Context Learning	9
2.3.3	Structured Output Generation	9
2.3.4	Constraints and Controllability in Regulated Domains	9
2.4	Knowledge Storage and Retrieval	10
2.4.1	Structured Knowledge Storage	10
2.4.2	Unstructured Knowledge Storage	11
2.4.3	Graph-Based Knowledge Storage	12
2.4.4	Retrieval Techniques	12
2.5	RAG	13
2.5.1	Knowledge in LLM-Based Systems	13
2.5.2	RAG Architecture	14
2.5.3	Benefits and Limitations of RAG	15
2.6	LLM Integration Strategies in Domain-Specific Systems	16
2.6.1	Fine-Tuning and Domain Adaptation	16
2.6.2	Parameter-Efficient Fine-Tuning	17
2.6.3	Prompt Engineering and In-Context Control	18
2.6.4	Tool-Augmented and Hybrid LLM Systems	19
3	Related Work	21
3.1	Automated and Synthetic Test Data Generation	21
3.2	RAG in Knowledge-Intensive and Regulated Domains	22
3.3	Structured Test Artifacts and Constrained Generation	23
3.4	Validation-Aware and Self-Correcting Generation Pipelines	24
3.5	Comparison of Integration Strategies for LLM-Based Test Generation	25
3.6	Knowledge Storage for RAG and Controlled Test Data	25
3.6.1	Relational Storage for Authoritative Reference Data	26

3.6.2	Vector Databases for Unstructured and Semi-Structured Text . . .	26
3.6.3	Graph-Based Storage for Relational Constraints and Multi-Hop Dependencies	26
3.6.4	Implications for Controllable Test Invoice Generation	27
4	Architecture	29
4.1	Current Process in the Sumex Environment	29
4.1.1	Invoice Submission and Ingestion	29
4.1.2	Transfer to Sumex Core	31
4.1.3	Formal Validation Layer	31
4.1.4	Expert Systems and Medical/Tariff Analysis	32
4.1.5	Analytics and Anomaly Detection	32
4.1.6	Manual Review and Case Handling	32
4.1.7	Downstream Processing	32
4.2	MVP Architecture: LLM-Assisted Test Invoice Generator	33
4.2.1	Design Objectives	33
4.2.2	Architectural Design Decisions	36
4.2.3	Generation Layer	37
4.2.4	Knowledge Retrieval Layer (RAG)	37
4.2.5	Transformation and Validation Layer	38
4.2.6	Artifact Handling and Traceability	39
4.2.7	End-to-End Generation Flow	39
4.2.8	Rationale and Scope of the MVP	40
5	Implementation	41
5.1	Implementation Overview	42
5.2	System Configuration and Deployment	43
5.3	Generation Layer	43
5.3.1	Prompt Interpretation	44

5.3.2	Scenario Building	44
5.3.3	Generation Service and LLM Client	45
5.4	Knowledge Retrieval Layer	46
5.4.1	Retrieval Orchestration	46
5.4.2	Structured Tariff Retrieval	46
5.4.3	Vector Document Retrieval and Ingestion	47
5.5	Transformation and Validation Layer	48
5.5.1	Draft Parsing and Date Normalization	48
5.5.2	XML Mapping and Marshalling	49
5.5.3	XSD Validation	49
5.6	Artifact Handling and Traceability	50
5.7	Summary	51
6	Evaluation	53
6.1	Evaluation Objectives and Design	53
6.2	Experimental Setup	54
6.2.1	Source Corpus and Provenance	54
6.2.2	Ground-Truth Labeling	55
6.2.3	Data Normalization	55
6.2.4	Golden Dataset Construction	56
6.2.5	Pipeline Configuration and Reproducibility	57
6.3	Evaluation Metrics	57
6.3.1	Syntactic Validity	58
6.3.2	Retrieval Quality	58
6.3.3	Semantic Correctness	59
6.3.4	Negative-Test Quality	59
6.3.5	Operational Metrics	59
6.3.6	Statistical Treatment	59

6.4	Quantitative Results	60
6.4.1	Syntactic Validity (RQ1)	60
6.4.2	Targeted Rule Violations (RQ2)	60
6.4.3	Retrieval and Grounding (RQ3)	61
6.4.4	Coverage and Diversity	62
6.4.5	Ablation and Component Analysis	62
6.4.6	Sensitivity to the Generation Model	63
6.4.7	Sensitivity to the Embedding Model	65
6.4.8	Computational Cost: Latency and Token Usage	66
6.5	Qualitative Evaluation: Use-Case Study	67
6.5.1	Study Design	68
6.5.2	Participants	70
6.5.3	Procedure and Materials	70
6.5.4	Measures	70
6.5.5	Task Outcomes	71
6.5.6	Perceived Usefulness	72
6.5.7	Participant Feedback and Observations	74
6.6	Discussion	75
6.6.1	Determinism versus Probabilistic Repair	79
6.6.2	Strengths and Limitations	79
6.7	Threats to Validity	79
7	Summary and Conclusions	83
7.1	Summary of the Work	83
7.2	Answers to the Research Questions	84
7.3	Contributions	84
7.4	Limitations	85
7.5	Future Work	86
7.6	Concluding Remarks	87

Bibliography	89
Abbreviations	95
Glossary	97
List of Figures	99
List of Tables	102
A Prototype Interface and Invoice-Draft Schema	105
A.1 User Interface of the Prototype	105
A.2 Full Invoice-Draft Schema	108
B Supplementary Evaluation Results	109
B.1 Metric Definitions	109
B.2 Per-Slice Quantitative Results	110
B.3 Use-Case Study Data	110
C Installation Guidelines	113
C.1 Prerequisites	113
C.2 The Invoicelab Prototype	114
C.3 Reproducing the Evaluation	115
C.4 Supporting Data-Preparation Tools	117
C.4.1 XML_Fixer (Normalization)	117
C.4.2 InvoiceXMLScreening (Labelling)	118
D Contents of the CD	119

Chapter 1

Introduction

1.1 Motivation

Healthcare insurance providers process large volumes of medical invoices that must comply with legal requirements, tariff regulations, and internal procedural rules. These invoices are typically submitted in *Extensible Markup Language* (XML) format and are automatically checked by validation systems against predefined schemas and domain-specific rules [1], [2]. Ensuring the correctness of such validation systems is critical, as errors may lead to incorrect reimbursements, regulatory violations, or financial losses. Because these systems operate at scale, even infrequent validation failures can have disproportionate financial and operational impact.

Testing invoice validation software requires the availability of realistic and diverse test invoices. In practice, however, generating such test data remains largely a manual task performed by analysts and other practitioners. Creating meaningful test cases often involves detailed knowledge of tariff structures, exception rules, and legal constraints. In particular, the construction of invoices that intentionally violate specific rules, while remaining structurally valid (i.e., schema-compliant) and otherwise realistic, is time-consuming and error-prone. As a result, test coverage is often limited, especially for rare edge cases and complex rule interactions. This is problematic not only for functional correctness but also for negative testing, where systems must reliably detect and explain violations under tightly controlled input conditions [3], [4], [5].

Recent advances in *Large Language Models* (LLMs) have demonstrated strong capabilities in generating structured and domain-specific content when appropriately guided [6], [7]. At the same time, *Retrieval-Augmented Generation* (RAG) techniques allow LLMs to access external knowledge sources, such as tariff catalogs or internal documentation, thereby improving factual grounding and reducing hallucinations [8], [9]. Prior research has shown that combining LLMs with retrieval mechanisms enables controlled generation in knowledge-intensive and regulated domains [10]. However, for test data generation in regulated workflows, plausibility alone is insufficient: generated invoices must be reproducible, auditable, and verifiably correct with respect to formal schemas and domain rules.

1.2 Problem Statement

There is currently no lightweight, analyst-facing approach that can (i) generate realistic XML test invoices, (ii) deliberately trigger specific semantic validation rules, and (iii) provide deterministic validation evidence, without requiring analysts to handcraft complex invoice examples. This thesis addresses that gap by combining LLM-based generation with retrieval from heterogeneous knowledge sources and a validation-aware generation loop.

These developments suggest an opportunity to support the automated generation of validated test invoices for healthcare insurance systems. By integrating LLMs with structured domain knowledge and explicit validation mechanisms, it becomes possible to generate both compliant and intentionally non-compliant invoices through natural-language interaction. Such a system may reduce manual effort, improve test coverage, and support more systematic testing of invoice validation software [9], [11], [12].

1.3 Research Questions and Contributions

This thesis is guided by the following research questions:

- RQ1** How can an LLM be guided to generate invoices that are consistently compliant with a given XML Schema Definition (XSD)?
- RQ2** How can the system generate invoices that intentionally violate a specific semantic rule while keeping all other constraints satisfied (targeted negative tests)?
- RQ3** How should structured and unstructured domain knowledge be stored and retrieved to support controllability and traceability during generation?
- RQ4** To what extent does the resulting prototype improve practical testing workflows in terms of syntactic validity, semantic correctness, coverage of edge cases, and analyst-perceived usefulness?

The main contributions of this work are as follows:

- C1** A validation-aware architecture for LLM-assisted generation of healthcare invoice test cases, integrating retrieval over heterogeneous knowledge sources and deterministic checking mechanisms.
- C2** A generation strategy for producing both compliant invoices and targeted rule violations while maintaining XSD validity.
- C3** A prototype implementation that produces XML invoices suitable for automated ingestion and validation pipelines.
- C4** An evaluation methodology covering syntactic validity, semantic correctness relative to retrieved knowledge, rule-violation targeting, and structured feedback from practitioners.

1.4 Description of Work

This Master’s thesis investigates the design and implementation of an LLM-assisted system for the automated generation of healthcare insurance test invoices. The work is structured into four stages that align with the thesis goals and deliverables.

In the first stage, the state of the art in LLM-driven data generation within regulated domains is reviewed. This includes an analysis of RAG approaches, post-training and prompt-engineering techniques, and methods for storing and retrieving domain-specific knowledge. Different storage technologies, such as relational databases, vector databases, and graph-based approaches, are examined with respect to their suitability for representing tariff data, partner information, and unstructured business documentation. The findings of this stage form the theoretical foundation of the thesis.

In the second stage, the focus is on system design. An architecture is defined for an LLM-assisted solution that supports the generation of realistic and rule-violating test invoices. This includes identifying components for knowledge storage and retrieval, LLM interaction, invoice generation, and validation. Special attention is given to mechanisms that ensure consistency between generated invoice content and domain constraints, such as XML Schema Definition (XSD) validation and semantic rule checks. The design further considers traceability, i.e., recording which retrieved sources and constraints influenced a generated invoice and its validation outcome.

In the third stage, a prototype of the proposed system is implemented. The prototype enables analysts to describe invoice scenarios using natural language and to receive generated XML invoices that conform to a given XSD. The implementation includes a retrieval pipeline for domain knowledge, integration of an LLM capable of generating both compliant and intentionally incorrect invoices, and validation steps to ensure syntactic correctness and semantic relevance for testing purposes. Where generation initially fails validation, the prototype supports iterative repair based on validator feedback, forming a generate-validate-repair loop.

The final stage consists of an evaluation of the prototype. The system is assessed with respect to the syntactic validity of generated XML invoices, the semantic correctness of invoice content relative to the knowledge base, and the coverage of rule-violation scenarios. In addition, a use-case study with practitioners is conducted to evaluate the practical usefulness of the system. The evaluation results are used to identify strengths, limitations, and directions for future work.

1.5 Thesis Outline

The thesis is structured as follows. Chapters 2 and 3 present background information and related work on LLM-based structured data generation, retrieval-augmented generation, knowledge representation in regulated domains, and automated test data generation. Chapter 4 describes the requirements and the overall system architecture of the proposed LLM-assisted invoice generation system. Chapter 5 details the implementation of the prototype, including knowledge storage, retrieval pipelines, LLM integration, and validation components. Chapter 6 presents the evaluation methodology and discusses the experimental results and feedback from practitioners. Finally, Chapter 7 concludes the thesis with a summary of the contributions, a critical discussion of the results, and an outlook on potential future work.

Chapter 2

Background

This chapter introduces the technical and conceptual foundations required to understand the design, implementation, and evaluation of the proposed LLM-assisted invoice generation system. The focus is on (i) domain concepts, (ii) data representations, and (iii) validation mechanisms that are fundamental to healthcare invoice processing and automated testing.

The chapter does not present a detailed comparison of existing research systems. Instead, it establishes a shared terminology and explains the core technologies used throughout the thesis. These foundations are required to motivate later design decisions and to interpret the results of the prototype implementation and evaluation.

2.1 Domain and Data Foundations

2.1.1 Healthcare Invoice Processing

Healthcare insurance systems rely on invoices submitted by medical service providers to determine reimbursement amounts. These invoices document performed services, associated costs, and contextual information such as provider identifiers, billing periods, and additional administrative attributes required for adjudication. Insurance providers process large volumes of such invoices on a daily basis, making manual inspection infeasible.

Invoice processing typically follows a multi-stage pipeline. Submitted invoices are first checked for structural correctness. They are then validated against tariff definitions, contractual agreements, and legal requirements. Finally, reimbursement decisions are derived based on the validated content. Errors at any stage may lead to incorrect payments or regulatory violations [13].

Due to the complexity of healthcare billing regulations, automation plays a central role in invoice validation. Automated systems enable scalable processing and consistent rule enforcement. At the same time, these systems must be carefully tested to ensure correctness under a wide range of valid and invalid input scenarios.

2.1.2 Structured Data and XML

Invoice data in healthcare systems is commonly represented as structured data. Structured data follows a predefined organization that allows automated processing and validation. This distinguishes it from unstructured data, such as free-text clinical notes, which lacks an explicit schema [14].

XML is widely used as an exchange format for structured healthcare data. XML provides a hierarchical representation of information and supports explicit tagging of data elements. Its text-based format ensures platform independence and long-term interoperability [1]. A key advantage of XML is its compatibility with schema-based validation mechanisms. By defining the expected structure of documents, XML enables early detection of malformed or incomplete data. This property makes XML well suited for automated invoice processing pipelines.

2.1.3 XML Schema Definition (XSD)

XSD specifies the structure, data types, and constraints of XML documents. An XSD defines which elements may appear, their ordering, and permissible values. Validation against an XSD ensures that an XML document conforms to the expected format [2]. XSD-based validation focuses on syntactic correctness. It verifies structural properties such as required elements, attribute types, and value ranges. This form of validation is deterministic and can be performed efficiently during invoice ingestion.

However, XSD validation alone is insufficient for ensuring semantic correctness. Many domain-specific constraints cannot be expressed at the schema level. As a result, invoices that are syntactically valid may still violate business rules or legal requirements.

2.1.4 Semantic Validation and Business Rules

Semantic validation addresses constraints that go beyond document structure. It evaluates whether invoice content is meaningful and compliant with domain-specific rules. These rules are derived from tariff systems, contractual agreements, and healthcare regulations [13]. Examples of semantic constraints include valid combinations of services, maximum reimbursable amounts, and context-dependent exceptions. Such rules often involve relationships between multiple invoice fields and cannot be captured by XSD alone [2]. Semantic validation is typically implemented using rule engines or procedural logic layered on top of syntactic checks. These mechanisms enable flexible rule definitions and support frequent updates as regulations evolve. Testing semantic validation logic requires carefully constructed invoice examples that target specific rule interactions.

2.2 Test Data and Test Case Generation

2.2.1 Test Data in Software Validation

Test data is a central component of software validation. It provides the concrete inputs used to assess whether a system behaves as intended under different conditions. In data-driven systems, such as invoice validation pipelines, the quality and diversity of test data strongly influence the effectiveness of testing.

A common distinction is made between functional testing and negative testing. Functional testing focuses on valid inputs and verifies that the system produces correct outputs for expected scenarios. Negative testing, in contrast, uses invalid or inconsistent inputs to ensure that the system detects errors and handles them appropriately [4], [5]. Both forms are required to achieve robust validation.

Edge cases play a critical role in negative testing. These cases occur at the boundaries of valid input domains or arise from uncommon combinations of parameters. Fault injection techniques deliberately introduce errors to assess system resilience and error-handling behavior. In rule-based validation systems, such as healthcare invoice validators, edge cases often involve complex interactions between multiple constraints rather than isolated errors [5].

Test coverage is used to describe how thoroughly a system is exercised by a given test dataset. High coverage does not only depend on the number of test cases but also on their diversity and relevance. Achieving sufficient coverage is particularly challenging in domains governed by extensive business rules and exceptions.

2.2.2 Challenges in Manual Test Data Creation

In many industrial settings, test data is created manually by domain experts. This approach allows fine-grained control over individual test cases but introduces several limitations. Creating realistic test data requires detailed knowledge of domain rules, data formats, and system behavior.

Manual test data creation does not scale well with system complexity [3]. As the number of rules and possible input combinations grows, the effort required to design comprehensive test suites increases significantly. Maintaining test data becomes difficult when rules change, as existing test cases must be reviewed and updated accordingly [3].

Another limitation concerns test coverage. Human-generated test data often focuses on common or previously observed scenarios. Rare edge cases and subtle rule interactions are easily overlooked. As a result, validation systems may perform well on standard inputs while failing in less frequent but critical situations.

2.2.3 Synthetic Test Data

Synthetic test data refers to artificially generated data produced by computer algorithms or simulations that are designed to resemble the statistical properties of real-world data, without directly exposing or releasing operational data. The primary purpose of synthetic data is to support model training, testing, and evaluation in scenarios where real data is scarce, sensitive, or subject to privacy and regulatory constraints [15].

Two key requirements are commonly associated with high-quality synthetic test data: correctness and diversity. Correctness ensures that generated data adheres to the relevant structural and semantic constraints of the domain. Diversity refers to the ability to cover a wide range of scenarios, including rare and extreme cases. These requirements are often in tension, as increasing diversity may increase the risk of producing unrealistic or invalid data [7].

In testing contexts, synthetic data is used to supplement manually created test cases. It enables systematic exploration of input spaces and supports the generation of negative and edge-case scenarios. For validation systems that process structured inputs, such as XML invoices, synthetic data generation must balance realism with controllability to ensure that generated test cases are both meaningful and targeted.

2.3 LLMs

2.3.1 Overview of LLMs

LLMs are neural language models designed to process and generate natural language at scale. Most modern LLMs are based on the Transformer architecture, which relies on self-attention mechanisms to model relationships between tokens within a sequence. This design enables efficient parallelization during training and reduces the path length between distant tokens, facilitating the learning of long-range dependencies in text [16]. Moreover, LLMs are typically pretrained on large corpora of text using self-supervised learning objectives such as next-token prediction. During pretraining, the model learns general linguistic regularities, factual associations, and latent structure present in natural language. This enables the model to generalize to a wide range of downstream tasks in zero or few-shot settings, without task-specific parameter updates [6].

To improve usability, many LLMs undergo additional alignment steps after pretraining, including instruction tuning and reinforcement learning from human feedback. These methods aim to align model behavior with user intent by optimizing for human-favored outputs. As a result, aligned models exhibit improved instruction following and can adapt to diverse task descriptions, including tasks that require structured or constrained generation [17].

2.3.2 In-Context Learning

A distinctive property of LLMs is their ability to perform tasks through in-context learning. Instead of updating model parameters, task behavior is specified directly in the input prompt. Depending on the amount of guidance provided, this can take the form of zero-shot, one-shot, or few-shot learning [6]. In zero-shot settings, the model is given only a task description and is expected to infer the desired behavior. One-shot and few-shot learning extend this approach by including one or more examples within the prompt. These examples serve as implicit demonstrations, allowing the model to infer patterns and constraints relevant to the task without explicit retraining.

The prompt-based task specification offers flexibility, but introduces variability. Model outputs may differ across runs due to probabilistic decoding and sensitivity to prompt phrasing. While this variability can be beneficial for generating diverse outputs, it poses challenges for tasks that require determinism and reproducibility, such as test data generation [7]. Controlling randomness through decoding parameters and prompt design is therefore an important consideration when using LLMs in testing contexts.

2.3.3 Structured Output Generation

Beyond free-form text, LLMs are increasingly used to generate structured outputs such as JSON, XML, or code. By explicitly specifying output formats in prompts, it is possible to guide the models toward producing syntactically constrained data representations [18]. This capability enables the application of LLMs to domains that rely on machine-readable formats, including configuration files, program inputs, and structured business documents. The generation of structured outputs offers several benefits. It allows direct integration of LLM outputs into automated pipelines and enables downstream validation using formal mechanisms such as schema validation. In testing scenarios, this makes it possible to automatically assess whether generated data conforms to expected structural constraints.

However, structured generation also exposes typical failure modes. LLMs may produce outputs that are nearly correct but violate structural rules in subtle ways, such as missing elements, incorrect nesting, or invalid values [18]. These issues arise because LLMs generate text autoregressively based on token-level probability distributions, rather than enforcing explicit formal grammars. As a result, post-generation validation and correction mechanisms are often required to ensure reliability when structured outputs are used in regulated or safety-critical domains [18].

2.3.4 Constraints and Controllability in Regulated Domains

In regulated domains such as healthcare insurance, data generation systems must satisfy requirements that go beyond syntactic correctness. Generated artifacts are expected to adhere to explicit domain constraints, support traceability, and behave predictably under repeated execution. These requirements stand in contrast to many open-ended generation tasks, where variability and creativity are often desirable [7], [12].

For test data generation, controllability is a central concern. Test cases must target specific rules or edge conditions in a deliberate manner, and the relationship between input data and triggered validation behavior should be transparent. Uncontrolled variability can reduce the usefulness of generated test data, as it becomes difficult to reproduce results or to isolate the causes of observed system behavior [7]. Prior work on LLM-based synthetic data generation highlights that unconstrained generation may lead to outputs that are plausible but misaligned with domain rules or testing objectives [7].

Structured output formats and explicit validation mechanisms provide partial control by enforcing formal constraints on generated data. However, many relevant domain rules cannot be expressed solely through schemas or type checks. As a result, effective control in regulated settings typically requires combining structured representations, external domain knowledge, and post-generation validation. Recent studies suggest that integrating retrieval mechanisms and validation steps can significantly improve reliability and reduce unsupported generations in structured and rule-driven tasks [12], [18].

These considerations motivate the design of hybrid generation pipelines that balance flexibility with control. Rather than relying exclusively on model-internal knowledge, such systems explicitly incorporate domain constraints and reference material to ensure that generated outputs are both meaningful and suitable for systematic testing [8], [12], [18].

2.4 Knowledge Storage and Retrieval

2.4.1 Structured Knowledge Storage

Structured knowledge storage refers to the representation of information according to a predefined schema that specifies entities, attributes, and relationships. In contrast to free-form text, structured data is organized in a way that supports deterministic querying, validation, and consistency guarantees. Classical data management research emphasizes that such representations are particularly well suited for domains where correctness, traceability, and reproducibility are essential requirements [14].

Relational data models are among the most widely used approaches for structured knowledge storage. In the relational model, information is represented in tables consisting of rows and columns, where each table corresponds to a specific entity type and relationships are expressed through keys. This representation enables precise queries using declarative languages such as SQL and supports strong integrity constraints, including type checks, uniqueness constraints, and referential integrity [14]. These properties make relational databases a natural choice for storing business-critical data that must be validated and processed reliably.

In healthcare insurance systems, structured storage is commonly used for reference data such as tariff catalogs, partner information, and contract parameters. Tariff data defines billable services, reimbursement rules, and pricing constraints, often in a highly formalized manner. Similarly, partner data specifies providers, insurers, and contractual relationships that must be consistently referenced across invoices. Because this information is relatively

stable, highly structured, and subject to strict consistency requirements, it is typically maintained in relational or similarly structured data stores [13].

From a system perspective, structured knowledge storage provides clear advantages for automated validation and testing. Queries over structured data yield deterministic results and can be combined with rule engines or procedural logic to enforce domain constraints. This determinism is particularly important when generating test cases, as it enables reproducible behavior and precise control over which rules are satisfied or violated. Prior work on structured data processing highlights that such representations are especially effective when downstream systems rely on exact matching and formal validation rather than probabilistic interpretation [18].

However, structured storage also imposes limitations. Schema design requires upfront modeling effort, and rigid schemas may struggle to capture informal rules, exceptions, or evolving regulatory interpretations. As a result, structured data models are often complemented by unstructured documentation that provides contextual explanations and legal background [14], [18]. This separation between structured reference data and unstructured domain knowledge motivates hybrid approaches that combine structured storage with retrieval mechanisms, which are discussed in subsequent sections.

2.4.2 Unstructured Knowledge Storage

Not all domain knowledge in healthcare insurance systems can be expressed in structured form. Many relevant sources, such as internal guidelines, legal regulations, contractual clauses, and explanatory documentation, are primarily available as unstructured text. These documents often describe rules, exceptions, and interpretations that complement formally defined tariff data, but lack a fixed schema suitable for direct validation or querying.

Unstructured knowledge plays an important role in invoice validation processes, as it provides contextual information that cannot be fully captured by structured reference tables alone. For example, legal texts may describe conditional exceptions or special cases that depend on narrative interpretation rather than explicit field constraints. As a result, such information is typically maintained in document repositories, knowledge bases, or content management systems rather than relational databases [14].

From a systems perspective, unstructured knowledge is more flexible than structured storage but also more difficult to integrate into automated pipelines. Querying unstructured text relies on information retrieval techniques rather than exact matching, and the interpretation of retrieved content is inherently ambiguous. Recent work on LLM-based systems highlights that language models are particularly well suited to processing and reasoning over unstructured text, but may produce unsupported or inconsistent outputs when relevant context is missing or poorly retrieved [7], [11].

The coexistence of structured reference data and unstructured documentation is therefore characteristic of regulated domains such as healthcare. This separation motivates architectures that combine structured storage with retrieval mechanisms over unstructured

text, allowing systems to leverage both deterministic data sources and flexible contextual knowledge. Such hybrid approaches form the basis for RAG, which is discussed in the following sections.

2.4.3 Graph-Based Knowledge Storage

In regulated domains, many constraints are relational and involve dependencies between multiple entities. Such constraints often require multi-step reasoning across interconnected data elements. Examples include service dependencies, exclusions between tariff items, or hierarchical classifications of medical services. Classical data management literature notes that highly connected data can be cumbersome to model and query efficiently in purely tabular representations [14], [19].

Graph-based knowledge storage represents data as nodes and edges. Nodes correspond to entities such as tariff items, providers, patient contexts, or billing settings, while edges encode relationships such as *requires*, *excludes*, *is-part-of*, or *valid-in-setting*. This explicit modeling elevates relationships to first-class elements of the data model. Such representations are commonly referred to as *knowledge graphs* and are widely used for modeling structured, relational domain knowledge [19].

Two graph paradigms are widely adopted. RDF graphs represent information as subject-predicate-object triples and are typically queried using SPARQL. Labeled property graphs attach attributes directly to nodes and edges and are commonly queried using languages such as Cypher [19]. A central advantage of graph databases is efficient traversal. They support multi-hop queries that follow chains of relationships across entities. This capability aligns well with real-world validation rules that depend on sequences of conditions rather than single-field lookups [19]. Compared to relational storage, graph models can reduce modeling complexity for many-to-many relationships, hierarchical classifications, and exception networks [14], [19]. They also support improved traceability, because retrieved subgraphs can be inspected and logged as explicit evidence for why a constraint applies. Compared to vector-based retrieval systems, graph queries are deterministic and interpretable. This property is particularly relevant in testing contexts, where reproducibility and explainability are important system requirements [9], [11].

Graph storage also introduces trade-offs. It requires ontology design, schema governance, and careful maintenance to ensure consistency as regulations evolve. In addition, graph representations typically complement rather than replace other storage paradigms. Authoritative reference data is often maintained in relational systems, while unstructured documentation is stored in text repositories or vector indexes. For this reason, hybrid knowledge architectures that combine relational, graph-based, and retrieval-oriented components are common in practice [14], [19].

2.4.4 Retrieval Techniques

Retrieval techniques determine how relevant information is selected from a knowledge store in response to a query. In systems that combine structured and unstructured data,

retrieval serves as the interface between stored knowledge and downstream processing components. The choice of retrieval method has a direct impact on both precision and coverage, particularly in domains where correctness and completeness are critical.

Traditional keyword-based retrieval relies on lexical matching between query terms and documents. Approaches such as inverted indexes score documents based on term frequency and inverse document frequency, favoring exact or near-exact matches. Keyword-based retrieval is efficient and transparent, and it performs well when queries and documents use consistent terminology. However, it is sensitive to vocabulary mismatches and struggles with paraphrases or semantically related expressions that do not share surface forms [14], [20].

Vector-based retrieval addresses these limitations by representing queries and documents as dense embeddings in a shared semantic space. Similarity between a query vector q and a document vector d is typically computed using cosine similarity,

$$\text{sim}(q, d) = \frac{q \cdot d}{\|q\| \|d\|}$$

and the top- k most similar documents are retrieved. Dense retrieval has been shown to improve recall for semantically related content, particularly in knowledge-intensive tasks [21]. As a result, vector-based methods are widely used in modern RAG systems [8], [9].

Despite their strengths, dense retrieval methods introduce trade-offs. Embedding-based similarity may retrieve documents that are topically related but contextually irrelevant, and the retrieval process is less interpretable than keyword matching [11]. In addition, retrieval quality depends heavily on embedding models, chunking strategies, and index maintenance, which can affect system reliability in practice [11].

Hybrid retrieval approaches combine keyword-based and vector-based techniques to leverage their complementary strengths. For example, keyword filtering can be applied to enforce strict constraints, while vector similarity is used to rank semantically relevant candidates. Prior work suggests that hybrid strategies often outperform purely lexical or purely dense retrieval in practical systems, especially when dealing with heterogeneous knowledge sources [8], [9]. Such approaches are well suited for regulated domains such as healthcare insurance, where both terminological precision and semantic flexibility are required.

2.5 RAG

2.5.1 Knowledge in LLM-Based Systems

LLM-based systems rely on two different “places” where knowledge can live. Parametric knowledge is stored implicitly in the model weights as a result of pretraining and fine-tuning. This makes it fast to use at inference time, but it is difficult to update, and the model may still be unreliable for long-tail or rarely observed facts. Non-parametric knowledge is stored outside the model, for example in a document collection or database that can be queried when needed [8], [11], [22].

