Integrating LLMs into DSL Engineering: Lessons from a DSL Evolution Study

Journal:	*IEEE Software*
Manuscript ID	SW-2025-10-0163
Manuscript Type:	Regular
Date Submitted by the Author:	13-Oct-2025
Complete List of Authors:	Garcia Molina, Jesus; Universidad de Murcia Pérez-Serrano, Antonio; Universidad de Murcia Fernández-Candel, Carlos J.; Universidad de Murcia
Keywords:	D.2.6 Programming Environments/Construction Tools < D.2 Software Engineering < D Software/Software Engineering, D.3.2.g Design languages < D.3.2 Language Classifications < D.3 Programming Languages < D Software/Software Engineering, D.3.3 Language Constructs and Features < D.3 Programming Languages < D Software/Software Engineering
Abstract:	***Large Language Models (LLMs) such as GPT-4o are changing the way software is built. They can turn natural language (NL) descriptions into working code or other software artifacts, bringing automation to tasks once reserved for experts. Yet Domain-Specific Languages (DSLs) continue to play a key role: they capture domain knowledge explicitly and enforce structure—something current LLMs often lack. In this article, we share our experience using GPT-4o to automate the evolution of two DSLs from the USchema family: Athena, for database design, and Orion, for schema evolution. Guided through a four-step prompting strategy, GPT-4o generated correct DSL scripts and corresponding target code, reducing the effort required to maintain and extend these languages. Our findings show that LLMs and DSLs can complement each other: LLMs accelerate development and evolution, while DSLs provide the formal backbone that ensures correctness and maintainability.***

SCHOLARONE™
Manuscripts

Article Type: Feature (LLM–DSL SYNERGY)

Integrating LLMs into DSL Engineering: Lessons from a DSL Evolution Study

Jesús García-Molina, University of Murcia, Spain

Antonio Pérez-Serrano, University of Murcia, Spain

Carlos J. Fernández-Candel, University of Murcia, Spain

Abstract— Large Language Models (LLMs) such as GPT-4o are changing the way software is built. They can turn natural language (NL) descriptions into working code or other software artifacts, bringing automation to tasks once reserved for experts. Yet Domain-Specific Languages (DSLs) continue to play a key role: they capture domain knowledge explicitly and enforce structure—something current LLMs often lack.

In this article, we share our experience using GPT-4o to automate the evolution of two DSLs from the USchema family: Athena, for database design, and Orion, for schema evolution. Guided through a four-step prompting strategy, GPT-4o generated correct DSL scripts and corresponding target code, reducing the effort required to maintain and extend these languages. Our findings show that LLMs and DSLs can complement each other: LLMs accelerate development and evolution, while DSLs provide the formal backbone that ensures correctness and maintainability.

Domain-Specific Languages (DSLs) have long served as a bridge between a developer's intent and its implementation in general-purpose languages (GPL). By focusing on a specific problem domain, DSLs enable developers to work at a higher level of abstraction, reducing effort and increasing productivity. Well-known examples include SQL for querying databases, Gherkin for defining acceptance tests, and Flutter for building cross-platform user interfaces.

The rise of Large Language Models (LLMs) such as GPT, Claude, and Gemini brings a new dimension to this picture. These models can generate code or other software artifacts from natural language (NL), offering an abstraction level higher than DSLs. This raises a natural question: if LLMs can understand our intent so broadly, do we still need DSLs?

Recent studies have started to explore the relationship between DSLs and LLMs. Böckelund [1] highlighted how the smaller the semantic gap between a prompt and the target language, the simpler the interaction. Netz et al. [2] trained GPT-4 to generate DSL scripts from NL, enabling non-experts to create web apps. García-González et al. [3] developed DSL-Xpert, capable of generating scripts for many DSLs through grammar prompting and few-shot learning. Mosthaf et al. [4] even built a GPT-4 assistant to create new DSLs, including grammar and examples. Together, these works show that LLMs not only generate DSL artifacts but can also assist across the DSL lifecycle.

Our own perspective builds on two decades of hands-on DSL design across diverse domains—from model-driven engineering (MDE) to context-aware. In recent years, this experience led us to create a family of DSLs based on the USchema generic data model: *Athena* for defining database schemas, *Orion* for evolving them, and *SkiQL* for querying them [5–7]. Building on this foundation, we recently explored how LLMs could support

DSL engineering tasks for *Athena* and *Orion*, comparing the outcomes with those obtained in earlier manual evolutions.

