Proceedings of the 20th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 531–542
DOI: 10.15439/2025F5613
ISSN 2300-5963 ACSIS, Vol. 43

Machine-Readable by Design: Language Specifications as the Key to Integrating LLMs into Industrial Tools

Alexander Fischer*, Louis Burk*, Ramin Tavakoli Kolagari, Uwe Wienkop
0009-0001-1737-7395
0009-0003-0727-6456
0000-0002-7470-3767
0009-0000-4487-2458
Nuremberg Institute of Technology,
Faculty of Computer Science,
Kesslerplatz 12,
90489 Nuremberg, Germany
Email: {a.fischer, louis.burk, ramin.tavakolikolagari, uwe.wienkop}@th-nuernberg.de

***Abstract*—We propose a meta-language-based approach enabling Large Language Models (LLMs) to reliably generate structured, machine-readable artifacts referred to as Meta-Language-defined Structures (MLDS) adapted to domain requirements, without adhering strictly to standard formats like JSON or XML. By embedding explicit schema instructions within prompts, we evaluated the method across diverse use cases, including automated Virtual Reality environment generation and automotive security modeling. Our experiments demonstrate that the meta-language approach significantly improves LLM-generated structure compliance, with an 88% validation rate across 132 test scenarios. Compared to traditional methods using LangChain and Pydantic, our MLDS method reduces setup complexity by approximately 80%, despite a marginally higher error rate. Furthermore, the MLDS artifacts produced were easily editable, enabling rapid iterative refinement. This flexibility greatly alleviates the "blank page syndrome" by providing structured initial artifacts suitable for immediate use or further human enhancement, making our approach highly practical for rapid prototyping and integration into complex industrial workflows.**

I. Introduction

INDUSTRIAL tools have become increasingly powerful and can now handle vast amounts of data. However, as their range of functions continues to grow, operating these tools often overshadows the core task of content or problem management. This situation is further exacerbated by the common "blank page syndrome," [1], a common phenomenon in complex design or modeling tasks, where the absence of any initial structure or guidance leads to hesitation, inefficiency, or even creative paralysis. In such cases, users face difficulty starting from scratch, especially when formal specifications or architectural expertise are required. This not only slows down the initial design process but may also prevent domain experts from effectively contributing to model-driven workflows. Consequently, the focus moves away from the substantive content toward the tool itself, lowering productivity and widening the gap between content and tooling

At the same time, LLMs offer new opportunities by helping domain experts generate creative impulses and mitigate the "blank page syndrome." These assisting tools have evolved rapidly—especially through the integration of AI capabilities—that they have become even more complex, making proper integration of LLMs into content creation workflows difficult. In particular, when domain-specific languages are used that deviate from common formats like JSON or XML, producing reliably machine-readable artifacts remains a significant challenge.

Our approach addresses this issue by defining a language specification (meta-language) that is adapted to the requirements of each domain, and which likewise does not rely on existing standard formats. This enables us to capture the structural differences and specific relationships of a domain. By combining LLMs with this meta-language, it becomes possible to produce customized, machine-readable artifacts without requiring domain experts to possess extensive knowledge of architectural or formal specifications.

To clarify our terminology, we define two key concepts central to this paper. A Meta-Language-defined Structure (MLDS) refers to a structured, machine-readable artifact that is generated based on natural language input, using a set of schema rules defined in a meta-language which we call Meta-Language-defined Structure Instructions (MLDSI) . The meta-language itself is a domain-specific specification that defines the structure of the expected artifact. It serves as a structural blueprint that guides the generation of MLDS artifacts and ensures that the resulting outputs conform to the expected semantics of the target domain.

Although recent approaches have explored how LLMs can be guided to generate structured outputs, they typically rely on rigid, domain-agnostic formats such as JSON or XML. These

*Both authors contributed equally to this work.

solutions offer limited semantic adaptability and often require specialized tools or substantial manual intervention. This highlights a persistent research gap: the lack of a lightweight, domain-adaptable framework for generating structured outputs from natural language in a way that is both semantically meaningful and technically accessible. Our work addresses this gap by introducing the MLDS approach, which enables domain experts to produce customized, machine-readable content with minimal setup effort and without requiring expertise in schema tooling or formal languages.

Whether this meta-language approach is viable and beneficial is examined by focusing on the following central question:

What conditions must be met for meta-languages to function as a bridging element between natural content descriptions and machine-readable structures, enabling the integration of LLMs into industrial toolchains?

This central question is addressed through the following three sub-questions:

- **R1:** Which methods can be used to incorporate meta-language definitions into a model?
- **R2:** What criteria can be considered to validate the output?
- **R3:** To what extent can reference examples be used to adapt the output to specific needs?

To ensure transparency and facilitate reproducibility, all relevant research artifacts, including evaluation results, domain-specific meta-language definitions and generated Meta-Language-defined Structures (MLDS) artifacts, have been made publicly available in our GitHub repository [2].

In the remainder of this paper, we present the technical background and motivation for our meta-language approach in Section II. An overview of related research follows in Section III, which positions our work within the broader field. Next, Section IV introduces our proposed methodology and the design of the meta-language. We then provide a comprehensive evaluation in Section V, showcasing its applicability across various domains. Section VI discusses our key findings and highlights potential limitations, while Section VII outlines promising avenues for future research. Finally, Section VIII concludes the paper with a summary of our main contributions.

II. Background and Motivation

In many technical and economic fields, there are specialized challenges that can only be successfully addressed with the help of experts. This is especially true for the development and implementation of complex software and system architectures [3]. A classic example is the collaboration between a person with architectural expertise (Architectural Domain Expert, ADE) and someone with substantive or specialized expertise (Content Domain Expert, CDE). While an architect generally possesses deep knowledge of a specific, clearly defined domain, the transfer of architectural concepts to other domains is often difficult and time-consuming. This hurdle complicates efficient collaboration and leads to practical inefficiencies when the same fundamental principle is applied in different fields.

