Journal of Applied Engineering and Technological Science
Vol 7(1) 2025:247-260

MODERN LANGUAGE TECHNIQUES FROM EMERGING TRENDS, EVALUATION, OBSTACLES, TO PROSPECTS: REVIEW

Samira Abdul-kader Hussain[1], Dina Riadh Alshibani[2]*, Noor A. Yousif [3]
Department of Computer Science, College of Science, Mustansiriyah University, Baghdad, Iraq[1]
Department of Computer Science, College of Science, Mustansiriyah University, Baghdad, Iraq[2]
Control and Systems Eng. Dept., University of Technology-Iraq, Bagdad, Iraq[3]
samiracs@uomustansiriyah.edu.iq, dinashibani@uomustansiriyah.edu.iq*,
noor.a.yousif@uotechnology.edu.iq

Received: 22 February 2025, Revised: 16 October 2025, Accepted: 27 October 2025
**Corresponding Author*

ABSTRACT
Languages designed for a particular application domain are known as domain-specific languages (DSLs). When compared to general-purpose programming languages (GPPLs) in their field of usage, they provide significant improvements in expressiveness and usability. In order to handle the concurrent expansion of areas like cloud-native, distributed, and modular architectures, this discipline has been undergoing enormous evolution. Finding current trends, gaps, and new prospects in the field of DSLs is the aim of this review. We use a novel approach in this review by grouping the state-of-the-art studies into several groups. Furthermore, the three primary implementation issues of DSLs abstract syntax, concrete syntax, and semantics were highlighted. In particular, they are distinguished by the functions they prioritize (modeling, visualizing), the mapping outcomes (textual/graphical symbols), and their parsing and mapping approach (external/internal) between the abstract and concrete languages. Focus on the development lifecycle while keeping up with contemporary trends, obstacles, and the assessment metrics that are employed to evaluate the DSLs. We concluded by summarizing the research overview of DSLs after integrating it with the literature.
Keywords: *Domain Specific Language (DSL), DSLs Implementation Aspects, Abstract Syntax, Concrete Syntax, Semantics.*

1. Introduction

Languages created and implemented to express and resolve a particular class of issues are known as domain-specific languages, or DSLs. The language's domain is represented by this class (Negm et al., 2019). The database domain is the focus of the DSL SQL (Structured Query Language), while the web domain is the focus of HTML (Hyper Text Markup Language) (Bucchiarone et al., 2021). When it comes to describing a particular domain, DSLs are far superior to general-purpose programming languages (GPPLs). On the one hand, it offers more abstractions for the specified domain, which improves the development process's quality and productivity. The user will develop IoT applications using notions like devices, sensors, and actuators and will employ DSL particular to the IoT domain. When using one of the GPPLs, such as Java or C#, the user is compelled to model his problem using ideas unrelated to his domain, such as classes, fields, and arrays. Compared to the direct IoT representation, the latter one takes more time and effort and could produce more inaccuracies. Because DSL makes use of domain-specific restrictions, it also offers improved validation and verification for the output programs. In addition, the error messages are also more meaningful since it utilizes the domain concepts. On the other hand, the DSLs' usage of domain notations enables the domain expert to participate more actively in the development process (Morales et al., 2022). This results in closing the gap between the technical model, which belongs to the programmer, and the business model, which belongs to the domain expert. Consequently, the end product's quality is improved (Attiogbé & Rocheteau 2023). Due to their ability to bridge the "semantic gap" between the implementation and the issue domain, DSLs are essential for simplifying complex domains. Enhance efficiency and close the knowledge gap between developers and subject matter specialists. They are used in every aspect of modern software ecosystems, from databases and machine learning frameworks to infrastructure automation and business rules. It's challenging to design and construct a DSL that addresses every

facet of the domain. If the standard GPPL approaches are applied, it is extremely challenging to implement a new language from scratch if no appropriate one is available. In addition to requiring specialized technical expertise, creating a compiler or interpreter from scratch takes a lot of time and effort. Martin Fowler coined the term "language workbenches" to describe comprehensive environments that attempt to address the aforementioned issues by offering high-level tools. These tools make it easier to create, update, and compose DSLs. The concepts of modular language and language expansion are used in the majority of language workbenches. They give the language developer the ability to produce completely interconnected language modules. Despite the fact that the DSL concept is not new, there are no established methods for developing DSL. To accomplish their objectives, the language workbenches must handle a variety of factors. Every workbench implements these elements in a different way (Méndez-Acuña et al., 2016). While some of these methods are documented in the workbench, others have been published in scientific journals (Chavarriaga et al., 2023).