In practice, knowledge freshness is a major motivation for non-parametric approaches. Updating parametric knowledge typically requires additional training, which is costly and may introduce regressions. With retrieval, the model can access updated documents simply by updating the index. This makes RAG-style systems attractive for domains where rules, tariffs, or internal procedures change over time [8], [9], [23]. Empirical studies also suggest that retrieval is especially valuable when questions depend on less common information, where memorization in the model weights tends to be weaker [11]. As illustrated in Figure 1, REALM augments a parametric language model with a neural knowledge retriever that accesses an external text corpus such as Wikipedia [22]. The retrieved passages provide non-parametric knowledge that complements what is stored in the model’s parameters.

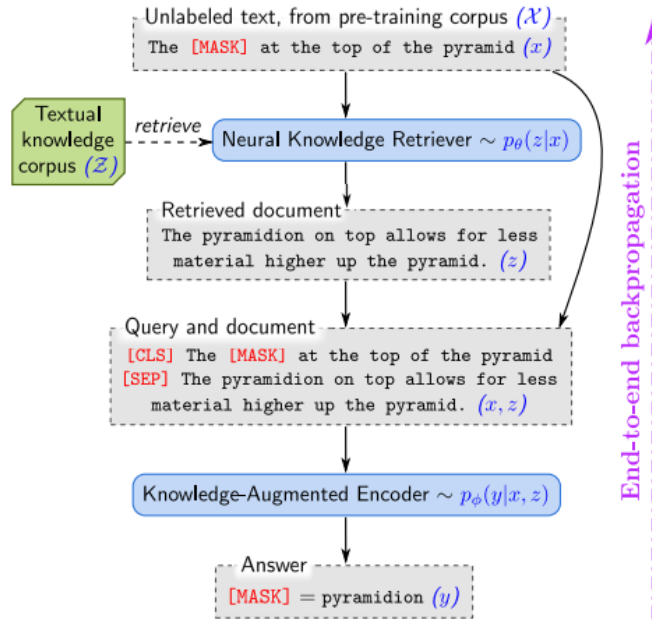


Figure 1: The REALM Architecture [22]

2.5.2 RAG Architecture

A typical RAG pipeline separates the system into a retriever and a generator. The retriever maps a user query (or task prompt) into a vector representation and returns the top- k most relevant passages from an indexed corpus. In many systems, this is implemented as a dense dual-encoder retriever, where queries and passages are embedded separately and matched using similarity search [21]. Figure 2 provides an overview of this retriever-generator decomposition, in which a dense retriever selects the top- k documents most relevant to the query; these are treated as latent variables and marginalized during sequence generation.

The generator is then responsible for producing the final output conditioned on the query and the retrieved context. In the original RAG formulation, retrieval is integrated into generation by marginalizing over retrieved passages, so that generation is probabilistically conditioned on retrieved evidence rather than relying only on the model’s internal memory

[8]. Later architectures explore different ways of “fusing” multiple retrieved passages, for instance by encoding each passage separately and combining them in the decoder attention mechanism (Fusion-in-Decoder) [24]. Figure 3 illustrates this Fusion-in-Decoder architecture, where evidence from multiple retrieved passages is combined during decoding. Other work shows that retrieval augmentation can be trained and used effectively even in few-shot settings, while keeping the document index modular, and that the underlying index can be swapped or refreshed without retraining the whole model [23].

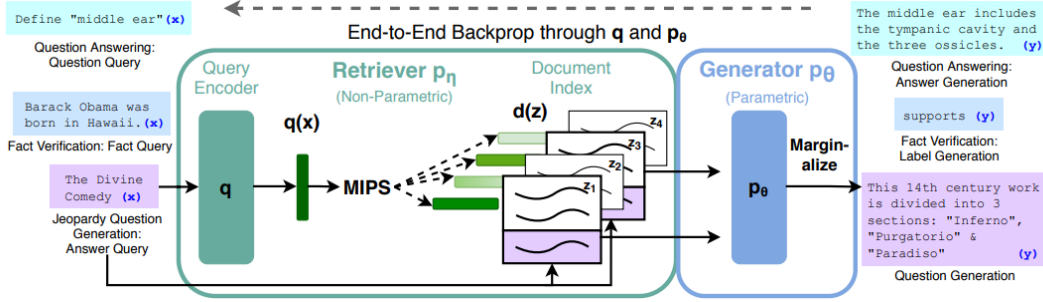


Figure 2: Overview of the RAG Architecture [8]

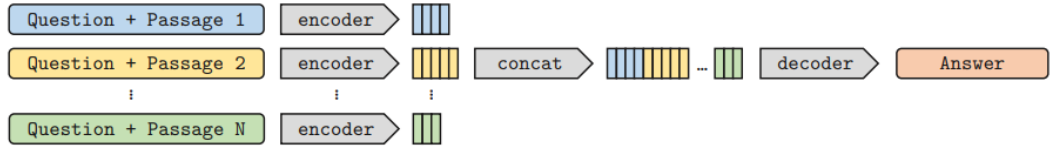


Figure 3: The Fusion-in-Decoder (FiD) Architecture [24]

2.5.3 Benefits and Limitations of RAG

The main practical benefit of RAG is grounding: instead of relying only on parametric memory, the model is given explicit evidence at inference time. This often improves factual accuracy and helps reduce hallucinations, especially when the needed information is rare, domain-specific, or updated after model training [8], [9], [11]. For structured or workflow-like outputs, retrieval can also reduce unsupported generations by constraining the model to relevant reference material [12].

At the same time, RAG systems inherit failure modes from retrieval. If the retriever returns irrelevant passages, missing information, or outdated documents, the generator may still produce incorrect outputs, sometimes with higher confidence because it “saw” something that looked authoritative [9], [11]. This creates a strong dependency on corpus quality, indexing strategy, chunking, and query formulation.

RAG also introduces engineering trade-offs. Retrieval adds latency and operational complexity (indexing, embedding updates, monitoring). On the model side, conditioning on multiple passages can increase compute costs substantially; for example, FiD-style architectures are powerful but can be expensive at inference because they attend over

many retrieved contexts [24], [25]. These trade-offs matter in production pipelines where throughput and predictability are important.

2.6 LLM Integration Strategies in Domain-Specific Systems

2.6.1 Fine-Tuning and Domain Adaptation

LLMs are typically pretrained on broad and heterogeneous text corpora using self-supervised objectives such as next-token prediction [6], [16]. While this pretraining phase enables strong generalization across tasks, it does not guarantee optimal performance in specialized domains. To adapt a pretrained model to a specific task or domain, additional training on domain-relevant data is often performed. This process is commonly referred to as *fine-tuning*.

Fine-tuning updates the model parameters using supervised task-specific data [26]. In its simplest form, a pretrained model is trained further on labeled examples from the target domain. For example, a model could be fine-tuned to generate domain-specific documents, follow structured output formats, or comply with certain stylistic conventions. Instruction tuning and reinforcement learning from human feedback are prominent variants of post-training alignment that adjust model behavior toward desired output characteristics [17]. Continued pretraining on domain-specific corpora is another adaptation strategy, aiming to shift the model's internal representations toward specialized terminology and knowledge distributions [27].

In domain-intensive settings, fine-tuning can yield measurable improvements in factual accuracy, terminology consistency, and format adherence. Empirical studies suggest that continued pretraining or task-specific fine-tuning can significantly improve performance in specialized domains, particularly when tasks require domain-specific terminology or structured outputs [27]. In industrial applications, domain-adapted models have been used to improve reliability and controllability in constrained generation scenarios [10].

Despite these benefits, fine-tuning introduces several trade-offs that are particularly relevant in regulated environments. First, updating model parameters couples domain knowledge tightly to a specific model version. When regulations, tariff definitions, or internal policies change, the model must be retrained. Additional fine-tuning is required to incorporate updated rules. This process can be computationally expensive and may require curated datasets reflecting the new rules. In contrast, non-parametric approaches such as retrieval allow knowledge updates by modifying external data sources without altering the model weights [8], [9].

Second, fine-tuning reduces modularity. Once domain knowledge is embedded in model parameters, it becomes difficult to isolate which parts of the training data influence specific outputs [11]. This lack of transparency may conflict with auditability requirements in regulated domains, where traceability of decisions is important. Furthermore, additional

training can lead to unintended side effects, such as catastrophic forgetting, where performance on previously learned capabilities degrades after adaptation [28]. These risks complicate maintenance and version control in production systems.

Third, fine-tuning alone does not guarantee compliance with formal constraints such as XML schemas or domain-specific validation rules. While a fine-tuned model may learn to approximate structural patterns, it remains a probabilistic generator and does not guarantee formal constraint compliance [29]. Formal correctness therefore still requires deterministic validation mechanisms.

For these reasons, fine-tuning represents a powerful but heavyweight integration strategy. It is particularly suitable when large, stable, and well-curated domain datasets are available and when performance gains justify the cost of parameter updates. In rapidly evolving regulatory environments, embedding domain knowledge directly into model parameters can limit flexibility and increase maintenance effort. Approaches that keep domain knowledge outside the model, such as retrieval augmentation combined with deterministic validation, allow updates without retraining and therefore support better adaptability over time.

2.6.2 Parameter-Efficient Fine-Tuning

While full fine-tuning updates all model parameters, recent work proposes more efficient adaptation strategies that modify only a small subset of parameters. These approaches are commonly referred to as *parameter-efficient fine-tuning* (PEFT) [30], [31]. The central idea is to adapt large pretrained models to new tasks or domains without retraining the entire network.

One widely used method is adapter-based fine-tuning. In this approach, small trainable modules are inserted into the Transformer layers, while the original model parameters remain frozen [32]. During adaptation, only the adapter parameters are updated. This significantly reduces memory requirements and training cost, while preserving the general capabilities of the base model.

Another influential technique is Low-Rank Adaptation (LoRA) [30]. Instead of inserting new modules, LoRA decomposes weight updates into low-rank matrices that are trained while the original weights remain fixed. This approach achieves competitive performance with a fraction of the trainable parameters. Similar strategies have been explored under different formulations, including prefix-tuning and prompt-tuning, which optimize small task-specific vectors appended to the input or intermediate representations [33], [34].

Parameter-efficient methods offer several practical advantages. They reduce computational cost, enable faster experimentation, and allow multiple domain adaptations to share a common base model [30], [31]. In industrial settings, this modularity simplifies deployment and version management, as different adapters or low-rank updates can be loaded depending on the target task.

However, the fundamental trade-offs of fine-tuning remain. Even when only a small subset of parameters is updated, domain knowledge remains embedded in model weights rather

than being stored in external, inspectable data sources [8], [22]. Updating regulatory knowledge still requires additional training. Moreover, like full fine-tuning, PEFT does not inherently guarantee compliance with formal constraints and therefore does not replace deterministic validation mechanisms [29].

In regulated domains, PEFT can therefore be seen as a compromise between full model retraining and purely non-parametric approaches [8]. It reduces cost and engineering overhead, but it does not eliminate the need for external knowledge integration and validation layers. As with full fine-tuning, its suitability depends on the stability of the domain and the availability of high-quality task-specific data.

2.6.3 Prompt Engineering and In-Context Control

An alternative to parameter updates is to guide model behavior directly through input design. This approach is commonly referred to as *prompt engineering*. Instead of modifying model weights, task instructions, constraints, and examples are embedded in the input prompt. The model then adapts its behavior at inference time without additional training [35].

Prompt engineering builds on the observation that LLMs can perform tasks through in-context learning [6]. By providing explicit instructions and representative examples, models can generate outputs that follow desired patterns or formats. In structured generation tasks, prompts often include output templates, schema descriptions, or example documents to increase format adherence [18].

Different prompting strategies have been proposed. Zero-shot prompting relies solely on instructions. Few-shot prompting includes one or more examples that demonstrate the expected structure or behavior [6]. Chain-of-thought prompting encourages the model to generate intermediate reasoning steps before producing the final output, which can improve reliability in complex tasks [36]. More recent approaches combine prompting with tool use or structured intermediate representations to increase control over generation [37].

Prompt engineering offers several advantages. It requires no additional training data, avoids parameter updates, and allows rapid iteration. Instructions can be modified at runtime, which makes this approach flexible in dynamic environments [35]. In regulated domains, prompts can explicitly reference domain rules, schema constraints, or required validation conditions.

However, prompt-based control has inherent limitations. Model outputs remain probabilistic and sensitive to prompt phrasing. Small changes in wording can lead to different outputs. This variability complicates reproducibility, which is critical in systematic testing scenarios [7], [38]. Furthermore, prompt engineering alone does not enforce hard structural guarantees. While examples and instructions can improve format adherence, they cannot replace deterministic schema validation or rule checking.

In practice, prompt engineering is often combined with external validation and retrieval mechanisms [8], [12]. Prompts define the intended behavior, retrieval provides relevant

domain context, and validation ensures formal correctness. For structured test artifact generation, prompting therefore serves as a lightweight control layer rather than as a standalone reliability mechanism.

2.6.4 Tool-Augmented and Hybrid LLM Systems

Beyond fine-tuning and prompting, recent work explores the integration of LLMs with external tools and system components. These approaches treat the language model not as an isolated generator, but as part of a broader architecture [8], [37]. The model may invoke databases, validation services, search engines, or symbolic reasoning modules during inference.

Tool-augmented systems extend the capabilities of LLMs by allowing them to interact with deterministic components [37], [39]. For example, a model may generate a structured query that is executed against a database, or it may call a schema validator to check structural correctness. The result is then incorporated into the subsequent generation step. Such architectures combine probabilistic language generation with precise and auditable computation [12], [40].

RAG is one prominent example of this paradigm, where a retriever supplies relevant external knowledge that conditions the generator [8], [9]. More generally, recent research investigates systems in which LLMs can select and use external tools dynamically, improving factual accuracy and controllability [37]. In industrial contexts, hybrid architectures often combine retrieval, validation, and corrective feedback loops to increase reliability in structured generation tasks [10], [12], [40].

Hybrid systems offer several advantages in regulated environments [9], [12]. They separate responsibilities across components [8], [37]. The LLM handles flexible language understanding and generation. External modules enforce formal constraints, execute deterministic computations, or provide authoritative data. This separation improves transparency and supports auditability, since validation steps and database queries can be logged independently of model outputs [9], [11].

However, tool integration increases architectural complexity [9], [25]. Systems must manage interfaces between components, error propagation, and latency introduced by external calls. In addition, designing appropriate tool invocation strategies requires careful engineering. The model must decide when to rely on internal knowledge and when to defer to external resources [9], [37].

Despite these challenges, tool-augmented approaches are increasingly viewed as a practical path toward controllable and reliable LLM applications [9], [12], [40]. In regulated testing scenarios, hybrid architectures allow structured generation, external knowledge grounding, and deterministic validation to coexist within a single pipeline [12], [40]. This aligns closely with the requirements of invoice test case generation, where correctness depends on both flexible scenario construction and strict rule enforcement.

Chapter 3

Related Work

This chapter reviews prior research related to automated test data generation, synthetic data generation with machine learning models, and the use of LLMs in knowledge-intensive and regulated domains. In contrast to the background chapter, which introduced the underlying concepts and technologies, this chapter focuses on existing approaches and systems, highlighting their strengths and limitations with respect to the generation of structured test data. The goal is to position the present thesis within the existing body of work and to identify gaps that motivate the proposed approach.

3.1 Automated and Synthetic Test Data Generation

Automated test data generation has long been studied as a means to improve the effectiveness and coverage of software testing. Traditional approaches focus on systematically deriving test inputs from specifications, models, or explicit constraints. Model-based testing and rule-driven generation techniques are commonly used to produce test cases that satisfy predefined structural and semantic conditions. These methods offer strong guarantees with respect to correctness and reproducibility, as generated test cases are derived from formal representations of system behavior or input constraints [3], [4], [41].

A key advantage of classical test data generation techniques is their suitability for negative testing. By explicitly encoding constraints and rules, these approaches can deliberately construct inputs that violate specific conditions, allowing targeted evaluation of error handling and validation logic [5], [42]. However, their effectiveness depends heavily on the availability of complete and precise specifications. In domains governed by complex and evolving business rules, such as healthcare billing, constructing and maintaining such models requires substantial manual effort and domain expertise. As the number of rules and interactions increases, scalability becomes a limiting factor [3].

To address data scarcity and privacy concerns, synthetic data generation techniques based on machine learning have gained attention. These approaches aim to produce artificial data that resembles real-world distributions without exposing sensitive information. In contrast to classical test generation, the primary objective is often statistical realism rather

than systematic coverage of validation logic [15]. As a result, while synthetic data can support testing by increasing data volume and diversity, it provides limited control over the generation of specific edge cases or rule violations.

Recent work explores the use of LLMs for synthetic data generation. LLM-based approaches leverage pretrained models to generate realistic and diverse data instances with minimal manual configuration [7]. These methods are attractive due to their flexibility and low setup cost, particularly when domain knowledge is partially informal or documented in natural language. However, prior studies highlight several limitations when LLMs are used for testing purposes. Generated outputs are probabilistic and may vary across runs, which complicates reproducibility. Moreover, while LLMs can generate plausible structured data, they do not provide explicit guarantees that specific constraints or validation rules are satisfied or violated as intended [7], [43].

Overall, existing work on automated and synthetic test data generation reveals a trade-off between control and flexibility. Classical approaches offer strong guarantees but require extensive manual modeling, while LLM-based methods scale well but lack precise controllability. These limitations therefore suggest the need for hybrid approaches that combine generative models with explicit domain knowledge and validation mechanisms, particularly in regulated settings where systematic negative testing is required [3], [7].

3.2 RAG in Knowledge-Intensive and Regulated Domains

LLMs rely primarily on knowledge encoded in their model parameters during pretraining. While this parametric knowledge enables fluent and context-aware generation, it is difficult to update and may be unreliable for domain-specific or rarely observed information. RAG addresses this limitation by augmenting language models with access to external knowledge sources at inference time, allowing generated outputs to be conditioned on retrieved documents rather than solely on internal model memory [8].

In a typical RAG setup, a retriever component selects relevant documents from an external corpus based on a query or prompt, and a generator conditions its output on both the input and the retrieved context. Dense retrieval methods based on learned embeddings have been shown to improve recall for semantically related information and form the basis of many modern RAG systems [21]. Subsequent work has explored different ways of integrating retrieved evidence into generation, such as encoding multiple documents independently and fusing them during decoding, which can improve robustness in knowledge-intensive tasks [23], [24].

RAG has been widely studied as a mechanism for grounding language model outputs and reducing hallucinations. Empirical studies show that providing explicit reference material can improve factual accuracy, particularly when tasks depend on domain-specific or less common knowledge [9]. However, retrieval does not eliminate errors entirely. If irrelevant, incomplete, or outdated documents are retrieved, the generator may still produce incorrect outputs, sometimes with increased confidence due to the presence of seemingly

authoritative context [11]. As a result, the quality of retrieval and the structure of the underlying knowledge base play a critical role in overall system reliability.

These considerations are especially important in regulated and industrial domains. Empirical studies suggest that RAG-based systems are most effective when combined with explicit validation mechanisms and domain constraints, rather than being used as standalone generators [12]. In applied settings such as system testing, retrieval is often used to inject up-to-date specifications or reference data. Downstream validation components then ensure that generated artifacts meet formal requirements. Feng et al. demonstrate this hybrid approach in an industrial testing context, showing that combining retrieval with targeted fine-tuning and validation can improve controllability without substantially increasing model complexity [10].

Overall, prior work suggests that retrieval augmentation improves grounding and adaptability in knowledge-intensive tasks, but does not by itself guarantee correctness or controllability. In regulated environments, RAG is therefore best viewed as a complementary mechanism that supports access to domain knowledge, rather than as a replacement for explicit validation and rule enforcement.

3.3 Structured Test Artifacts and Constrained Generation

A central requirement of test invoice generation is the production of machine-readable artifacts. For healthcare invoice processing, this includes XML documents that must satisfy strict structural constraints. Prior work shows that LLMs can generate structured representations when the desired format is made explicit in the prompt [18]. This enables rapid prototyping and supports flexible scenario descriptions [7]. However, prompt-only approaches do not provide formal guarantees. Even small structural errors, such as missing closing tags or invalid nesting, can make the output unusable in downstream pipelines. This motivates combining LLM generation with explicit checking and correction [40].

A common response to this limitation is to combine generation with explicit constraints. In practice, constraints can be enforced either during generation or after generation [29], [44]. Approaches that constrain generation restrict outputs to a known format, for example by applying grammar- or schema-constrained decoding [29], [39], [44], [45]. Post-generation approaches validate the produced artifact using an external checker, and then trigger correction steps when violations are detected [40]. In regulated settings, this validation-first pattern is attractive because the checks are deterministic and auditable, and they integrate well into existing pipelines [9], [40].

For structured outputs, retrieval can support correctness by grounding the generation process with relevant reference material [8], [9]. Ayala and Bechard show that retrieval augmentation can reduce hallucinations when producing structured outputs, but also emphasize that correctness still depends on retrieval quality and on validation mechanisms [11], [12]. For this thesis, these findings motivate a design where LLM generation is not treated as a standalone component. Instead, it is embedded in a controlled pipeline that includes schema validation and domain rule checks [2], [12].

3.4 Validation-Aware and Self-Correcting Generation Pipelines

Producing syntactically valid artifacts is only one aspect of test data generation. In rule-based domains, correctness is defined not only by format compliance but also by controlled semantic behavior. In invoice validation, valuable negative tests are not arbitrary corruptions. Instead, they are realistic inputs that violate a specific rule while remaining consistent with all other constraints. For example, an invoice may remain XSD valid while deliberately violating a tariff exclusion rule between two service codes that must not be billed together. Such cases require targeted and controllable generation rather than unconstrained synthesis [3], [7]. Prior studies indicate that correctness in LLM-based systems emerges from the interaction of multiple components, not from the language model alone [8], [9]. Retrieval augmentation improves grounding by conditioning generation on external evidence. However, retrieval errors can propagate into the final output, and models may generate confident but unsupported statements when context is incomplete or misleading [11]. These findings motivate system designs in which generation is continuously monitored and constrained by deterministic validation steps.

Validation-aware pipelines introduce explicit checking mechanisms downstream of generation. Structural validation can be implemented through schema-based mechanisms such as XSD, which ensure syntactic correctness and enforce structural constraints [2]. Semantic validation is typically realized through rule engines or procedural logic that encode domain-specific constraints [13]. This layered validation approach separates probabilistic generation from deterministic verification, thereby improving reliability in regulated environments.

When a generated artifact fails validation, feedback can be provided to the model in a subsequent refinement step. This generate-validate-repair pattern has been studied in the context of structured output correction and automated error repair [40]. Rather than regenerating the entire artifact, corrective prompts can target specific violations identified by the validator. This iterative refinement strategy reduces error propagation and increases controllability [12], [40]. For structured outputs, retrieval and validation play complementary roles. Retrieval provides contextual grounding and reduces hallucinations, while validation enforces hard constraints that cannot be guaranteed by probabilistic decoding alone [11], [12]. Empirical evidence suggests that combining retrieval with deterministic validation significantly improves reliability in knowledge-intensive tasks [9].

In regulated testing scenarios, such as healthcare invoice validation, this hybrid approach is particularly relevant. It enables deliberate rule violation targeting, reproducible behavior, and traceable decision paths. These characteristics align with industrial requirements for auditability and systematic negative testing [3], [5]. Consequently, validation-aware and self-correcting generation pipelines form an essential conceptual foundation for the system proposed in this thesis [9], [12].

3.5 Comparison of Integration Strategies for LLM-Based Test Generation

Different strategies exist for integrating LLMs into domain-specific systems. These strategies differ in how domain knowledge is incorporated, how controllability is achieved, and how system maintenance is handled over time. For regulated structured artifact generation, such as healthcare invoice testing, these differences are particularly relevant. Table 1 summarizes the main characteristics of different LLM integration strategies with respect to knowledge integration, maintenance effort, and formal guarantees.

Table 1: Comparison of LLM Integration Strategies

Strategy	Knowledge location	Update cost	Formal guarantees
Full fine-tuning	Model weights	High	None
PEFT	Model weights (partial)	Medium	None
Prompting / ICL	Prompt only	Low	None
RAG	External documents	Low-Medium	Grounding only
Tool-augmented/ Hybrid	External knowledge + validation tools	Medium	Deterministic validation possible

The background chapter (Chapter 2) details each strategy individually; here the emphasis is comparative, as distilled in Table 1. *Full fine-tuning* and *PEFT* embed knowledge in model weights, which couples the model to a specific regulatory state, requires retraining when rules change, and still does not guarantee formal constraint compliance [26], [27], [28], [29], [30], [31], [32]. *Prompt engineering and in-context learning* keep knowledge in the input and enable rapid iteration, but leave outputs sensitive to phrasing, with no reproducibility or format guarantees absent external checks [6], [7], [35], [38]. *RAG* externalizes knowledge so it can be updated without retraining and logged for traceability, yet retrieval quality alone does not ensure correctness [8], [9], [11]. *Tool-augmented and hybrid* architectures treat the LLM as one component alongside databases, schema validators, and rule engines, and are the only family in which deterministic validation can be enforced rather than assumed [12], [37], [40]. Only this last family satisfies the regulated-testing requirements of knowledge freshness, traceability, and formal guarantees simultaneously, which is why this thesis adopts it.

3.6 Knowledge Storage for RAG and Controlled Test Data

RAG is often described as a retriever combined with a generator. In practice, the storage layer is equally important. Storage determines what can be retrieved, how precisely it can be filtered, and how traceable the results are. These aspects are particularly relevant in regulated domains, where reproducibility, auditability, and explainability are central requirements [8], [9], [11].

In the invoice-generation scenario of this thesis, the knowledge base is heterogeneous. Some information is highly structured, such as tariff catalogs, partner master data, and code lists. Other information is primarily textual, such as internal guidelines or legal interpretations. A third class consists of semantic constraints with explicit relations, such as exclusion rules between services or context-dependent exceptions. The coexistence of structured, semi-structured, and unstructured data is characteristic of enterprise systems and has been widely discussed in data management literature [14], [19]. Different storage technologies support these classes with different strengths.

3.6.1 Relational Storage for Authoritative Reference Data

Relational storage is the natural home for this thesis’s authoritative, stable reference data (tariffs, provider registries, and contract parameters), because its declarative SQL queries and deterministic semantics provide the reproducibility that systematic testing depends on (Chapter 2) [14]. Its limitation is equally relevant here: rigid schemas are awkward for densely connected constraints such as exclusion relationships between many tariff items, which lead to join-heavy queries [19]. Relational stores therefore cover the stable core facts, while rule interactions and documentation are left to complementary representations.

3.6.2 Vector Databases for Unstructured and Semi-Structured Text

Vector databases fit the unstructured half of the knowledge base (guidelines, policy fragments, and legal interpretations), retrieved by semantic similarity to ground generation where the relevant knowledge is not encoded in tables (Chapter 2) [8], [9], [11], [21]. Two limitations bear directly on controllability in this setting: similarity search is approximate and may surface topically related but insufficient chunks, and efficient hybrid queries that combine metadata filters (tariff version, insurer, period) with similarity remain an open challenge [46]. Vector retrieval is therefore best treated as a grounding mechanism to be combined with filtering and validation, not as a correctness guarantee [9], [11].

3.6.3 Graph-Based Storage for Relational Constraints and Multi-Hop Dependencies

Graph representations fit the third class of domain knowledge: explicit dependencies and exception networks, such as exclusions between services or context-dependent conditions, where correctness depends on multi-hop relations rather than single-field lookups (Section 2.4.3) [19]. This is exactly where negative tests often live, so graph traversal can isolate the minimal set of related entities a specific constraint requires. Recent *GraphRAG* work extends RAG by retrieving connected subgraphs rather than independent passages [47], [48]; it aligns closely with this domain, although the prototype in this thesis implements only the relational and vector channels and leaves graph storage to future work.

3.6.4 Implications for Controllable Test Invoice Generation

Prior research suggests that no single storage technology covers all requirements of knowledge-intensive systems [9], [14]. Relational storage provides authoritative facts and deterministic filtering. Vector stores provide access to unstructured policy knowledge. Graph representations capture explicit dependencies and exception networks [19], [47].

For an invoice-generation system, this motivates a hybrid architecture. Structured fields, such as service codes and allowable ranges, can be retrieved through SQL queries. Policy explanations and informal constraints can be retrieved through vector search with metadata filtering [8], [46]. Dependency constraints and exclusions can be retrieved through graph traversal when multi-hop relations matter [48].

Generation should then be combined with XSD validation and semantic rule checks to ensure syntactic correctness and targeted rule violations [12]. This closes the loop between retrieved knowledge, generated XML, and the intended validation objective, and reflects current best practices in retrieval-augmented and validation-aware generation pipelines [8], [9].

To position the proposed system within the broader landscape of related approaches, Table 2 summarizes key characteristics of different research directions. The comparison reflects typical representatives of each approach family, without assuming additional constraint decoding, tool use, or external validation components unless explicitly stated. Furthermore, the comparison focuses on four dimensions that are particularly relevant for regulated invoice testing: (i) the ability to produce structured, machine-readable artifacts, (ii) support for deliberate and targeted semantic rule violations, (iii) explicit integration of external knowledge through retrieval, and (iv) the presence of deterministic validation and correction loops.

In this context, *Structured output* refers to the ability to generate artifacts that conform to a defined schema or format (e.g., JSON or XML). *Targeted rule violations* denote deliberate triggering of specific semantic validation rules rather than accidental or random inconsistencies. *Retrieval* indicates the use of external non-parametric knowledge sources during generation. *Validation loop* describes the integration of deterministic structural or semantic checks with iterative refinement.