Results showed that, with modest training, a general-purpose LLM can perform DSL evolution tasks such as code generation and language conversions, while enabling users to express their intent in NL instead of writing DSL scripts. In turn, DSLs provide a formal structure that constrains LLM output and ensures syntactic and semantic correctness, although human supervision remains necessary. This article reports on our experience using GPT-4o as the LLM in our project. The following sections summarize the automated tasks, explain our prompting strategy, describe the evaluation and results, and conclude with the main lessons learned.

WHAT DSL TASKS CAN BE AUTOMATED WITH LLM?

Many DSLs, particularly those created using MDE techniques, are built from three core elements [8]:

1) *Domain model or metamodel*— that captures the concepts and relationships of the language, providing a conceptual basis.

2) *Notations*— that allow developers to express solutions in textual and/or graphical form.

3) *Code generators*— that consist of *model* transformation chains that produce executable code or other software artifacts.

The following sidebar briefly explains how these generators are typically implemented in MDE solutions.

****** BEGIN SIDEBAR ******

How DSL Generators are built in MDE

A generator takes as input a DSL script, which is internally converted into a model conforming to the DSL metamodel. This model, commonly represented as an object graph in memory, is then processed through a transformation chain composed of zero or more model-to-model (M2M) transformations and a final model-to-text (M2T) transformation. In many cases, generators consist solely of an M2T transformation when the mapping between the model and the generated artifacts is not complicated.

- ➔ *Model-to-Model (M2M) transformations* map between metamodels and can be implemented either with DSLs (e.g., Acceleo) or GPLs (e.g., Java or Xtend)
- ➔ *Model-to-Text (M2T)* transformations specify how to generate textual artifacts from models. String templates are commonly used; Acceleo, Python, and Xtend natively support template-based generation.

****** END SIDEBAR ******

DSL users learn the notation to write scripts that serve as input to generators to obtain the desired result, while DSL developers build new languages or evolve existing ones. To create a DSL, developers define the metamodel, specify one or more notations (grammars for textual DSL), and implement one or more generators. To evolve an existing DSL, they may extend DSLs — for instance, adding new generators or establishing mapping to well-known notations. Traditionally, all these tasks required significant manual effort and technical expertise. LLMs now open the door to automating them or users can rely on them to generate scripts directly from NL requirements. This shift reduces the cost of DSL evolution and lowers the entry barrier for adoption, while still preserving the benefits of DSLs.

Our focus here will be on evolution tasks of two DSLs based on the USchema metamodel that provides a generic data model to represent databases schemas. Athena specifies database schemas, and Orion describes schema change operations (SCO). These DSLs were created using the Xtext framework, which provides the Xtend language for writing model transformations [9].

Figure 1 provides simplified Athena and Orion examples for a social network database. Figure 1(a) shows an Athena script defining four entities: *User*, *Profile*, *Post* and *Comment*. Figure 1(b) illustrates Orion operations on that same schema, including renaming attributes, adding fields, and extracting entities. Readers do not need to grasp the full syntax or semantics of these DSLs; the examples are included only to illustrate the level of detail that LLMs must handle in our case. Still, the syntax and semantics of Athena and Orion can be easily inferred from these examples, which capture their key constructs and conventions.

Box 1 summarizes the tasks automated by using GPT-4o. While the examples are specific to these DSLs, the approach applied here can be generalized to others. As discussed later, human supervision through semantic verification remains essential to ensure the correctness and consistency of the generated artifacts

```
SCHEMA TodAI:1
ROOT ENTITY User {
   +id: Identifier,
   !name: String /[a-zA-Z0-9_]
                 {5,15}/,
   !email:String/^.+@.+\\.com$/,
   age: Number <0..100>,
   ?profile: aggr<Profile>&
   followers: ref<User>*,
   following: ref<User>*,
   comments: ref<Comment>*,
   posts: ref<Post>*
}
ENTITY Profile: {
   name: String,
   ?avatar_url: String,
   description: String
}
```

```
ROOT ENTITY Comment{
  +id: Identifier,
  text: String,
  post: ref<Post>&,
  replyTo: ref<Comment>&,
  user: ref<User>&
}
ROOT ENTITY Post {
    +id: Identifier,
    create_at: Timestamp,
    update_at: Timestamp,
    title: String,
    user: ref<User>&,
    ?content: String,
    numLikes: Number
}
```

(a) Example Athena script defining entities, relationships, and constraints in a social-network schema

```
-- Rename attribute 'name' to
--'handle' in entity User
RENAME User.name to handle

-- Add attribute 'numPosts'
-- of type Number to User

ADD ATTR User.numPost: Number

-- Extract 'create_at' and
-- 'update_at' into a
-- separate entity

EXTRACT ENTITY Post INTO
    PostMetadata (create_at,
                  Update_at)
```

(b) Example Orion script specifying SCOs on TodAI schema.