In this review article, we provide a comprehensive study for DSLs. Several surveys and literature reviews have been conducted around modeling languages and specific language; most of these studies concentrate on one or two aspects of specific language-constructing aspects. However, our review is, more useful and forward-looking than previous surveys since it specifically addresses current challenges, such as scalability, interoperability, usability, and tool support, in addition to synthesizing existing information and offering a roadmap for future. we pose our main research questions: What are the essential elements for defining a new DSL? What are the main implementation techniques? In answering these questions, our review makes the following contributions:

- An explanation of the main elements that are essential for designing and building new DSL.
- An exploration of the main approaches that are used for DSL implementation.
- An overview of the development DSL lifecycle phases.
- Connects DSL methodologies to modern development trends that were mostly overlooked in earlier work, such as low-code/no-code platforms, IoT, AI/ML, and cyber-physical systems.
- Addresses the absence of formal assessment criteria in previous research by introducing a systematic set of evaluation metrics (usability, expressiveness, performance, productivity, integration, and scalability) to evaluate DSLs.
- A reflection on the opportunities and challenges of using DSLs.

This review article's remaining content is organized as follows: Methodology is presented in section 2. Related studies on DSL implementation are reviewed in Section 3. The background concepts of the programming language are covered in Section 4, along with its implementation concepts and workbenches. The latest developments in DSL implementation are presented in Section 5. They include examples of each of the existing methods. The evaluation metric is percent in Section 6 percent. Section 7: Obstacles and Current Limitations. Discussion is in section 8. The paper's conclusion and some future study directions are provided in Section 9.

2. Methodology

This study followed a systematic literature review (SLR) methodology to ensure transparency, reproducibility, and comprehensive coverage of relevant works. The objectives, research questions guiding the review, databases searched, search strategy, and the inclusion, exclusion criteria, selection Process, and data extraction are present in this section. The process was guided by the PRISMA framework for systematic reviews.

2.1. Research Questions

Our review aims to answer the following research questions:

RQ1. What are the historical developments and differences between DSLs and GPPLs, and how have these differences contributed to the growing adoption of DSLs?

RQ2. How do implementation aspects (abstract syntax, concrete syntax, and semantics) influence the efficiency and adoption of DSLs?

RQ3. What are the phases of the DSL development lifecycle?

RQ4. What are the current trends in DSL development?

RQ5. Which evaluation metrics best capture the effectiveness of DSLs?

RQ6. What obstacles currently hinder the scalability, interoperability?

2.2 Databases Searched

To identify relevant studies, we searched five major scientific databases that are widely used in software engineering and computer science research: Scopus, IEEE Xplore, ACM Digital Library, Springer Link, and Web of Science. These databases were selected because they provide extensive coverage of journals, conference proceedings, and surveys in the area of programming languages and domain-specific languages.

2.3 Search Strategy

The search queries combined keywords related to domain-specific languages and their development. The main search string was: ("Domain-Specific Language" OR "DSL") AND ("development" OR "implementation") AND "evaluation"). Boolean operators (AND, OR) were used to refine results. The search was restricted to the time span 2000–2025, as this period covers the most significant developments in modern DSL environments such as Xtext and JetBrains MPS.

2.4 Inclusion and Exclusion Criteria

Studies were adapted in this review explicitly focus on Domain-Specific Languages (DSLs), the paper presents new methods, comparative analyses, empirical studies, or surveys that advance the understanding of DSL techniques, evaluation.

The review address the inclusion criteria are as follows:

- Implementation aspects (abstract syntax, concrete syntax, semantics, tooling).
- Development environments (e.g., Xtext, JetBrains MPS, ANTLR, Eclipse-based tools).
- Evaluation metrics (usability, expressiveness, scalability, productivity, integration).
- Emerging trends and applications (e.g., DSLs in AI/ML, Cyber-Physical Systems, low-code/no-code).
- Obstacles or challenges in DSL adoption (scalability, interoperability, tooling limitations).

