Towards a Multi-Agentic Pipeline to Automate Model-to-Text Transformations for Domain-Specific Languages

Theodoros Tsampouris, Emmanouil Tsardoulias, Konstantinos Panayiotou, Andreas L. Symeonidis
Dept. of Electrical and Computer Engineering
Aristotle University of Thessaloniki
Thessaloniki, 54124, Greece
ttsampob@ece.auth.gr, etsardou@ece.auth.gr, klpanagi@ece.auth.gr, symeonid@ece.auth.gr

***Abstract*—Domain-Specific Languages, apart from the design and formal definition aspects involved to build, also require Model-to-Text (M2T) transformations to generate executable code. Authoring the M2T templates is a manual, labor-intensive process that often generates frustrating, time consuming errors. And, although Large Language Models have advanced code and grammar generation, the automated synthesis of downstream Model-to-Text templates still remains largely unaddressed. In the context of software engineering and model-driven development, automating such transformations addresses a core challenge in modern software toolchain construction. This paper presents a work-in-progress multi-agent system that automatically generates production-ready Jinja2 templates from a textX grammar, natural language requirements and sample model instances. The architecture decomposes the generation task into seven stages orchestrated by eleven specialized agents, employing deterministic runtime introspection to provide ground-truth structural data. To ensure reliability, the generation process is coupled with a rigorous three-level deterministic validation loop that evaluates template syntax, rendering execution and target-language correctness without relying on self-assessment by the language models. We demonstrate the pipeline on two structurally versatile domains, specifically a recursive calculator and a declarative smart home automation language, illustrating its ability to produce functional templates, rendered code and test artifacts across diverse grammar paradigms.**

***Index Terms*—Domain-Specific Languages, Model-to-Text Transformation, Large Language Models, Multi-Agent Systems, Automated Code Generation, Model-Driven Engineering**

I. Introduction

Domain-Specific Languages (DSLs) have long been recognized as powerful instruments for bridging the gap between domain expertise and software implementation, enabling practitioners to express solutions using terminology and abstractions native to their problem space while hiding the inherent complexity of the domain behind well-defined language constructs [1]. A complete DSL toolchain requires not only a formal grammar and parser, but also Model-to-Text (M2T) transformations that convert parsed model instances into correct target General Purpose Language (GPL) code [2]. These transformations are typically authored as templates in languages such as Acceleo, EGL, Xpand, or Jinja2 and their correctness depends on knowledge of the grammar's internal object model, the target GPL idioms and the runtime attribute structure of parsed models [2], [3].

In the era of Large Language Models (LLMs), this template-based code generation paradigm takes on renewed significance. While LLM-based code generation is inherently stochastic, with studies reporting that at least 48% of AI-generated code snippets contain security vulnerabilities [4], DSLs combined with M2T transformations offer a fundamentally different and complementary approach. In a DSL workflow, the domain expert or an LLM with knowledge in the corresponding domain writes a model that expresses *what* the system should do using domain-native abstractions. The M2T templates then deterministically render this model into executable code that is pre-validated, quality-checked and safe, because the templates themselves have been verified once and are then applied reliably to any conforming model instance [5]. However, building this trustworthy infrastructure remains prohibitively labor-intensive. M2T template authoring requires not only language engineering expertise, but also deep knowledge of the target GPL's implementation conventions and deployment patterns, demanding collaboration between domain experts, language engineers and target-GPL specialists [1].

Having that said, it is also a fact that LLMs have demonstrated remarkable capabilities in automated code generation [6], [7] and LLM agents can iteratively improve their outputs by reflecting on execution feedback [8], [9]. Multi-agent architectures such as ChatDev [10] and MetaGPT [11] have shown that role-specialized agents can collaboratively produce structured software artifacts. These capabilities are now finding traction to Model-Driven Engineering (MDE) [12], with LLMs generating DSL-conforming code from grammar specifications [13], [14] and early prototypes chaining LLM-based generation with formal model validation [15]. Malamas et al. [16] demonstrated LLM-based DSL model generation through prompt engineering, while our prior work [17] indicates that LLM agents can produce syntactically valid and structurally optimized textX grammars from natural language requirements.