Table 2: Approaches for Structured Invoice Test Generation

Approach type	Structured Output	Targeted rule violations	Retrieval	Validation loop
Classical test generation	✓	✓	✗	✓
ML synthetic data	✗	✗	✗	✗
RAG systems	Partial	✗	✓	✗
Constrained decoding	✓	✗	✗	✗
Proposed system	✓	✓	✓	✓

As shown in Table 2, existing approaches typically address only a subset of these dimensions. Classical generators offer strong controllability but lack retrieval-based grounding. RAG-based systems improve grounding but do not inherently provide deterministic validation or targeted semantic violation control. The proposed system integrates all four aspects in a single architecture, thereby addressing limitations identified in prior work.

Research Gap and Positioning

Despite significant advances in automated test generation, RAG, and structured output control, existing approaches rarely address all of the following aspects simultaneously: (i) XSD-compliant XML generation, (ii) targeted semantic rule violations, (iii) hybrid knowledge storage across structured and unstructured sources, and (iv) validation-aware self-correction within a single integrated system. Prior work often emphasizes statistical realism, retrieval grounding, or constrained decoding in isolation. A unified framework that combines structured knowledge access, deliberate rule targeting, deterministic validation, and iterative repair for regulated invoice testing remains underexplored. This gap provides the rationale for the architecture proposed in this thesis. The following chapter presents the system architecture designed to address these limitations in an integrated manner.

Chapter 4

Architecture

This chapter describes the architectural context of the proposed solution. It first presents the current invoice processing workflow within the Sumex environment and the insurer IT landscape. Subsequently, the architecture of the proposed MVP for LLM-assisted test invoice generation is introduced, highlighting the additional components for knowledge retrieval, invoice generation, validation, and artifact management.

Sumex is a commercial claims- and invoice-processing suite for the Swiss healthcare-insurance market, offered by Sumex AG, a company that originated from a collaboration between ELCA and Suva.¹ Swiss health and accident insurers use it to ingest, process, and validate the medical invoices submitted by care providers, automating at scale the checks that would otherwise be carried out by hand. The Sumex Suite coordinates these checks (structural schema validation, tariff and contract lookups, expert-system rule analysis, and analytical scoring) across the invoice lifecycle, and it is the production validation system that the test invoices generated in this thesis are ultimately meant to exercise.

4.1 Current Process in the Sumex Environment

Figure 4 illustrates the current high-level invoice processing architecture within the insurer’s IT landscape and the Sumex Suite. The process begins outside the insurer’s system boundary with invoice submission and concludes with a validated and routed claim in downstream systems. The described architecture reflects the existing production setup without LLM-based support.

4.1.1 Invoice Submission and Ingestion

Invoices are submitted by healthcare providers, such as physicians or hospitals. Depending on the submission channel, transmission takes place either electronically (XML) or via paper/PDF. Electronic invoices are typically routed through an intermediary clearing or

¹<https://www.sumex.ch/sumex>

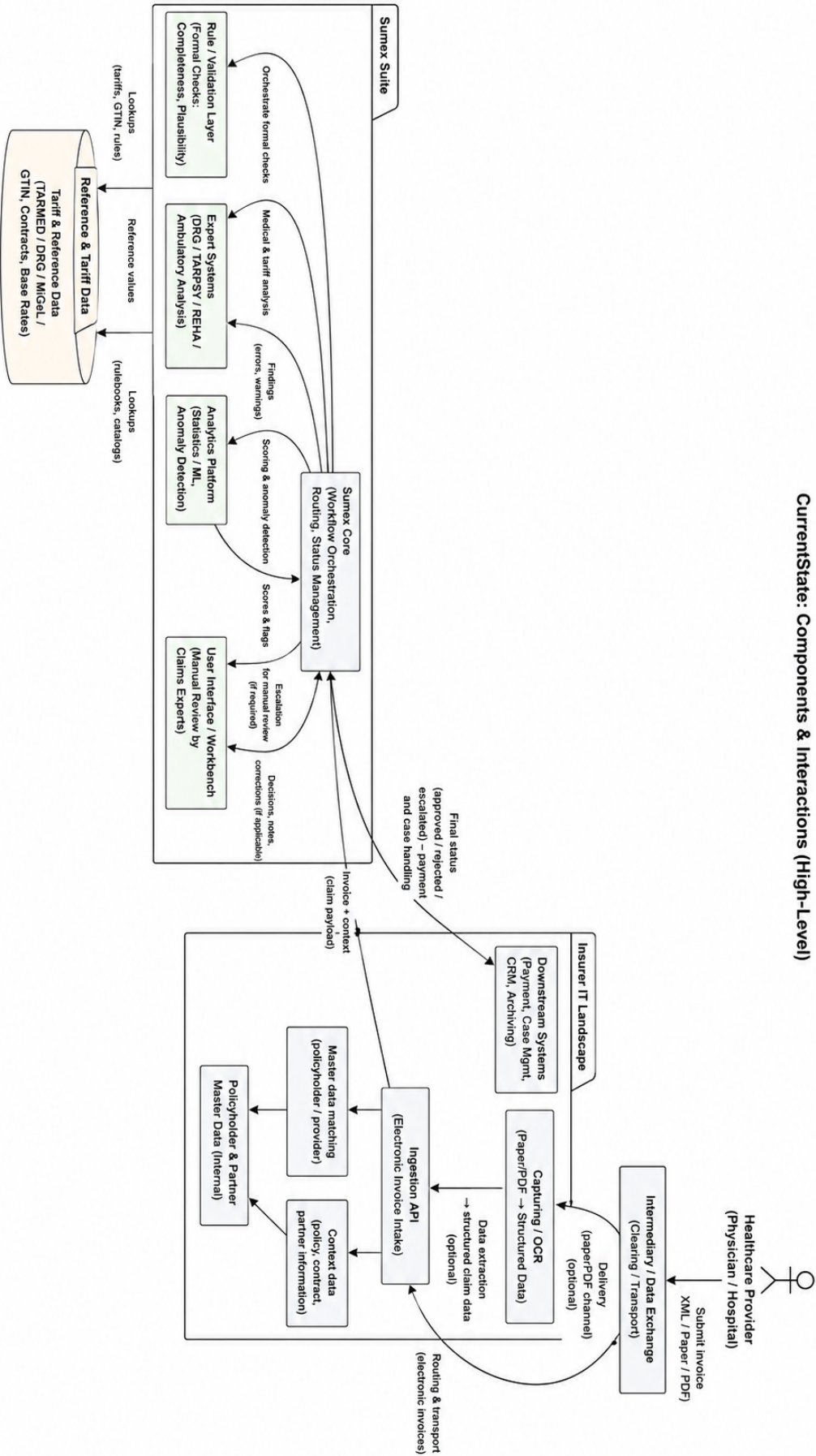


Figure 4: Current High-Level Invoice Processing Workflow

transport service. Paper or PDF invoices may pass through a capturing or OCR component before structured data is extracted.

All structured invoice data is forwarded to the insurer's ingestion interface, which serves as the technical entry point for further automated processing. At this stage, initial technical validations are performed and the invoice is formally registered in the insurer's system.

Before further processing, master data matching is carried out. The ingestion component queries internal partner and insured master data. This step enriches the invoice with contextual information, such as policyholder details, contract parameters, and partner identifiers. The result is a structured invoice payload combined with relevant contextual data.

4.1.2 Transfer to Sumex Core

The enriched invoice is then transmitted to the Sumex Suite. Within Sumex, the central coordination component is the Sumex Core. It is responsible for workflow orchestration, routing, and status management.

The Sumex Core determines the required validation and analysis steps. It orchestrates formal checks, expert analyses, and analytical scoring, while managing intermediate states and ensuring traceable workflow execution. All intermediate findings and decisions are centrally coordinated within this component.

4.1.3 Formal Validation Layer

The first analytical stage consists of formal and technical validations. These checks are implemented in the rule and validation layer. They cover mandatory fields, structural completeness, and plausibility constraints.

Structural validation typically includes schema-based checks, such as XML Schema Definition (XSD) validation, which ensures conformity with predefined document structures. In addition, plausibility checks evaluate field-level constraints and basic logical consistency.

During these validations, the system performs lookups against external reference and tariff data. This includes catalogs such as TARMED, DRG, MiGeL, GTIN, contract parameters, and base rates. Among these, the outpatient tariff TARMED is being superseded by TARDOC (`tariff_type 007`), the tariff on which the MVP of this thesis focuses. The validation layer compares invoice content against these authoritative reference values. Detected inconsistencies are recorded as errors or warnings and returned to the Sumex Core.

4.1.4 Expert Systems and Medical/Tariff Analysis

In addition to formal checks, domain-specific expert systems perform medical and tariff-related analyses. These components evaluate complex rule contexts, such as DRG groupings, outpatient billing rules, or rehabilitation-specific constraints.

The expert systems rely on structured reference and catalog data to assess rule compliance. They produce structured findings, including detected rule violations, contextual annotations, and indications of non-reimbursable service combinations. These findings are returned to the Sumex Core and integrated into the overall case assessment.

4.1.5 Analytics and Anomaly Detection

Beyond deterministic rule checks, the process includes analytical components. The analytics platform applies statistical methods or machine learning models for scoring and anomaly detection. The objective is to identify unusual billing patterns or risk indicators that are not captured by explicit rules.

The analytics component returns scores and flags to the Sumex Core. These indicators influence subsequent routing and escalation decisions, but they do not replace deterministic rule-based validation.

4.1.6 Manual Review and Case Handling

If required, the invoice is routed to a user interface or workbench for manual review. Claims experts can inspect findings, review flagged issues, and add annotations or corrections where procedural rules permit manual intervention.

Decisions taken during manual review are fed back into the workflow. The Sumex Core updates the case status accordingly and records the final outcome in a traceable manner.

4.1.7 Downstream Processing

After completion of all automated and manual steps, the final result is transmitted to downstream systems. These include payment processing, case management, CRM, and archiving components.

The outcome may involve acceptance, rejection, escalation, or routing for further case handling. Downstream systems execute payment decisions, update case management records, and persist the final processing state of the claim.

Summary and Limitations of the Current Process. The current process relies on predefined validation rules, expert systems, and analytical components to ensure scalable and largely automated invoice processing. Structural correctness is enforced through schema validation, while semantic correctness is assessed through deterministic rule engines and domain-specific analyses.

However, the generation of structured test invoices that deliberately target specific validation rules remains an external and largely manual activity. The current process does not provide integrated mechanisms for controllable test case creation or validation-aware generation. The following sections introduce an extension of this process that integrates LLM-assisted generation and deterministic validation loops to support systematic and reproducible test data creation.

4.2 MVP Architecture: LLM-Assisted Test Invoice Generator

This section presents the architecture of the proposed MVP for LLM-assisted test invoice generation. The MVP extends the existing validation ecosystem by introducing a controlled generation pipeline that combines natural language interaction, retrieval-augmented knowledge access, and deterministic validation mechanisms.

The MVP operates outside the production claim-processing workflow and serves as a supporting test-data generation and validation system. Generated invoices can subsequently be used within existing validation and testing environments. Figure 5 illustrates the high-level component structure, while Figure 6 shows the corresponding sequence of interactions during test invoice generation.

4.2.1 Design Objectives

The MVP is designed to address the gap identified in the previous chapters: the lack of an integrated mechanism for generating realistic, XSD-compliant, and semantically controlled test invoices. Rather than relying on manual test case creation or purely prompt-based generation, the proposed architecture combines scenario-driven generation, retrieval-augmented knowledge access, and deterministic validation within a single workflow. The design is guided by three key objectives:

- **Grounded generation:** Generated invoices should be based on structured tariff data and relevant domain documentation rather than relying solely on the internal knowledge of the language model [8], [9].
- **Deterministic validation:** Structural correctness and selected domain constraints should be checked through explicit validation mechanisms, including XSD validation and rule-oriented checks, rather than assumed from probabilistic generation alone [2], [40].

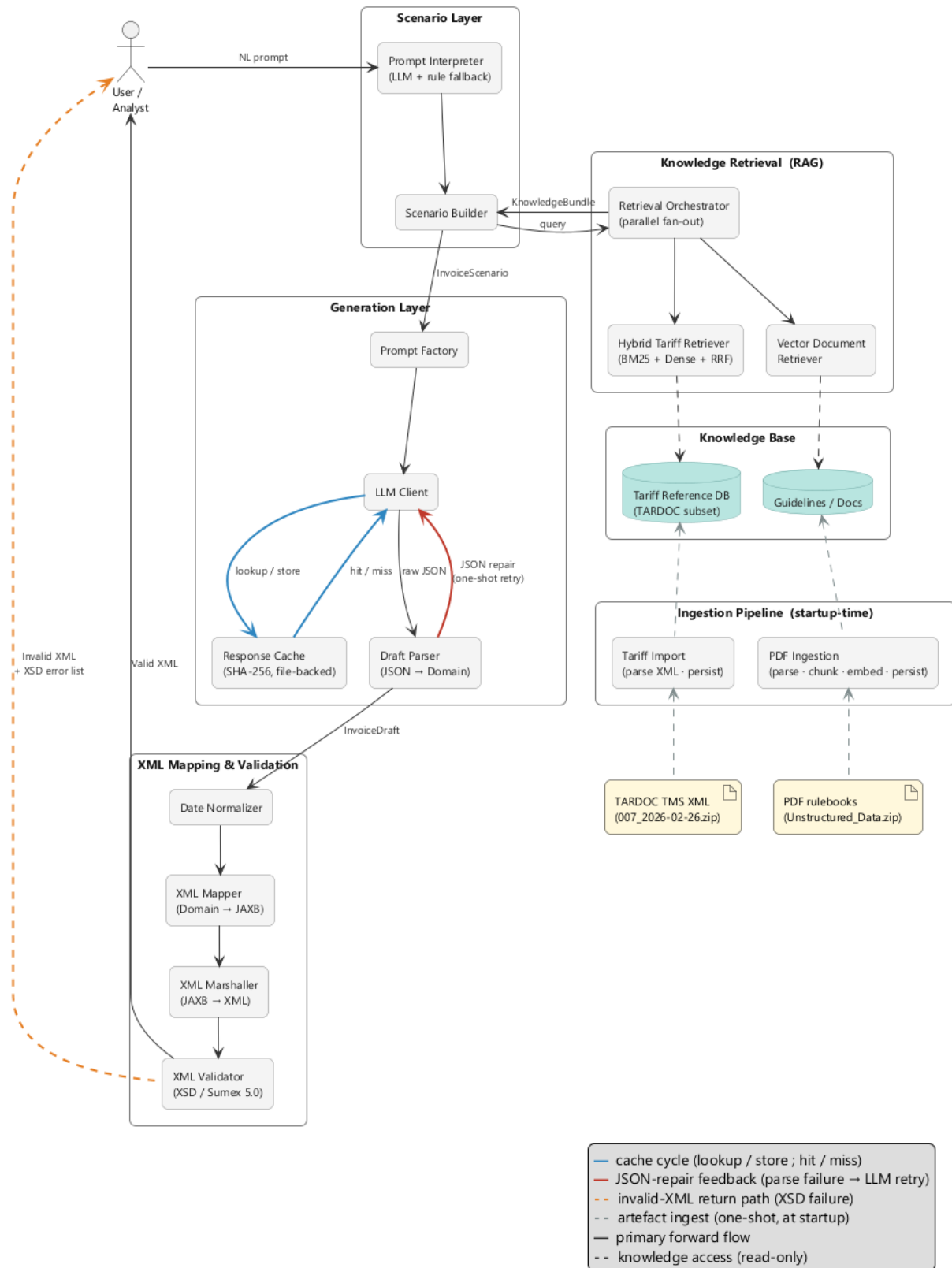


Figure 5: MVP Architecture for LLM-Assisted Test Invoice Generation

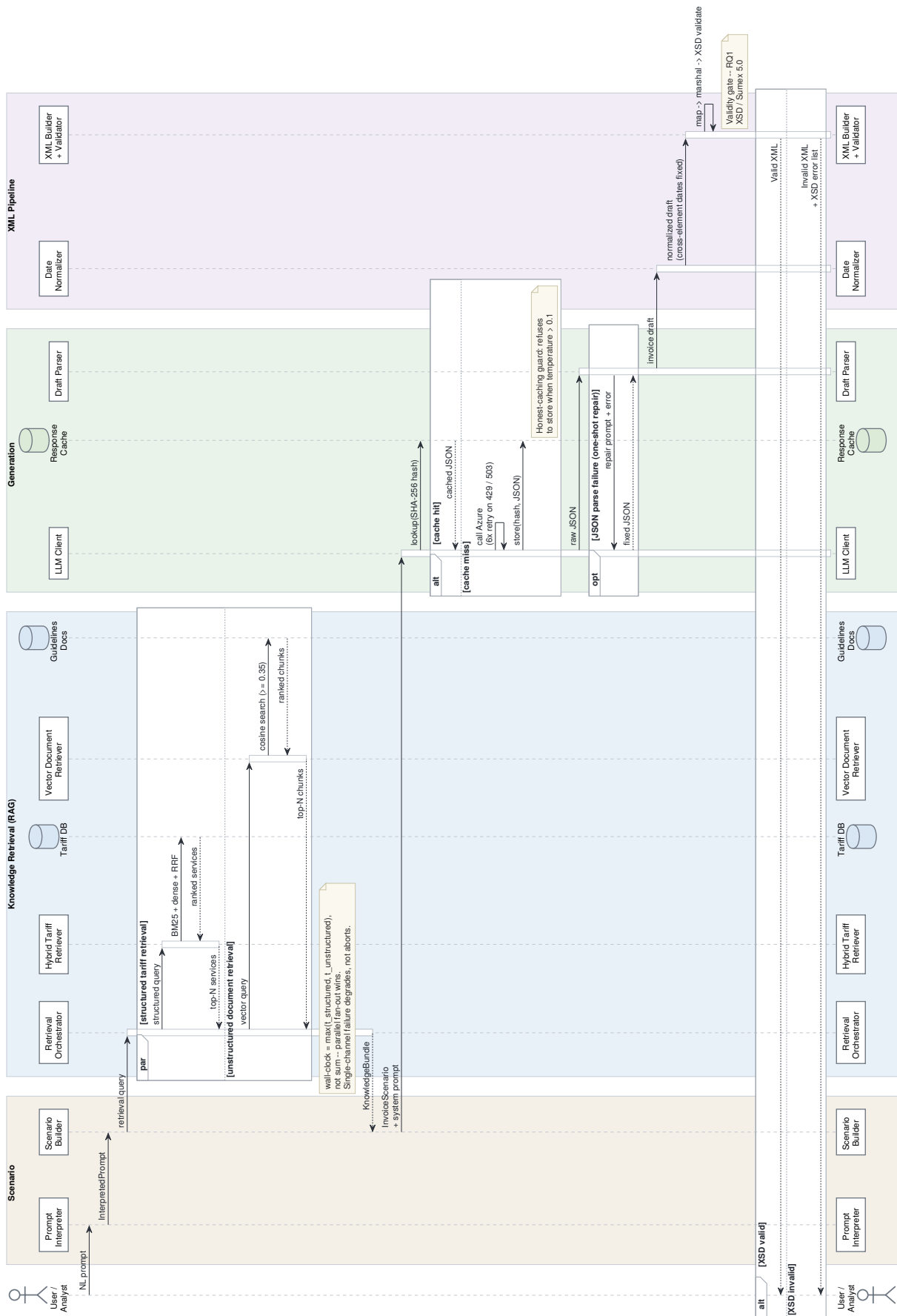


Figure 6: MVP End-to-End Sequence Diagram

- **Traceability and reproducibility:** The architecture should support reproducible scenarios and traceability by keeping scenario definitions, retrieved knowledge, and validation results observable during generation and evaluation.

These objectives reflect the requirements of healthcare invoice testing, where correctness, controllability, and transparency are essential. Table 3 summarizes the main architectural layers of the MVP.

Table 3: Main MVP Architecture Layers

Layer	Components	Responsibility
Generation	Prompt Interpreter, Scenario Builder, Generation Service, LLM Client	Creates structured invoice drafts from analyst requests.
Retrieval	Retrieval Orchestrator, Hybrid Tariff Retriever, Vector Document Retriever	Provides tariff facts and guideline context.
Transformation & Validation	Draft Parser, Date Normalizer, XML Builder/Mapper, XSD Validator	Converts drafts to XML and enforces structural validity.
Artifact & Traceability	XML response/download, request logging, correlation IDs	Returns artifacts and supports request-level tracing.

4.2.2 Architectural Design Decisions

The MVP architecture is based on several design decisions that follow from the requirements of controllable test invoice generation. First, the system uses RAG instead of fine-tuning the language model. This keeps domain knowledge outside the model parameters and allows tariff data, guideline documents, and reference information to be updated independently of the LLM [8], [9].

Second, the retrieval layer follows a hybrid design. Structured tariff retrieval is used to access precise and authoritative reference data, while vector-based document retrieval provides contextual information from unstructured guideline documents. This combination supports both deterministic lookup and semantic flexibility.

Third, validation is performed outside the LLM. The generated draft is transformed into XML and checked through deterministic mechanisms such as XSD validation and structural consistency checks. This separation ensures that formal correctness does not depend solely on probabilistic model output [12], [40].

Finally, the MVP follows a modular pipeline design, applying the classical principle of separation of concerns [49]. Generation, retrieval, transformation, validation, artifact handling, and evaluation are separated into distinct components. This allows individual parts of the system to be extended or replaced independently, for example by adding further tariff sources, improving retrieval, or extending the evaluation logic in future iterations.

4.2.3 Generation Layer

The generation layer serves as the primary interface between the analyst and the system. It transforms natural language requests into structured generation tasks and produces invoice drafts that are subsequently processed by the validation pipeline. The layer consists of three main components: the *Prompt Interpreter*, the *Scenario Builder*, and the *Generation Service*, which communicates with the *LLM Client*.

The *Prompt Interpreter* uses an LLM-assisted interpretation step to analyze the analyst's natural language request and extract relevant parameters, such as scenario intent, expected validation outcome, targeted rule violations, keywords for retrieval, or specific tariff positions. A rule-based fallback can be used when LLM-based interpretation fails. This step converts informal user input into a structured representation that can be processed by subsequent components.

The *Scenario Builder* transforms the extracted parameters into a formal scenario definition. The resulting scenario specifies required invoice characteristics, intended validation outcomes, relevant evidence from retrieved knowledge sources, and constraints that must remain satisfied. This intermediate representation improves controllability and reduces ambiguity before generation begins.

The *Generation Service* produces invoice drafts based on the scenario definition and the retrieved domain knowledge provided by the retrieval layer. It constructs the prompt payload, forwards the request to the configured *LLM Client*, and receives a structured invoice draft as output. Rather than relying solely on prompt instructions, generation is explicitly grounded in structured tariff information and supporting guideline documents. This follows the principles of RAG, where external knowledge is incorporated at inference time to improve relevance and factual consistency [8], [23].

The generated draft serves as the input for the subsequent transformation and validation stages, where it is parsed, converted into a structured XML representation, and checked against deterministic validation rules.

To reduce repeated LLM calls and improve reproducibility for identical generation contexts, the MVP includes a response cache. Cached responses can be reused when the same scenario and retrieved context are encountered again, while new LLM calls are performed when the generation context changes.

4.2.4 Knowledge Retrieval Layer (RAG)

The knowledge retrieval layer provides controlled access to domain information and grounds the generation process in authoritative external knowledge. It consists of three components: the *Retrieval Orchestrator*, the *Hybrid Tariff Retriever*, and the *Vector Document Retriever*.

The *Retrieval Orchestrator* coordinates knowledge access based on the generated scenario definition. It determines which tariff information and supporting documents are required and orchestrates the corresponding retrieval operations. In the MVP, the structured tariff

retrieval and vector-based document retrieval channels are executed in parallel. If one retrieval channel fails or returns no useful results, the pipeline can continue with partial context while recording the degraded retrieval outcome.

The *Hybrid Tariff Retriever* retrieves authoritative reference data from the Tariff Reference Database (MVP subset). This includes tariff positions, service descriptions, and related reference information. By combining structured database retrieval with ranking mechanisms, the component identifies the most relevant tariff entries for a given scenario while maintaining deterministic access to authoritative domain data. This supports both reproducibility and precise scenario construction.

The *Vector Document Retriever* performs semantic search over unstructured guideline documents. It retrieves relevant text fragments containing explanatory rules, contextual constraints, and policy information that may not be represented directly in structured reference data. These retrieved documents provide additional context for the generation process and support more realistic invoice creation.

The combination of structured tariff retrieval and unstructured document retrieval reflects a hybrid RAG architecture [9], [11]. Structured retrieval provides precision, consistency, and deterministic filtering, whereas vector-based retrieval contributes semantic flexibility and contextual knowledge. Retrieved information is passed explicitly to the LLM, enabling traceable grounding of generated invoices.

The architecture is designed to be extensible. While the MVP focuses on a limited subset of tariff and guideline data, additional structured repositories, vector collections, or graph-based knowledge sources can be integrated without requiring changes to the overall generation workflow.

4.2.5 Transformation and Validation Layer

The transformation and validation layer converts the generated draft into a machine-readable XML invoice and verifies its structural correctness. It consists of the *Draft Parser*, the *Date Normalizer*, the *XML Builder/Mapper*, and the *XSD Validator*.

The *Draft Parser* processes the structured draft returned by the *Generation Service* and converts it into an internal invoice representation. If the model output cannot be parsed as a valid structured draft, the *Generation Service* performs a single corrective regeneration pass before processing continues.

The *Date Normalizer* adjusts date-related fields across the generated invoice draft. This step ensures that dependent dates, such as treatment, invoice, and billing dates, remain internally consistent before XML construction.

The *XML Builder/Mapper* transforms the normalized invoice representation into the target XML structure. This separation prevents raw LLM output from directly becoming the final artifact and ensures that XML construction follows explicit mapping logic.

The *XSD Validator* checks the generated XML against the configured Sumex/general invoice schema. XSD validation ensures structural correctness, including required elements,

data types, and ordering constraints [2]. This deterministic validation step is independent of the LLM. When validation fails, the schema errors are returned to the *Generation Service*, which regenerates a corrected draft; this generate-validate-repair cycle repeats for a bounded number of attempts until the XML is schema-valid or the attempt budget is exhausted. Because structural validity and targeted rule violations are orthogonal, a deliberately rule-violating invoice retains its intended violation while only incidental structural errors are repaired.

In contrast to a full production validation engine, the MVP does not perform complete runtime semantic claim validation. Targeted rule behavior and semantic correctness are assessed primarily through the evaluation setup, using predefined scenarios and oracle-based checks. This keeps the runtime pipeline lightweight while still allowing generated invoices to be evaluated against the intended test objectives.

This layered approach separates probabilistic generation from deterministic transformation and validation. It reflects validation-aware generation principles, where generated artifacts are checked by explicit mechanisms rather than assumed to be correct from model output alone [12], [40].

4.2.6 Artifact Handling and Traceability

The artifact handling layer is responsible for returning the generated invoice output and supporting traceability during execution. In the current MVP, generated XML invoices are provided to the analyst as downloadable artifacts or API responses. The runtime system does not persist generated invoices in a dedicated long-term test dataset store.

Traceability is supported through request-scoped logging and correlation identifiers. These mechanisms make it possible to relate a generation request to its retrieval, generation, transformation, and validation steps during debugging and evaluation. However, persistent audit storage of prompts, retrieved sources, validation results, and generated artifacts is outside the scope of the current MVP implementation.

The project also contains a golden dataset used for evaluation. This dataset is separate from runtime artifact handling: it serves as a controlled reference for assessing generation quality and rule-targeting behavior, rather than as a storage mechanism for newly generated invoices.

4.2.7 End-to-End Generation Flow

Figure 6 illustrates the runtime sequence of the MVP. An analyst submits a natural language request describing the desired invoice scenario. The *Prompt Interpreter* converts this request into a structured interpretation, which is then transformed by the *Scenario Builder* into a formal invoice scenario. Based on this scenario, the retrieval layer gathers relevant tariff information and guideline fragments through the structured tariff and vector document retrieval channels.

The retrieved context is combined with the scenario definition and passed to the *Generation Service*. If a matching cached response is available, it can be reused; otherwise, the *LLM Client* is invoked to generate a structured invoice draft. The draft is then parsed by the *Draft Parser*, normalized by the *Date Normalizer*, and transformed into XML by the *XML Builder/Mapper*.

Finally, the generated XML is checked by the *XSD Validator* against the configured Sumex/general invoice schema. If validation fails, the schema errors are fed back to the *Generation Service* for a bounded number of repair attempts; the resulting XML and its final validation status are then returned to the analyst. This workflow combines LLM-based generation with deterministic transformation and validation steps, resulting in a controlled pipeline for producing structurally verifiable test invoices.

4.2.8 Rationale and Scope of the MVP

The MVP deliberately focuses on the core capabilities required for controlled test invoice generation: scenario interpretation, knowledge retrieval, XML transformation, and XSD validation. Its scope is limited to a selected subset of tariff data and guideline documents, which reduces implementation complexity while still validating the architectural principles of the proposed approach.

Certain capabilities are intentionally outside the scope of the current MVP. These include production-grade artifact storage, persistent audit logging, and full runtime semantic claim validation with rule-level self-correction. Structural self-correction is supported through the bounded XSD generate-validate-repair loop described above, whereas semantic correctness is assessed separately in the evaluation (Chapter 6). In general, the MVP emphasizes a lightweight and modular pipeline in which generated invoices can be structurally validated and returned as test artifacts for use in existing validation and testing environments.

Despite these limitations, the architecture demonstrates how LLM-based generation can be embedded within a validation-aware and retrieval-augmented workflow. This hybrid design addresses limitations of unconstrained LLM generation [7], [11] and aligns with best practices in retrieval-augmented and tool-augmented systems [8], [12]. The modular structure also allows future extensions, such as additional structured repositories, graph-based rule stores, persistent tracing, or more advanced semantic validation components.

Chapter 5

Implementation

This chapter describes how the MVP architecture introduced in Chapter 4 (Section 4.2) is realized as a running system. The same layered decomposition used in the architecture chapter is followed: the generation layer (Section 5.3), the knowledge retrieval layer (Section 5.4), the transformation and validation layer (Section 5.5) and the handling and traceability of artifacts (Section 5.6). Rather than restating the architectural rationale, each section focuses on *how* the corresponding components are implemented and configured. The evaluation corpus, the invoice ground-truth labeling, and the golden data set used to assess the system are not part of the runtime pipeline and are therefore described separately in Chapter 6.