The goal, therefore, is to allow CDEs in a given domain to create structured artifacts directly processable by machines, even without extensive knowledge of software or system architecture. These artifacts, which we refer to as MLDS, make it possible to model specialized requirements in detail. By using MLDS, ideas and requirements expressed in natural language are transformed into formal structures that adhere to certain rules and constraints. Consequently, the ADE's role shifts to a higher-level task: defining a structural specification (the "meta-language") that ensures the resulting MLDS meet the desired quality and formatting requirements.

Currently, structured outputs from natural language models, such as JSON-based responses, typically require specialized tools and significant manual intervention. Tools like Pydantic and LangChain have emerged to facilitate structured outputs by validating and enforcing schemas, thereby ensuring JSON structures are accurately generated and adhered to [4], [5]. However, effective use of these tools demands substantial technical expertise and familiarity with their underlying frameworks [6]. Moreover, these tools require ongoing adaptation to handle differences in how newer LLM architectures generate structured output, which introduces additional maintenance overhead and complexity.

Furthermore, not all LLMs inherently support structured output generation effectively. This limitation restricts their applicability to scenarios requiring strictly formatted outputs [7], [8]. For models that do support structured outputs, outputs often need post-processing and manual corrections due to subtle inconsistencies, ambiguities, or deviations from predefined structures, increasing complexity and effort [9].

To circumvent these challenges, our approach introduces MLDS Instructions—structured prompts explicitly designed to guide LLMs in generating precisely defined structured artifacts. LLMs are powerful models that can interpret and produce natural language [8]. However, a key issue arises because LLMs generally do not generate precise, user-defined structures unless these are already standard formats. This is precisely where our solution comes into play: with so-called "MLDS instructions," we provide the LLM with a meta-language characterized by a clear structure and defined relationships, without relying on existing standard formats.

The MLDS instructions inform the model about both the desired output format and the rules needed to produce it. This distinguishes our approach from classic procedures that require extensive fine-tuning of the model or force the output into a fixed format. Through the meta-language and its meta-levels, complex relationships within the MLDS can also be represented. In this way, MLDS builds a bridge between the conceptual problem statements formulated by CDE and a strictly formatted output that can be processed by machines.

Our approach is inspired by established metamodeling architectures, such as those found in the Meta-Object Facility (MOF) [10] and the Unified Modeling Language (UML) [11], and is structured into four layers:

- At the M0 level, the focus is on concrete data or instance-level elements—the real-world objects or executions that

result from applying the MLDS. This level is where the model "comes to life" and is directly processed by software systems.

- The M1 level, known as the model level, is where the MLDS artifacts reside. These artifacts are formal representations that capture the concrete, structured requirements of a domain, serving as blueprints derived from the natural language input provided by domain experts.
- The M2 level, or metamodel level, contains the meta-language. This layer defines the rules, constraints, and abstract syntax and semantics required for constructing valid MLDS artifacts. The MLDS instructions used to guide the LLM in generating structured outputs are derived from this meta-language.
- At the highest M3 level, the meta-meta model is defined. This layer establishes the fundamental constructs and abstractions upon which the meta-language is built. Although users do not normally interact directly with this level, it is necessary to ensure that the meta-language (M2) is consistent, which in turn ensures that the rules governing the MLDS (M1) are sound.

By organizing our approach into these four interconnected layers, M0 for instances, M1 for MLDS artifacts, M2 for the meta-language, and M3 for the meta-meta model, we create a structured, hierarchical approach that ensures domain-specific requirements are transformed consistently into machine-processable formats. While content domain experts focus on formulating concrete requirements at the M1 level, the meta-language at the M2 level ensures that these requirements are implemented within a consistent, formal scope. The M3 layer supports the whole system by providing a basis for the meta-language.

In this paper, we will illustrate the functionality of our MLDS instructions and the resulting MLDS outputs using two application scenarios: (1) Automotive Software Security and (2) Virtual Reality in business applications. By "Automotive Software Security," we mean the specific security and reliability requirements in vehicle software engineering, while "Virtual Reality in business applications" refers to immersive and interactive digital learning and work environments used within companies. These examples highlight how varied the domains can be in which experts (CDE)—for instance, from the security or VR fields—wish to express their specialized requirements in MLDS, without being trained as architects in the sense of software system architecture.

An illustrative use case demonstrates this: a VR specialist wants to model a virtual learning space for internal corporate training. Although this expert (CDE) has a clear understanding of the requirements for the learning space, they lack the experience to describe it in a precise architectural form. The existing tool for creating VR environments requires a clearly defined structure in the form of MLDS. Until now, it was the architect's job to convert these specialized requirements into the corresponding formal structure. With our approach, a natural language description of the desired learning space and the MLDS instructions satisfy for an LLM to generate the appropriate MLDS. Thanks to the MLDS instructions, the LLM knows exactly how the resulting artifact should be structured. As a result, the CDE's specialized requirements are transformed into a formally processable data format, without the need for the CDE to possess deeper knowledge of software architecture or formal languages.

III. Related Work

In our approach, we propose a meta-language framework that allows the creation of formally defined structures from purely natural-language descriptions *without* relying on standard formats like JSON or UML, nor requiring extensive fine-tuning.

A variety of prior studies already employ LLMs for structured output generation, yet they pursue either *more specialized* or *fundamentally different* techniques. For instance, Wang et al. [12] demonstrate a *grammar prompting* procedure, where LLMs are constrained by a Backus–Naur Form (BNF) grammar to produce strictly formal outputs. While this approach works very well for specific languages that fit into a context-free grammar, it often requires explicit modifications to the decoding process (such as restricting token choices) or relies on well-defined, existing grammar rules. By contrast, our focus is on designing a *flexible meta-language* that need not be encoded in BNF.

A related but narrower method is illustrated by Scholak et al. [13] with their *PICARD* approach, where an autoregressive model is incrementally validated against the desired language (e. g., SQL). Similar to grammar prompting, token decoding is continuously constrained during generation. However, in contrast to our meta-language paradigm, PICARD involves *invasive modifications* to the decoding algorithm. We rely instead on clearly defined meta-language instructions and LLM prompt processing, yielding more latitude for *language structures*.

Further research in retrieval augmentation or fine-tuning comes from Bassamzadeh and Methani [14], comparing whether structured code is best produced via specialized fine-tuning or optimized prompting techniques. While these findings are valuable for reducing errors and hallucinations, they typically assume a *known* format (e. g. JSON/YAML). In our method, no specialized model training is needed; we *generate* all target structures solely via prompt-based instructions and meta-language rules.