The exclusion criteria were:

- Papers that mention DSLs only briefly without addressing their design, evaluation, or application.
- Studies focusing only on general-purpose programming languages (GPLs) without comparison or relation to DSLs.
- Works limited to domain applications (e.g., a DSL applied in one case study) without methodological insights into DSL development or evaluation.

3. Literature Review

Previous research on Domain-Specific Languages (DSLs) spans several application domains, reflecting the diversity of their design and use. (Shen et al., 2021) focused on the core implementation concepts of DSLs abstract syntax, concrete syntax, and semantics and categorized them based on their target application areas. Their survey highlighted Domain-Specific Modeling Languages (DSMLs) for system modeling, Domain-Specific Embedded Languages (DSELs) for embedded systems, and Domain-Specific Visualization Languages (DSVLs) for visualization tasks. This categorization demonstrates how DSLs are tailored to different contexts, a trend that continues today with Verilog and VHDL in hardware design, MATLAB-based DSLs in control systems, and graphical DSLs for system visualization.

In contrast, (Mernik et al., 2005) provided a general lifecycle model for DSL development, outlining phases such as decision, analysis, design, and implementation. While not tied to a single domain, his framework remains broadly applicable, guiding DSL creation across areas such as business process automation, embedded software, and graphical user interfaces (GUIs). This domain-agnostic methodology is particularly relevant in the current rise of low-code/no-code

platforms, where DSL lifecycle practices underpin rapid application development environments like Mendix and Microsoft Power Apps.

(Negm et al., 2019) extended the methodological perspective by distinguishing between text-based and model-based DSLs. Text-based DSLs, represented by traditional grammar rules, are common in domains such as configuration and DevOps pipelines (e.g., YAML, JSON, and TOML, now essential for CI/CD). On the other hand, model-based DSLs, built on meta-models, dominate in simulation and systems engineering, with modern parallels including TensorFlow and Keras, which adopt DSL-like abstractions for machine learning. Their work emphasized semantics as a key driver of usability and performance, reinforcing the importance of design choices in industry adoption.

(Kosar et al., 2010) investigated the human factors of DSL usage by empirically comparing DSLs and general-purpose languages (GPLs). Using the XAML DSL for UI design against C# Forms, they showed that developers performed about 15% better with DSLs across comprehension tasks, even with less prior experience. This evidence underscores how DSLs can improve productivity, error reduction, and program comprehension. Today, similar trends are observed in widely adopted DSLs such as SQL, which dominates database management due to its simplicity and expressiveness, and domain-specific configuration languages in DevOps, which streamline workflows compared to GPPL alternatives.

As a whole, previous research shows how DSLs meet different needs in different domains, such as general lifecycle methodologies (Mernik), text-based versus model-based engineering approaches (Negm et al.), embedded systems and visualization (Shen Liming et al.), and human factors in productivity and cognition (Kosar et al.). These fundamental ideas not only organize scholarly knowledge but also speak to current industrial adoption patterns, where specialized DSLs like SQL, YAML, Verilog, and TensorFlow-like DSLs support productivity, usability, and efficiency in their fields.

4. Background

This section provides an overview of the fundamentals of programming languages, taking into account their historical chronology, paradigms, and purpose.

4.1 RQ1. What are the historical developments and differences between DSLs and GPPLs, and how have these differences contributed to the growing adoption of DSLs?

Through a structured communication system, people can give computers instructions on how to carry out particular activities. It is made up of a collection of rules, syntax, and semantics that specify how computer programs should be written and understood. Programming languages act as a link between machine-executable code and human reasoning (Cardozo et al., 2022). As illustrated in Figure 1, this study offers a thorough classification that comprises three taxonomies to provide a summary of the development methodologies. The three main topics of taxonomies were 1) the purpose, 2) the kind of programming paradigm, and 3) the chronological sequence.