Whichever the advancements made in this LLM-enabled

aspect, a critical gap still exists. While grammar generation and DSL model synthesis have received growing attention, the downstream task of automatically producing M2T transformations remains entirely unaddressed. Existing surveys catalog over 70 template-based code generation tools [2], yet all assume templates are manually authored by language engineers. No prior work tackles automatic M2T template generation from a grammar specification, its requirements and sample model instances.

Despite the promise of this approach, several limitations should be acknowledged upfront. The current system supports only textX grammars and Jinja2 as the template language, limiting its immediate generalizability. The evaluation presented in this paper is qualitative and restricted to two DSL domains, meaning broader applicability across diverse grammar paradigms has not yet been empirically established. Additionally, the pipeline's performance is inherently tied to the capabilities of the underlying LLM, and no cross-model comparison has been conducted.

To this end, this paper presents a work-in-progress multi-agent system that automates M2T transformation generation. Given a textX DSL grammar, natural-language requirements and valid sample model instances, the system produces production-ready Jinja2 templates that generate correct, executable target-GPL code. The system decomposes the task into seven stages orchestrated by eleven specialized LLM agents, with deterministic runtime introspection providing ground-truth structural information and three-level deterministic validation ensuring correctness at every step. We showcase the system on two structurally distinct DSL domains.

The remainder of this paper is organized as follows. Section II describes the multi-agent pipeline architecture in detail. Section III demonstrates the system on two structurally contrasting DSL domains. Section IV concludes the paper and outlines future research directions.

II. Multi-Agent M2T Generation Architecture

The proposed pipeline comprises seven stages (Fig. 1) orchestrated by eleven specialized LLM agents alongside deterministic validation components. It requires three types of input: a) a textX grammar file, b) natural-language transformation requirements and, c) sample DSL model instances. Its primary output is a set of production-ready Jinja2 templates. As part of the validation process, it also produces rendered code for each sample model, unit tests that verify semantic correctness and Dockerfiles for sandboxed test execution.

Work is organized in six Stages, as follows: **Stage 0 (Grammar Parsing & Model Introspection).** The grammar is parsed with textX and structurally analyzed to extract all rules, attributes, types and hierarchies. Each sample model is parsed and deeply introspected to produce *runtime object dumps* that record every attribute name and its actual value in the parsed object graph. These dumps provide agents with ground-truth runtime structure, which is critical since attribute names in parsed textX objects are not always directly recoverable from the grammar syntax alone.

Stage 1 (Parallel Semantic Analysis). Three agents analyze the inputs concurrently. The *Metamodel Semantician* interprets the DSL's semantic meaning and rule purposes. The *Requirements Analyst* converts free-form requirements into a structured transformation specification. The *Sample Analyzer* identifies usage patterns and edge cases from the sample models and their object dumps.

Stage 2 (Blueprint Design). The *Blueprint Designer* synthesizes all Stage 1 outputs into an ordered sequence of implementation steps, specifying which template files and macros each step produces and which earlier steps it depends on. This decomposition ensures foundational structures are generated before complex logic.

Stage 3 (Synthetic Test Model Generation). The *Test Model Generator* produces synthetic DSL instances designed to exercise every grammar path. Each is validated against the textX grammar and invalid models are discarded, yielding a test corpus for Stage 4.

Stage 4 (Incremental Generational Loop). For each blueprint step, the *Step Generator* produces Jinja2 templates and the *Template Reviewer* inspects them for structural issues. A three-level deterministic validator then checks Jinja2 syntax, renders templates against synthetic models and verifies target-language correctness. If validation fails, an *Error Analyst* diagnoses root causes while consulting previous fix attempts to avoid cycling and a *Refiner* implements fixes. This loop repeats with escalation to broader restructuring if targeted fixes are insufficient. Accumulated templates advance once all steps complete.

Stage 5 (Final Validation Against Real User Samples). The final templates are validated against the original user-provided samples using the same three-level validation and refinement loop, separating grammar path completeness (tested in Stage 4) from real-world correctness.

Stage 6 (Test & Deployment Artifact Generation). For each sample model with rendered output, the system generates unit tests that assert on observable behavior and Dockerfiles for containerized execution. These artifacts enable safe, sandboxed evaluation where execution results can be collected and fed back into refinement if needed.