In the context of model-driven software development, Netz et al. [15] show how UML-like class diagrams (CD4A) can be generated by GPT-3.5/GPT-4 and transformed directly into complete web applications. However, they rely on *UML notations* and existing generator frameworks (e. g. MontiGem). This differs from our approach, which does *not* require any model-driven development toolchain or standardized notation. Instead, we allow arbitrary domain-specific structures—for instance, automotive security or VR scenarios—specified with a freely definable meta-language.

Key Contribution of Our Approach: While the mentioned studies employ various forms of grammar-based decoding,

fine-tuning, or predefined formats, our method provides a universal meta-language definition. Thus, architects or tool developers can prescribe any target structure (from security modeling to 3D scenes) at the meta-level so that an LLM subsequently outputs *machine-readable* artifacts in exactly that format–without training, without invasive decoder modifications, and without restricting to JSON/UML. Because our approach does not fix a specific output format, it supports both machine-driven and human refinements of the same structure, fostering an iterative co-creation process. Domain experts bring in new scenario descriptions, the LLM generates or extends the MLDS, and humans can then fine-tune or correct the result in the meta-language immediately. This openness, combined with a clear metamodel architecture, constitutes the central distinguishing feature of our work.

IV. Methodology

In this Section, we explain our proposed approach (Section II) in detail, showing how we use our custom meta-language together with the MLDS framework to solve the problem. We begin by outlining how the meta-language is derived from the tool's requirements, providing a starting point for the process. Next, we describe how to structure the descriptions of tasks and scenarios, setting up a foundation for generating the MLDS using LLMs. The complete procedure is shown in Figure 1.

To keep the process transparent, we provide examples that demonstrate how prompts are combined with the meta-language. A notable feature of our approach is that it allows users to influence different parts of the MLDS in various ways, making the method adaptable. After fully explaining the MLDS, we show how it can be directly imported into tools, and we conclude by demonstrating how this method addresses the blank page syndrome.

A. Conception of the Meta-Language

The core of our approach lies in developing a meta-language that incorporates the specific requirements of a domain or tool and organizes them into a structured form that is not bound to common standards. In contrast to JSON or XML, the meta-language may deliberately allow specialized object types, relationships, and value ranges that thoroughly reflect the peculiarities of a domain. The meta-language thus pursues two primary objectives: on the one hand, it provides the LLM with explicit guidelines on which structures, attributes, and relationships should appear in the generated MLDS. On the other hand, it is designed to be easily understandable and extendable by human domain experts, allowing them to add new elements or rules when necessary.

1) Reasons for a Meta-Language: One of the main challenges in automatically generating structured output is that LLMs initially lack any knowledge of custom formats. Although models can be adapted to standard formats, they often reach their limits when specialized domains require formats that differ from these standards or go beyond their semantics. Many advanced tools employ proprietary definitions that cannot readily be mapped onto common standards. Our meta-language addresses this gap by:

- Allowing freely definable object and relationship types to represent specific domain concepts.
- Defining clearly delimited value ranges (e.g., discrete selection options, predefined numerical ranges, or boolean variables) for each attribute, so that the LLM is not forced to "improvise" unnecessarily but instead chooses from a set of valid options.
- Providing structural guidelines that define the "skeleton" of an MLDS, ensuring that the generated output follows a meaningful and machine-readable architecture.

In this way, it is not necessary to extensively fine-tune the LLM. Instead, we present the model with precise rules via the meta-language and corresponding prompt instructions, which it follows when generating the MLDS.

2) Analysis of the Required Input: Before implementing the meta-language, it is necessary to determine which information a particular tool or domain actually needs. A standard example is a VR development tool that manages various object types (e.g., *rooms*, *avatars*, or *interactions*), each with specific properties (size, access rights, animation logic, etc.). These aspects are collected, categorized, and eventually transferred into the meta-language. On this basis, one can also estimate the relationships between objects—an *avatar*, for instance, may be assigned to a *room*, and an *interaction* may be tied to an *object*.

3) Extracting Relevant Objects and Properties: Once the domain analysis is complete, relevant objects are defined. An *object* corresponds to a central concept in the domain (e.g., learning environment, avatar, security component). A *learning environment*, for example, might have properties such as *name*, *maximum number of users*, or *room configuration*. Each property is assigned a corresponding value range (e.g., a fixed set of room types, numerical ranges for the number of users) to ensure consistency and comprehensibility.

4) Defining Relations: Another key to understanding and modeling complex relationships is specifying the links between objects. For instance, a *human actor* object at a certain *resilience level* could refer to an *attack* object, or multiple *vehicle features* could form a single item. These relationships may be configured hierarchically within the meta-language or refer to the same level, each implying different semantic connections. Such a relational network dictates the subsequent structure of the MLDS, guiding the LLM in how individual elements should interrelate.

5) Domain-Specific Meta-Language Instructions: In the following, we present three excerpts from a possible domain-specific meta-language design, illustrating its key concepts. Although the example involves a 3D scene for VR applications, the same underlying mechanisms can be applied to virtually any domain.

a) Defining the Scene Structure: The first step is to establish a top-level object that acts as a global container for all subsequent elements (Listing 1). In the VR domain, we call