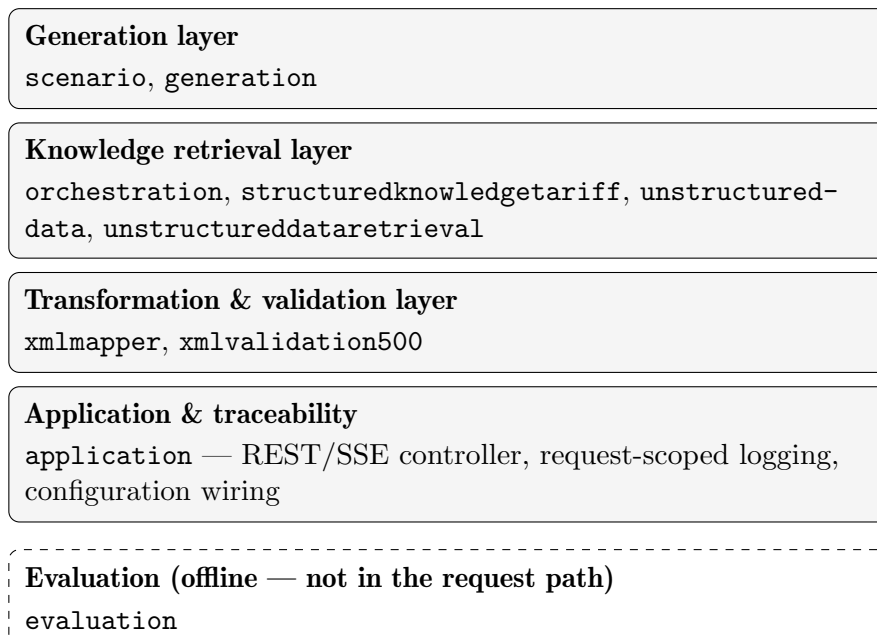


Figure 7: MVP Java Package Structure by Architectural Layer

5.1 Implementation Overview

The MVP is implemented as a single Spring Boot service written in Java 21 and exposed over a REST and Server-Sent Events (SSE) interface. The service realizes the full generation pipeline end to end: it interprets a natural-language request, retrieves grounding knowledge from a hybrid knowledge base, prompts an LLM to produce a structured invoice draft, and deterministically transforms and validates that draft into a schema-compliant XML invoice. Table 4 summarizes the principal technologies and the role each one plays in the pipeline.

Internally, the codebase is organized into Java packages that map directly onto the architectural layers of Section 4.2, as illustrated in Figure 7. The `scenario` and `generation` packages implement the generation layer; `orchestration`, `structuredknowledgetariff`, `unstructureddata`, and `unstructureddataretrieval` implement the knowledge retrieval layer; `xmlmapper` and `xmlvalidation500` implement the transformation and validation layer; and the `application` package provides the REST controller, request-scoped logging, and configuration wiring. A separate `evaluation` package, used only for the assessment in Chapter 6, is not part of the production request path.

Beyond the runtime service, two standalone supporting tools were developed for evaluation-data preparation rather than for the live pipeline: *XML_Fixer*, which deterministically normalizes selected fields of corpus invoices and pre-checks them against the XSD schema, and *InvoiceXMLScreening*, which classifies invoices against the production Sumex validation database to obtain ground-truth validity labels. As these tools serve to construct and label the evaluation corpus rather than to generate test invoices, they are described together with the experimental setup in Chapter 6 (Section 6.2).

Table 4: Technology Stack of the MVP Implementation

Technology	Role in the pipeline
Java 21, Spring Boot	Application runtime, dependency injection, and REST/SSE endpoints.
Azure AI Foundry (Kimi-K2.5)	LLM used for prompt interpretation and invoice-draft generation, accessed through an OpenAI-compatible chat-completions API.
Azure OpenAI (text-embedding-3-small)	Embedding model used to vectorize documents and queries for semantic retrieval.
PostgreSQL with pgvector	Vector store for unstructured document chunks; similarity search via an HNSW index over cosine distance.
Sumex tariff library	Authoritative structured tariff data (TARDOC and related catalogs) queried during structured retrieval.
Apache PDFBox	Text extraction from guideline PDFs during document ingestion.
JAXB	Object-to-XML marshalling of the generated invoice into the Sumex <i>generalInvoiceRequest</i> envelope.
JAXP / XSD validation	Deterministic schema validation of the marshalled XML.

5.2 System Configuration and Deployment

The service is configured declaratively through Spring Boot’s `application.yaml`, with all application settings bound from the `invoicelab` property namespace into a typed configuration record. Endpoints and credentials (the Azure LLM and embedding endpoints, API keys, and the database password) are not hard-coded but supplied through environment variables, so no secret is committed to the repository. The same binding mechanism allows individual parameters to be overridden per run without code changes; the specific values used for the evaluation are reported in Chapter 6.

The application is packaged as a container image based on a Java 21 runtime and deployed together with its datastore via `docker-compose`, which provisions a PostgreSQL instance with the `pgvector` extension. The service connects to this database through a pooled connection and uses it as the vector store for the retrieval layer. Domain data is supplied externally rather than bundled: structured tariff data is imported from a TARDOC archive into a file-backed tariff store, and the guideline PDFs are read from a ZIP archive. On first startup, when the vector store is empty, the document-ingestion pipeline runs automatically (load, parse, chunk, embed, store); on subsequent starts it detects the populated store and skips ingestion, so the step is idempotent. Table 5 gives an overview of the principal runtime parameters and default values.

Table 5: Principal Runtime Parameters and Default Values

Parameter	Default	Description
Retrieval top- k	10	Candidates retrieved per knowledge channel.
Min. similarity score	0.35	Cosine-similarity threshold for retained document chunks.
Chunk size (min/max)	300 / 1200	Character bounds for paragraph chunking.
Chunk overlap	150	Overlap (characters) between adjacent chunks.
Embedding dimension	1536	<code>text-embedding-3-small</code> ; HNSW index over cosine distance.
LLM temperature	0.7	Sampling temperature for generation.
LLM max output tokens	8192	Upper bound on generated tokens per call.
LLM request timeout	120 s	Per-request timeout for the LLM client.
Max XSD repair attempts	2	Bound on the generate-validate-repair loop (Section 5.5).

5.3 Generation Layer

The generation layer turns a free-text analyst request into a structured invoice draft. It realizes the generation components of Section 4.2 as three stages that run in sequence:

prompt interpretation, scenario building, and grounded generation. Figure 8 shows the data flow and the intermediate artifacts passed between them.

5.3.1 Prompt Interpretation

The entry point is the *Prompt Interpreter*, implemented by `LlmPromptInterpreter`. It sends the analyst's request to the LLM with a system prompt that constrains the answer to a small JSON object (Listing 5.1): the scenario `intent`, the requested validity, an explicit rule-violation flag, and a list of retrieval keywords. The free-text request may be written in English or German, and the keyword list is lower-cased, de-duplicated, and stripped of stop words in both languages. When the analyst names a specific tariff position, it is extracted directly from the request with a regular expression matching the TARDOC code pattern `LL.NN.NNNN`. Contradictory combinations are normalized deterministically: a request for an invalid invoice forces the rule-violation flag and the rule-violation intent, so downstream components see a single coherent signal.

Because generation depends on this step, it is made robust rather than strict: if the LLM call or the JSON parsing fails, the interpreter falls back to a rule-based implementation (`RuleBasedPromptInterpreter`) that derives the same fields from keyword and pattern matching, so the pipeline never stalls on an unparseable interpretation. The stage emits an immutable `InterpretedPrompt` carrying the original text, the intent, the requested validity, the rule-violation flag, the optional service code, and the keywords.

```
{
  "intent": "GENERATE_TEST_INVOICE | GENERATE_RULE_VIOLATION_CASE |
    EXPLAIN_TARIFF_RULES | UNKNOWN",
  "requestedValidity": "VALID | INVALID | UNSPECIFIED",
  "ruleViolationRequested": true,
  "keywords": ["..."]
}
```

Listing 5.1: Prompt-Interpretation Schema

5.3.2 Scenario Building

The *Scenario Builder* (`ScenarioBuilder`) combines the interpreted prompt with the knowledge returned by the retrieval layer (Section 5.4) into a single, self-contained `InvoiceScenario`. It derives the retrieval query from the service code, keywords, and validity signals, and assembles the retrieved knowledge into a typed list of evidence items: tariff facts are recorded as `TARIFF` evidence (category, code, and description), and document fragments as `DOCUMENT` evidence with their excerpt truncated to a fixed length. The builder also produces a short scenario summary and collects warnings for conditions that weaken controllability, such as an unrecognized intent or an empty knowledge bundle. This intermediate representation is what makes generation reproducible: every input the model will see is fixed and inspectable before the LLM is called.

5.3.3 Generation Service and LLM Client

The *Generation Service* (`GenerationService`) drives the actual generation. It delegates prompt construction to `PromptFactory`, which builds a system prompt that fixes the model’s role and output contract – JSON only, TARDOC (`tariff_type` 007) by default, and exact tariff codes – and a user prompt that lays out the scenario summary, the requested validity and rule-violation flag, the retrieved evidence, and the target draft schema (Listing 5.2). Grounding the request in explicit evidence, rather than in prompt instructions alone, is what distinguishes this from unconstrained prompting.

The prompt is sent through the `LlmClient` abstraction. In the deployed configuration this is `AzureKimiLlmClient`, which calls the `Kimi-K2.5` model on Azure AI Foundry through an OpenAI-compatible chat-completions API. The client can be wrapped by a caching decorator (`CachingLlmClient`) that memoizes responses for identical requests, which keeps repeated runs – in particular the evaluation in Chapter 6 – reproducible and avoids redundant model calls. The raw model output is decoded by `InvoiceDraftParser`, which strips any markdown fences, extracts the fields with sensible defaults, and requires at least one service record; malformed output raises a `GenerationException`. If the output cannot be parsed, the service performs a single corrective regeneration pass. The result is an `LlmGeneratedInvoiceDraft`, which becomes the input to the transformation and validation layer (Section 5.5), where the bounded XSD generate-validate-repair loop is applied.

```
{
  "scenarioType": "...",
  "intentionallyInvalid": false,
  "patientReference": "...",
  "providerReference": "...",
  "justification": "...",
  "declared_total_amount": 0.0,
  "serviceRecords": [
    {
      "record_id": 1,
      "tariff_type": "007",
      "code": "AA.00.0010",
      "name": "...",
      "quantity": 1,
      "date_begin": "YYYY-MM-DDThh:mm:ss",
      "provider_id": "...",
      "responsible_id": "...",
      "amount": 0.0
      /* remaining fields omitted; full schema in Appendix \ref{app:draft-schema} */
    }
  ]
}
```

Listing 5.2: Invoice-Draft Schema (Abbreviated)

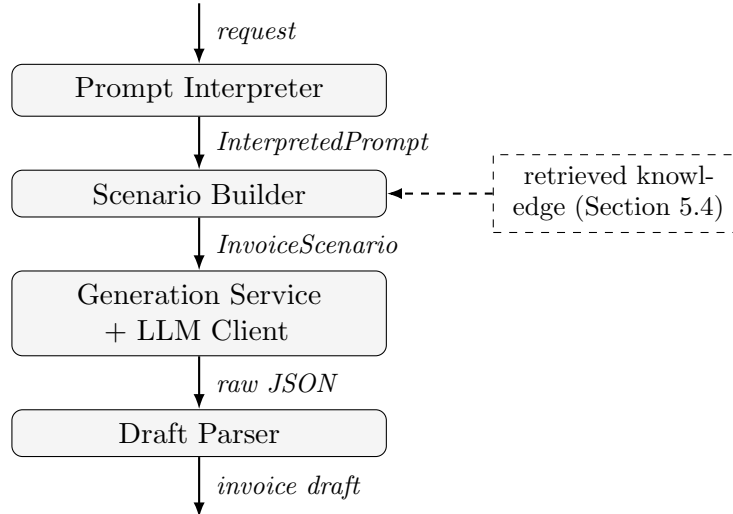


Figure 8: Generation-Layer Data Flow

5.4 Knowledge Retrieval Layer

The retrieval layer supplies the evidence that grounds generation. It implements the hybrid design of Section 4.2: a structured channel that returns authoritative tariff facts and an unstructured channel that returns semantically relevant guideline passages. Both are driven from the query derived during scenario building and feed their results back as a single `KnowledgeBundle`.

5.4.1 Retrieval Orchestration

The *Retrieval Orchestrator* (`RetrievalOrchestrator`) runs the two channels concurrently. Each retrieval is dispatched as a `CompletableFuture`, so the slower channel determines the latency rather than the sum of both. The orchestrator is deliberately tolerant of partial failure: if one channel throws or returns nothing useful, the error is demoted to a warning and generation proceeds on the remaining context; only when both channels fail is a `RetrievalException` raised. The assembled `KnowledgeBundle` carries the tariff facts, the retained document chunks, the number of chunks discarded by the relevance filter, and any warnings, so a degraded retrieval is visible downstream rather than silently masked.

5.4.2 Structured Tariff Retrieval

The structured channel (`StructuredKnowledgeService`, backed by `StructuredQueryService`) queries the Sumex tariff library, which holds the TARDOC catalogue and related reference data. It follows two paths. When the query contains an explicit tariff code, the service resolves it by direct lookup, which gives deterministic access to the named position. Otherwise it ranks the catalogue against the free-text query using a pluggable ranker: BM25 lexical scoring [50] by default, or a hybrid ranker that fuses BM25 with

dense embedding similarity through reciprocal rank fusion [51] when configured. Either path returns a bounded set of **TariffFact** entries – category, code, and description – capped at twenty candidates.

To bridge the vocabulary gap between clinical or lay phrasing and the German tariff terminology, an optional *query expander* can precede the lexical lookup. When enabled, it asks the LLM to translate the query into a handful of German TARDOC keywords that are appended to the search terms; it is skipped for code-explicit queries, where the exact-code lookup is already optimal, and is disabled entirely when not configured.

5.4.3 Vector Document Retrieval and Ingestion

The unstructured channel (**VectorDocumentRetriever**) embeds the query with the Azure **text-embedding-3-small** model and searches the vector store for its nearest neighbours by cosine distance. Each candidate’s distance is converted to a similarity score, and candidates below a configurable threshold are discarded so that only sufficiently relevant passages reach the model; the number dropped is reported in the bundle. The store itself is a PostgreSQL table with a **pgvector** column and an HNSW index [52] over cosine distance (Listing 5.3), which keeps approximate nearest-neighbour search efficient as the corpus grows.

The corpus is prepared by the ingestion pipeline (**DocumentIngestionService**), shown in Figure 9. It reads the guideline PDFs from their source, extracts text with PDFBox, splits each document into overlapping paragraph-sized chunks, embeds every chunk, and writes the chunk together with its vector and provenance metadata to the store. Ingestion runs once on first startup, when the store is empty, and is skipped thereafter, so the corpus is built idempotently. Table 6 lists the parameters that govern chunking, embedding, indexing, and retrieval.

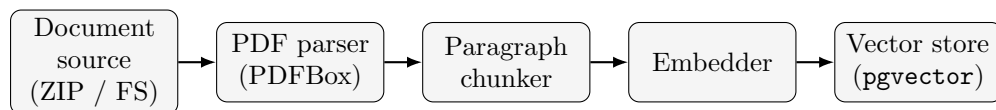


Figure 9: Document Ingestion Pipeline

```

CREATE TABLE vector_store (
  chunk_id TEXT PRIMARY KEY,
  document_id TEXT,
  document_name TEXT,
  chunk_index INT,
  content TEXT,
  embedding vector(1536)
);

CREATE INDEX vector_store_embedding_idx
  ON vector_store USING hnsw (embedding vector_cosine_ops);
  
```

Listing 5.3: Vector Store Schema and HNSW Index

Table 6: Retrieval and Ingestion Parameters

Parameter	Value	Description
Chunk size (min / max)	300 / 1200	Character bounds for paragraph chunking.
Chunk overlap	150	Overlap (characters) between adjacent chunks.
Embedding model / dim.	<code>text-embedding-3-small</code> / 1536	Query and chunk embeddings.
Vector index	HNSW, cosine	Approximate nearest-neighbour search in <code>pgvector</code> .
Retrieval top- <i>k</i>	10	Candidates returned per channel.
Min. similarity score	0.35	Threshold below which document chunks are discarded.
Structured result cap	20	Maximum tariff facts returned by the structured channel.

5.5 Transformation and Validation Layer

This layer turns the model’s structured draft into a schema-compliant XML invoice. It is deliberately deterministic: the probabilistic model proposes content, but the conversion to the target format and the verdict on its correctness are produced by fixed logic. The draft passes through date normalization, object-to-XML mapping, marshalling, and XSD validation, with a bounded repair loop closing the gap when validation fails. Figure 10 shows the sequence.

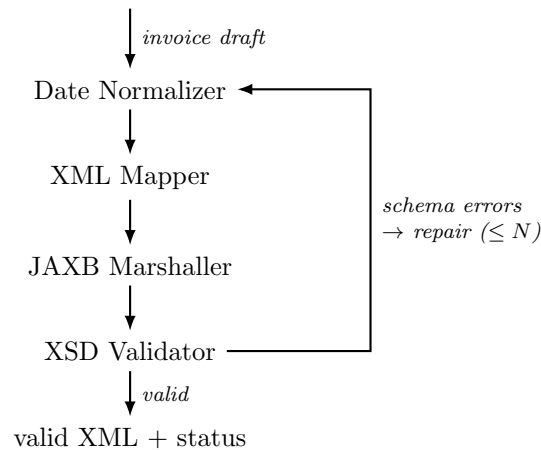


Figure 10: Transformation and Validation Sequence with Bounded Repair Loop

5.5.1 Draft Parsing and Date Normalization

The structured draft produced by the generation layer (Section 5.3) enters this stage already parsed into an `LlmGeneratedInvoiceDraft`. Before it can be mapped to XML, its dates must satisfy the cross-field constraints of the Sumex envelope, which the model does

not reliably enforce on its own. The `InvoiceDateNormalizer` applies these constraints deterministically.

Its guiding principle is *respect-when-consistent, overwrite-when-not*: a date supplied by the model is kept whenever it already satisfies the relevant constraint, and recomputed otherwise. The treatment range, for example, is preserved if it brackets every service date and falls within the permitted window; if it does not, the range is derived from the service dates themselves. The service dates are treated as the ground truth of the clinical episode and are never moved – the normalizer only fills in a missing end date or corrects an inverted one. A well-known placeholder date emitted when the model leaves a field blank is recognized and treated as absent rather than as a real value. The component is stateless apart from an injectable clock, which lets the surrounding tests pin “now” deterministically. Table 7 summarizes the rules it enforces.

Table 7: Cross-Field Date-Consistency Rules

Element	Constraint
Invoice	Request timestamp and request date denote submission time and are mutually consistent; the request date is strictly after the treatment end date.
Treatment	<code>date_begin ≤ date_end</code> ; both lie within the last five years up to now; the range covers every service date.
Law	The case date is on or before the treatment start date.
Service	Per service, <code>date_begin ≤ date_end</code> when an end date is present.

5.5.2 XML Mapping and Marshalling

The normalized draft is converted to XML in two explicit steps. The `InvoiceXmlMapper` maps the draft onto a JAXB-bound object tree that mirrors the Sumex *xmlInvoice* 5.0 structure, splicing in the normalized dates, and the `InvoiceXmlMarshaller` serializes that tree into an XML string. Keeping mapping and marshalling separate from generation ensures that raw model output never becomes the final artifact directly: the element structure, ordering, and value formats are dictated by the binding classes, not by the text the model happened to produce.

5.5.3 XSD Validation

The marshalled XML is checked by `InvoiceXmlValidator` against the Sumex `generalInvoiceRequest_500` schema and its dependencies (the XML digital signature and encryption schemas). The schema set is compiled once into an immutable, shared schema, and a fresh validator is created per call as required by the JAXP contract; a collecting error handler gathers all violations so that the full set of structural problems is reported rather than only the first. Validation yields an `XmlValidationResult` – a boolean verdict and the list of error messages.

Validation drives a bounded generate-validate-repair loop, coordinated by `XmlValidationService`. When the XML is invalid, the collected schema errors are handed back to the generation layer, which regenerates a corrected draft; the new draft is normalized, mapped, marshalled, and validated again. The cycle repeats until the XML is schema-valid or a configured attempt budget is reached, after which the final XML and its validation status are returned. The budget keeps the loop terminating and the cost bounded. Crucially, the repair is targeted at structural defects only: because schema validity and the targeted tariff-rule violations are independent, a deliberately rule-violating invoice retains its intended violation while only incidental structural errors are corrected. Semantic correctness itself is not checked at runtime; it is assessed offline against a labeled dataset (Chapter 6), which keeps the runtime path lightweight.

5.6 Artifact Handling and Traceability

The generated invoice is delivered to the analyst rather than persisted: the service exposes the pipeline over a small REST and SSE interface and returns each invoice as a response or a file download. The JSON endpoint returns the structured draft for inspection, while the streaming endpoint emits per-step lifecycle events (`step-start`, `step-detail`, `step-finish`, and a final `result` or `error`) so the single-page frontend can show progress through the pipeline instead of an opaque wait. The XML endpoint runs the full pipeline including the bounded repair loop of Section 5.5 and returns the invoice as a downloadable attachment, with the validation outcome surfaced in response headers – whether the XML is schema-valid, whether the first pass was already valid, how many repair attempts were made, and the schema errors when validation ultimately fails. A separate conversion endpoint re-serializes an already-generated draft to XML without re-invoking the model, which is useful for exporting a result the analyst has already reviewed. The available endpoints are listed in Table 8. Consistent with the MVP scope, no long-term test-dataset store is maintained; artifacts are handed back to the caller.

Traceability is provided at request granularity. Each incoming request is tagged with a short correlation identifier placed in the logging context (MDC), so every log line emitted while interpreting, retrieving, generating, and validating that request carries the same identifier and a single generation can be reconstructed from the logs. Together with the streamed step events, this gives enough observability to debug runs and to follow the evaluation in Chapter 6. Persistent audit storage of prompts, retrieved sources, and validation outcomes is deliberately out of scope for the MVP.

The frontend that consumes these endpoints is shown in Appendix A.1.

Table 8: REST Endpoints Exposed by the Service

Endpoint	Purpose	Returns
POST /api/generate	Run the full pipeline.	Invoice draft (JSON) with warnings.
POST /api/generate/stream	Same, with progressive step events.	SSE event stream.
POST /api/xml	Run the pipeline incl. bounded XSD repair.	XML attachment + validation headers.
POST /api/xml/convert	Re-serialize an existing draft (no model call).	XML attachment + validation headers.

5.7 Summary

The implementation realizes the layered architecture of Section 4.2 as a single Spring Boot service that takes a natural-language request and returns a schema-compliant XML test invoice. Each architectural layer maps onto a distinct part of the codebase: prompt interpretation and scenario building in the generation layer, parallel structured and vector retrieval in the knowledge layer, and deterministic transformation and validation in the final layer. The intermediate artifacts passed between them – the interpreted prompt, the invoice scenario, the generated draft, and the validated XML – are explicit and inspectable, which is what makes a generation run reproducible and traceable.

Three engineering decisions follow directly from the design rationale in Section 4.2. First, domain knowledge is supplied through retrieval rather than baked into the model by fine-tuning, so tariff data and guideline documents can be updated independently of the LLM and the provenance of generated content remains visible. Second, everything that must be correct rather than plausible is handled by deterministic logic outside the model: cross-field date normalization, the JAXB mapping to the Sumex envelope, and XSD validation. The model proposes content; fixed logic decides its form and its validity. Where the generated XML does fail validation, a bounded generate-validate-repair loop feeds the schema errors back for a limited number of regeneration attempts, correcting structural defects without disturbing a deliberately injected rule violation. Third, the pipeline is modular: generation, retrieval, transformation, validation, and artifact handling are separable components, so a ranker, an embedding model, or a knowledge source can be replaced without touching the rest.

In keeping with the MVP scope, several capabilities are intentionally excluded. The service does not maintain a persistent test-dataset or audit store – artifacts are returned to the caller – and it does not perform full runtime semantic claim validation with rule-level self-correction. Semantic correctness and the targeting of specific rule violations are instead assessed offline, against a labeled reference dataset, which is the subject of Chapter 6.

Chapter 6

Evaluation

This chapter evaluates the prototype along two complementary lines. The first is an automated, quantitative evaluation of the generation pipeline against a labeled golden dataset, measuring syntactic validity, retrieval quality, semantic correctness, and the system’s ability to target specific rule violations. The second is a qualitative use-case study with practitioners who build test invoices, who perform a realistic test-data task once with the tool and once without it, so that the prototype’s practical value can be assessed against current manual practice rather than in isolation. Together, the two parts answer the research questions stated in Section 1.3: RQ1-RQ3 are addressed primarily by the quantitative evaluation, while RQ4 – the prototype’s effect on practical testing workflows – draws on both, combining objective coverage measures with the analysts’ first-hand assessment of usefulness.

6.1 Evaluation Objectives and Design

The evaluation is structured around the four research questions and is designed so that each is answered by an appropriate method rather than by a single aggregate score. RQ1 concerns whether the system reliably produces schema-compliant XML; it is measured by the syntactic validity of the generated invoices. RQ2 concerns controllable negative testing – generating an invoice that violates one specific rule while remaining otherwise well-formed – and is measured by how often the intended violation is actually realized. RQ3 concerns how structured and unstructured knowledge support controllable, traceable generation, and is measured through the quality of retrieval and the provenance of the codes the model emits. RQ4 concerns practical usefulness and is assessed both by the diversity and coverage of the generated test cases and by the qualitative study.

The quantitative evaluation runs the full pipeline over a frozen golden dataset of labeled scenarios spanning several prompt-detail tiers and both German and English (Section 6.2), and reports a set of metrics defined in Section 6.3. Because individual scenarios exercise different aspects of the system, metrics are computed only where they are meaningful and are reported with confidence intervals and per-slice breakdowns rather than as single

point estimates. The qualitative study (Section 6.5) uses a within-subjects, tool-versus-no-tool design to capture effects that the automated metrics cannot observe, such as task completion under a fixed time budget and perceived usefulness. Table 9 summarizes how each research question maps onto methods and metrics.

Table 9: Research Questions Mapped to Methods and Metrics

RQ	Method	Primary metrics
RQ1	Quantitative (golden dataset)	XSD validity (first-pass and post-repair); JSON structural validity.
RQ2	Quantitative (golden dataset)	Violation-injection success; scenario correctness.
RQ3	Quantitative (golden dataset)	Retrieval precision, recall, recall@ K ; code provenance.
RQ4	Quantitative + qualitative study	Coverage across tiers and languages; task completion and Sumex import; perceived usefulness (SUS).

6.2 Experimental Setup

The quantitative evaluation rests on a curated, labeled dataset built from real production invoices. This section describes how that dataset is sourced, labeled, cleaned, and assembled, and records the pipeline configuration under which the evaluation was run. Figure 11 shows the data lineage end to end, from the raw invoices to the evaluation harness.

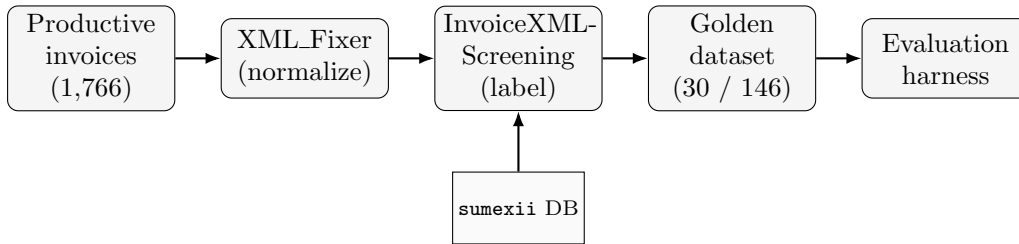


Figure 11: From Productive Invoices to the Golden Dataset

6.2.1 Source Corpus and Provenance

The corpus consists of real medical invoices drawn from a Sumex II test environment, in the production *generalInvoiceRequest* XML format. Using genuine invoices rather than hand-crafted examples keeps the evaluation grounded in the structures, value ranges, and edge cases that occur in practice.

Table 10 traces the path from these raw invoices to the evaluation dataset. Of 1,766 distinct invoices submitted to the screening tool, 744 fell outside the TARDOC tariff (`tariff_type` 007) and were skipped as out of scope, leaving 1,022 TARDOC invoices

labeled by the production validation engine: 802 valid, 212 contract violations, 4 rejections, and 4 unresolved cases. From the 1,018 usable labeled invoices, 30 records were curated into the golden dataset and expanded into 146 prompt variants. Making the funnel explicit shows that the dataset is a deliberate, balanced subset of real labeled invoices rather than a hand-picked sample.

Table 10: Data Provenance: From Screened Invoices to the Golden Dataset

Stage	Invoices
Distinct invoices submitted to screening	1,766
skipped – non-TARDOC-007 (out of scope)	744
TARDOC-007 labeled by the production engine	1,022
valid	802
contract violation (VLC)	212
rejection (RWC)	4
unresolved (error)	4
Curated golden dataset (records)	30
Prompt variants generated	146

6.2.2 Ground-Truth Labeling

The validity label of each invoice is not assigned heuristically but taken from the production validation engine itself. The *InvoiceXMLScreening* tool reads each invoice, looks it up in the operational *sumexii* database, and inspects the validation codes the engine recorded against it. Invoices carrying a rejection code (RWC) are labeled as rejected, those carrying a contract violation code (VLC) but no rejection as contract-violating, and those with neither as valid; cases that cannot be resolved are set aside as errors. Because the labels are the verdicts of the same engine the generated invoices are ultimately meant to test, they constitute trustworthy ground truth rather than an approximation, which is essential for interpreting the downstream metrics.