Three design principles underpin the pipeline architecture. First, **validation is never entrusted to the LLM** since all correctness judgments are made by deterministic tools. Second, **generation is incremental and isolated** through the blueprint decomposition. Third, **each agent operates at a temperature calibrated to its role**, with analysis agents at 0.2 for precision, generation agents at 0.3–0.4 for balanced creativity and refinement agents at 0.5–0.6 in escalation mode.

III. Showcase on Two DSL Domains

To illustrate the pipeline's applicability across structurally versatile languages, we apply the proposed methodology on two contrasting DSL domains and present representative snippets of the inputs and generated outputs. The two domains were selected to exercise fundamentally different capabilities

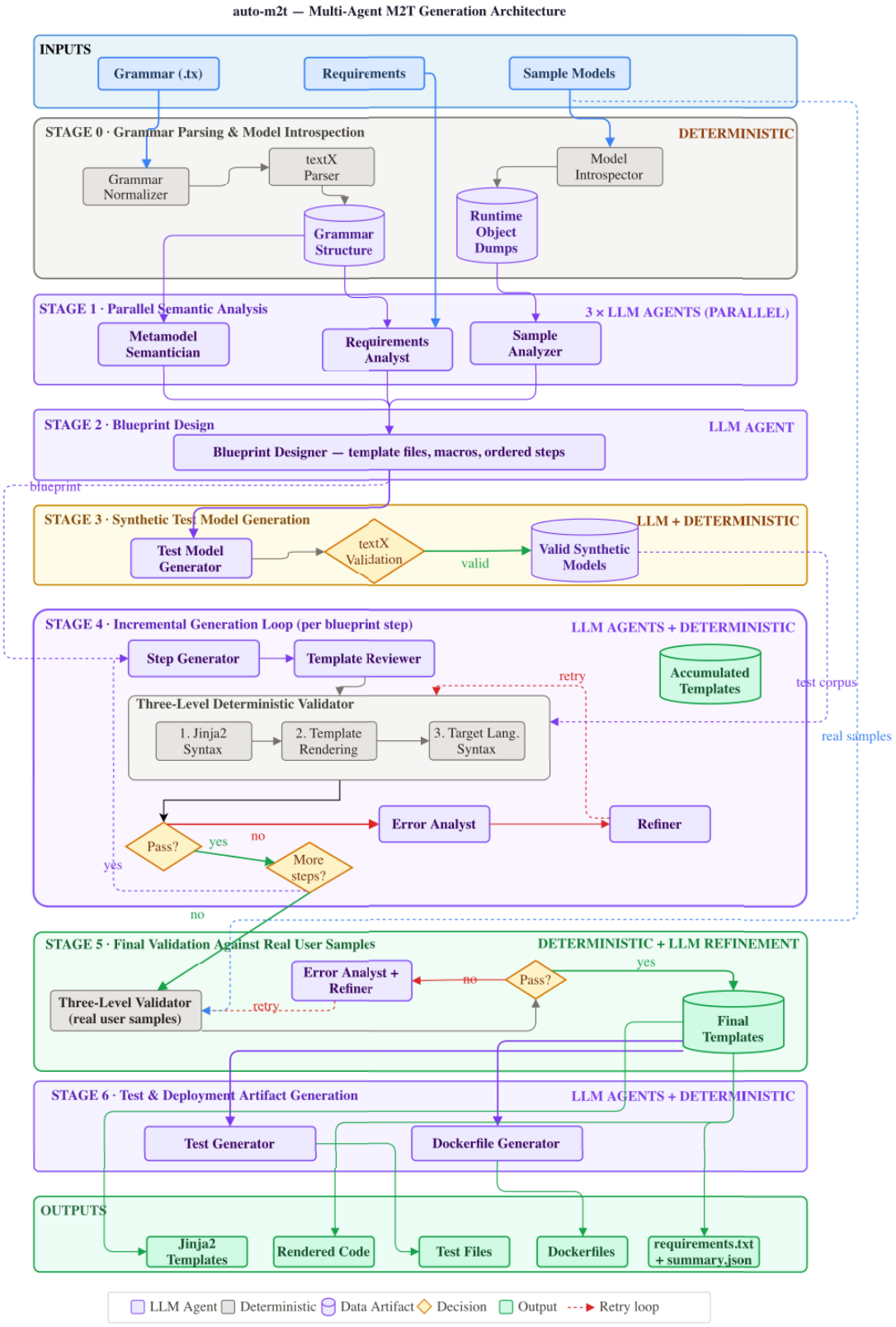

Figure 1. The proposed multi-agent pipeline architecture. Purple nodes represent LLM agents, grey nodes deterministic components, cylinders data artifacts, diamonds decision points and green nodes outputs.