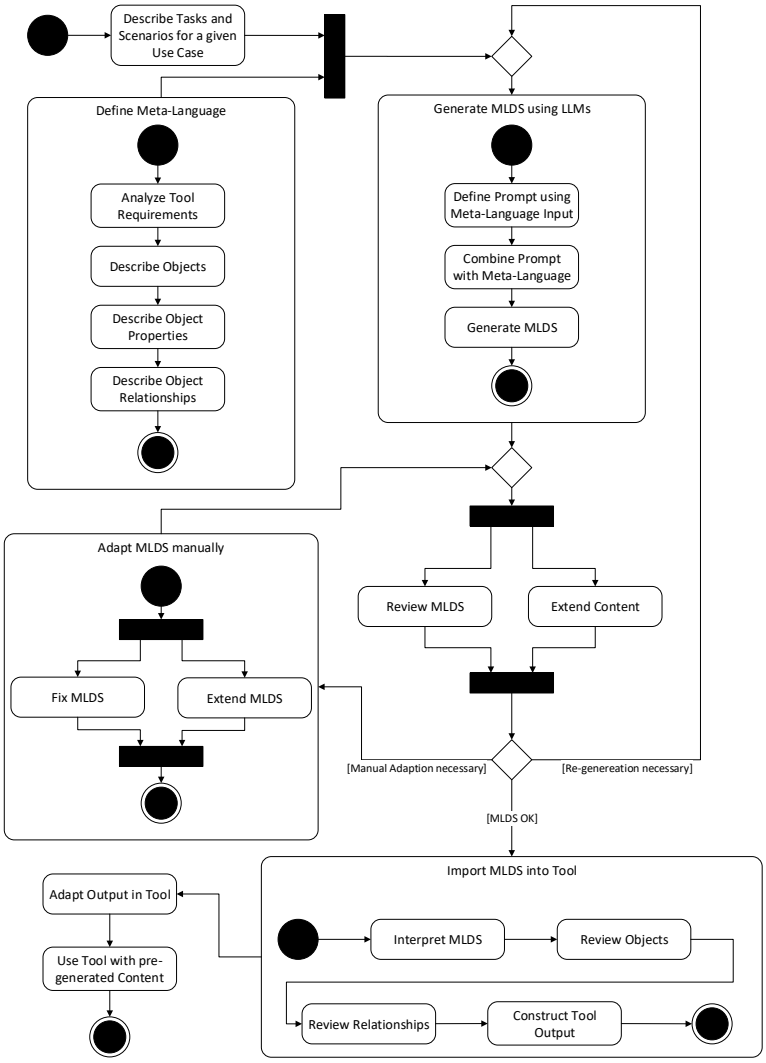

Fig. 1. Workflow for generating and refining MLDS from natural language input, combining meta-language specification, LLM-based structure generation, and iterative human-AI co-editing for tool integration.

this object `scene`, which holds basic environmental details and a list of objects.

```
{
  "scene":
  {
    "sceneName": "VR Training Room",
    "environment":
    {
      "type": "indoor",
      "dimensions": { "width": 10.0, "
          height": 3.0, "depth": 15.0 }
    },
    "objects":
    [
      /* Objects go here */
    ]
  }
}
```

Listing 1. Definition of the Scene Structure

In other domains, what is here called `"scene"` could be labeled `"system"` or `"project"` and might include completely different properties (e.g., `systemCategory` or `budget`). Nevertheless, the principle remains the same: a clearly defined parent object provides the framework in which all detailed elements and their attributes are nested.

b) Modeling Objects and Their Relationships: Each object has a unique `objectId` and is characterized by properties (e.g., `objectType`, `position`, `rotation`) as well as optional relational information. Listing 2 shows how to place an object (a table) relative to another object using `relativePositioning`.

```
{
  "objectId": "object_01",
  "objectType": "table",
  "position": { "x": 2, "y": 0, "z": 3 },
  "rotation": { "x": 0, "y": 45, "z": 0
      },
  "dimensions": { "width": 1.5, "height":
       0.75, "depth": 1.0 },
  "relativePositioning":
  {
    "referenceObject": "object_02",
    "relation": "next_to"
  },
  "offset": { "x": 0.2, "y": 0, "z": 0 }
}
```

Listing 2. Objects with Relative Positioning

In other domains, objects may have entirely different properties (for instance, `memoryLimit` or `location` in a cloud application). Similarly, a relationship like `next_to` could be replaced by something like `depends_on` if you want to describe software dependencies. By specifying such structures in a meta-language, you ensure that the LLM knows exactly which attributes exist and which values are permissible.

c) Hierarchical Nesting of Objects: To describe more complex scenarios, objects can themselves have *children* (Listing 3). This enables a hierarchical organization with subordinate elements.

```
{
  "objectId": "object_02",
  "objectType": "shelf",
  "relativePositioning":
  {
    "referenceObject": "object_03",
    "relation": "on_top_of"
  },
  "children":
  [
    {
      "objectId": "object_02_child_01",
      "objectType": "box",
      "relativePositioning": {
        "referenceObject": "object_02",
        "relation": "next_to"
      },
      "offset": { "x": 0.1, "y": 0, "z": 0
          }
    }
  ]
}
```

Listing 3. Object Hierarchies Using Children

This type of `children` property could represent, for example, submodules within a larger module in a software architecture or smaller processes within a bigger industrial workflow. Likewise, a relationship such as `"on_top_of"` is just an example of a hierarchical link; in other contexts, it might be `"contained_in"` or `"extends"`.

B. Generation of the MLDS

Once the meta-language is established, natural language descriptions or scenarios adapted to a specific use case can be created. As illustrated in Figure 1, the meta-language is developed as a separate module that, once finalized, can be reused for all subsequent generations.

In the most compact workflow, we concatenate the meta-language and the scenario into one composite prompt and pass it to a pretrained LLM via its API. Because the model already possesses broad world knowledge, it needs no task-specific fine-tuning; instead, the embedded meta-language rules channel the model towards a syntactically correct MLDS while the scenario text supplies the domain content. A prompt segment might therefore read:

```
<<META LANGUAGE>>
object room { width:float height:float
    depth:float ... }
...
<<SCENARIO>>
Create a training classroom with 20
    student desks, one teacher desk,
and a projector mounted to the ceiling.
```

Listing 4. Excerpt of a single-shot prompt that merges meta-language and scenario

For projects that demand stricter traceability or staged quality control, we separate the process into two calls. The first call feeds nothing but the meta-language to the LLM and receives back a condensed, token-optimised representation, the second call injects the scenario plus the condensed rules. This split makes it easy to inspect intermediate artifacts, attach unit tests to the generated schema, or swap in a different scenario without retransmitting an unchanged meta-language, an advantage when prompt length becomes a bottleneck.

Regardless of the single- or multi-step variant, the raw MLDS is fed through an automatic validator that flags missing attributes, illegal value ranges or broken references. Only after the MLDS passes this gatekeeper is it handed to the target tool, guaranteeing that the downstream pipeline never receives malformed input.