6.2.3 Data Normalization

Before the corpus is used, the *XML-Fixer* tool applies a deterministic cleanup pass. It normalizes a small set of fields whose malformed values would otherwise block schema validation – party and provider identifiers, social-security numbers, and contact addresses – substituting well-formed placeholder values, and it pre-checks each invoice against the XSD schema. The pass is purely mechanical and rule-based: it standardizes input quality without altering the clinical or tariff content that the evaluation depends on.

6.2.4 Golden Dataset Construction

The evaluation does not run against all 1,022 labeled invoices but against a smaller, curated *golden dataset* derived from them by the `GoldenDatasetPipeline`. The construction follows a reverse-engineering approach: rather than authoring scenarios by hand, each labeled invoice is treated as the ground truth from which the expected scenario fields, retrieval labels (tariff codes, treatment context, and, for invalid cases, the violation type), and validity outcome are recovered; a canonical business intent is then derived, and natural-language prompts are generated to target that known answer. This mirrors recent work on generating evaluation benchmarks from enterprise corpus data [53].

For each record, prompts are produced at increasing levels of abstraction, from an explicit code request (T0) through a complete specification (T1), a clinical description (T2), and a lay description (T3), to an explicit violation request (T4), in both English and German. The result is frozen as 30 human-validated records expanded into 146 prompt variants, which falls within the 50-200 scenarios targeted at the design stage. The invalid records cover nine violation types: eight synthetic structural violations (amount mismatch, duplicate code, invalid quantity, cumulation violation, missing base service, treatment mismatch, invalid billing role, and side duplication), and corpus-derived contract violations (CORPUS_VLC). Freezing the dataset makes every run comparable and the evaluation reproducible. Table 11 summarizes its composition.

Table 11: Golden Dataset Composition

Dimension	Count
<i>Records (30)</i>	
Valid	13
Invalid	17
synthetic violations	8
corpus violations	9
<i>Prompt variants (146) – by tier</i>	
T0 code-explicit	13
T1 specification-complete	26
T2 clinical	13
T3 lay	60
T4 violation	34
<i>Prompt variants – by language</i>	
English	86
German	60
<i>Scale</i>	
Service lines	282
Unique tariff codes	120

6.2.5 Pipeline Configuration and Reproducibility

The evaluation was run with the Kimi-K2.5 generation model and the `text-embedding-3-small` embedding model, using a low sampling temperature to favour structural consistency over variety, a retrieval depth of ten candidates per channel, hybrid structured retrieval (BM25 fused with dense similarity) with query expansion enabled, and the bounded XSD repair loop. The full configuration is listed in Table 12.

Each run is self-documenting. The harness writes a typed configuration snapshot (`config.json`) and run metadata (`run-meta.json`, including the model identifiers, the scenario source, timing, and the git commit) into a dedicated per-run directory, and appends one row per run to an append-only leaderboard. This makes any reported number traceable to the exact configuration and code revision that produced it, and lets runs be compared directly, which is what makes the model and embedding sweeps of Sections 6.4.6 and 6.4.7 practical. Response caching keeps repeated runs over identical inputs reproducible and avoids redundant model calls.

Table 12: Pipeline Configuration for the Evaluation Run

Setting	Value
Generation model	Kimi-K2.5 (Azure AI Foundry)
Embedding model	<code>text-embedding-3-small</code> (1536-dim)
Sampling temperature	0.2
Max output tokens	16,384
Retrieval top- k	10
Min. similarity score	0.30
Structured strategy	Hybrid (BM25 + dense, RRF)
Query expansion	Enabled
Max XSD repair attempts	2
Scenario source	Golden dataset

The headline first-pass figures are measured with the XSD repair loop disabled; the loop’s effect is reported separately in Section 6.4.5. The minimum similarity score is lowered from the shipped default of 0.35, which was too aggressive for this corpus (top observed similarity ≈ 0.39), dropping most chunks.

6.3 Evaluation Metrics

This section defines the metrics used in the quantitative evaluation, grouped by the dimension they measure, so that the results in Section 6.4 can be read without interruption. Throughout, a single generation run produces one invoice per scenario; rates are reported as the fraction of applicable scenarios that satisfy a condition. For the code-based metrics, let C_{gen} denote the set of tariff codes present in the generated invoice, C_{exp} the expected codes recorded in the golden dataset, and C_{ret} the codes contained in the retrieved knowledge for that scenario. All metrics and the research question each one informs are summarized in Table 24 (Appendix B.1).

6.3.1 Syntactic Validity

Syntactic validity captures whether a generated invoice is well-formed at two levels. *JSON structural validity* is the fraction of drafts that satisfy the structural contract checked before XML mapping (required fields present, correct types, at least one service record). *XSD validity* is the fraction of marshalled invoices that validate against the Sumex schema. Because the pipeline includes the bounded repair loop of Section 5.5, both are reported *first-pass* (before any repair) and *post-repair* (after the loop), and the *repair effectiveness* is the share of first-pass failures that repair converts into valid output:

$$\text{repair effectiveness} = \frac{\# \text{ first-pass failures repaired to valid}}{\# \text{ first-pass failures}}. \quad (6.1)$$

6.3.2 Retrieval Quality

Retrieval quality is measured on the codes the system commits to, relative to the gold codes, and is only computed for scenarios that specify expected codes; lay (T3) and violation (T4) prompts deliberately do not, and are excluded from these means rather than counted as zero. Precision, recall, and their harmonic mean [20] are defined as

$$P = \frac{|C_{\text{gen}} \cap C_{\text{exp}}|}{|C_{\text{gen}}|}, \quad R = \frac{|C_{\text{gen}} \cap C_{\text{exp}}|}{|C_{\text{exp}}|}, \quad F_1 = \frac{2PR}{P + R}. \quad (6.2)$$

TARDOC recall is a variant of recall whose expected set C_{exp} is first restricted to its TARDOC-format subset. Non-TARDOC identifiers (for example GTIN-13 pharmacy codes) can never appear in the `code` field of a valid TARDOC invoice, so counting them against recall conflates a genuine retrieval miss with a schema constraint; TARDOC recall removes that cap and is undefined for scenarios whose expected codes contain no TARDOC-format code.

Two recall variants isolate the retriever from the generator. *Recall@K* is a diagnostic probe measuring the fraction of expected codes found among the top- K retrieved document chunks, reported for $K \in \{5, 10, 15, 20, 30, 50\}$; *structured recall@20* is the fraction of expected codes present among the (at most twenty) tariff facts surfaced by the structured channel:

$$\text{recall@K} = \frac{|\{c \in C_{\text{exp}} : c \text{ appears in the top-}K \text{ chunks}\}|}{|C_{\text{exp}}|}. \quad (6.3)$$

Finally, two grounding proxies are reported. *Code-field faithfulness* is the fraction of emitted codes that are supported by the retrieved knowledge rather than drawn from the model’s parametric memory, and *context relevance* is approximated by the precision of the codes the model chose to use:

$$\text{faithfulness} = \frac{|\{c \in C_{\text{gen}} : c \in C_{\text{ret}}\}|}{|C_{\text{gen}}|}. \quad (6.4)$$

6.3.3 Semantic Correctness

Semantic correctness concerns whether the content is coherent and matches the intended scenario. *Required-field completeness* is the average fraction of mandatory invoice fields that are populated with non-default values. *Constraint-satisfaction rate* is the fraction of intra-invoice consistency constraints that hold (for example, the agreement between quantities, unit values, and line totals, and the ordering of dates). *Scenario correctness* is the fraction of scenarios whose generated validity matches the expected one: a valid scenario must yield an invoice not marked as intentionally invalid, and an invalid scenario must yield one that is.

6.3.4 Negative-Test Quality

Negative-test quality assesses whether a requested rule violation is actually realized. *Violation-injection success* is the fraction of negative scenarios in which the intended violation is detected in the generated invoice by a deterministic oracle (`ViolationOracle`). It is computed only for violation types with a structural oracle; violations that the oracle cannot adjudicate from the draft alone are excluded from the mean rather than scored as failures.

6.3.5 Operational Metrics

Three operational metrics characterize cost. *Latency* is the wall-clock time per scenario, reported as median (p50) and tail (p95), and includes any repair calls. *Repair attempts* is the mean number of repair LLM calls per scenario, separating the single JSON-structure pass from the bounded XSD loop. *Token usage* is the number of tokens consumed per scenario, reported separately as prompt (input) and completion (output) tokens and summed across the initial generation and any repair calls; it is taken from the provider-reported `usage` field rather than counted locally, and aggregated as a per-run sum and a per-scenario median and tail.

6.3.6 Statistical Treatment

Point estimates are reported with 95% confidence intervals obtained by non-parametric bootstrap resampling over scenarios [54] (`BootstrapCi`), which makes no distributional assumption and is appropriate for the small, fixed dataset. To expose where performance varies rather than only its average, every metric is additionally sliced by prompt tier (T0-T4), language (English/German), and validity class (`EvaluationSlicer`); a metric is reported for a slice only where it is meaningful for that slice.

6.4 Quantitative Results

This section reports the pipeline’s performance on the golden dataset under the configuration of Table 12. Unless stated otherwise, figures are from the baseline run; the first-pass-versus-post-repair comparison for XSD validity is drawn from the paired runs of the repair-loop ablation (Section 6.4.5). Aggregate values, with 95% bootstrap confidence intervals, are collected in Table 18.

6.4.1 Syntactic Validity (RQ1)

The system produces structurally well-formed invoices at a high rate. JSON structural validity is 95.2% on the first pass and rises to 99.3% after the single JSON-repair pass, which fired on 7 of 146 scenarios and corrected 6 of them (a repair effectiveness of 86%). XSD validity is 99.3% on the first pass, a direct consequence of the deterministic date normalization and JAXB mapping of Section 5.5, which leave little for the schema to reject. The bounded XSD repair loop is therefore a safety net rather than the main driver of validity: it engages only on the small residual of first-pass failures. In the paired repair-loop ablation, the loop-off run leaves a single schema-invalid invoice (145 of 146; 99.3%), which the loop-on run repairs, raising XSD validity to 100.0% (146 of 146). Repair effectiveness on the XSD residual is thus 100% (one of one), and because only this single scenario ever requires schema repair, the mean number of XSD repair calls per scenario is negligible (below 0.01). Figure 12 contrasts first-pass and post-repair XSD validity.

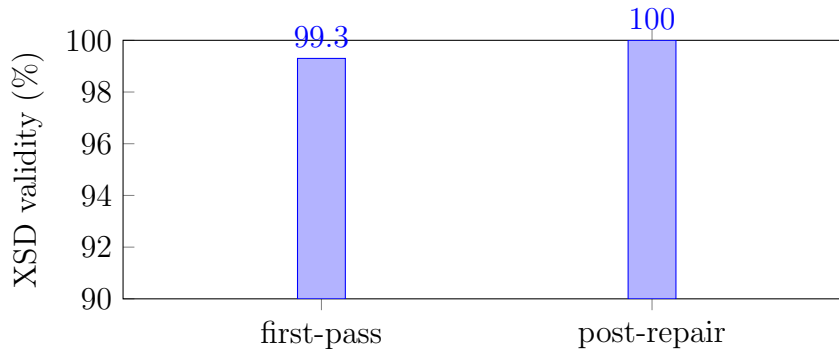


Figure 12: First-Pass vs. Post-Repair XSD Validity

6.4.2 Targeted Rule Violations (RQ2)

Scenario correctness is 100%: in every case the system produced an invoice whose validity (valid or intentionally invalid) matched the requested scenario. Realizing the *specific* requested violation is harder. Across the 28 negative scenarios with a structural oracle (seven violation types, four prompt variants each), the intended violation was injected successfully in 64.3% of cases (18 of 28), but the rate varies sharply by type (Table 13): amount mismatch, duplicate code, invalid billing role, and side duplication are

reproduced reliably, whereas missing base service and treatment mismatch are not yet achieved. Cumulation violations and corpus-derived contract violations (CORPUS_VLC) have no draft-side oracle (the violation cannot be adjudicated from the generated draft alone) and are therefore excluded from this measurement rather than scored as failures; they account for the remaining 40 of the 68 negative prompt variants.

Crucially, the negative invoices remain schema-valid while violating their target rule. On the invalid subset specifically, XSD validity is 98.5% on the first pass (the single schema-invalid invoice in the entire dataset falls in this subset) and reaches 100% after the repair loop. Structural validity and rule violation are thus independent: an invoice can carry a deliberate rule violation and still satisfy the schema, which is precisely the property that makes these invoices usable as targeted negative tests.

Table 13: Violation-Injection Success by Type

Violation type	n	Injected
Amount mismatch	4	100%
Duplicate code	4	100%
Invalid billing role	4	100%
Side duplication	4	100%
Invalid quantity	4	50%
Missing base service	4	0%
Treatment mismatch	4	0%
Overall	28	64.3%

6.4.3 Retrieval and Grounding (RQ3)

On the scenarios that specify expected codes, the production retrieval achieves a precision of 83.3%, a recall of 63.7% ($F_1 = 72.2\%$), and a TARDOC-only recall of 79.1%; the structured channel alone reaches a recall@20 of 61.3%. Grounding is strong: code-field faithfulness is 96.9%, meaning almost every emitted code is supported by retrieved knowledge rather than invented. The provenance of the 324 emitted codes confirms this: 76.2% trace to the structured tariff channel, 3.1% to retrieved document chunks, and 20.7% to the model’s parametric memory; the last figure is the share that query expansion is designed to convert into retrieved candidates.

The diagnostic recall@ K probe (Figure 13) is far lower (6-11%) and rises only marginally with K . This is expected and not a contradiction: the probe is a lower bound that queries the *document* channel with the raw prompt, whereas the production figures above use the full hybrid retrieval with code lookup and keyword distillation. The gap quantifies how much the structured channel and query construction contribute over naive document retrieval. The individual contribution of the structured strategy and query expansion is isolated in the ablation study of Section 6.4.5.

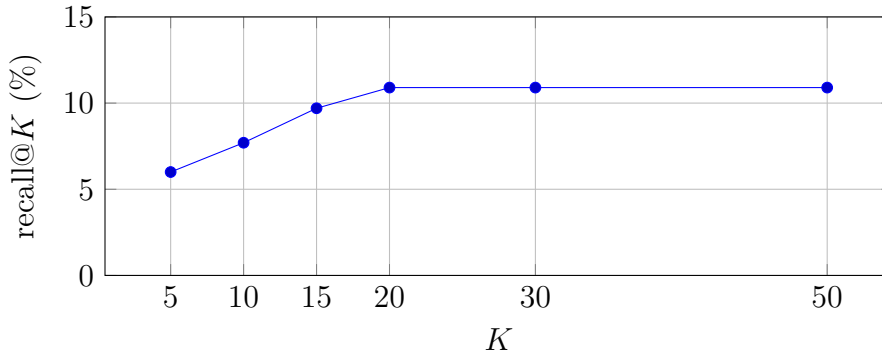


Figure 13: Diagnostic Recall@K (Document-Probe Lower Bound)

6.4.4 Coverage and Diversity

Performance holds across prompt tiers and both languages, with the expected gradient as prompts become less explicit (Table 25, Appendix B.2). Retrieval is strongest on code-explicit prompts (T0: precision 1.00, recall 0.79) and weakest on the fully specified T1 prompts (recall 0.53), while JSON validity stays at or near 1.00 across all tiers. German and English are comparable (recall 0.60 vs. 0.65; violation injection 0.64 for both), indicating the bilingual prompting does not disadvantage either language. Retrieval metrics are not applicable to the lay (T3) and violation (T4) tiers, which do not specify expected codes. Content quality is likewise high across the dataset: required-field completeness averages 86.9% and intra-invoice constraint satisfaction 99.9% (Table 18), so the invoices are not merely schema-valid but internally consistent across fields.

6.4.5 Ablation and Component Analysis

Beyond the repair mechanisms, the pipeline exposes several configurable levers whose individual effect can be isolated. Rather than a dedicated ablation batch, each lever was introduced as a single-variable change during development and measured against the immediately preceding configuration on the same golden dataset; the append-only leaderboard (Section 6.2) keeps each such pair directly comparable. Because these runs were executed on different days, they are subject to run-to-run and infrastructure noise; a conservative noise band (precision within 2pp, recall within 3pp, latency within 10%) is therefore applied, and only differences beyond it are read as genuine effects. Table 14 summarizes the levers, grouped into generation controls, retrieval components, and the structural safety nets.

The most consequential generation control is the sampling temperature. Lowering it from 0.7 to 0.2 raised violation-injection success from 0.27 to 0.68, by far the largest single-lever effect in the pipeline. At high temperature the model tended to paraphrase away the seeded violation, silently “repairing” the very defect the scenario was meant to introduce; the low temperature makes targeted negative generation controllable, which is why it is used throughout the evaluation (Table 12). Separately, raising the output-token budget from 8,192 to 16,384 removed a class of truncation failures and lifted scenario correctness

Table 14: Ablation of the Pipeline’s Configurable Levers

Lever	Change	Effect (key metric)
<i>Generation controls</i>		
Sampling temperature	0.7 $\rightarrow$ 0.2	Violation injection 0.27 $\rightarrow$ 0.68
Max output tokens	8 192 $\rightarrow$ 16 384	Scenario correctness 0.92 $\rightarrow$ 0.97
<i>Retrieval components</i>		
Query expansion	off / on	Recall 0.46 $\rightarrow$ 0.55; precision 0.65 $\rightarrow$ 0.69
Hybrid retrieval	BM25 / BM25+dense	Structured recall@20 0.40 $\rightarrow$ 0.59 (CI-sig.)
<i>Structural safety nets</i>		
JSON repair pass	off / on	JSON validity 95.2% $\rightarrow$ 99.3%
XSD repair loop	off / on	XSD validity 99.3% $\rightarrow$ 100.0%

from 0.92 to 0.97, confirming that some early failures were budget-bound rather than capability-bound.

On the retrieval side, two levers improve grounding. Query expansion (translating clinical or lay phrasing into German TARDOC keywords before the structured lookup) raised retrieval recall from 0.46 to 0.55 and precision from 0.65 to 0.69, and shifted code provenance away from parametric memory (from 38% to 33%). Adding the dense channel to form the hybrid ranker produced the clearest retriever-side result: structured recall@20 rose from 0.40 [0.30, 0.51] to 0.59 [0.49, 0.68], a lift whose bootstrap intervals barely fail to overlap and is therefore statistically robust. The end-to-end retrieval recall lift (0.61 to 0.63) stays within the noise band, so the hybrid channel’s benefit is best described as significantly improving the candidate pool while the LLM only partially consumes it, consistent with the provenance shift from memory toward the structured channel (memory share 23% to 18%). This gap between an improved candidate pool and the LLM’s selection from it is the clearest open lever for future work.

Finally, the two deterministic safety nets act only at the margin: the JSON-repair pass lifts JSON validity from 95.2% to 99.3%, and the bounded XSD loop closes the single residual schema failure (99.3% to 100.0%). Read together, the ablation localizes where each part of the pipeline earns its place: temperature governs controllability, query expansion and the hybrid channel govern grounding, and the repair loops guarantee structural validity at the margin. Notably, the retrieval-side levers move provenance from parametric memory toward the structured channel at every step, which is the mechanism behind the strong grounding reported in Section 6.4 and the model-invariance observed in Section 6.4.6. The one lever not evaluated here is the relevance threshold on the vector channel, which is left to future work.

6.4.6 Sensitivity to the Generation Model

The results reported so far use a single generation model. To test whether the approach depends on that choice or generalizes, the full pipeline was re-run on the golden dataset

with the generation model swapped for each of eight alternatives, spanning frontier and small models from five families, while every other component was held fixed: the embedding model, retrieval configuration, prompts, and dataset of Table 12. All sweep runs share a single code revision and are therefore mutually comparable; the Kimi-K2.5 configuration used elsewhere in this chapter is re-run here as the in-sweep reference. Because that revision differs slightly from the baseline of Section 6.4, absolute values should be read within the sweep rather than against the earlier baseline. Table 15 reports the outcome, ordered by median latency.

Table 15: Sensitivity to the Generation Model

Model	Structural (%)		Semantic (%)		Retrieval & grounding (%)					Latency (s)	
	XSD	JSON	Scen	Viol	Prec	Rec	sR@20	Faith	Mem	p50	p95
GPT-4.1-nano	100.0	89.0	99.3	46.4	67.0	59.2	57.9	93.3	16.2	3.5	10.8
Gemma-4-31B	100.0	96.6	100.0	71.4	82.1	66.8	58.1	96.1	22.3	7.2 [‡]	25.2 [‡]
GPT-5.1	100.0	98.0	100.0	71.4	76.3	63.8	60.0	95.3	8.6	9.1	23.8
DeepSeek-V4-Flash	93.8	96.6	100.0	71.4	75.9	62.7	58.5	95.5	11.3	10.7	36.5
Mistral-Large-3	46.9	99.3	99.3	67.9	75.8	66.7	62.5	94.7	25.9	16.6	50.8
DeepSeek-V4-Pro	97.9	94.5	99.3	70.4	70.7	61.9	59.6	94.1	27.8	18.1	50.9
Kimi-K2.6	99.3	97.3	98.0	60.7	87.6	68.1	60.1	97.7	12.5	35.3	87.9
Kimi-K2.5 [†]	95.9	98.6	100.0	71.4	83.1	62.2	57.3	96.9	14.1	37.8	101.4

Metrics: XSD/JSON validity, Scen(ario correctness), Viol(ation injection), Prec(ision), Rec(all), structured recall@20 (sR@20), Faith(fulness), Mem = share of emitted codes from parametric memory (lower is better-grounded). p_{50}/p_{95} are median and tail latency. [†] Default configuration used throughout the rest of the chapter. [‡] Gemma-4-31B is served via a rate-limited gateway; its latency is a serving-stack artifact and is not comparable to the Azure-hosted cells. GPT-5.1 is a reasoning model whose endpoint does not honour the temperature setting, so its single-scenario precision may drift more than the other cells.

The central observation is that the pipeline is robust to the choice of model. Scenario correctness remains at or near 100% for every model ($\geq 98\%$), JSON structural validity stays between 89.0% and 99.3%, and code-field faithfulness never falls below 93.3%. Retrieval quality is especially stable: recall lies in a narrow 59.2 - 68.1% band, precision in 67.0 - 87.6%, and structured recall@20 in a tight 57.3 - 62.5% span – despite the wide capability gap between, for example, GPT-4.1-nano and the frontier Kimi and GPT-5.1 models. This stability is by design: retrieval and grounding are performed by model-independent components (the shared embedding index and the structured tariff channel), and the generation model influences only which of the retrieved codes it commits to, while the deterministic transformation and validation stages operate identically regardless of model. Most of the system’s quality is therefore carried by its architecture rather than by any single model, which is exactly the property required for the approach to generalize. The complementary provenance view tells the same story: the share of codes traceable only to parametric memory ranges from 8.6% (GPT-5.1) to 27.8% (DeepSeek-V4-Pro), so even the least-grounded model still draws roughly three-quarters of its codes from retrieved knowledge, and this residual memory share is precisely what query expansion is designed to reduce.

Three metrics do separate the models. First, XSD validity, though high for most (93.8 - 100%), collapses to 46.9% for Mistral-Large-3, while that same model produces the *highest*

JSON structural validity (99.3%). The pattern indicates that Mistral emits drafts that are well-formed as JSON but carry values the schema rejects (out-of-domain codes, identifier formats, or value ranges), which the deterministic mapper standardizes structurally but does not rewrite semantically. Model choice therefore still matters for the value-level schema conformance that transformation alone cannot guarantee. Second, violation injection is stable around 61 - 71% for the capable models but drops to 46.4% for the smallest model, GPT-4.1-nano, consistent with targeted negative generation being the most capability-sensitive task in the pipeline. Third, and most sharply, end-to-end latency spans an order of magnitude, from a 3.5s median for GPT-4.1-nano to 37.8s for Kimi-K2.5.

Taken together, these results reframe the generation model as a quality-latency lever rather than a correctness prerequisite. The frontier Kimi models achieve the best retrieval precision (Kimi-K2.6: 87.6%) but are also the slowest ($p_{50} \approx 35\text{--}38\text{ s}$). Several far faster models match or exceed the Kimi-K2.5 default on the primary metrics at a fraction of the cost: GPT-5.1 reaches 100% XSD validity, 63.8% recall, and 95.3% faithfulness at a 9.1s median, and Gemma-4-31B reaches comparable recall (66.8%) on a mid-size model, though its latency is not directly comparable (rate-limited gateway). For a test-data generation tool, where throughput and predictability matter, this is a favourable finding: the prototype is not tied to the most expensive model, and a deployment can trade a few points of retrieval precision for a four- to fivefold latency reduction. More fundamentally, the consistency of the core metrics across eight heterogeneous models substantially strengthens the external validity of the evaluation: the conclusions drawn for Kimi-K2.5 are not an artifact of that model but hold across the model landscape.

6.4.7 Sensitivity to the Embedding Model

The retrieval side depends on a second model choice: the dense embedding used by the vector channel and the dense half of the hybrid ranker. To test whether a larger embedding improves retrieval, the default `text-embedding-3-small` (1536 dimensions) was swapped for `text-embedding-3-large` (3072 dimensions) on three generation models (Kimi-K2.5, DeepSeek-V4-Pro, and Mistral-Large-3), holding the LLM and all other configuration fixed. Mistral is the sharpest test: it held the highest small-embedding structured recall@20 (62.5%), so if any model’s retriever benefits from a larger embedding, it should. Table 16 reports the outcome.

The larger embedding is a null result. The retriever-only metric (structured recall@20) moves only marginally and with mixed sign (+3.5 pp for Kimi, +0.9 pp for DeepSeek-V4-Pro, and −1.5 pp for Mistral), and all three deltas lie inside the bootstrap noise band. Critically, the model with the most to gain, Mistral, actually *regresses*: 3-large does not push structured recall@20 past the 62.5% small-embedding ceiling. No model consumes an end-to-end lift either: recall is flat-to-down and precision falls on all three. The parametric-memory share even moves in opposite directions (Kimi +10 pp, DeepSeek-V4-Pro −4 pp, Mistral −2 pp), confirming that the provenance differences are run-to-run wobble rather than an embedding effect.

The interpretation is that the dense channel is already saturated at 1536 dimensions for this tariff catalogue of roughly 1,400 services: the bottleneck is the LLM’s code selection

Table 16: Sensitivity to the Embedding Model

Model	Embedding	sR@20	Recall	Prec.	Mem	XSD
Kimi-K2.5	3-small (1536)	57.3	62.2	83.1	14.1	95.9
Kimi-K2.5	3-large (3072)	60.8	62.0	79.5	24.3	99.3
DeepSeek-V4-Pro	3-small (1536)	59.6	61.9	70.7	27.8	97.9
DeepSeek-V4-Pro	3-large (3072)	60.5	60.5	70.0	23.7	95.1
Mistral-Large-3	3-small (1536)	62.5	66.7	75.8	25.9	46.9
Mistral-Large-3	3-large (3072)	61.0	64.7	71.9	23.8	44.4

All values in percent. sR@20 = structured recall@20 (retriever-only); Mem = parametric-memory share (lower is better-grounded).

from the retrieved candidates, not the embedding’s representational capacity, consistent with the earlier observation that recall@ K plateaus beyond $K = 20$ (Section 6.4). The larger embedding also carries a real cost: at 3072 dimensions it exceeds the limit for `pgvector`’s HNSW index, forcing an exact scan and a full re-ingestion, and it roughly doubled the wall-clock time on Kimi. The recommendation is therefore to stay on `text-embedding-3-small` across all model families: it is cheaper, index-friendly, and loses nothing measurable.

6.4.8 Computational Cost: Latency and Token Usage

End-to-end latency is dominated by the LLM calls. Under the baseline configuration the median scenario completes in 46.8s and the 95th percentile in 109.3s, the tail reflecting the longer, fully specified prompts and any repair calls. That the generation model is the dominant cost is confirmed directly by the model sweep (Section 6.4.6): median latency tracks the choice of model across an order of magnitude (from 3.5s for the smallest model to 37.8s for Kimi-K2.5), while the deterministic retrieval, transformation, and validation stages contribute a comparatively small, model-independent overhead. Repair adds latency only to the minority of scenarios it touches: the JSON-repair pass fires on 7 of 146 scenarios and the bounded XSD loop on a single one, so it lengthens the tail rather than the median.

Token consumption tells the same story from the cost side and explains where these latency differences originate. Because the harness records the provider-reported usage (`prompt_tokens` and `completion_tokens`) of every LLM call and sums it across the initial generation and any repair passes, per-scenario and per-run token totals are directly comparable across models. Table 17 reports these totals for four generation models re-run on the full 146-scenario golden dataset with token capture enabled. The four were chosen to span the cost–latency spectrum observed in the sweep (Section 6.4.6), from the fastest and leanest (Gemma-4-31B) to the slowest and most token-hungry (Kimi-K2.5), and to include both the quantitative baseline (Kimi-K2.5) and the model deployed in the use-case study (GPT-5.1); token accounting was added to the harness after the main sweep, so a representative subset rather than all eight models was re-measured. All four runs share the configuration of Table 12 and captured provider-reported usage on all 146 scenarios.

Total token spend varies roughly twofold across models on an identical workload (from 478,829 tokens for Gemma-4-31B to 990,501 for Kimi-K2.5), while the median scenario costs between 2,884 and 5,697 tokens. The spread is driven almost entirely by output rather than input: prompt totals lie in a narrow 336k–395k band, since the same retrieved context and instructions are sent to every model, whereas completion totals range from 121k to 655k. The completion-to-prompt ratio captures this directly: it is 1.95 for Kimi-K2.5, whose reasoning-style deployment counts an internal scratchpad against completion tokens, but only 0.34 for Gemma-4-31B, which emits compact structured JSON with no visible reasoning trace. Token usage broadly tracks latency (the cheapest model is also the fastest and the most expensive the slowest), though the two mid-range models trade places, since a reasoning model can be slow without emitting the most tokens. Either way, the generation model is the dominant, and essentially the only substantial, cost lever in the pipeline: the deterministic stages add a fixed overhead that does not scale with model choice. These figures include the tokens consumed by the repair passes when they fired, so they reflect the true end-to-end cost of a scenario rather than the initial call alone, which slightly penalizes models whose first-pass output triggers more repairs.¹

Table 17: Model Token Usage on the Golden Dataset

Model	Total tokens			Per scenario		
	Prompt	Compl.	Total	p50	p95	C:P
Gemma-4-31B	357,589	121,240	478,829	2,955	4,776	0.34
DeepSeek-V4-Flash	342,294	158,788	501,082	2,884	6,145	0.46
GPT-5.1	395,000	303,293	698,293	3,474	12,176	0.77
Kimi-K2.5	335,986	654,515	990,501	5,697	13,439	1.95

Prompt/Compl./Total are summed over all scenarios (initial generation plus any repair calls); p50/p95 are the per-scenario total-token median and tail. C:P = completion-to-prompt ratio. Token counts are provider-reported (`usage` field); monetary cost is not computed.