FIGURE 1. Example Athena and Orion scripts

BOX 1. LLM-based Automation in Athena and Orion

For DSL developers

- ➔ Generate SQL, Cassandra CQL, and MongoDB schemas from Athena scripts.
- ➔ Generate SQL, Cassandra CQL, and MongoDB data update scripts from Orion scripts.
- ➔ Implement bidirectional translation capabilities for both Athena and Orion, enabling conversions between DSL scripts and multiple languages (SQL, CQL, Cypher, MongoDB API).

For DSL users

- ➔ Generate Athena and Orion scripts directly from natural-language requirements.
- ➔ Create datasets of DSL scripts for validating and testing the automated tasks.

GUIDING THE LLM FOR DSL TASKS: A FOUR-STEP PROMPTING STRATEGY

To enable GPT-4o to work effectively with *Athena* and *Orion*, we designed a four-step prompting strategy instead of training a new model from scratch. The goal was to guide the LLM so it could handle DSL scripts with enough correctness and consistency. We conducted two independent sessions—one for *Athena* and the other for *Orion*—where we wrote prompts (textual instructions provided to the model to elicit specific responses) tailored to each step's objective. After applying the four steps, we verified that the model produced correct outputs before relying on it for automation. The complete process is explained below and illustrated in Figure 2.

Step 1. System Prompt.

We began by describing the overall problem, the structure of the DSLs, and the strategy to solve it, setting the context for the model. The system prompt defined three sequential training phases—formal definition, explanatory material, and practical examples—each requiring the model to acknowledge receipt before continuing. When the command "End of training" was issued, the LLM was asked to summarize what it had learned and raise any questions or ambiguities it

detected. This staged setup established the model's role and delimited the scope of the tasks to be performed. All the prompts and additional material are available in our public GitHub repository, referenced at the end of this section.

Step 2. Grammar Prompting.

We provided the syntactic rules required to generate structurally valid *Athena* and *Orion* scripts, derived from their original Xtext grammars. To reinforce correctness, we also included Xtend validation rules (e.g., entities must have distinct names), ensuring the LLM respected both grammar and semantic constraints. This approach constrained the model's output and reduced syntactic errors.

Step 3. Knowledge Acquisition.

The LLM was exposed to publicly available documentation about both DSLs, including research papers, thesis excerpts, and example schemas with their corresponding evolution operations. The goal was to provide the background knowledge required to understand the semantics of *Athena* and *Orion*, enabling the model to interpret and generate valid DSL scripts. This exposure improved its grasp of domain-specific details and reduced ambiguities in the outputs.

Step 4. Few-Shot Learning.

We gave the model detailed examples of bidirectional conversions to help it generalize beyond the examples introduced in the previous steps.

a) *For Athena*: five examples covering translations between Athena and three target technologies—SQL, Cassandra CQL, and Mongoose (MongoDB ORM)—resulting in 30 conversion cases.
b) *For Orion*: two examples covering conversions with four target technologies—SQL, Cassandra CQL, MongoDB, and Neo4J—resulting in 8 cases.

Since *Athena* scripts define complete database schemas, they are inherently more complex and required more examples than *Orion*, which focuses on individual SCO. Even so, the overall number of examples remained small, yet sufficient for the model to generalize effectively. This allowed it to perform accurate translations and SCOs on new cases.

Beyond these four steps, the process revealed some unexpected behaviors and additional challenges. After the End of training command, the model not only demonstrated understanding of the DSL syntax but also raised clarification questions about semantic aspects not explicitly covered in the specifications — for example, how Athena manages versioning or how to combine ranges with regular expressions. Based on our answers, it even suggested possible ways to address these issues.

To ensure coverage, we curated the example set to include all relevant DSL constructs—aggregates, references, hierarchies, and many-to-many relationships—so the model would encounter the range of patterns present in real schemas. Because Orion was given fewer examples, we ran additional iterations until the model consistently applied schema-evolution operations correctly.

Outputs were validated through automatic syntax checks and manual semantic verification, including execution on real databases. The detailed results of this validation are presented in Section 4.