C. Adaption of the MLDS

In practice, generating the initial MLDS is only the first step in its full utilization. Because the MLDS can be produced in a format that is easily readable by humans, domain experts can readily validate its contents without a steep learning curve. Once generated, the MLDS may be scrutinized for correctness, and any missing information can be added. As illustrated in Figure 1, these iterative improvements can be carried out by the LLM itself, which can respond to feedback or correction prompts. At the same time, domain experts remain free to make manual adjustments, ensuring a high degree of precision where AI suggestions may not capture every nuance. This dual approach of automated and manual editing fosters a co-creative process, merging AI-driven speed and consistency with human expertise. Moreover, the MLDS's structured nature facilitates intuitive graphical interfaces, allowing content specialists to modify or enrich the MLDS without delving into the underlying code. By accommodating both fully automated and personalized interventions, the workflow remains flexible, transparent, and accessible to all involved stakeholders.

D. Integration into the Tool Environment

Once the finalized version of the MLDS is ready, it can be loaded directly into the target tool. The tool interprets the generated structure and creates the required output, representing all defined objects and their relationships according to the MLDS specifications. Additional modifications and enhancements can then be carried out directly within the tool, giving users the flexibility to refine or extend the setup as needed. This ensures that the produced content is fully leveraged, while minimizing both development effort and potential errors.

V. Evaluation

A. Objective and Approach

To demonstrate the developed Meta-Language approach, we selected two distinct domains: (1) the automated generation of Virtual Reality (VR) worlds and (2) the Security Modeling of embedded systems in the automotive context. The aim is to show that from a defined meta-language, a domain-specific MLDS can be generated that consistently follows the given specification. Furthermore, we highlight how this method mitigates the so-called "blank page syndrome," where a completely unstructured starting point often leads to high efforts in initial model or environment creation. By providing a machine-readable foundation, users can directly refine and adapt the generated MLDS artifacts.

B. Use Case: Automated VR World Generation

The first domain focuses on automatically generating VR worlds in Unity. A structured description is created to form a basic layout of the virtual environment. Designers provide a natural-language prompt (for instance: "Generate a classroom for 20 students"), which is then transformed into a JSON-based MLDS using our meta-language. Unity processes the resulting MLDS to position objects and define relationships in the 3D scene. Initially, placeholders (e.g., cubes) are placed to help visualize object arrangements. Designers can later replace these placeholders with final assets, ensuring a quick start for prototyping. This effectively reduces the blank page syndrome since each new scene already has a predefined structure.

Over 200 test scenarios (see Section IX) were conducted, demonstrating that LLMs are able to generate coherent object hierarchies and spatial relationships. For example, Listing 5 shows a parent-child relationship where a desk acts as the parent object, and a chair is placed relative to the desk. Positioning attributes and relational references (e.g., `in_front_of_positive_z`) are used to automate the placement in the 3D environment.

```
{
  "objectId": "studentDesk_r0_c0",
  "objectType": "student_desk",
  "assetName": "student_desk_asset",
  "position": {"x": -6, "y": 0.5, "z":
      8},
  "dimensions": {"width": 2, "height": 1
      ,"depth": 2},
  "group": "furniture",
  "children":
  [
    {
      "objectId": "studentChair_r0_c0",
      "objectType": "chair",
      "assetName": "student_chair_asset",
      "relativePositioning": {
        "referenceObject": "
            studentDesk_r0_c0",
        "relation": "in_front_of_positive_z
            ",
        "distance": 1
      },
      "dimensions": {"width": 0.5, "height
          ": 0.5, "depth": 0.75},
      "group": "furniture"
    }
  ]
}
```

Listing 5. VR Parent-Child Relation

By incorporating such parent-child relations, users benefit from an iterative co-creation process: the generated foundation can be refined either through extended prompts (in natural language) or direct modifications of the MLDS output.

C. Use Case: Security Modeling in Automotive

The second use case addresses Security Modeling for embedded systems in the automotive context. Building upon the metamodel proposed in [16], we define a meta-language for an MLDS that helps create a foundational security model. Security experts or systems engineers can outline potential threats or vulnerabilities in natural language (e.g., "unsecured wireless communication" or "man-in-the-middle attack"). An LLM then translates these descriptions into the MLDS, which is directly processable by a security modeling tool. Such a tool subsequently calculates risk levels or highlights vulnerabilities. Manual adjustments remain possible at any stage, ensuring a flexible co-creation process.

Listing 6 illustrates a JSON-based MLDS snippet with typical attributes for attack characterization, such as `AttackComplexity` and `UserInteraction`. By assigning initial values to these fields, the LLM provides a starting point for further risk assessments.

```
{
  "Name": "Man-in-the-middle:
      intercepting and manipulating
      wireless traffic",
  "AccessRequired": "Network",
  "AttackComplexity": "(H)igh",
  "PrivilegesRequired": "(N)one",
  "Urgency": "(L)ow",
  "UserInteraction": "(N)one",
  "AvailabilityImpact": "(L)ow",
  "ConfidentialityImpact": "(H)igh",
  "IntegrityImpact": "(L)ow",
  "SafetyRelevance": "(H)igh",
  "vulnerabilities":
  [
    {
      "Name": "Unsecured Wireless
          Communication Protocol"
```

```
16        }
17    ]
18 }
```

Listing 6. Automotive Security Attack Representation

Since risk or vulnerability scores are computed within the modeling tool itself, attributes assigned by the LLM can be refined later. This aligns with the cooperative design principle, where the model's first version is automatically generated and then iteratively improved.

D. Results and Insights

In both domains, we combined the language specification with natural-language prompts across several LLMs (including GPT 4o mini, GPT o3 mini, DeepSeek-R1, and DeepSeek-R1 14B running locally). Most generated MLDS files were syntactically correct and machine-readable. Occasional issues arose mainly from incomplete API responses or minor inaccuracies in the meta-language. Notably, even complex problem descriptions in natural language led to well-structured and valid MLDS artifacts. Key findings are:

- **High Machine-Readability:** The LLM outputs conform closely to the defined meta-language, minimizing parsing errors.
- **Blank Page Syndrome Mitigation:** In both VR world creation and security modeling, users start with a usable blueprint rather than a completely empty project.
- **Ease of Extension:** Generated models or VR scenes can be expanded using extended prompts or direct modifications.
- **Domain-Agnostic Feasibility:** The approach works effectively in multiple domains, provided a domain-specific meta-language is well-defined—for example, structuring content in education, automating routines in smart environments, modeling workflows in business, formalizing scenarios in healthcare, or generating logic in games and legal contexts.