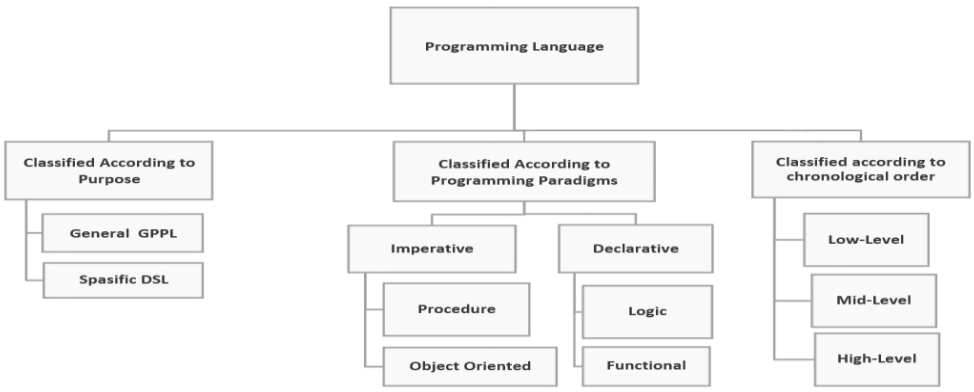

Fig. 1. Taxonomy of programming Language approaches (Qassir et al., 2024)

There are two primary methods for the first taxonomy, which is based on the purpose: general and specific. Programming languages that are intended for use in creating software for a broad range of operating systems, hardware configurations, and application domains are known

as general-purpose programming languages (GPPLs) (Johanson et al., 2017). It is distinguished by its multi-paradigm compatibility, portability, scalability, rich syntax and features, and versatility. GPPLs are flexible and able to handle a variety of computational problems, in contrast to domain-specific languages (DSLs), which are designed for certain applications or industries (Lutalo, 2024) DSL and GPPL are contrasted in Table 1 from various perspectives:

Table 1 - Comparing DSLs with GPPLs

Standard	DSL	GPPL
Its definition	A language created for a particular field or application.	a language that is intended to be widely used in many different fields.
Examples	SQL for databases. HTML for web markup. VHDL for hardware description.	JavaScript Python Java C#
Its Scope	limited with an emphasis on a particular problem domain.	Broad and appropriate for many different applications.
Ease of Use	Easier for domain experts	More complex.
Abstraction Level	High level.	The abstractions for classes, functions, ..etc.
Performance	Very efficient.	depends on its compiler.
Learning ability	Easier for domain experts.	More complex for domain experts.
Extensibility	Extending outside of the original realm is challenging.	incredibly extensible with plugins, frameworks, and libraries.
Tool Support	Limited.	Extensive.
Flexibility	Low.	High.
Expressiveness	High.	depends on the language's libraries.
Error Handling	Errors are within scope only.	Strong error handling.
Pros	- It is simple and intuitive for domain experts. - It is highly optimized for specific tasks. - It is Easy for users familiar with the domain. - It is Easy if domain remains stable.	- It is versatile, works for many applications. - It is good performance with optimization. - It is applicable in many domains once learned. - It is easier with established standards and tools.
Cons	- Useless outside its domain. - Limited scope. - Steep learning curve for non-experts. - Difficult if domain evolves or expands.	-Complex syntax and broader scope. - May require significant effort to optimize. - High initial effort to master syntax and tools. - Codebases may become large and complex.

Declarative and imperative programming are the two primary approaches that are taken into account in the second classification; each of these approaches has its own subcategories. The broad overview of programming paradigms, the languages that implement them, and the ideas they encompass is provided in figure (1). Programming paradigms are far less common than programming languages. For this reason, concentrating on paradigms instead of languages is intriguing. This perspective shows that, at least from the perspective of paradigms, languages like Java, Javascript, C#, Ruby, and Python are all essentially the same: they all implement the object-oriented paradigm with very slight variations. The path from languages to paradigms and concepts is depicted in figure (2). One or more paradigms are realized by every programming language (Van Roy. 2009). A collection of programming ideas that are arranged into a basic core language known as the paradigm's kernel language defines each paradigm. Programming languages abound, but paradigms are far less numerous (Gabbrielli et al., 2010).