6.5 Qualitative Evaluation: Use-Case Study

The quantitative evaluation establishes how well the pipeline performs in isolation, but it cannot tell whether the tool actually helps in practice. This section reports a use-case study with practitioners who build test invoices as part of their work, addressing the practical side of RQ4: whether generating test invoices with the tool is faster, broader in coverage, and perceived as more useful than the current manual approach. The study follows a within-subjects design in which each participant performs a comparable test-invoice task once with the prototype and once using their current manual workflow, so that the prototype is assessed against real practice rather than in the abstract.

¹The token-usage runs are a separate, later run set from the model sweep of Section 6.4.6 and were executed on four models spanning the observed latency range; per-scenario latencies therefore differ marginally from Table 15. Monetary cost is not computed by the harness: it depends on per-provider pricing and can be derived as $\Sigma_{\text{prompt}} \cdot p_{\text{in}} + \Sigma_{\text{completion}} \cdot p_{\text{out}}$ from the totals reported here.

Table 18: Aggregate Metrics for the Baseline Configuration

Metric	Value	95% CI
XSD validity (first-pass)	99.3%	[97.9, 100.0]
XSD validity (post-repair)	100.0%	[100.0, 100.0]
JSON validity (first-pass)	95.2%	[91.8, 98.6]
JSON validity (post-repair)	99.3%	[97.9, 100.0]
Retrieval precision	83.3%	[75.0, 90.9]
Retrieval recall	63.7%	[54.3, 73.2]
Retrieval F_1	72.2%	—
TARDOC recall	79.1%	[70.6, 86.9]
Structured recall@20	61.3%	[51.9, 70.7]
Code-field faithfulness	96.9%	[95.0, 98.5]
Required-field completeness	86.9%	[83.9, 89.8]
Constraint satisfaction	99.9%	[99.7, 100.0]
Scenario correctness	100.0%	[100.0, 100.0]
Violation injection	64.3%	[46.4, 82.1]
Latency (p50 / p95)	46.8 s / 109.3 s	—

95% percentile bootstrap, 10,000 resamples over scenarios. First-pass XSD is from the repair-off run, post-repair from the repair-on run (Section 6.4.5). F_1 is derived from precision and recall; latency percentiles are reported without a CI. Intervals are wide for small- n metrics (violation injection, $n = 28$).

The design, procedure, and measures are described first, followed by the objective task outcomes (Section 6.5.5), the subjective ratings (Section 6.5.6), and the recurring themes from the questionnaires and free-text feedback (Section 6.5.7). Because the participant pool is small and drawn from a single organization, the results are read as indicative rather than statistically conclusive, and are interpreted alongside the quantitative findings in the discussion (Section 6.6).

6.5.1 Study Design

The study uses a within-subjects design [55]: each participant carries out the same three test-invoice scenarios twice – first manually (the *no-tool* condition) and then with the prototype (the *tool* condition). A within-subjects design is chosen because the pool of qualified participants is small, and having each participant serve as their own control yields more signal than a between-subjects split. The two conditions are run in a fixed order – manual first, then tool – so that each participant’s current manual practice is captured as a baseline before the tool is introduced; within each condition the participant is free to choose the order of the three scenarios. Each condition is time-boxed to ten minutes (a hard cut), and the same three scenarios are used in both. This keeps the task realistic and the sessions short, but it introduces a potential order and learning effect – having just attempted a scenario manually can ease the subsequent tool attempt – which is revisited in the threats to validity (Section 6.7).

Participants are practitioners who construct test invoices as part of their work – business analysts, engineers, and domain experts. Only one participant had prior hands-on ex-

perience with the prototype; the rest encountered it for the first time during the study, so the usability ratings largely reflect first-time use. The three scenarios increase in difficulty: (S1) a valid invoice containing a radiology (X-ray) service; (S2) an invoice with a service billable only for female patients; and (S3) a deliberately faulty invoice combining a consultation with a service that may not be billed in the same session – the kind of *targeted rule violation* the system is designed to produce. In the tool condition the prototype is configured with the GPT-5.1 model and German prompting (Section 6.4.6). The invoices from both conditions are read into Sumex – the validation system the generated invoices are ultimately meant to test – and each participant records whether the invoice was imported successfully before completing the questionnaire. Figure 14 traces this flow end to end: the solid path is the task (the same three scenarios worked first manually and then with the tool, ten minutes per condition, each read into Sumex and its import outcome recorded), and the dashed path is the subjective measurement via the post-task questionnaire. Table 19 summarizes the design.

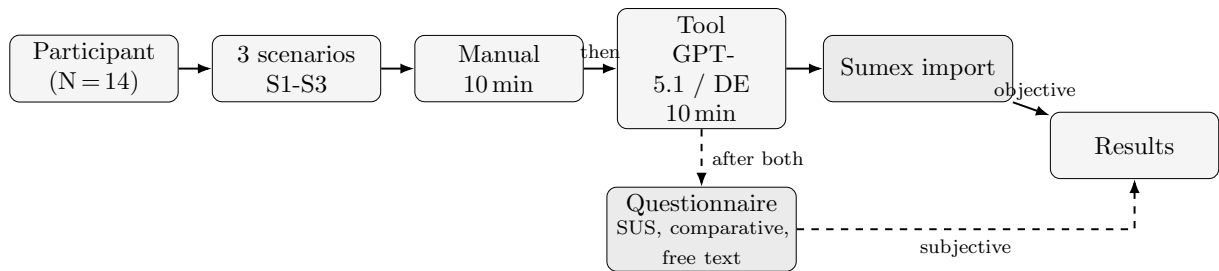


Figure 14: Use-Case Study: Task and Measurement Flow

Table 19: Use-Case Study Design

Element	Description
Design	Within-subjects; each participant works the same three scenarios first manually, then with the tool.
Order	Fixed (manual → tool), not counterbalanced; scenario order within a condition is free.
Participants	Practitioners who author test invoices (business analysts, engineers, domain experts).
Scenarios	S1 valid X-ray invoice; S2 female-only service; S3 illegal service combination (increasing difficulty; identical in both conditions).
Conditions	<i>Tool</i> : the prototype UI (GPT-5.1, German prompting); <i>no-tool</i> : current manual approach (editor, schema, tariff lookup).
Time budget	Ten minutes per condition (hard cut); questionnaire ≈25 min.
Materials	Task sheet/instructions; post-task questionnaire (SUS, comparative ratings, free-text items).
Measures	Scenario completion within the time budget; Sumex import success; perceived usefulness (SUS and comparative ratings); open feedback.

6.5.2 Participants

Fourteen practitioners took part, spanning the roles that build test invoices in practice: seven software engineers, two business analysts, two domain experts, one project leader, and two who classified themselves as *other*. Their anonymized profiles are listed in Table 26 (Appendix B.3).

Both self-rated familiarity dimensions skew toward the experienced end, which is appropriate for the target user group. Familiarity with XML/XSD invoice structures is high (mean 4.1 of 5, median 4), and familiarity with healthcare invoice validation is somewhat lower but still solid (mean 3.6, median 3.5); only two participants rate below the midpoint on either dimension. The pool therefore represents competent practitioners rather than novices, which strengthens the relevance of the usability ratings but limits how far they generalize to first-time or non-technical users – a point revisited in the threats to validity (Section 6.7). Figure 21 in Appendix B.3 shows the two distributions.

6.5.3 Procedure and Materials

Each session begins with a short briefing based on the written instructions. Each participant then works the three scenarios first manually and afterwards with the tool – ten minutes per condition as a hard cut (Section 6.5.1). In the no-tool condition they work as they normally would, with access to a text editor, the XSD schema, and the tariff catalogue but without the prototype; in the tool condition they use the prototype to generate and export an invoice per scenario. The same three scenarios (S1-S3) are used in both conditions. After completing both conditions, each participant reads the produced invoices into Sumex, records whether each was imported successfully, and fills in the questionnaire.

For this study the prototype was configured with the GPT-5.1 generation model rather than the Kimi-K2.5 baseline of the quantitative evaluation. The choice follows directly from the generation-model sweep (Section 6.4.6) and the token-usage comparison (Table 17): GPT-5.1 lies on the quality-latency Pareto frontier, matching the baseline on the core correctness metrics (100% XSD validity, 100% scenario correctness, on-par violation injection) and grounding best of all models tested, while running roughly four times faster (median ≈ 9 s vs. 38 s per scenario). Because responsiveness is itself a usability factor in an interactive, human-in-the-loop setting, the deployment used the frontier-fast model; the near model-invariance of the core metrics (Section 6.4.6) keeps this consistent with the quantitative findings.

6.5.4 Measures

The study combines objective task measures with subjective assessment, mirroring the facets of RQ4:

- **Efficiency** – the number of scenarios each participant completed within the ten-minute budget in each condition, as an objective proxy for the manual effort the tool removes (elapsed time itself was not recorded).

- **Coverage** – which of the three scenario types (valid, gender-restricted, illegal-combination) each condition managed to produce; with a fixed three-scenario set this is read as the breadth of completed cases rather than a continuous measure.
- **Validity** – whether each produced invoice was read into Sumex successfully, recorded per scenario and condition as the import outcome.
- **Perceived usefulness** – a standardized usability questionnaire (SUS) together with targeted tool-experience and direct tool-versus-manual Likert items, complemented by open-ended feedback.

The efficiency, coverage, and validity measures are taken from participants' own records of what they completed and imported into Sumex; the perceived-usefulness measures come from the post-task questionnaire.

6.5.5 Task Outcomes

The objective outcomes favour the tool decisively. Across the 42 scenario attempts per condition (14 participants $\times$ three scenarios), every tool-generated invoice was completed within the ten-minute budget and read into Sumex successfully (42/42, 100%).² The manual condition reached 69% (29/42): one invoice was completed but rejected on import, and twelve were not finished within the budget at all. Figure 15 contrasts the two conditions.

The manual shortfall is concentrated rather than diffuse. Four participants did not complete the manual block within ten minutes for any scenario – in each case because they lacked a suitable example invoice to adapt and spent the budget searching for or anonymizing one, exactly the bottleneck the free-text responses describe (Section 6.5.7). A single further invoice was completed but not imported, owing to a hand-produced malformed XML. The tool removed both failure modes: it always emitted a schema-valid invoice that Sumex accepted.

Two caveats temper this result. First, a successful import is a floor, not a ceiling: it confirms that the invoice is structurally acceptable to Sumex, not that its business content is correct. Several participants noted that imported tool invoices still carried issues that real use would expose – an identical `request_id` across generations (which Sumex would reject as a duplicate on re-import) and placeholder insurer and provider identifiers needing correction – which are analysed in Section 6.5.7. Second, because the conditions ran in fixed order (manual then tool) on the same scenarios, part of the tool's completion advantage reflects the familiarity participants had already built manually; this order effect is discussed in the threats to validity (Section 6.7).

²One tool attempt was initially blocked by the model provider's content filter and succeeded on a retry; it is counted as imported.

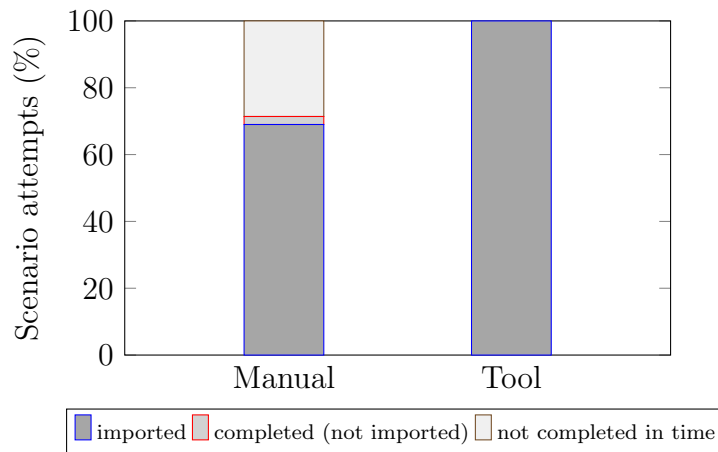


Figure 15: Scenario Completion and Sumex Import by Condition

6.5.6 Perceived Usefulness

On the direct comparison with manual work, the tool is rated clearly positively on effort and speed but only neutral on confidence. Averaged over the fourteen participants, four of the five comparative items sit near the top of the scale: the tool reduced the effort of creating test invoices (mean 4.4 of 5), helped complete more scenarios within the time limit (4.3), made the targeted scenarios easier to generate (4.3), and made it easier to keep invoices XSD-valid while meeting the scenario goal (4.3). The one exception is confidence in the resulting test cases, which lands exactly at the neutral midpoint (3.0). Figure 16 shows the contrast.

The item-level ratings of the tool itself explain that gap (Table 20). Participants rate the structural quality of the output highly (structural correctness 4.6, natural-language input 4.8) and find the explanations reasonably clear (3.9), but they are more reserved about the business content: usefulness for their testing needs (3.3), realism of the generated invoices (3.1), the amount of manual correction still required (3.1), and trust in the invoices for real use (3.1) all sit around the midpoint. The pattern is consistent: what the tool produces is structurally sound and quickly obtained, but its non-tariff content is not yet trusted without review, which matches the import caveats of Section 6.5.5 and the free-text themes in Section 6.5.7.

Three factors explain these reserved items (realism, remaining manual correction, and trust for real use), and together they place the scores as a floor rather than a ceiling (Table 21). The primary factor is a deliberate scope boundary. The prototype grounds the tariff-code selection but not the surrounding value fields: the prices and tax points, the provider and insurer identifiers (GLN, ZSR, Mandant) in the invoice body, and several per-line fields such as the reference code and the entry reason. These are filled from static defaults rather than derived dynamically, and doing so is explicitly left to future work (Chapter 7). The reserved ratings therefore fall on fields the prototype does not yet populate; one participant made the point directly, noting that such service-line values are “sometimes not realistic ... I guess, not considered yet.”

A second, smaller factor is the genuine residual the quantitative evaluation already measures:

a retrieval recall of 63.7% leaves some expected codes unselected, and a violation-injection success of 64.3% means a targeted rule is not always hit. A third factor is trust calibration: because all but one participant met the tool for the first time (Section 6.5.2), the trust rating is a first impression formed before its reliable and unreliable areas are known, and appropriate reliance on an automated aid is generally learned through repeated, verified use [56]. Consistent with this, the one already-familiar participant rated trust above the neutral midpoint (4 of 5), though at $n = 1$ this is only suggestive. Taken together, the neutral trust is best read as a floor set by an out-of-scope feature gap, a bounded and measured content residual, and first-impression uncertainty that calibrated use would resolve, rather than as a ceiling on what the approach can achieve.

Table 20: Tool-Experience Ratings

Item	Mean
Structural correctness	4.6
Natural-language input	4.8
Explanation clarity	3.9
Confidence in targeting	3.9
Usefulness for testing	3.3
Realism / plausibility	3.1
Manual correction still needed	3.1
Trust for real use	3.1

All items on a 1-5 scale; higher is more positive, except “manual correction still needed,” where higher means more correction.

Table 21: Reserved Business-Content Ratings and Their Interpretation

Item	Mean	Likely cause	Addressed by
Realism / plausibility	3.1	Value fields not yet populated dynamically (tax points, reference code, entry reason); minor recall/violation residual	Ground value fields; extend retrieval and oracles
Manual correction still needed	3.1	Ungrounded fields (prices, GLN, ZSR, Mandant) corrected by hand	User-fixable values plus field grounding
Trust for real use	3.1	Ungrounded value fields; also a first-impression rating (calibrates with use)	Field grounding; repeated calibrated use

All items on a 1 to 5 scale; higher is more positive, except “manual correction,” where higher means more correction. Causes and remedies summarize Sections 6.5.6 to 6.5.7 and Chapter 7.

By contrast, overall usability is rated excellent. The System Usability Scale [57] gives

a mean of 87.0 (median 86.3, range 75 to 100, 95% CI [83.4, 90.5]), placing the tool well above the established SUS average of 68 [58] and inside the top (*excellent*) adjective band [59] (Figure 17). As noted, first use for all but one participant means this reflects first-impression usability rather than familiarity built up over repeated use.

Adoption intent is correspondingly positive. Asked whether they would use such a tool in their daily workflow, eight of the fourteen answered yes and six were tentatively positive (*maybe*); none declined. The tentative responses cluster with the confidence and trust reservations above, indicating that willingness to adopt is gated less by usability than by the business-content gaps.

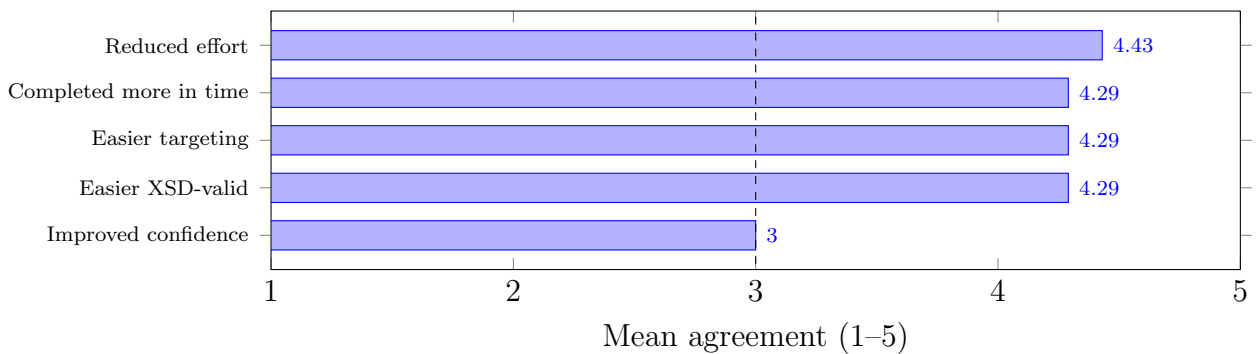


Figure 16: Tool vs. Manual: Mean Agreement on the Comparative Items

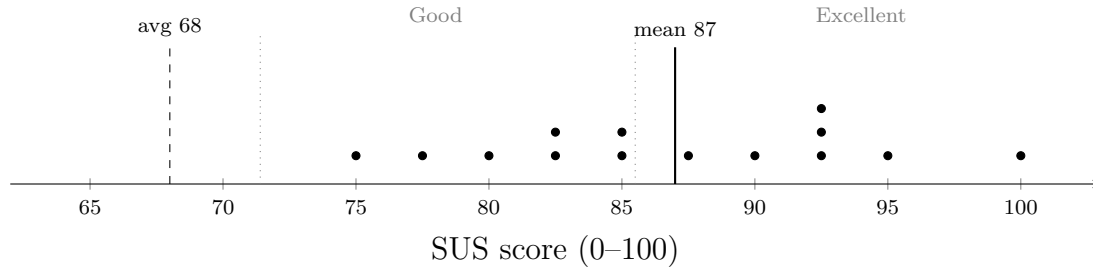


Figure 17: SUS Scores Against the Reference Average and Adjective Bands

6.5.7 Participant Feedback and Observations

The open-ended responses converge on a consistent picture. Participants value the tool for the same three things across almost all questionnaires: it produces a schema-valid invoice quickly, it selects the correct tariff positions, and it accepts a plain natural-language description of the scenario. Their reservations are equally consistent and concern the content the tool does not yet ground in authoritative data: the prices and tax points are placeholder values rather than real tariff figures, and the provider and insurer identifiers (GLN, ZSR, Mandant) are largely hardcoded and must be corrected by hand. A third group of comments concerns specific-rule fidelity, where the generated invoice sometimes realizes a different violation than intended or silently breaks a limit the participant did not target. Recurring themes were identified by thematic analysis of the free-text responses

[60]. Table 22 collects them with representative comments, and Figure 22 (Appendix B.3) plots their frequencies.

These themes line up closely with the quantitative results. The praise for tariff-code selection mirrors the high code-field faithfulness (96.9%) and the large share of codes traced to the structured channel, so participants confirm from experience what the provenance analysis measures. The difficulty with specific rules, in turn, matches the violation-injection result (64.3%). The hardest study scenario, S3, is an illegal service combination, which belongs to the cumulation family that the quantitative evaluation could not adjudicate with a draft-side oracle; one domain expert indeed obtained an inclusion violation where an exclusion was intended. The complaints about prices, tax points, and partner data fall outside the tariff-grounding scope of the current prototype, so they are best read as feature gaps rather than retrieval failures. The full cross-comparison is drawn together in the discussion (Section 6.6).

A few individual observations are worth recording. One participant’s second scenario was initially blocked by the model provider’s content filter and succeeded only on a retry. Several noted that the tool reuses an identical `request_id`, which Sumex rejects as a duplicate when the same invoice is imported twice. The validity explanation divided opinion: some found it the most helpful feature, while others ignored it or found the long free-text form off-putting. The requested improvements form a coherent roadmap: user-fixable field values for partners and insurers, support for tariffs beyond TARDOC, batch generation of several invoices, and an iterative refinement mode with an in-tool XML preview.

6.6 Discussion

This section interprets the evaluation against the four research questions and brings the two strands of evidence together. Each question is discussed in turn, drawing on the quantitative results of Section 6.4 and, for RQ4, on the use-case study of Section 6.5. Two cross-cutting themes then follow: the division of labour between deterministic and probabilistic components, and the overall strengths and limitations of the approach.

RQ1 – XSD-compliant generation. The system reliably produces schema-valid invoices: XSD validity is 99.3% on the first pass, and JSON structural validity rises from 95.2% to 99.3% after a single repair pass. The notable finding is *where* this reliability comes from. It is not the language model guaranteeing structure, nor primarily the repair loop, but the deterministic post-processing (date normalization and the JAXB mapping onto the Sumex envelope) that leaves little for the schema to reject. The bounded XSD repair loop is consequently a safety net for the small residual rather than the main mechanism.

With the repair loop enabled, XSD validity reaches 100.0%, closing the single first-pass failure, and the mean number of XSD repair calls per scenario is negligible (below 0.01), so the residual is closed at essentially no cost. The use-case study corroborates this from the user side: every invoice generated with the tool was read into Sumex successfully, and

Table 22: Recurring Themes from the Free-Text Feedback

Theme	n	Representative comment
<i>Strengths</i>		
Fast, schema-valid starting template	9	“Quick generation of schema-valid XML.”
Correct tariff-code selection	6	“It found matching tariff positions for me.”
Simple natural-language input	4	“Typing your idea and just generating an invoice from it.”
Clear validity explanations	4	“The explanations of an invalid invoice were very good.”
<i>Weaknesses</i>		
Prices / tax points not from tariff	6	“The tax points used were not the real ones from the tariff.”
Hardcoded provider / insurer data	6	“Provider information appeared to be hardcoded rather than generated.”
Explanation not matching the XML	4	“There is a disconnect between the explanation and what is written into the XML.”
Specific rule or limit not respected	3	“A daily limit was exceeded: allowed once per day, generated with quantity ten.”
Limited to TARDOC	3	“Support for tariffs beyond TARDOC, especially drugs.”
<i>Requested improvements</i>		
User-fixable field values	5	“Allow fixed values, e.g. the insurer EAN and the patient’s insurance number.”
Batch / multi-invoice generation	3	“Generate one-to-many invoices via an API for parameterized tests.”
Iterative refinement + XML preview	3	“An XML preview to adjust and regenerate directly in the tool.”
<i>Best-fit situations</i>		
Edge cases / complex combinations	4	“In very complex invoices where edge cases need to be tested.”
Automated / CI test pipelines	3	“In automated parameterized tests that call the tool via an API.”
No template or weak tariff knowledge	3	“When you don’t know the tariff rules well or lack an example invoice.”

structural correctness was the highest-rated tool attribute (4.6 of 5). RQ1 is therefore answered affirmatively: with deterministic transformation in front of a validating loop, an LLM can be guided to produce XSD-compliant invoices consistently.

RQ2 – Targeted rule violations. The system can be steered to the intended validity in every case (scenario correctness 100%), and the generated negative invoices remain schema-valid while violating their target rule. This confirms that structural validity and rule violation are independent, the property that makes them usable as targeted negative tests. Realizing the *specific* violation, however, succeeds only partially: 64.3% overall, ranging from reliable (amount mismatch, duplicate code, invalid billing role, side duplication) to not yet achieved (missing base service, treatment mismatch). The use-case study echoes this limitation: its deliberately faulty scenario (S3) is an illegal service combination, precisely the relational kind of violation that is hardest to inject, and one domain expert obtained an inclusion violation where an exclusion was intended. RQ2 is thus answered with a qualification: targeted negative generation works well for violations that can be expressed as local edits to the invoice, and less well for those requiring more global or relational manipulation.

RQ3 – Knowledge storage and retrieval. The hybrid design delivers well-grounded generation: code-field faithfulness is 96.9%, retrieval precision 83.3% and recall 63.7%, and 76.2% of emitted codes trace to the structured tariff channel against only 20.7% to the model’s parametric memory. This supports the architectural choice of combining structured tariff lookup with vector retrieval: the structured channel carries most of the grounding, while the residual memory share is exactly what query expansion targets. The use-case study corroborates this from the user side: the most frequently praised strength of the tool was that it selected the correct tariff positions, the same behaviour the faithfulness and provenance metrics capture. Retrieval quality degrades as prompts become less explicit, which localizes where improvement is needed. The model sweep (Section 6.4.6) substantiates this: retrieval quality is nearly model-invariant (structured recall@20 within a 5.2 pp band across eight models), and the best-grounded model, GPT-5.1, relies least on parametric memory (8.6%), confirming that grounding is an architectural property rather than a model artifact. The embedding sweep (Section 6.4.7) further shows the dense channel is saturated at 1536 dimensions. This grounding is specific to the tariff codes, however: the surrounding business fields such as prices, tax points, and partner identifiers are not retrieved from authoritative data, which is why participants flagged them for manual correction (Section 6.5.7).

RQ4 – Practical usefulness. The quantitative evidence shows broad, consistent coverage across prompt tiers and both languages, which speaks to the prototype’s applicability. The practical, human-centred part of RQ4 is addressed by the use-case study (Section 6.5), and its verdict is positive with one clear reservation. Objectively, the tool improved task completion: every invoice generated with it was finished within the ten-minute budget and imported into Sumex (42 of 42), against 69% in the manual condition, where four participants ran out of time and one produced a malformed XML. Subjectively, usability

was rated excellent (SUS 87.0, well above the 68 benchmark), and on the direct comparison the tool was rated clearly better than manual work for reducing effort, completing more scenarios within the budget, hitting the target scenario, and keeping invoices XSD-valid (all near 4.3 of 5). Adoption intent was correspondingly positive: eight of the fourteen participants would use the tool in their daily workflow and six were tentatively positive, with none declining.

The reservation concerns confidence rather than capability. The one comparative item that did not improve over manual work was confidence in the resulting test cases (3.0, exactly neutral), and trust in the invoices for real use remained only neutral (3.1). Participants traced this to the non-tariff business content (prices, tax points, and partner identifiers) that the prototype does not yet ground and that therefore still requires manual correction (Section 6.5.7). As Section 6.5.6 explains, this reservation reads as a floor rather than a ceiling: it falls on value fields the prototype does not yet populate (in the header and within the service lines), which are outside the current scope, and where it touches the grounded content it coincides with the recall and violation-fidelity residual already reported. It is, moreover, a first-impression rating formed before participants had learned the tool's reliable and unreliable areas, which calibrated use would be expected to raise. The two strands therefore agree on where the tool is not yet sufficient. RQ4 is thus answered affirmatively but with a scope: the prototype clearly improves the efficiency and usability of test-invoice construction, while full trust for unreviewed use awaits grounding of the business fields beyond the tariff codes. Two caveats bound this reading: the sample is small and drawn from a single organization, and because the manual condition always preceded the tool condition on the same scenarios, part of the completion advantage reflects the familiarity built up manually (Section 6.7).

Triangulation. The two strands largely reinforce each other, and where they diverge the divergence is itself informative. They agree most clearly on grounding and structure: the quantitative faithfulness (96.9%) and the objective import result (100%) are matched by the analysts' two most-praised strengths, correct tariff-code selection and fast, schema-valid output. They also agree on the limits of targeted negative generation: the violation-injection rate (64.3%) is lowest for relational violations, and the study's hardest scenario (S3, an illegal service combination) is exactly such a case, where one participant obtained an inclusion violation instead of the intended exclusion. Agreement across an automated metric and independent human judgement strengthens both conclusions.

The strands diverge on what they can see. The analysts' dominant reservation, that prices, tax points, and partner identifiers are placeholder values needing correction, concerns fields the quantitative metrics do not measure, since those metrics target tariff codes and schema validity. The study therefore surfaces a practical bottleneck that the quantitative evaluation was structurally blind to, and locates the clearest next step: extending grounding from the tariff codes to the surrounding business fields. Conversely, the high objective validity (100% import) coexists with only neutral trust (3.1), marking the gap between an invoice that Sumex accepts and one an analyst would rely on unreviewed. Read together, the strands agree that the prototype is structurally sound and well-grounded in tariff data, and agree on where it is not yet sufficient: the content it does not retrieve.