We further evaluated our approach by systematically analyzing 132 generated MLDS files to quantify adherence to the prescribed meta-language schema. This analysis was automated through a custom Python-based parser validating each file against the defined schema.

E. Evaluation Setup

The evaluation dataset consisted of files categorized into two complexity levels: *easy* (n=72) and *medium* (n=60), selected from the VR dataset based on the required content complexity. The dataset aimed to enable an end-to-end pipeline for generating virtual environments. Furthermore, each complexity level had two variations: one employing semantic validators (*withValidator*) and one without validators (*withoutValidator*). These validators were applied downstream of the MLDS translation process and specifically assessed semantic correctness, such as checking for overlapping objects, rather than syntactic accuracy. As our analysis indicated no significant differences in syntactic correctness between these variations, we decided to merge them into a single unified dataset for evaluation purposes. Each file was evaluated based on structural validity, completeness of schema components, and the presence of specific parsing errors.

We defined the metrics as follows:

$$V = \sum \text{valid files}, \quad I = N - V \tag{1}$$

$$\text{Validity Rate } (\%) = \frac{V}{N} \times 100 \tag{2}$$

$$\text{Presence Rate } (\%) = \frac{\text{Files containing schema part}}{N} \times 100 \tag{3}$$

where N is the total number of evaluated files, V is the number of valid (correctly structured) files, and I is the number of invalid (incorrect) files.

F. Quantitative Results

Out of $N = 132$ files, $V = 116$ (87.9%) were valid, while $I = 16$ (12.1%) contained parsing errors (Fig. 2). Errors primarily arose from JSON parsing issues due to incorrect string formatting.

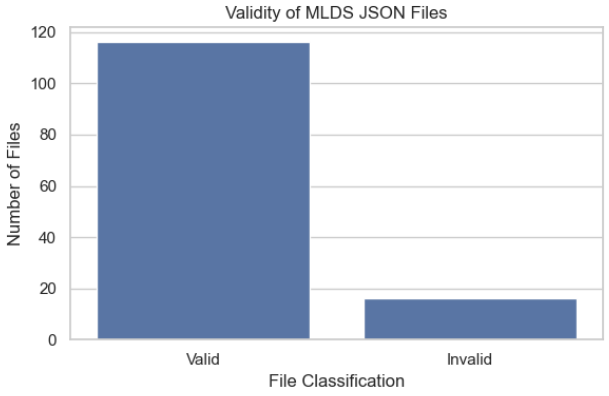

Fig. 2. Distribution of valid vs. invalid MLDS JSON files.

Out of the 16 invalid files, 15 exhibited a parsing error indicated by the message 'str' object has no attribute 'get'. This error occurred because the JSON parser expected a structured object but encountered a string instead. Only one file showed a different issue, specifically missing essential schema components: *environment* and *sceneName*.

Table I summarizes the presence rates of schema parts. Most elements showed high presence rates, indicating strong schema adherence, except for optional fields like `env.background`, which appeared less frequently (72.7%).

TABLE I
SCHEMA PART PRESENCE SUMMARY

Schema Part	Presence	Missing files
sceneName	87.9%	16
environment	87.9%	16
env.type	87.9%	16
env.dimensions	87.9%	16
env.lighting	85.6%	19
env.background	72.7%	36
objectGroups	87.9%	16
objects	88.6%	15
obj.objectId	87.9%	16
obj.objectType	87.1%	17
obj.position	87.9%	16
obj.rotation	87.1%	17
obj.dimensions	87.1%	17

Since we did not explicitly request the inclusion of background components and instead relied on the LLM's initiative to incorporate them as part of the MLDS, the observed number of omissions was anticipated. This could be improved by explicitly addressing background components within the input prompt, thus providing clearer guidance to the LLM.

Figure 3 visually presents these results and highlights areas for possible improvements.

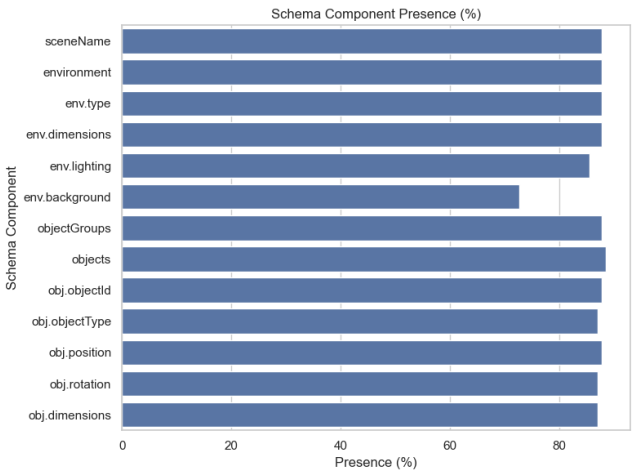

Fig. 3. Presence rates of schema parts in MLDS files.

Figures 4 and 5 highlight the complexity of generated files. Most scenarios had a moderate number of objects, suitable for realistic industrial applications, while nesting depth indicated well-structured JSON.

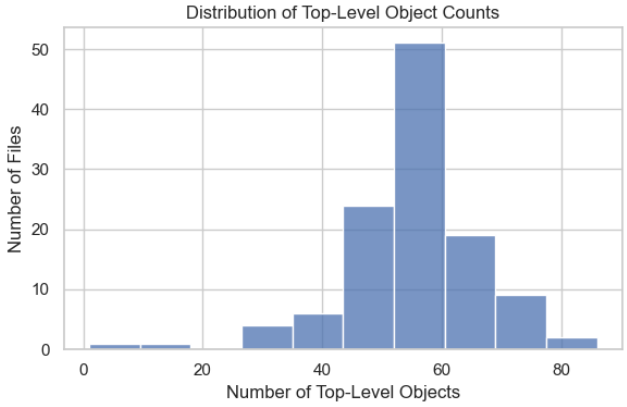

Fig. 4. Distribution of object counts per MLDS file.