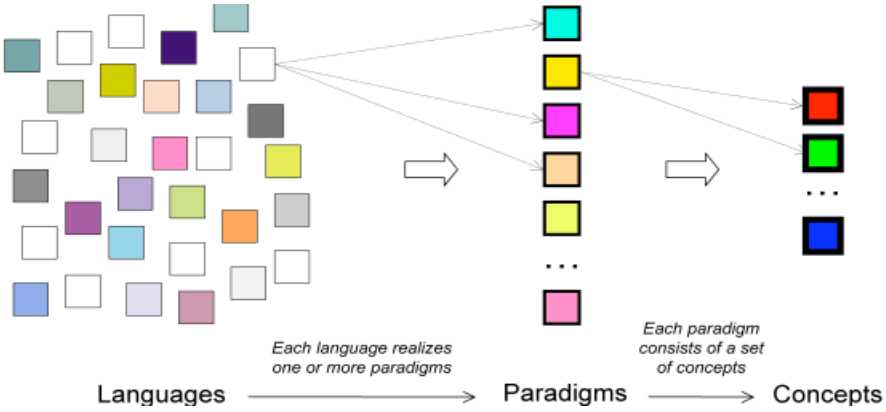

Fig. 2. The path from languages to paradigms and concepts (Van Roy. 2009)

Programming language classification based on chronological sequence is the final taxonomy shown in figure (1). Low-level programming languages come first, followed by high-level and low-code languages. From low-level programming languages like machine code and assembly to contemporary low-code development platforms, figure (3) presents the development of programming languages. Since low-level languages are closest to hardware, they provide exact control over system resources, but they also need a great deal of technical know-how and work. Higher-level languages arose as software complexity rose, offering abstractions that increased efficiency and decreased the possibility of mistakes. With the advent of scripting languages and domain-specific languages (DSLs), which provided specialized solutions for specific fields, this trend persisted. With the help of pre-built components and user-friendly graphical interfaces, low-code and no-code platforms have lately transformed software development by empowering users with little to no programming experience to create apps. This evolution illustrates how technology developments continuously alter the software development landscape by reflecting the continuous trade-off between control and abstraction (Chaudhary et al., 2022).

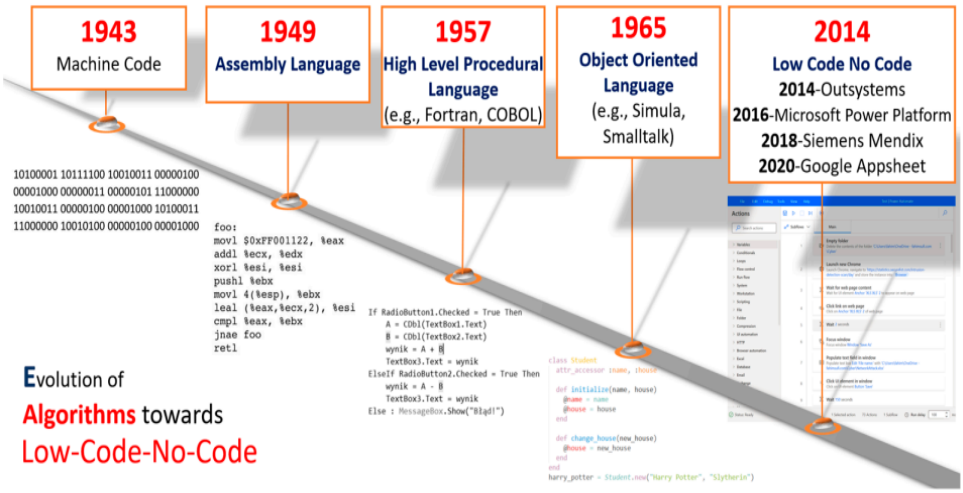

Fig. 3. Programming language evolution from machine code to low-code/no-code (Sufi, 2024).

4.2 RQ2. How do implementation aspects (abstract syntax, concrete syntax, and semantics) influence the efficiency and adoption of DSLs?

The demand for various computer languages is rising quickly. Furthermore, there are benefits to creating a new programming language specifically for a given use case; it can serve as a counterbalance to the growing complexity and scale of software. DSLs are specialized tools that are effective at solving a small number of problems, whereas programming languages are

generally tools to solve problems (Qassir et al., 2023). DSL is a programming language with expressiveness tailored to the application domain that is utilized by both beginners and experts in a certain field. In order to reduce the programming difficulties of GPPLs while preserving accurate expression in particular domains, this language provides a higher degree of abstraction, ease of use, and conciseness while offering significantly more flexibility. However, as illustrated in Figure 4, the definition of a DSL includes the three essential concepts: abstract syntax, concrete syntax, and semantics (Qassir et al., 2023).