of the system. The calculator DSL tests recursive expression traversal, operator dispatch and paired-list normalization, while the smart home DSL tests polymorphic type switching across dozens of grammar alternatives, optional attribute handling and complex cross-entity references. In both cases, the runtime object dumps produced in Stage 0 are essential for providing agents with accurate structural information about the parsed models. The complete description of the domains, along with the input artifacts and generated outputs for both domains are publicly available and are omitted due to space limitations.[1]

A. Calculator DSL (Recursive, Computational)

The first domain is a calculator DSL with recursive expression trees, operator dispatch, variable bindings and nested arithmetic. The grammar defines 13 rules including abstract alternatives (e.g., `Factor` can be a `Number`, `Variable`, parenthesized `Expression`, or `UnaryOp`), paired-list patterns for operator precedence and recursive nesting. Fig. 2 shows an excerpt from a sample model and Fig. 3 shows the corresponding rendered Python code produced by the generated templates.

```
calc 3.14 * 2
calc -(10 + 5)
var pi = 3.14159
var radius = 5.0
calc pi * radius * radius
```

Figure 2. Calculator DSL sample model (excerpt).

```
variables = {}
print(3.14 * 2.0)
print(-(10.0 + 5.0))
variables['pi'] = 3.14159
variables['radius'] = 5.0
print(variables['pi'] * variables['radius']
      * variables['radius'])
```

Figure 3. Rendered Python output for the calculator model.

The pipeline generates two template files (main and macros) totaling 53 lines of Jinja2 code. These templates correctly handle recursive expression traversal, operator precedence through paired-list iteration and dispatch of unary operators, parenthesized sub-expressions and variable references through macro-based type switching. The pipeline also generated 9 unit tests that execute the rendered code and assert on computed values, all executed successfully.

B. Smart Home IoT DSL (Flat, Declarative)

The second domain is a smart home automation DSL with 30+ grammar rules covering metadata declarations, multi-protocol broker configurations (MQTT, AMQP, Redis) with authentication, sensor and actuator entity definitions with typed attributes and value generators and automation rules with conditions, actions and orchestration dependencies. In contrast to the calculator, this grammar has no recursion and is predominantly declarative, with many optional attributes, polymorphic alternatives (e.g., six action types, six generator types, three broker types) and list-based constructs. Fig. 4 shows an excerpt from a sample model.

```
Entity outdoorTemp
  type: sensor
  topic: "home/outdoor/temperature"
  broker: mqttMain
  freq: 30
  attributes:
    - outdoor_temp : float ->
        gaussian(18.0, 45.0, 4.0)
        with noise gaussian(0.0, 0.2)
    - outdoor_humidity : float ->
        saw(30.0, 95.0, 1.0)
end

Automation climateControl
  condition:
    indoorClimate.indoor_temp > 25.0
    AND outdoorTemp.outdoor_temp > 20.0
  actions:
    - hvacUnit.hvac_power : true
    - hvacUnit.target_temp : 22
  enabled: true
  continuous: true
end
```

Figure 4. Smart Home DSL sample model (excerpt).

For this domain, the pipeline generates three template files (main, macros and helpers) totaling over 700 lines of Jinja2 code. The rendered output for the full sample model exceeds 800 lines of Python, producing a complete simulation framework with polymorphic broker and generator implementations, entity classes, automation logic and a `SmartEnvironment` orchestrator. The pipeline also generated 11 unit tests covering broker configuration, sensor generation, actuator initialization and automation orchestration.

IV. Conclusions and Future Work

This paper presented a multi-agentic methodology that automates the generation of Model-to-Text transformations for textX DSLs. To the best of our knowledge, this is the first system to address automated M2T template generation, a task that has historically required manual authoring by language engineers. The system decomposes the generation task across seven stages and eleven specialized agents, grounds every generation step in three-level deterministic validation and produces not only Jinja2 templates but also rendered code, semantic unit tests and containerized test environments. We demonstrated the system on two structurally contrasting DSL domains, showing that it produces functional templates for both recursive computational grammars and flat declarative grammars.