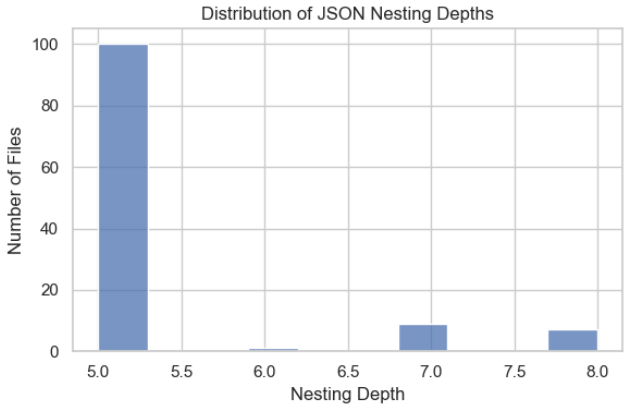

Fig. 5. Distribution of nesting depths in MLDS files.

The nesting depth was computed with the following Python function:

```python
def compute_depth(obj, depth=0):
    if isinstance(obj, dict):
        return max([compute_depth(v, depth
            +1) for v in obj.values()] or [
            depth])
    if isinstance(obj, list):
        return max([compute_depth(i, depth
            +1) for i in obj] or [depth])
    return depth
```

Listing 7. Computing JSON nesting depth

G. Comparison with Traditional Approaches

We compared our MLDS approach directly against a traditional structured-output pipeline using LangChain and Pydantic [5], [4]. The implementation of the traditional approach took approximately five times longer than preparing our MLDS instructions, primarily due to the complexity and steep learning curve associated with LangChain and Pydantic. While the traditional approach consistently produced compilable results when successfully executed, approximately 7%

of pipeline runs failed entirely due to validation errors or incomplete responses. In contrast, our MLDS-based approach resulted in a slightly higher error rate (around 6% more errors), but significantly reduced the overall complexity, effort, and required expertise. Therefore, especially for rapid prototyping, iterative adjustments, or when specific expertise in Pydantic and LangChain is not readily available, our MLDS approach offers a clear practical advantage.

H. Implications

Explicitly embedding meta-language schema instructions within prompts enhances structural compliance and machine readability. This efficiency allows rapid prototyping and seamless integration into diverse industrial workflows, underscoring the practical effectiveness of our approach.

VI. Discussion

In this Section, we revisit the insights gained from our evaluation in light of the research questions posed in Section I. We then compare our approach to existing work (see Section III) and critically examine potential limitations as well as threats to validity.

A. Interpretation of Evaluation Results

1) Research Question R1: Methods for Incorporating Meta-Language Definitions into Models: Our experiments in the domains of Virtual Reality (VR) world generation and Automotive Security indicate that a well-specified meta-language can effectively convey structural and relational constraints to LLMs. The results suggest that this method scales to both creative (VR design) and safety-critical (automotive security) fields. We observed that generating complex MLDS artifacts worked particularly well when the meta-language was sufficiently fine-grained and included specific object definitions and enumerated attribute values (e.g., *relativePositioning*). This confirms that incorporating meta-language definitions does not require adherence to standardized formats, so long as the underlying structure is clear and domain-relevant.

2) Research Question R2: Criteria for Validating the Generated Output: Our evaluation revealed that, on one hand, formal aspects such as syntax and mandatory attributes can be automatically validated. On the other hand, certain domain-specific or semantic aspects (e.g., the correctness of spatial relationships in VR scenarios or the accuracy of security attributes in automotive models) often benefit from human review. Thus, a purely syntactic validation is a valuable first step but does not eliminate the need for *semantic* validation. A cooperative feedback loop—where experts refine machine-generated MLDS artifacts and tools provide automated format checks—leverages the strengths of both human expertise and automated structure enforcement.

3) Research Question R3: Adapting Output via Reference Examples: Including examples (e.g., prototype *scene* definitions for VR environments or sample *attack* entities in security modeling) improved both the consistency and format of the generated outputs. The LLM relies on these "exemplary" structures to align with domain-specific expectations. Notably, references can demonstrate not only value ranges but also relational patterns (*children* objects, hierarchy levels, dependencies). Moreover, the use of *templates* allows for targeted adjustments of the desired model detail, for instance by distinguishing between *minimal* or *extended* output variants.

B. Comparison with Related Work

Meta-languages for architecture and modeling are well-established in research, typically appearing in the form of DSMLs (Domain-Specific Modeling Languages) or UML profiles. Our approach extends these ideas by leveraging LLMs: instead of conforming strictly to standardized outputs (e.g., UML diagrams or XML), we demonstrate that "custom" meta-languages can be reliably processed by LLMs, provided the rules are clearly specified. This addresses a gap often left by other methods: reliable *and* flexible structured modeling without extensive fine-tuning of the model or proprietary export layers.

C. Implications and Practical Benefits

Our findings suggest that the proposed MLDS approach offers a flexible means of translating domain-centric (natural-language) descriptions into formally processable artifacts, even for users lacking deep architecture knowledge. In both domains, the technique reduced the "blank page syndrome," as users begin with a ready-to-use framework rather than an empty project. At the same time, the generated MLDS remains open to subsequent modifications, either by further prompting or by direct model editing.

Looking ahead, this could reshape workflows in software architecture and engineering. Architects may focus on designing and maintaining the meta-language, while domain experts continue to think in natural-language scenarios. This division of responsibilities can foster more productive, model-based processes and raise the adoption of structured modeling methods.

D. Limitations and Threats to Validity

While our results are encouraging, some constraints remain:

- **Domain-Specific Complexity:** As domains grow more detailed, the meta-language itself can become large and complex. An insufficiently granular metamodel may produce vague or incomplete MLDS artifacts.
- **Dependence on Prompt Quality:** The clarity of the natural-language input significantly affects the generated output. Ambiguous or insufficiently detailed prompts may result in incomplete models.
- **Technical Limitations of LLMs:** Issues such as model confusion or output truncation can arise with especially lengthy or nested structures. While example MLDS patterns help mitigate these risks, LLMs are not foolproof.
- **Generalizability:** Although we explored two notably different domains (VR and automotive security), other domains could impose even more specialized requirements that warrant extended validation.