To foster reuse, we published the full prompting workflow—system prompt, grammar rules, documentation excerpts, and few-shot pairs—together with the complete GPT-4o chat session and validation scripts in a public GitHub repository (https://github.com/modelum/IEEE-Software-2025). We deliberately opted for a repository rather than a proprietary GPT in ChatGPT to ensure portability across LLM providers, transparent versioning and peer review, easier citation and long-term availability, and to avoid vendor lock-in. This makes it straightforward for developers to adopt or adapt the prompts and replicate our results when working with Athena and Orion.

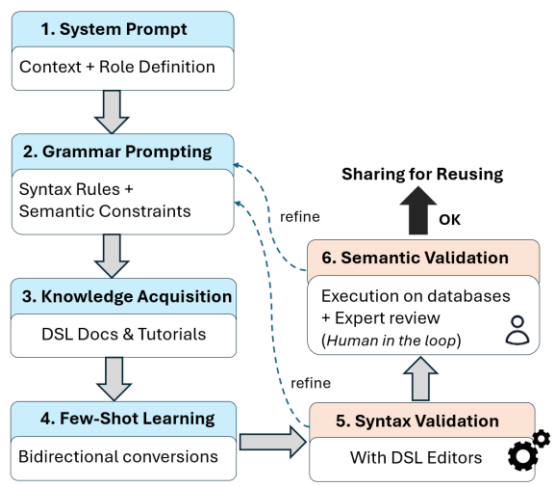

FIGURE 2. Prompting strategy and validation process.

EVALUATION AND RESULTS

We evaluated our prompting-based approach along two dimensions: the coverage and correctness of the generated outputs, and the reduction of manual effort compared to traditional implementations.

Correctness. We conducted two validation experiments to assess both the coverage and correctness of the LLM-generated outputs for Athena and Orion. In the first experiment, we used small-scale cases involving 3 to 5 entities that covered the full variability in the source DSLs. Conversions from Athena to the target database languages and APIs (SQL, CQL of Cassandra, and MongoDB API) were evaluated using a five-entity schema describing students' comments on courses taught by teachers. For the reverse conversions, different schemas were employed to ensure the independence of results. All transformations were syntactically and semantically correct, except for a single case (MongoDB to Athena) where a range type was generated without an upper bound.

Next, an Orion script was created for the course review schema, covering its ten main change operations. All direct conversions to database scripts were correct, and the reverse ones—from each database language to Orion using their corresponding schemas—were also successfully validated.

The second experiment used two larger and more complex schemas for Athena: a sports e-commerce application for direct conversions and a 15-table extract of the IMDB schema for reverse ones. In the e-commerce case, the Athena script—automatically generated from natural language—defined 11 entities covering users, products, discounts, payments, and sales. Validation achieved full correctness for database script generation and nearly 100% for the natural language–to–Athena step, with a single inaccurate mapping of payment information that the LLM promptly corrected after minimal feedback. In the IMDB case, all conversions between SQL, Cassandra, and MongoDB schemas to Athena were fully successful.

Overall, the validation performed in the two experiments demonstrated that the LLM accurately learned the Athena and Orion DSLs and consistently applied their deterministic mapping rules across all target data models, achieving complete correctness in all schema transformations.

Validation was performed using the Xtext editors to verify grammar and semantic rules, and by executing the SQL, CQL, and MongoDB outputs on real databases to confirm their correct behavior. The generated scripts were reviewed by a domain expert; only minor adjustments were needed, confirming the high reliability of the automated generation process.

Effort reduction. In the traditional approach, M2T transformations for Athena and Orion were manually implemented in Xtend, each producing the scripts for target databases. Table 1(a) summarizes their size in lines of code (LOC).

Although Athena provides a rich set of constructs for defining complex database schemas, its M2T transformations are relatively straightforward: each schema element is mapped to a corresponding construct in the target language. Instead, the SCOs of Orion often require multi-step updates to existing databases. These operations must be decomposed into several low-level DDL and DML commands (e.g., ALTER TABLE, CREATE, or UPDATE statements), sometimes involving temporary structures. Consequently, Orion's M2T transformations are significantly longer, as reflected in the higher LOC values shown in Table 1(a).

(a) Lines of code (LOC) for Athena and Orion M2T transformations.

	SQL	Cassandra	MongoDB
LOC (Athena)	327	278	266
LOC (Orion)	1321	561	768

(b) Internal complexity metrics for Orion-to-SQL M2T transformation.