6.6.1 Determinism versus Probabilistic Repair

A recurring theme across RQ1 and RQ2 is the division of labour between deterministic and probabilistic components: structural correctness is carried almost entirely by deterministic logic, and the probabilistic repair loop contributes only at the margin. This argues for keeping correctness-critical steps outside the model wherever a deterministic formulation exists, and using the model where flexibility is genuinely required (scenario interpretation, content generation, and repair of the residual). The use-case study extends the same lesson from measured correctness to user trust: practitioners relied most on the parts that are deterministic or grounded, which is the practical lesson the architecture embodies and the evaluation bears out.

6.6.2 Strengths and Limitations

The approach has three core strengths. Generation is grounded in authoritative tariff data; the transformation and validation path is deterministic and auditable; and the evaluation rests on *production* ground-truth labels and a frozen, reproducible dataset. The two-strand design adds a fourth: the pipeline is measured against a labeled golden dataset, while the use-case study tests the prototype against real manual practice rather than in isolation. In that study the tool raised task completion from 69% to 100% and was rated highly usable (SUS 87.0).

Several limitations temper these conclusions. The golden dataset is small, at 30 records and 146 variants, so the estimates carry non-trivial uncertainty; this is why every figure is reported with a 95% bootstrap confidence interval (Table 18). The headline numbers come from a single generation model, though the model sweep (Section 6.4.6) shows the core results hold across eight. Three quality gaps also remain: some violation types are not yet reliably injected, retrieval recall is only moderate on under-specified prompts, and end-to-end latency is high (p95 of 109s), driven by the LLM calls.

Two further limitations come from the use-case study. First, grounding currently covers the tariff codes but not the surrounding business fields, so prices, tax points, and partner identifiers still require manual correction. Second, the study is indicative rather than conclusive: the sample is small and from a single organization, the completion and import outcomes are self-reported, and the fixed manual-then-tool order introduces a learning effect (Section 6.7). Together these points define the scope within which the conclusions hold and motivate the future work discussed in Chapter 7.

6.7 Threats to Validity

This section discusses the main threats to the validity of the evaluation and the steps taken to limit them, following the usual distinction between internal, external, construct, and reliability threats.

Internal validity. The validity *labels* of the dataset are a particular strength: they are taken from the production validation engine rather than assigned heuristically, so the ground truth for syntactic and contract validity is trustworthy. The check on *violation injection* is weaker, however. It relies on deterministic draft-side oracles that exist only for the structural violation types; violations without such an oracle (cumulation violations and corpus-derived contract violations) are excluded from the measurement rather than scored, so the reported 64.3% characterizes a subset of violation types, not all of them. A further internal threat is that the golden dataset is curated by a single pipeline from one corpus, which could introduce selection effects; the provenance funnel (Section 6.2) makes the selection explicit so that any such bias is visible. Finally, the paired runs behind the first-pass-versus-post-repair comparison share the same configuration and dataset; the small residual differences in their first-pass drafts reflect sampling variance, which the ablation attributes to noise rather than to the repair loop. A distinct internal threat applies to the use-case study: because its two conditions were not counterbalanced but always run manual-first on the same scenarios, part of the observed completion advantage may be a learning effect rather than a pure tool effect. This is acknowledged rather than controlled for, so the objective outcome is read alongside the subjective ratings rather than on its own.

External validity. The results are obtained with a single generation model (Kimi-K2.5) on a single tariff (TARDOC 007) and a limited guideline corpus, so they may not transfer directly to other models, tariffs, or document collections. The model sweep (Section 6.4.6) directly addresses the model dependence, since the core metrics hold across eight models from five families, but the tariff and corpus scope remain a limitation of the MVP. The source invoices come from a test environment rather than live production traffic, and the qualitative study involves fourteen participants from a single organization (only one with prior exposure to the prototype), so the small, single-site sample limits how far the findings generalize to the wider analyst population. Finally, the two evaluation strands use different generation models: Kimi-K2.5 for the quantitative baseline and GPT-5.1 for the use-case study, a deliberate deployment choice justified in Section 6.5. Because the model sweep (Section 6.4.6) shows the core metrics are near-invariant across models, this does not materially affect the comparability of the two strands, but it is noted for transparency.

Construct validity. Several metrics are proxies for the constructs they stand in for. Context relevance is approximated by the precision of the codes the model uses, code presence is checked by substring matching, and grounding (faithfulness) is approximated by whether an emitted code appears in the retrieved knowledge. These are reasonable operationalizations but not direct measurements of relevance or grounding. Likewise, the $\text{recall}@K$ probe queries the document channel with the raw prompt and is therefore a deliberate lower bound rather than a measure of the production retriever, and, in the use-case study, the completion and import outcomes are self-reported by participants rather than logged automatically, while perceived usefulness is captured by a standardized but inherently subjective questionnaire. The metrics should be read as consistent indicators rather than exact quantities.

Reliability. Because the language model is probabilistic, repeated runs can differ. This is mitigated in three ways: a low sampling temperature reduces output variation, response caching makes runs over identical inputs reproducible, and point estimates are reported with bootstrap confidence intervals so that the remaining uncertainty is quantified rather than hidden (the 95% intervals in Table 18). The small size of the dataset means these intervals are non-trivial, and results close together should not be over-interpreted. Reproducibility at the configuration level is supported by the per-run snapshots (configuration, model identifiers, and git commit) described in Section 6.2, so any reported number can be traced to the exact setup that produced it.

Chapter 7

Summary and Conclusions

This thesis set out to reduce the manual effort of producing test invoices for healthcare-insurance validation software. Today analysts hand-craft XML invoices that must be schema-valid and, for negative testing, deliberately violate a specific rule while remaining otherwise realistic. That work is slow, demands detailed tariff knowledge, and leaves edge cases under-tested. The central question was whether an LLM, given access to authoritative domain knowledge and wrapped in deterministic validation, can generate such invoices from a plain natural-language description. This closing chapter summarizes the work, states what the evaluation established for each research question, revisits the contributions, and outlines the limitations and future directions they motivate.

7.1 Summary of the Work

After reviewing the state of the art in LLM-based structured generation, RAG, and knowledge representation in regulated domains (Chapters 2 and 3), the thesis designed a validation-aware architecture (Chapter 4) and implemented it as a working prototype (Chapter 5). The prototype takes an analyst’s natural-language description of an invoice scenario and returns a generated *generalInvoiceRequest* invoice. It combines hybrid retrieval over a structured tariff catalogue and an unstructured guideline corpus, LLM generation into a typed intermediate representation, deterministic transformation (date normalization and a JAXB mapping onto the Sumex envelope), and a bounded generate-validate-repair loop against the XSD schema.

The prototype was evaluated along two complementary strands (Chapter 6). The quantitative strand ran the full pipeline over a frozen golden dataset of 30 records and 146 prompt variants, reverse-engineered from real production invoices whose validity labels come from the production validation engine itself. The qualitative strand was a within-subjects use-case study with fourteen practitioners, who each built the same three test invoices first manually and then with the tool, so that the prototype could be judged against current practice rather than in isolation.

7.2 Answers to the Research Questions

RQ1 (XSD-compliant generation). The system produces schema-valid invoices reliably: 99.3% on the first pass and 100% after a single repair. The decisive factor is not the language model but the deterministic post-processing in front of the schema, with the bounded repair loop closing only a negligible residual. An LLM can therefore be guided to consistent XSD compliance when correctness-critical steps are handled deterministically.

RQ2 (targeted rule violations). The system reaches the intended validity in every case, and its negative invoices stay schema-valid while carrying their target defect, which is what makes them usable as negative tests. The system reliably injects violations expressible as local edits (e.g. amount mismatch, duplicate code); relational violations (missing base service, treatment mismatch) are not yet achieved. This gives 64.3% over the oracle-adjudicable types, a characterized boundary rather than an undifferentiated miss.

RQ3 (knowledge storage and retrieval). The hybrid design delivers well-grounded generation, with code-field faithfulness of 96.9% and roughly three-quarters of emitted codes traceable to the structured tariff channel. This grounding is an architectural property rather than a model artefact: it holds across eight generation models and does not improve with a larger embedding, which indicates the dense channel is already saturated for this tariff. The grounding covers the tariff codes but not the surrounding business fields, which marks the boundary of the current design.

RQ4 (practical usefulness). The use-case study answers RQ4 affirmatively, with one clear reservation. The tool raised task completion from 69% to 100% and was rated highly usable (SUS 87.0), and analysts judged it clearly better than manual work for effort, speed, targeting, and schema validity. The reservation concerns confidence and trust: because prices, tax points, and partner identifiers are not yet grounded, analysts still review and correct the output before relying on it. The prototype thus improves the efficiency and usability of test-invoice construction. Trust is neutral primarily because the business fields beyond the tariff codes are not yet grounded, and grounding them is the main lever to raise it.

7.3 Contributions

The thesis delivered the four contributions set out in Section 1.3. It presented a validation-aware architecture that combines retrieval over heterogeneous knowledge with deterministic checking (C1); a generation strategy that produces both compliant invoices and targeted rule violations while preserving XSD validity (C2); a prototype that emits invoices suitable for automated ingestion and validation (C3); and an evaluation methodology built on production ground-truth labels and a frozen, reproducible dataset, extended with a human-centred use-case study (C4). A recurring lesson cuts across all four: correctness is carried

by the deterministic and grounded parts of the pipeline, and the same division of labour predicts where practitioners place their trust. The generation model, by contrast, behaves as a quality-latency lever rather than a correctness prerequisite.

7.4 Limitations

The conclusions hold within a defined scope. The limitations below are ordered from the most consequential to the more contained; each is treated in full in the threats to validity (Section 6.7), and the future-work agenda (Section 7.5) refers back to them by label.

- L1 – Grounding is limited to the tariff codes.** The system grounds the tariff-code selection in authoritative data, but the surrounding business fields – prices and tax points, the provider and insurer identifiers (GLN, ZSR, Mandant), and several per-line values – are filled from static defaults rather than derived dynamically. These fields therefore still require manual correction, and they are the reason analyst trust in the output stayed neutral rather than positive. This is the single most consequential limitation: the tool is trustworthy exactly where it is grounded, and not yet beyond that.
- L2 – Targeted rule violations are only partially realized, and only partially measurable.** The intended violation is injected in 64.3% of the adjudicable cases, reliably for violations expressible as local edits (for example amount mismatch or duplicate code) but not yet for relational ones (missing base service, treatment mismatch). Moreover, that figure covers only the violation types with a deterministic draft-side oracle; cumulation and corpus-derived contract violations have no such oracle and are excluded from the measurement rather than scored, so negative-test quality is characterized on a subset of types.
- L3 – Retrieval recall is moderate on under-specified prompts.** Code recall is 63.7% overall and degrades as prompts become less explicit. The hybrid channel demonstrably improves the candidate pool more than the language model then consumes from it, so part of the shortfall is a selection gap rather than a pure retrieval gap.
- L4 – End-to-end latency is high.** The median scenario takes tens of seconds and the tail exceeds one hundred ($p_{95} \approx 109$ s), driven almost entirely by the LLM calls. The model sweep shows this is a quality-latency lever rather than a fixed cost, but the default configuration is slow for interactive use.
- L5 – Small dataset, single tariff, single headline model.** The golden dataset is small (30 records, 146 prompt variants), so estimates carry non-trivial uncertainty, which is why every figure is reported with a 95% bootstrap confidence interval. It is curated by one pipeline from one corpus, on a single tariff (TARDOC 007) drawn from a test environment, and the headline numbers use a single generation model – although the eight-model sweep indicates the core results generalize.

L6 – The use-case study is indicative, not conclusive. The sample is small (fourteen practitioners) and from a single organization, the completion and import outcomes are self-reported rather than logged, and the fixed manual-then-tool order on the same scenarios introduces a learning effect that inflates part of the measured completion advantage.

7.5 Future Work

The evaluation and the participant feedback point to a coherent agenda, ordered roughly by expected impact. Each item names the limitation (Section 7.4) it primarily addresses.

- **Ground the business fields (*addresses L1*).** Deriving prices and tax points from the tariff, and providing correct or user-fixable partner and insurer identifiers (GLN, ZSR, Mandant), is the single change most likely to turn the analysts' neutral confidence into everyday trust.
- **Broaden tariff coverage (*addresses L5*).** Support tariffs beyond TARDOC, in particular drugs and outpatient flat rates, which frequently appear alongside TARDOC positions in practice.
- **Batch generation and integration.** Generate several invoices per request and expose an API, so the tool can feed parameterized tests in a continuous-integration pipeline.
- **Iterative refinement (*eases L1*).** Add an in-tool XML preview and a follow-up-prompt mode, so analysts can adjust and regenerate without starting over, cutting the manual correction the ungrounded fields still require.
- **Close the retrieval-to-selection gap (*addresses L3*).** The hybrid channel improves the candidate pool more than the model consumes; tuning the vector-channel relevance threshold, the one lever left un-ablated, is the clearest remaining retrieval lever.
- **Extend the violation oracles (*addresses L2*).** Draft-side oracles for cumulation and corpus-derived contract violations would let negative-test quality be measured across all violation types rather than a structural subset.
- **A larger user study (*addresses L6*).** A multi-site, counterbalanced study with logged task times would put the practical findings on firmer statistical ground.

7.6 Concluding Remarks

The prototype shows what the approach can do. Grounded in authoritative tariff data and wrapped in deterministic validation, LLM-assisted generation turns a slow, knowledge-intensive manual task into a fast, natural-language one. Its output is trustworthy exactly where it is grounded. This result is carried by the architecture rather than by any single model, which is what gives the approach a path beyond the prototype. As grounding extends from the tariff codes to the full invoice, the same design should support more trustworthy and realistic test-data generation across the healthcare-insurance domain.

Bibliography

- [1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation, 2008. [Online]. Available: <https://www.w3.org/TR/xml/>.
- [2] D. C. Fallside and P. Walmsley, *XML Schema Part 0: Primer Second Edition*. W3C Recommendation, 2004. [Online]. Available: <https://www.w3.org/TR/xmlschema-0/>.
- [3] M. Fewster and D. Graham, *Software Test Automation: Effective Use of Test Execution Tools*. Boston, MA: Addison-Wesley, 1999.
- [4] L. Baresi and M. Pezze, “An introduction to software testing”, *Electronic Notes in Theoretical Computer Science*, vol. 148, no. 1, pp. 89–111, 2006.
- [5] A. Causevic, R. Shukla, S. Punnekkat, and D. Sundmark, “Effects of negative testing on tdd: An industrial experiment”, in *International conference on agile software development*, Springer, 2013, pp. 91–105.
- [6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, et al., “Language models are few-shot learners”, *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020.
- [7] L. Long, R. Wang, R. Xiao, J. Zhao, X. Ding, et al., “On llms-driven synthetic data generation, curation, and evaluation”, *Findings of the Association for Computational Linguistics*, 2024.
- [8] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, et al., “Retrieval-augmented generation for knowledge-intensive nlp tasks”, *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.
- [9] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, et al., “Retrieval-augmented generation for large language models: A survey”, *arXiv preprint arXiv:2312.10997*, 2023.
- [10] C. Feng, Z. Liu, S. Gupta, G. Cui, J. von der Assen, and B. Stiller, “Smarter, not bigger: Fine-tuned rag-enhanced llms for automotive hil testing”, *arXiv preprint arXiv:2511.22584*, 2025.
- [11] A. Mallen, A. Asai, V. Zhong, R. Das, D. Khashabi, and H. Hajishirzi, “When not to trust language models: Investigating effectiveness of parametric and non-parametric memories”, in *Proceedings of ACL*, 2023.

- [12] O. Ayala and P. Bechard, “Reducing hallucination in structured outputs via retrieval-augmented generation”, in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, 2024.
- [13] S. J. Duckett and T. J. Jackson, *Healthcare Finance: An Introduction to Accounting and Financial Management*. Amsterdam: Elsevier, 2011.
- [14] S. Abiteboul, P. Buneman, and D. Suciu, *Data on the Web: From Relations to Semistructured Data and XML*. San Francisco, CA: Morgan Kaufmann, 1997.
- [15] Y. Lu et al., “Machine learning for synthetic data generation: A review”, *arXiv preprint arXiv:2302.04062*, 2023.
- [16] A. Vaswani, N. Shazeer, N. Parmar, et al., “Attention is all you need”, *Advances in Neural Information Processing Systems*, vol. 30, 2017.
- [17] L. Ouyang et al., “Training language models to follow instructions with human feedback”, *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.
- [18] S. Arora et al., “Language models enable simple systems for generating structured views of heterogeneous data lakes”, *arXiv preprint arXiv:2304.09433*, 2023.
- [19] M. Besta et al., “Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries”, *ACM Computing Surveys*, vol. 56, no. 2, pp. 1–40, 2023.
- [20] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008, vol. 39.
- [21] V. Karpukhin et al., “Dense passage retrieval for open-domain question answering”, in *Proceedings of EMNLP*, 2020.
- [22] K. Guu, K. Lee, Z. Tung, P. Pasupat, and M.-W. Chang, “Realm: Retrieval-augmented language model pre-training”, in *Proceedings of ICML*, 2020.
- [23] G. Izacard et al., “Atlas: Few-shot learning with retrieval augmented language models”, *Journal of Machine Learning Research*, vol. 24, 2023.
- [24] G. Izacard and E. Grave, “Leveraging passage retrieval with generative models for open domain question answering”, in *Proceedings of EACL*, 2021.
- [25] M. De Jong et al., “Fido: Fusion-in-decoder optimized for stronger performance and faster inference”, in *Findings of the Association for Computational Linguistics: ACL 2023*, 2023.
- [26] J. Howard and S. Ruder, “Universal language model fine-tuning for text classification”, in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2018, pp. 328–339.
- [27] S. Gururangan et al., “Don’t stop pretraining: Adapt language models to domains and tasks”, in *Proceedings of the 58th annual meeting of the association for computational linguistics*, 2020, pp. 8342–8360.
- [28] J. Kirkpatrick et al., “Overcoming catastrophic forgetting in neural networks”, *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

- [29] S. Geng et al., “Generating structured outputs from language models: Benchmark and studies”, *arXiv preprint arXiv:2501.10868*, 2025.
- [30] E. J. Hu et al., “Lora: Low-rank adaptation of large language models”, in *International Conference on Learning Representations (ICLR)*, 2022.
- [31] J. He, C. Zhou, X. Ma, T. Berg-Kirkpatrick, and G. Neubig, “Towards a unified view of parameter-efficient transfer learning”, *arXiv preprint arXiv:2110.04366*, 2021.
- [32] N. Houlsby et al., “Parameter-efficient transfer learning for nlp”, in *International conference on machine learning*, PMLR, 2019, pp. 2790–2799.
- [33] X. L. Li and P. Liang, “Prefix-tuning: Optimizing continuous prompts for generation”, in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2021, pp. 4582–4597.
- [34] B. Lester, R. Al-Rfou, and N. Constant, “The power of scale for parameter-efficient prompt tuning”, in *Proceedings of the 2021 conference on empirical methods in natural language processing*, 2021, pp. 3045–3059.
- [35] 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.
- [36] J. Wei et al., “Chain-of-thought prompting elicits reasoning in large language models”, *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.
- [37] T. Schick et al., “Toolformer: Language models can teach themselves to use tools”, *Advances in neural information processing systems*, vol. 36, pp. 68 539–68 551, 2023.
- [38] Z. Zhao, E. Wallace, S. Feng, D. Klein, and S. Singh, “Calibrate before use: Improving few-shot performance of language models”, in *International conference on machine learning*, Pmlr, 2021, pp. 12 697–12 706.
- [39] 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*, 2021, pp. 9895–9901.
- [40] L. Pan, M. Saxon, W. Xu, D. Nathani, X. Wang, and W. Y. Wang, “Automatically correcting large language models: Surveying the landscape of diverse automated correction strategies”, *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 484–506, 2024.
- [41] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Elsevier, 2010.
- [42] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing”, *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.
- [43] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, “The curious case of neural text degeneration”, *arXiv preprint arXiv:1904.09751*, 2019.
- [44] L. Beurer-Kellner, M. Fischer, and M. Vechev, “Guiding llms the right way: Fast, non-invasive constrained generation”, *arXiv preprint arXiv:2403.06988*, 2024.

- [45] F. Raspanti, T. Ozcelebi, and M. Holenderski, “Grammar-constrained decoding makes large language models better logical parsers”, in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, 2025, pp. 485–499.
- [46] J. J. Pan, J. Wang, and G. Li, “Survey of vector database management systems”, *The VLDB Journal*, vol. 33, no. 5, pp. 1591–1615, 2024.
- [47] B. Peng et al., “Graph retrieval-augmented generation: A survey”, *ACM Transactions on Information Systems*, vol. 44, no. 2, pp. 1–52, 2025.
- [48] Y. Hu, Z. Lei, Z. Zhang, B. Pan, C. Ling, and L. Zhao, “Grag: Graph retrieval-augmented generation”, in *Findings of the Association for Computational Linguistics: NAACL 2025*, 2025, pp. 4145–4157.
- [49] D. L. Parnas, “On the criteria to be used in decomposing systems into modules”, *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.
- [50] S. Robertson and H. Zaragoza, *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc, 2009, vol. 4.
- [51] G. V. Cormack, C. L. Clarke, and S. Buettcher, “Reciprocal rank fusion outperforms condorcet and individual rank learning methods”, in *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, 2009, pp. 758–759.
- [52] Y. A. Malkov and D. A. Yashunin, “Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs”, *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 4, pp. 824–836, 2018.
- [53] S. Filice, G. Horowitz, D. Carmel, Z. Karnin, L. Lewin-Eytan, and Y. Maarek, “Generating Q&A benchmarks for RAG evaluation in enterprise settings”, in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, Vienna, Austria: Association for Computational Linguistics, Jul. 2025, pp. 469–484. DOI: 10.18653/v1/2025.acl-industry.33. [Online]. Available: <https://aclanthology.org/2025.acl-industry.33/>.
- [54] B. Efron and R. J. Tibshirani, *An introduction to the bootstrap*. Chapman and Hall/CRC, 1994.
- [55] J. Lazar, J. H. Feng, and H. Hochheiser, *Research methods in human-computer interaction*. Morgan Kaufmann, 2017.
- [56] J. D. Lee and K. A. See, “Trust in automation: Designing for appropriate reliance”, *Human factors*, vol. 46, no. 1, pp. 50–80, 2004.
- [57] J. Brooke et al., “Sus-a quick and dirty usability scale”, *Usability evaluation in industry*, vol. 189, no. 194, pp. 4–7, 1996.
- [58] J. Sauro, *A practical guide to the system usability scale: Background, benchmarks & best practices*. Measuring Usability LLC, 2011.
- [59] A. Bangor, P. Kortum, and J. Miller, “Determining what individual sus scores mean: Adding an adjective rating scale”, *Journal of usability studies*, vol. 4, no. 3, pp. 114–123, 2009.

-
- [60] V. Braun and V. Clarke, “Using thematic analysis in psychology”, *Qualitative research in psychology*, vol. 3, no. 2, pp. 77–101, 2006.

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
BM25	Best Matching 25 (lexical ranking function)
cf.	confer (compare)
CI	Confidence Interval
CRM	Customer Relationship Management
DRG	Diagnosis-Related Group
e.g.	exempli gratia (for example)
et al.	et alii (and others)
FiD	Fusion-in-Decoder
GLN	Global Location Number
GPT	Generative Pre-trained Transformer
GTIN	Global Trade Item Number
HNSW	Hierarchical Navigable Small World
ICL	In-Context Learning
i.e.	id est (that is)
IT	Information Technology
JAXB	Jakarta XML Binding (Java Architecture for XML Binding)
JAXP	Java API for XML Processing
JSON	JavaScript Object Notation
LLM	Large Language Model
LoRA	Low-Rank Adaptation
MDC	Mapped Diagnostic Context

MiGeL	Mittel- und Gegenständeliste (Swiss list of medical aids and appliances)
MVP	Minimum Viable Product
NL	Natural Language
OCR	Optical Character Recognition
PDF	Portable Document Format
PEFT	Parameter-Efficient Fine-Tuning
RAG	Retrieval-Augmented Generation
RDF	Resource Description Framework
REALM	Retrieval-Augmented Language Model Pre-Training
REST	Representational State Transfer
RRF	Reciprocal Rank Fusion
RWC	Rejection code (Sumex validation code type)
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
SSE	Server-Sent Events
Sumex	Sumex invoice-validation suite (product name)
SUS	System Usability Scale
TARDOC	Swiss outpatient medical tariff (tariff type 007)
TARMED	Swiss outpatient medical tariff (predecessor of TARDOC)
UI	User Interface
VLC	Contract-violation code (Sumex validation code type)
XML	Extensible Markup Language
XSD	XML Schema Definition
ZSR	Zahlstellenregister number (Swiss provider registration number)

Glossary

Base service In TARDOC, a principal service that must be present for certain add-on (additional) services to be billable. An invoice that bills an add-on without its required base service triggers the “missing base service” violation.

Chunking Splitting a document into overlapping, paragraph-sized passages so that each can be embedded and retrieved independently.

Contract violation An invoice that breaks a contractual or tariff rule while remaining structurally valid. The production engine records a contract-violation (VLC) code but no rejection (RWC) code, and the invoice is labeled *invalid* (contract-violating).

Cumulation rule A tariff rule that restricts which services may be billed together, for example an exclusion between two service codes that must not appear on the same invoice. Such rules underlie the “illegal service combination” negative test.

DRG Diagnosis-Related Group: a case-based classification that groups inpatient stays into flat-rate reimbursement categories; one of the reference catalogues checked by the formal validation layer.

Embedding A dense numerical vector representing a piece of text, so that semantically related texts lie close together and can be found by similarity search.

Faithfulness The fraction of tariff codes in a generated invoice that are supported by the retrieved knowledge rather than produced from the model’s internal memory; used as a proxy for grounding.

Fine-tuning Adapting a pretrained language model by further updating its parameters on domain-specific data. This thesis deliberately avoids fine-tuning in favour of retrieval, so that domain knowledge can be updated without retraining.

generalInvoiceRequest The Swiss XML invoice format (envelope) that generated invoices conform to and that is validated against its XSD schema.

Generate-validate-repair loop The bounded cycle in which a generated invoice that fails XSD validation is regenerated, with the schema errors fed back to the model, until it is schema-valid or the attempt budget is exhausted.

GLN Global Location Number: a globally unique identifier for a healthcare provider or organization, used as a partner identifier on the invoice.

- Golden dataset** The frozen, human-validated benchmark of records and tiered prompt variants, reverse-engineered from real labeled production invoices, against which the quantitative evaluation is run.
- Grounding** Basing generated content on retrieved authoritative knowledge rather than on the model’s parametric memory. In this thesis the tool is considered trustworthy exactly where its output is grounded.
- Ground truth** The reference validity label of an invoice, taken directly from the production Sumex validation engine rather than assigned heuristically.
- GTIN** Global Trade Item Number: a product identifier for billable goods such as pharmaceuticals (e.g. GTIN-13 pharmacy codes), distinct from TARDOC service codes.
- Hallucination** Model output that is fluent and plausible but not supported by any underlying evidence or source.
- Hybrid retrieval** Combining lexical retrieval (BM25) with dense embedding-based retrieval, fusing the two candidate lists via reciprocal rank fusion.
- In-context learning** Specifying a task to a language model through instructions and examples in the prompt, without updating the model’s parameters.
- Mandant** (German: client/tenant) An insurer or organization identifier carried on the invoice; one of the partner fields the prototype does not yet ground and that analysts correct by hand.
- Marshalling** Serializing an in-memory object tree into an XML document; here performed by JAXB when producing the final invoice.
- MiGeL** *Mittel- und Gegenständeliste*: the Swiss list of reimbursable medical aids and appliances, one of the reference catalogues used during validation.
- Negative test** A deliberately invalid input that violates one specific rule while remaining otherwise well-formed, used to verify that a validation system detects and reports the error.
- Prompt engineering** Designing the input given to a language model (instructions, examples, output schema) to steer it toward a desired output.
- Query expansion** Translating a clinical or lay query into a handful of German TARDOC keywords before the structured lookup, so that lexical retrieval has matching vocabulary.
- Rejection** An invoice for which the production engine records a rejection (RWC) code. It is the most severe validity label and takes priority over a contract violation: an invoice carrying any RWC code is labeled *rejected* regardless of its VLC codes.
- Retrieval-Augmented Generation** An approach that conditions a language model’s output on external knowledge retrieved at inference time, rather than on the model’s parameters alone.

- Sampling temperature** A decoding parameter controlling the randomness of generated text. A low value (0.2) is used here to make targeted negative generation controllable.
- Schema validation** The deterministic check that an XML document conforms to its XSD schema, covering required elements, data types, and ordering.
- Sumex** The invoice-checking suite whose validation the generated invoices are ultimately meant to test; its core orchestrates formal, expert-system, and analytics checks on submitted invoices.
- TARDOC** The Swiss outpatient medical tariff catalogue (tariff type 007) in which the prototype grounds the service codes of generated invoices.
- Tariff position** A single billable service entry in a tariff catalogue, identified by a tariff code (in TARDOC following the pattern LL.NN.NNNN).
- TARMED** The Swiss outpatient medical tariff that preceded TARDOC; still referenced as an authoritative catalogue in the validation landscape.
- Vector store** A database of embeddings supporting similarity search; realized here as PostgreSQL with the `pgvector` extension and an HNSW index over cosine distance.
- Within-subjects design** A study design in which each participant experiences every condition, here performing the same tasks once manually and once with the tool, serving as their own control.
- ZSR** *Zahlstellenregister* number: a Swiss provider registration number identifying an accredited biller; a partner identifier the prototype does not yet ground.