VII. Future Directions

The integration of LLMs and meta-languages opens up several research areas. One is the development of automated *quality checks* that address both syntactic and semantic coherence (e.g., inconsistencies in hierarchical relationships). Another is exploring multi-prompting strategies or incremental fine-tuning to further improve MLDS consistency. There is also potential value in establishing a cross-domain *vocabulary* for reusable structural elements (e.g., `children`, `parent`, `referenceObject`).

Lastly, a more collaborative platform could be developed, where architects design new meta-languages while domain experts contribute reference MLDS examples. Such a shared knowledge base could be continuously updated and utilized by LLMs, enabling more efficient and accurate generation of structured domain artifacts.

VIII. Conclusion

We presented a domain-adaptable method for generating machine-readable, structured outputs from natural language prompts using Large Language Models. Our approach combines a custom-defined meta-language with schema-aware prompting to enable the creation of Meta-Language-defined Structures (MLDS). Evaluation results across two distinct use cases demonstrated a high validation rate (88%) and a substantial reduction in setup complexity (approx. 80% compared to LangChain and Pydantic). The resulting artifacts were interpretable and easy to refine, supporting rapid iterations in industrial workflows.

This approach is particularly valuable in domains where structured modeling is required but formal schema design presents a bottleneck. In addition to the use cases analyzed in this paper, we have successfully applied MLDS-based methods in a parallel study on immersive learning environments in Virtual Reality. There, natural language descriptions were used to generate didactically structured 3D scenes, allowing instructors to prototype entire VR labs without 3D design expertise, significantly reducing creation time while maintaining structural consistency.

These examples illustrate the broader potential for MLDS-based modeling in industries such as digital twin development, technical education, process automation, and safety-critical system design. Here, our method enables faster prototyping, reduces reliance on technical tooling, and facilitates interdisciplinary collaboration between domain experts and system architects.

Future work may explore the integration of MLDS with execution engines, real-time simulation tools, or graphical editors, as well as an expansion of the approach to additional modeling paradigms.

To reproduce our findings, please consult the readme file in our repository as referenced in Section IX.

IX. Data Availability

We have made all relevant research artifacts publicly available in the interest of fostering open science. Our repository contains two domain-specific meta-languages as well as additional MLDS artifacts generated with the help of these meta-languages. All materials can be accessed at the following GitHub repository: https://github.com/CoNaLaDe-MLDS/Artifacts.

Statement Concerning Generative AIs

We would like to clarify that this paper does not include substantial Sections of text directly produced by Generative AI, including LLMs.

However, since the primary objective of our work is to explore effective integration and utilization of LLMs in structured data generation and domain-specific modeling, we have naturally employed GenAI-generated outputs as part of our technical approach. In particular, LLMs were utilized to automatically generate structured MLDS files, code snippets, and example artifacts crucial to our experiments. Additionally, in selected cases, GenAI models assisted in translating specific text segments from German into English to enhance readability and accuracy.

We have clearly indicated within the paper all instances where GenAI-generated content was integrated, and provided an explicit analysis of its impact, including both benefits and limitations.

References

[1] C. K. Joyce, "The blank page: Effects of constraint on creativity," Social Science Research Network (SSRN), 2009.

[2] A. Fischer, L. Burk, R. Tavakoli Kolagari, and U. Wienkop, "Conalade mlds research artifacts," https://github.com/CoNaLaDe-MLDS/Artifacts.

[3] J. A. Díaz-Pace, A. Tommasel, and R. Capilla, "Helping novice architects to make quality design decisions using an llm-based assistant," in *Software Architecture*, M. Galster, P. Scandurra, T. Mikkonen, P. Oliveira Antonino, E. Y. Nakagawa, and E. Navarro, Eds. Cham: Springer Nature Switzerland, 2024, pp. 324–332.

[4] S. Colvin, "Pydantic: Data validation and settings management using python type annotations." [Online]. Available: https://pydantic.dev/

[5] H. Chase, "Langchain: Building applications with large language models," 2023. [Online]. Available: https://github.com/langchain-ai/langchain

[6] E. Filipovska, A. Mladenovska, M. Bajrami, J. Dobreva, V. Hillman, P. Lameski, and E. Zdravevski, "Benchmarking openai's apis and other large language models for repeatable and efficient question answering across multiple documents," in *2024 19th Conference on Computer Science and Intelligence Systems (FedCSIS)*, 2024, pp. 107–117.

[7] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22. Red Hook, NY, USA: Curran Associates Inc., 2022.

[8] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, "A survey of large language models," 2025. [Online]. Available: https://arxiv.org/abs/2303.18223

[9] S. Geng, H. Cooper, M. Moskal, S. Jenkins, J. Berman, N. Ranchin, R. West, E. Horvitz, and H. Nori, "JSON-SchemaBench: A rigorous benchmark of structured outputs for language models," *arXiv preprint arXiv:2501.10868*, 2025.

[10] Object Management Group (OMG), "Meta object facility (MOF) 2.0 core specification," 2006. [Online]. Available: http://www.omg.org/spec/MOF/2.0

[11] ——, "Unified modeling language (UML) specification version 2.5.1," 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/

[12] 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.

[13] 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.

[14] N. Bassamzadeh and C. Methani, "A Comparative Study of DSL Code Generation: Fine-Tuning vs. Optimized Retrieval Augmentation," *arXiv preprint arXiv:2407.02742*, 2024.

[15] L. Netz, J. Michael, and B. Rumpe, "From Natural Language to Web Applications: Using Large Language Models for Model-Driven Software Engineering," in *Modellierung*, 2024.

[16] A. Fischer, J.-P. Tolvanen, and R. Tavakoli Kolagari, "Automotive cybersecurity engineering with modeling support," in *2024 19th Conference on Computer Science and Intelligence Systems (FedCSIS)*, 2024, pp. 319–329.