	Attribute Promote	Reference Add	Aggregate Add	Average (≈)
Max Depth	3	2	1	**1.5**
Iteration/Cond. constructs	6 / 3	5 / 3	6 / 3	**4.3 / 2.6**
Artifacts per operation	6	9	1	**2.3**
Mapping Density	4	5	4	**2.9**

TABLE 1. Code size and complexity of M2T transformations.

Beyond code size, we also measured the internal complexity of the handcrafted transformations. We used the following metrics: *Max Depth* (nesting depth of control flow and scopes); *Iteration* (lambdas included); and *Conditional constructs*; *Artifacts per operation* (distinct artifacts generated per SCO); and *Mapping density* (number of mappings applied by a SCO).

We applied these metrics to the most demanding conversion direction, Orion to database scripts—concretely, the generation of SQL code for schema change operations, which synthesizes data-update scripts and typically orchestrates DDL and DML steps. Table 1(b) reports the results for the three most complex SCOs—*AttributePromote* (which converts an attribute into a key), *ReferenceAdd*, and *AggregateAdd*—together with the overall average computed across all SCOs. The results show that even a single SCO combines nested control flow, multiple iterations and conditionals, and several cross-artifact mappings—highlighting the cognitive and maintenance burden of handcrafted M2T code that the LLM-based approach helps avoid.

The LLM-based workflow removes the need to design, implement, and maintain these M2T transformations. Once the grammar constraints and few-shot examples are prepared, new conversions can be generated directly from prompts, without writing transformation code. This shifts the engineering effort from coding to prompt design and validation, which are lighter and reusable tasks across targets.

LESSONS LEARNED

Our work combining DSLs with LLMs yielded several lessons that may guide other practitioners, reflecting both the benefits and the practical challenges we encountered when applying LLMs to DSL evolution and code generation.

LLMs can learn DSLs with structured guidance. A generic LLM lacks native understanding of most DSLs, but it can learn one when guided through a structured prompting strategy that constrains syntax and semantics. Once trained, the same setup can be reused across related tasks or extended DSL versions with minimal adjustments.

Rigorous validation is required. Although only a few minor issues appeared during validation—about three in total—systematic checking was indispensable to ensure full syntactic and semantic correctness. Automated grammar validation and expert review confirmed the reliability of the generated scripts and helped fine-tune the prompts for consistent results.

Effort shifts rather than disappears. Manual work in M2T transformations was replaced by prompt engineering and validation. The first three steps of our prompting strategy—system prompt, grammar constraints, and DSL documentation—are lightweight and reusable across tasks. For each new conversion, the main cost lies in preparing a few representative examples and validating outputs—still far less than writing and debugging hundreds of lines of transformation code.

LLMs can cope with complex DSLs. Athena includes sophisticated constructs such as aggregates, references, cardinalities, and regular-expression constraints. The LLM handled these correctly when supported by precise grammar rules, high-quality examples, and validation. Similarly, in Orion, the model managed SCOs that required multi-step updates across different database technologies.

LLMs can reason about DSL design. After completing the training phase, we asked the model to provide feedback on the learning process (doubts, and improvement suggestions). The LLM produced a list of thoughtful questions and proposals addressing ambiguities, formal semantics, and tool integration. For instance, it inquired about operator associativity, multiple-inheritance conflicts, and the semantic validity of feature qualifiers. It also raised questions about JSON Schema generation. This behavior indicates that, beyond generating or transforming DSL artifacts, LLMs can engage in meta-level reasoning about a language's design, helping identify unclear semantics or underspecified rules. Such reflective feedback can support DSL engineers in improving the clarity, consistency, and tooling of their languages.

DSL expertise remains indispensable. While LLMs lower the technical barriers to using DSLs, their effective deployment still depends on experts who can define constraints, interpret outputs, and validate results. In practice, LLMs act as accelerators, rather than replacements, for DSL evolution.

CONCLUSIONS AND OUTLOOK

The results show that LLMs can effectively assist DSL evolution, provided that systematic validation and expert oversight are in place.