This work is at an early stage and several limitations remain. The system has been demonstrated qualitatively on two domains but lacks systematic empirical evaluation across a larger and more diverse corpus of DSLs. Template quality has not yet been measured quantitatively in terms of correctness rates,

[1] https://github.com/robotics-4-all/agentic-m2t-results

refinement iteration counts, or comparison against manually authored templates.

Several directions for future work follow naturally. First, we plan a systematic evaluation across DSLs of varying complexity, measuring template generation success rates and refinement convergence behavior. Second, we intend to integrate this methodology with our prior work on automated grammar generation [17] to create a fully end-to-end pipeline from natural language requirements to executable code generation infrastructure. Third, we will evaluate whether the architecture generalizes across different LLM families. Finally, we aim to explore incremental template evolution for DSLs whose grammars change over time.

Acknowledgment

Parts of this research have been supported by the Horizon Europe project Nostradamus (Grant Agreement No 101134888), funded by the European Union, and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH–CREATE–INNOVATE (project code: ΕΚΠΑΡ01-0080512).

References

[1] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[2] E. Syriani, L. Luhunu, and H. Sahraoui, "Systematic mapping study of template-based code generation," *Computer Languages, Systems & Structures*, vol. 52, pp. 43–62, 2018.

[3] I. Dejanović, R. Vaderna, G. Milosavljević, and Ž. Vuković, "Textx: a python tool for domain-specific languages implementation," *Knowledge-based systems*, vol. 115, pp. 1–4, 2017.

[4] C. Negri-Ribalta, R. Geraud-Stewart, A. Sergeeva, and G. Lenzini, "A systematic literature review on the impact of ai models on the security of code generation," *Frontiers in Big Data*, vol. 7, p. 1386720, 2024.

[5] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, and G. Wachsmuth, *DSL engineering-designing, implementing and using domain-specific languages*. M Volter/DSLBook. org, 2013.

[6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[7] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[8] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," *Advances in neural information processing systems*, vol. 36, pp. 8634–8652, 2023.

[9] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang *et al.*, "Self-refine: Iterative refinement with self-feedback," *Advances in neural information processing systems*, vol. 36, pp. 46 534–46 594, 2023.

[10] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong *et al.*, "Chatdev: Communicative agents for software development," in *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, 2024, pp. 15 174–15 186.

[11] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin *et al.*, "Metagpt: Meta programming for a multi-agent collaborative framework," in *The twelfth international conference on learning representations*, 2023.

[12] L. Burgueño, D. Di Ruscio, H. Sahraoui, and M. Wimmer, "Automation in model-driven engineering: A look back, and ahead," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, pp. 1–25, 2025.

[13] B. Wang, Z. Wang, X. Wang, Y. Cao, R. A Saurous, and Y. Kim, "Grammar prompting for domain-specific language generation with large language models," *Advances in Neural Information Processing Systems*, vol. 36, pp. 65 030–65 055, 2023.

[14] V. Lamas, M. R. Luaces, and D. Garcia-Gonzalez, "Dsl-xpert: Llm-driven generic dsl code generation," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, 2024, pp. 16–20.

[15] N. Petrovic, F. Pan, K. Lebioda, V. Zolfaghari, S. Kirchner, N. Purschke, M. A. Khan, V. Vorobev, and A. Knoll, "Synergy of large language model and model driven engineering for automated development of centralized vehicular systems," *arXiv preprint arXiv:2404.05508*, 2024.

[16] N. Malamas, E. Tsardoulias, K. Panayiotou, and A. L. Symeonidis, "Toward efficient vibe coding: An llm-based agent for low-code software development," *Journal of Computer Languages*, p. 101367, 2025.

[17] T. Tsampouris, E. Tsardoulias, K. Panayiotou, and A. L. Symeonidis, "Rq2dsl: An ai agent for automated domain-specific language grammar generation from requirements," in *2025 IEEE 37th International Conference on Tools with Artificial Intelligence (ICTAI)*, 2025, pp. 1300–1307.