List of Figures

1	The REALM Architecture [22]	14
2	Overview of the RAG Architecture [8]	15
3	The Fusion-in-Decoder (FiD) Architecture [24]	15
4	Current High-Level Invoice Processing Workflow	30
5	MVP Architecture for LLM-Assisted Test Invoice Generation	34
6	MVP End-to-End Sequence Diagram	35
7	MVP Java Package Structure by Architectural Layer	41
8	Generation-Layer Data Flow	46
9	Document Ingestion Pipeline	47
10	Transformation and Validation Sequence with Bounded Repair Loop	48
11	From Productive Invoices to the Golden Dataset	54
12	First-Pass vs. Post-Repair XSD Validity	60
13	Diagnostic Recall@ K (Document-Probe Lower Bound)	62
14	Use-Case Study: Task and Measurement Flow	69
15	Scenario Completion and Sumex Import by Condition	72
16	Tool vs. Manual: Mean Agreement on the Comparative Items	74
17	SUS Scores Against the Reference Average and Adjective Bands	74
18	Prototype UI	105
19	Streamed Pipeline Progress	106
20	Generated Invoice	107

21	Self-Rated Familiarity Across Participants	110
22	Frequency of Recurring Feedback Themes	111

List of Tables

1	Comparison of LLM Integration Strategies	25
2	Approaches for Structured Invoice Test Generation	27
3	Main MVP Architecture Layers	36
4	Technology Stack of the MVP Implementation	42
5	Principal Runtime Parameters and Default Values	43
6	Retrieval and Ingestion Parameters	48
7	Cross-Field Date-Consistency Rules	49
8	REST Endpoints Exposed by the Service	51
9	Research Questions Mapped to Methods and Metrics	54
10	Data Provenance: From Screened Invoices to the Golden Dataset	55
11	Golden Dataset Composition	56
12	Pipeline Configuration for the Evaluation Run	57
13	Violation-Injection Success by Type	61
14	Ablation of the Pipeline’s Configurable Levers	63
15	Sensitivity to the Generation Model	64
16	Sensitivity to the Embedding Model	66
17	Model Token Usage on the Golden Dataset	67
18	Aggregate Metrics for the Baseline Configuration	68
19	Use-Case Study Design	69
20	Tool-Experience Ratings	73

21	Reserved Business-Content Ratings and Their Interpretation	73
22	Recurring Themes from the Free-Text Feedback	76
24	Evaluation Metrics by Dimension	109
25	Per-Slice Results	110
26	Participant Profile	111
27	Delivered Projects	113
28	Principal Environment Variables for the Evaluation Harness	116

Appendix A

Prototype Interface and Invoice-Draft Schema

A.1 User Interface of the Prototype

The prototype exposes the generation pipeline through a single-page web interface. Figures 18-20 show a complete run: the analyst enters a natural-language scenario and selects a generation model (Figure 18); the pipeline streams its progress step by step (Figure 19); and the generated invoice is presented with its scenario type, validity, total, natural-language explanation, the selected tariff positions, and the raw model output, together with an XML download (Figure 20).

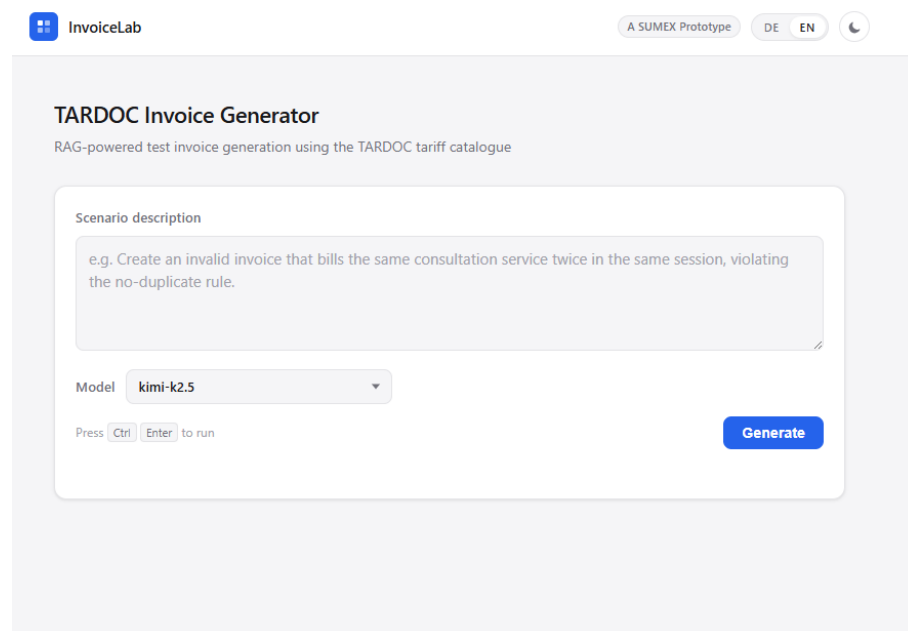


Figure 18: Prototype UI

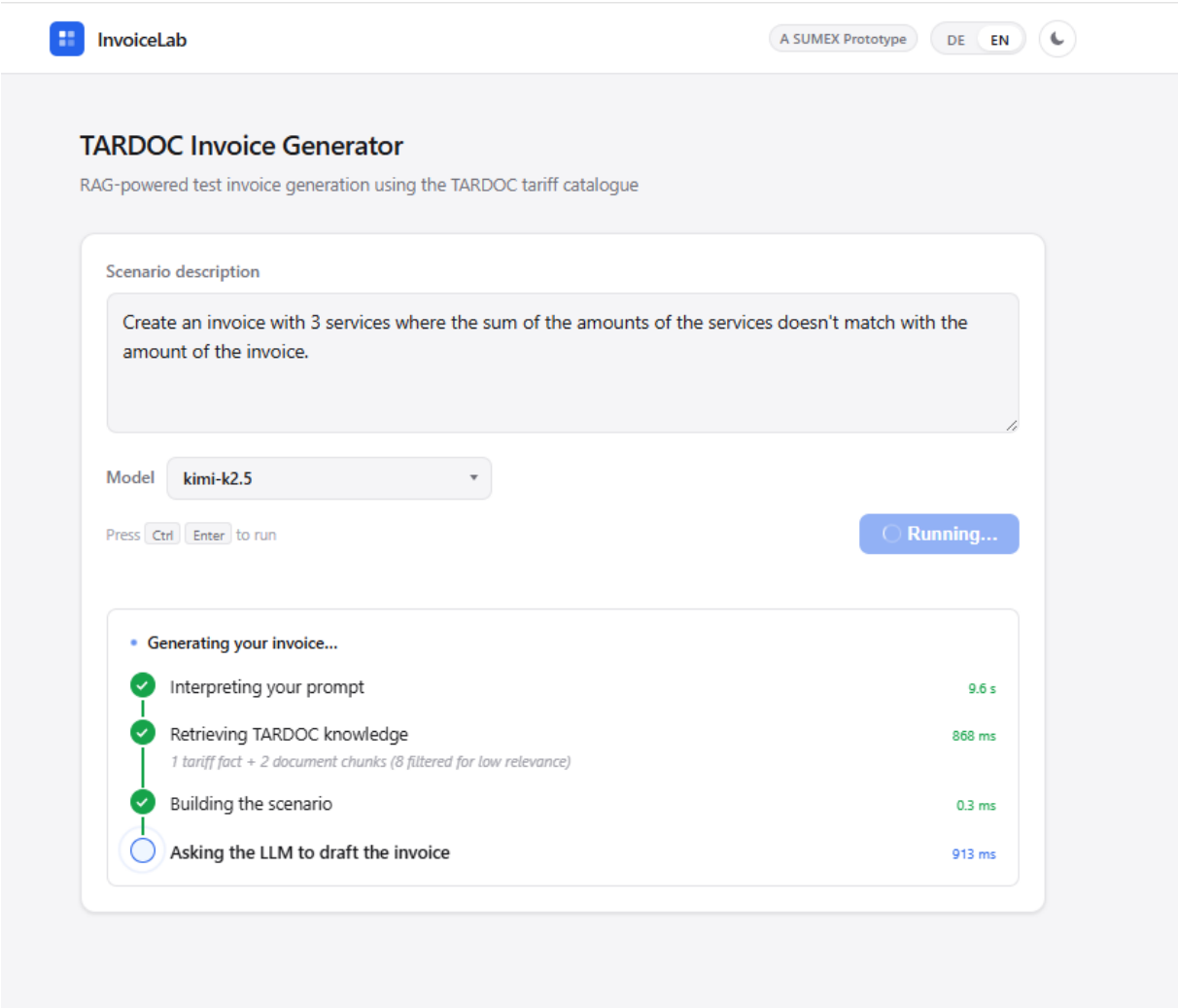


Figure 19: Streamed Pipeline Progress

RESULT

SCENARIO TYPE
RULE_VIOLATION_CASE

VALIDITY
Invalid

TOTAL (CHF)
CHF 250.00

MODEL
kimi-k2.5

EXPLANATION

This draft intentionally violates the invoice arithmetic consistency rule by setting declared_total_amount to 250.00 while the sum of the three service records equals 200.00 (100.00 + 80.00 + 20.00). This creates a header-to-line-item total mismatch. The service codes selected are AA.00.0010 and AA.00.0020 (ärztliche Konsultation segments) from the Rechnungsbeispiel table in the provided document evidence, and RC.00.0110 (Objektive Trübungsmessung) from the TARIFF evidence. All codes conform to the TARDOC LL.NN.NNNN structure and use tariff_type '007' as required for TARDOC services.

CODE	DESCRIPTION	QTY	UNIT PRICE	TOTAL
AA.00.0010	Ärztliche Konsultation, erste 5min.	1	CHF 100.00	CHF 100.00
AA.00.0020	Ärztliche Konsultation, weitere 5min.	1	CHF 80.00	CHF 80.00
RC.00.0110	Objektive Trübungsmessung der brechenden Medien, summarisch, beidseitig	1	CHF 20.00	CHF 20.00
GRAND TOTAL				CHF 250.00

▼ Raw LLM Output

```
```json
{
 "scenarioType": "RULE_VIOLATION_CASE",
 "intentionallyInvalid": true,
 "justification": "This draft intentionally violates the invoice arithmetic consistency rule by setting declared_total_amount to 250.00 while the sum of the three service records equals 200.00 (100.00 + 80.00 + 20.00). This creates a header-to-line-item total mismatch. The service codes selected are AA.00.0010 and AA.00.0020 (ärztliche Konsultation segments) from the Rechnungsbeispiel table in the provided document evidence, and RC.00.0110 (Objektive Trübungsmessung) from the TARIFF evidence. All codes conform to the TARDOC LL.NN.NNNN structure and use tariff_type '007' as required for TARDOC services.",
 "declared_total_amount": 250.00,
 "serviceRecords": [
 {
 "record_id": 1,
 "tariff_type": "007",
 "code": "AA.00.0010",
 "name": "Ärztliche Konsultation, erste 5min.",

```

Download XML

Figure 20: Generated Invoice

## A.2 Full Invoice-Draft Schema

Listing A.1 gives the complete invoice-draft schema abbreviated in Listing 5.2; *optional* fields take the defaults shown when omitted.

```
{
 "scenarioType": "RULE_VIOLATION_CASE", // required
 "intentionallyInvalid": true, // required
 "patientReference": "...", // optional
 "providerReference": "...", // optional
 "justification": "...", // optional
 "declared_total_amount": 500.00, // optional; may differ from line-item sum
 "invoiceContext": { // optional block
 "patient": { "gender": "...", "birthdate": "YYYY-MM-DD", "ssn": "..." },
 "treatment": { "dateBegin": "YYYY-MM-DD", "dateEnd": "YYYY-MM-DD", "canton": "ZH",
 "type": "ambulatory", "reason": "disease" },
 "law": { "type": "KVG", "insuredId": "...", "caseDate": "YYYY-MM-DD" }
 },
 "serviceRecords": [// required, at least one
 {
 "record_id": 1,
 "tariff_type": "007",
 "code": "AA.00.0010",
 "name": "Consultation, first 5 min.",
 "quantity": 1,
 "date_begin": "2026-01-15T00:00:00",
 "date_end": null, // optional
 "provider_id": "7601002534277",
 "responsible_id": "7601002534277",
 "unit_mt": 1, // medical (MT) fee component
 "unit_factor_mt": 1.0,
 "scale_factor_mt": 1.0, // optional (default 1.0)
 "external_factor_mt": 1.0, // optional (default 1.0)
 "amount_mt": 105.00,
 "unit_tt": 1, // technical (TT) fee component
 "unit_factor_tt": 1.0,
 "scale_factor_tt": 1.0, // optional (default 1.0)
 "external_factor_tt": 1.0, // optional (default 1.0)
 "amount_tt": 45.00,
 "amount": 150.00, // line-item total
 "vat_rate": 0.0, // optional (default 0.0)
 "ref_code": null, // optional
 "section_code": "ORL", // optional
 "remark": null, // optional
 "session": 1, // optional (default 1)
 "group_size": 1, // optional (default 1)
 "billing_role": "both", // optional (default "both")
 "medical_role": "self-employed", // optional (default "self-employed")
 "body_location": "none", // optional (default "none")
 "treatment": "ambulatory", // optional (default "ambulatory")
 "service_attributes": 0 // optional (default 0)
 }
]
}
```

Listing A.1: Full Invoice-Draft Schema

# Appendix B

## Supplementary Evaluation Results

This appendix collects reference and detailed-data floats that support the evaluation of Chapter 6 but are not needed to follow its argument: the full metric catalogue, the per-slice results, and the participant-level data of the use-case study.

### B.1 Metric Definitions

Table 24 lists every metric used in the quantitative evaluation and the research question it informs.

Table 24: Evaluation Metrics by Dimension

Metric	Measures	RQ
JSON validity	Drafts passing the pre-mapping structural contract (first-pass / post-repair).	RQ1
XSD validity	Invoices passing the Sumex schema (first-pass / post-repair).	RQ1
Repair effectiveness	First-pass failures converted to valid by the repair loop.	RQ1
Precision / recall / $F_1$	Agreement between emitted and expected tariff codes.	RQ3
Recall@ $K$	Expected codes among the top- $K$ retrieved chunks ( $K = 5, 10, 15, 20, 30, 50$ ).	RQ3
Structured recall@20	Expected codes among the structured channel’s tariff facts.	RQ3
TARDOC recall	Recall over the TARDOC-format subset of expected codes.	RQ3
Code-field faithfulness	Emitted codes grounded in retrieved knowledge.	RQ3
Context relevance	Precision of the codes the model used (proxy).	RQ3
Required-field completeness	Mandatory fields populated with non-default values.	RQ2
Constraint satisfaction	Intra-invoice consistency constraints that hold.	RQ2
Scenario correctness	Generated validity matches the expected validity.	RQ2
Violation injection	Intended rule violation detected by the oracle.	RQ2
Latency (p50 / p95)	Wall-clock time per scenario, including repair.	RQ4
Token usage	Prompt/completion tokens per scenario (provider-reported).	RQ4
Repair attempts	Mean repair LLM calls (JSON pass; XSD loop).	RQ1

## B.2 Per-Slice Quantitative Results

Table 25 breaks the aggregate results (Table 18) down by prompt tier, language, and validity class.

Table 25: Per-Slice Results

Slice	n	Precision	Recall	JSON	Violation
T0 code-explicit	13	1.00	0.79	1.00	n/a
T1 spec-complete	26	0.75	0.53	1.00	n/a
T2 clinical	13	0.83	0.71	1.00	n/a
T3 lay	60	n/a	n/a	0.98	0.64
T4 violation	34	n/a	n/a	1.00	0.64
German	60	0.87	0.60	0.98	0.64
English	86	0.82	0.65	1.00	0.64
Valid	78	0.83	0.64	1.00	n/a
Invalid	68	n/a	n/a	0.99	0.64

n/a indicates a metric does not apply to that slice. Violation injection is the overall rate over the 28 oracle-adjudicable negatives (Section 6.4) and is not further sub-sliced.

## B.3 Use-Case Study Data

This section collects the participant-level data and the theme-frequency breakdown of the use-case study (Section 6.5).

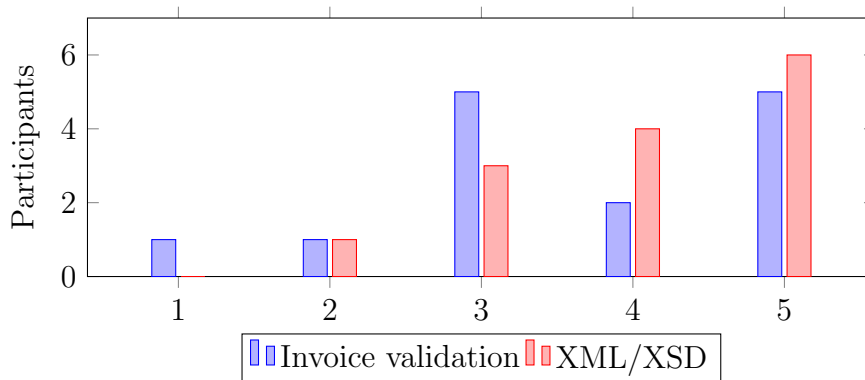


Figure 21: Self-Rated Familiarity Across Participants

Table 26: Participant Profile

Participant	Role	Invoice validation	XML/XSD
P1	Software Engineer	3	4
P2	Other	5	5
P3	Domain Expert	5	5
P4	Project Leader	3	4
P5	Other	5	5
P6	Software Engineer	3	3
P7	Business Analyst	5	5
P8	Software Engineer	2	2
P9	Software Engineer	5	5
P10	Business Analyst	4	4
P11	Software Engineer	3	4
P12	Software Engineer	3	5
P13	Domain Expert	4	3
P14	Software Engineer	1	3
<b>Mean</b>	—	3.6	4.1

Familiarity is self-rated on a 1 (none) to 5 (expert) scale.

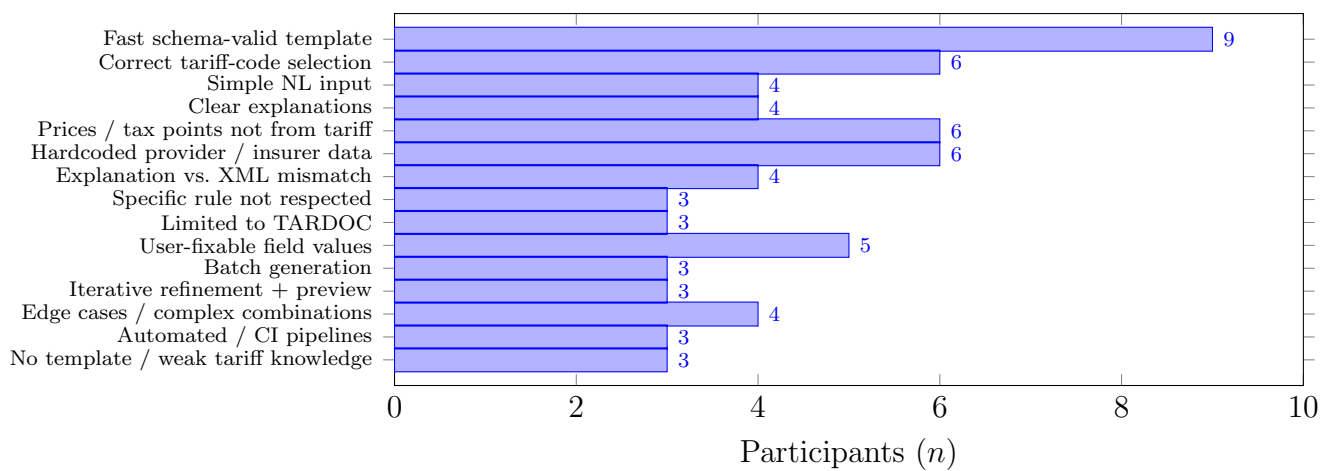


Figure 22: Frequency of Recurring Feedback Themes



# Appendix C

## Installation Guidelines

This appendix describes how to install and run the artifacts produced in this thesis. Three components are covered: the *InvoiceLab* prototype (the LLM-assisted test invoice generator of Chapters 4 and 5); the offline *evaluation* that reproduces the quantitative results of Chapter 6; and the two standalone *data-preparation tools*, *XML\_Fixer* and *InvoiceXMLScreening*, used to normalize and label the evaluation corpus (Section 6.2). The stack is based on Java, Maven, and Docker and is platform-independent: it was developed on Windows, and the instructions apply equally to Linux and macOS.

The source is delivered with the thesis submission as three project directories:

Table 27: Delivered Projects

Directory	Role	JDK
invoicelab/	Prototype service, UI, and evaluation harness	21
XML_Fixer/	Deterministic corpus normalization	17
InvoiceXMLScreening/	Ground-truth labelling against Sumex	17

### C.1 Prerequisites

Before installation, ensure the following are available:

- **Java Development Kit.** JDK 21 for the prototype and the evaluation; JDK 17 for the two data-preparation tools.
- **Apache Maven** (3.9 or newer), configured with access to the Sumex/ELCA artifact repository (via `~/.m2/settings.xml`). This is required to resolve the Sumex tariff library (`ch.sumex.tariff:tarifflibrary-dist`) and the internal Spring Boot dependencies.
- **Docker** and **Docker Compose**, used to provision the PostgreSQL 16 database with the `pgvector` extension.

- **Azure credentials.** An Azure AI Foundry chat-completions endpoint and key for generation, and an Azure OpenAI embeddings endpoint and key for retrieval.
- **Domain data.** The TARDOC tariff store and its tariff archive, and the ZIP archive of unstructured guideline PDFs.
- **SQL Server** (only for InvoiceXMLScreening): a reachable instance hosting the operational `sumexii` validation database.

## C.2 The Invoicelab Prototype

The prototype is a single Spring Boot service (Java 21) that exposes the generation pipeline over REST/SSE and serves a single-page web UI.

### 1. Obtain the source and start the database

```
cd invoicelab
docker compose up -d # PostgreSQL + pgvector on localhost:5432
```

The Compose file provisions a database `invoice-generator` with user `invoice-generator` and password `password`, matching the service defaults.

### 2. Provide the domain data

Place the domain data under `invoicelab/etc/data/`, so that the paths configured in `application.yaml` resolve:

```
etc/data/tariff-store/ # extracted TARDOC tariff store
etc/data/007_2026-02-26.zip # TARDOC tariff archive
etc/data/Unstructured_Data.zip # guideline PDFs (ingested on first start)
```

### 3. Configure credentials

Copy the local-profile template and fill in the two Azure API keys; endpoints, model list, and retrieval parameters are already set in `application.yaml`:

```
cp src/main/resources/application-local.example.yaml \
 src/main/resources/application-local.yaml
```

```
src/main/resources/application-local.yaml
invoicelab:
 azure:
 embedding:
 api-key: <YOUR_AZURE_EMBEDDING_KEY>
 kimi:
 api-key: <YOUR_AZURE_FOUNDRY_KEY>
```

## 4. Build and run

```
mvn clean package # compile, run unit tests, build the jar
mvn spring-boot:run # start the service (default profile: local)
```

On first start, when the vector store is empty, the service ingests the guideline PDFs into `pgvector` (load, parse, chunk, embed, store); on subsequent starts this step is detected as complete and skipped, so it is idempotent. Once started, the service is available at:

- **Web UI & API:** `http://localhost:8080`
- **Health/actuator:** `http://localhost:8081`

## 5. Generate a test invoice

1. Open `http://localhost:8080` in a browser.
2. Describe the desired invoice scenario in natural language (English or German), optionally selecting the generation model.
3. Generate the invoice; the UI streams the pipeline steps (interpret, retrieve, generate, validate).
4. Inspect the structured draft and the validity explanation, then download the schema-validated XML.

## C.3 Reproducing the Evaluation

The quantitative evaluation runs the full pipeline over the frozen golden dataset. The dataset ships committed under `invoicelab/golden-dataset/`, so the evaluation can be reproduced without regenerating it.

### 1. Configure the evaluation harness

The standalone harness reads its configuration from a `.env` file at the `invoicelab/` project root (searched up to five parent directories) and from the process environment. Create it from the following template and fill in the placeholders:

Table 28: Principal Environment Variables for the Evaluation Harness

Variable	Description
AZURE_EMBEDDING_ENDPOINT	Azure OpenAI embeddings endpoint URL
AZURE_EMBEDDING_API_KEY	Azure OpenAI key
AZURE_KIMI_ENDPOINT	Azure AI Foundry chat-completions endpoint
AZURE_KIMI_API_KEY	Azure AI Foundry key
EMBEDDING_MODEL / _DIMENSIONS	e.g. <code>text-embedding-3-small</code> / 1536
LLM_MODEL	Generation deployment name (see model sweep)
TARIFF_STORE_PATH	Absolute path to the tariff store
TARDOC_TARIFF_ID	TARDOC tariff ID to load
UNSTRUCTURED_ZIP_PATH	ZIP of guideline PDFs
PG_HOST/PORT/DB/USER/PASSWORD	PostgreSQL connection
LLM_TEMPERATURE	Sampling temperature (evaluation: 0.2)
LLM_MAX_TOKENS	Output-token cap (evaluation: 16384)
RETRIEVAL_MIN_SCORE	Cosine-similarity threshold (0.30)
QUERY_EXPANSION_ENABLED	<code>true</code> to expand queries into TARDOC keywords
STRUCTURED_STRATEGY	BM25 or HYBRID (BM25 + dense, RRF)
XSD_REPAIR_ENABLED	Enable the bounded generate-validate-repair loop
EVAL_RUN_ID	Label identifying the run in the leaderboard

## 2. Run the harness

With the database running, launch the harness. It wires the full real pipeline, loads the golden scenarios, executes them, and writes a self-documenting per-run directory<sup>1</sup>:

```
docker compose up -d
mvn -q exec:java \
 -Dexec.mainClass=ch.sumex.invoivelab.evaluation.EvaluationHarness
```

Results are written under `evaluation-output/<timestamp>/` (`evaluation-results.json`, `evaluation-summary.csv`, `evaluation-slices.csv`, `config.json`, `run-meta.json`), and one row per run is appended to the append-only leaderboard `runs-index.csv`.

## 3. Export figure data

The chapter-ready figure CSVs are produced with a single command:

```
mvn -q exec:java \
 -Dexec.mainClass=ch.sumex.invoivelab.evaluation.FigureDataExporter
```

<sup>1</sup>The registered Java package identifier is `ch.sumex.invoivelab`; the name predates this thesis and is retained as-is for consistency with the deployed artifact.

## 4. Model and embedding sweeps

To reproduce the sensitivity analyses (Sections 6.4.6 and 6.4.7), run the harness once per cell, changing only `LLM_MODEL` (and, for the embedding axis, the embedding block) and giving each cell a distinct `EVAL_RUN_ID`, while holding all other settings fixed.

## 5. (Optional) Regenerating the golden dataset

The frozen dataset is already provided. To rebuild it from the source corpus, generate the candidates, review them, and freeze:

```
generate a balanced sample with LLM-authored prompts -> golden-dataset-output/
mvn -q exec:java \
 -Dexec.mainClass=ch.sumex.invoivelab.xmlgoldendataset.GoldenDatasetCli \
 -Dgolden.sample=30 -Dgolden.llm=true

after reviewing golden-dataset-output/golden-dataset-review.csv, freeze:
mvn -q exec:java \
 -Dexec.mainClass=ch.sumex.invoivelab.xmlgoldendataset.GoldenDatasetFreeze \
 -Dexec.args="<review.csv> <golden-dataset.json> <frozen.json>"
```

The frozen files are then copied into `golden-dataset/`.

## C.4 Supporting Data-Preparation Tools

The two tools prepare the evaluation corpus and are run once, before the golden dataset is built. They correspond to the first two stages of the data lineage in Figure 11: normalization (`XML_Fixer`) followed by labelling (`InvoiceXMLScreening`).

### C.4.1 XML\_Fixer (Normalization)

`XML_Fixer` streams each corpus invoice, replaces malformed field values with schema-valid placeholders, mirrors the output directory structure, and pre-checks each file against the XSD schema. It has no external dependencies.

1. Set the input and output directories in `src/main/resources/config.properties`:

```
input.dir=C:\\path\\to\\raw-invoices
output.dir=C:\\path\\to\\fixed-invoices
```

2. Compile and run:

```
cd XML_Fixer
mvn -q clean compile
java -cp target/classes ch.sumex.invoivelab.Main
```

3. Alternatively, pass the two directories directly as arguments:

```
java -cp target/classes ch.sumex.invoivelab.Main <input-dir> <output-dir>
```

The console reports **FIXED** per transformed file and **VALID/INVALID** for the schema result; a bad file does not abort the batch.

### C.4.2 InvoiceXMLScreening (Labelling)

**InvoiceXMLScreening** assigns each invoice its ground-truth validity label by looking it up in the operational **sumexii** database and inspecting the validation codes the production engine recorded against it. Only invoices with **tariff\_type="007"** (TARDOC) are validated; others are skipped.

1. Copy the configuration template and set the values:

```
cd InvoiceXMLScreening
cp .env.example .env
```

Required keys: **INPUT\_DIR**, **VALID\_DIR**, **INVALID\_RWC\_DIR**, **INVALID\_VLC\_DIR**, **ERROR\_DIR**, **REPORT\_DIR**, **LOG\_DIR**, and the database connection (**DB\_URL**, **DB\_USERNAME**, **DB\_PASSWORD**, **DB\_DRIVER**). For a SQL Server instance:

```
DB_URL=jdbc:sqlserver://localhost:1433;databaseName=sumexii;\
encrypt=true;trustServerCertificate=true
DB_DRIVER=com.microsoft.sqlserver.jdbc.SQLServerDriver
```

2. (Optional) run the unit tests, then run the tool:

```
mvn clean test
mvn exec:java
```

Optionally pass the directory containing **.env** as the first argument:

```
mvn exec:java -Dexec.args="C:\\path\\to\\env\\dir"
```

Each invoice is classified with the priority **ERROR > INVALID\_RWC > INVALID\_VLC > VALID** (with **SKIPPED** for non-TARDOC invoices), moved into the corresponding output folder, and recorded as a row in the CSV execution report. These labels are the ground truth from which the golden dataset (Section C.3) is constructed.

By following these guidelines, the prototype can be installed and operated, the quantitative evaluation reproduced, and the labelled evaluation corpus rebuilt from raw invoices.

# Appendix D

## Contents of the CD

1. This thesis as PDF.
2. Declaration of Authorship as a separate signed document.
3. This thesis as LATEX source in a `.zip` file (if required).
4. Midterm presentation slides as PDF (if required).
5. The source code of the prototype (if required).
6. The responses from the participants used for the evaluation (if required).