Beyond these practical findings, a broader question arises: can LLMs fully replace DSLs? Based on our experience, the answer is no. DSLs embody domain knowledge in a precise and verifiable form, enabling validation, auditing, and long-term maintainability. Moreover, some DSLs—such as Athena—include advanced constructs that make manual scripting difficult, especially when expressed through prompts rather than structured editors. While LLMs cannot replace DSLs, they can simplify the creation of complex DSL scripts from natural language. They help reduce effort but do not eliminate the need for formal definitions and expert oversight. In practice, LLMs and DSLs should be viewed as complementary: *DSLs provide the formal backbone, while LLMs enhance usability and lower the effort of evolution tasks.*

Looking ahead, we foresee hybrid workflows where domain experts express needs in natural language, LLMs generate DSL scripts and suggest evolution steps, and formal validators check correctness. Through iterative refinement, DSLs evolve while LLMs assist in adjusting metamodel, grammars, and model transformations. As LLMs evolve into autonomous agents capable of managing context and coordinating complex workflows, their collaboration with DSLs may give rise to self-adaptive systems that continuously refine themselves through feedback and validation. In the coming years, DSLs will remain central to many LLM-based development workflows. For instance, AI workflow automation platforms, such as n8n, already employ a graphical DSL for process definition.

Ultimately, LLMs do not displace DSLs—they enrich them: reducing development effort, strengthening tooling, and extending their accessibility beyond traditional expert roles. While our focus has been on DSL evolution, LLMs could also support the early stages of DSL creation—for example, by proposing notations, drafting grammar rules, or generating metamodel prototypes from natural language descriptions. Exploring this direction is a promising direction for future work.

ACKNOWLEDGMENTS

This work was supported by the Spanish Ministry of Science, Innovation and Universities, the State Research Agency (MICIU/AEI /10.13039/ 5011000 11033) under project PID2020-117391GB-I00, and by the European Union's ERDF program.

REFERENCES

1) B. Böckelund, "How is GenAI different from other code generators?", in *Exploring Gen AI*, M. Fowler, Ed., Thoughtworks, Sep. 19, 2023. [Online]. Available: https://martinfowler.com/articles/exploring-gen-ai/07-how-is-this-different.html
2) L. Netz, J. Michael, and B. Rumpe, "From natural language to web applications: Using large language models for model-driven software engineering," Proc. Modellierung 2024, 2024, pp. 179–195.
3) V. Lamas, M. R. Luaces, and D. Garcia-Gonzalez, "DSL-Xpert: LLM-driven Generic DSL Code Generation," Proc. ACM/IEEE 27th Int. Conf. Model-Driven Eng. Lang. Syst. Companion (MODELS Companion '24), 2024, pp. 16–20.
4) M. Mosthaf and A. Wasowski, "From a natural to a formal language with DSL Assistant," Proc. ACM/IEEE 27th Int. Conf. Model-Driven Eng. Lang. Syst. (MODELS '24), 2024, pp. 541–549.
5) A. Hernández-Chillón, D. Sevilla-Ruiz, and J. García-Molina, "Athena: A database-independent schema definition language," in *Advances in Conceptual Modeling: ER 2021 Workshop CoMoNoS*, 2021, pp. 33–42.
6) A. Hernández-Chillón, M. Klettke, D. Sevilla Ruiz, and J. García-Molina, "A generic schema evolution approach for NoSQL and relational databases," *IEEE Trans. Knowl. Data Eng.*, vol. 36, no. 7, pp. 2774–2789, 2024.
7) Carlos J. Fernández-Candel, Jesús García-Molina, Diego Sevilla Ruiz, "SkiQL: A unified schema query language", Data Knowl. Eng., 148: 102234, 2023
8) M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, 2nd ed., Synthesis Lectures on Software Engineering. San Rafael, CA, USA: Morgan & Claypool Publishers, 2017.
9) L. Bettini, *Implementing Domain Specific Languages with Xtext and Xtend*, 2nd ed. Birmingham, U.K.: Packt Publishing, 2016.

Jesús García Molina received the Ph.D. degree from the University of Murcia, Spain, in 1987. He is a Full Professor with the Department of Informatics and Systems, University of Murcia, where he leads the Modelum group, an R&D team focused on task automation—mainly in software engineering and database engineering. His current research interests include domain-specific languages, and LLM-based automation. Contact him at jmolina@um.es.

Antonio Pérez-Serrano received the BSc degree in Computer Science from the University of Murcia, Spain, in June 2025. He is a member of the Modelum research group at the Faculty of Computer Science, University of Murcia. Within this group, he completed the bachelor's thesis on agile schema evolution with Orion. His research interests include defining methods and developing tools powered by LLMs to automate tasks, as well as software modernization. Contact him at a.perezserrano1@um.es.

Carlos J. Fernández-Candel received his Ph.D. in computer science from the University of Murcia, Spain, 2022. He was a member of the ModelUM research group at the Faculty of Computer Science, University of Murcia. His research interests include model-driven engineering, data engineering, and artificial intelligence. Contact him at cjferna@um.es.