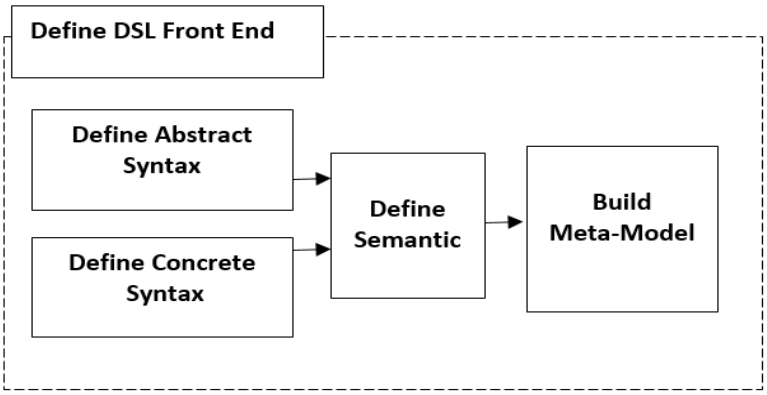

Fig. 4. The Elements of a New DSL Definition (Qassir et al., 2023).

Determining the abstract syntax, which should compile the domain's primitives, is the first step. Grammar-based and model-based are the two defining approaches that can be used. Context-Free Grammar (CFG) is used to create the grammar in order to define the DSL when employing the grammar-based approach. The programmer inputs characters into a text buffer when employing the grammar-based strategy, which is predicated on a parser-based methodology. The buffer is then analyzed to see if a string of characters adheres to a grammar. The parser first creates a parse tree, and then it creates the Abstract Syntax Tree (AST), which contains the relevant program structure (Diallo & Adda, 2024), (Brandon & Margaria, 2023). However, the model-based approach, which is based on the projection approach, uses the meta_modeling method to specify the language's structure rather than grammatical rules to define the language. The user-written program is a model that conforms to the language developer's stated meta_model. A language meta_model that describes the abstract syntax describes the concepts, relationships, and constraints of the domain (Staroletov, 2022). For the semantics, which deal with meaning and constitute the third component. For the "static" type of semantics, there aren't many design considerations. This type is frequently chosen because of its technological compatibility with the formalism used to create abstract syntax. Operational, denotational, and axiomatic semantics are the three methods available for the definition of the "dynamic" type. In software and other engineering domains, there are two different ways to express models, code, or system behavior: textual and graphical languages. Depending on the application domain, each has advantages and disadvantages (Predoaia et al., 2023).

Depending on their application, environment, or implementation method, DSLs go by a number of different names. The following are some other names for DSLs and concepts present in Table 2 that are closely related:

Table 2 - Examples of DSL with its specific name and problem domain

DSLs Names	Problem Domain	Example
Problem-Oriented Language (POL)	focuses on resolving a certain class of issues; it is frequently applied in a clearly defined problem domain.	SQL (Structured Query Language).
Task-Specific Language	focuses on automating particular processes.	- Make - Gradle
Configuration Language	designed to provide organized configuration specifications.	-YAML - JSON

		- TOML
Little Language	It alludes to tiny, specifically designed languages that are included into bigger systems.	Scripting languages used in text editors or graphics processing
Special-Purpose Language	describes languages that have a highly precise application domain focus.	-Regex
Query Language	Data retrieval and manipulation using a subset of DSLs	-SQL - GraphQL
Markup Language	utilized for content annotation.	-HTML - XML
Rule-Based Language	used to specify guidelines and limitations in particular fields	-Drools
Scripting Language	When DSLs are employed for certain tasks, like automating a bigger system.	-Bash -Lua

4.3 RQ3. What are the phases of the DSL development lifecycle?

There are several steps involved in creating a DSL, starting with conception to upkeep and development. A thorough explanation of the normal DSL development lifecycle is provided below: DSL is developed via a consistent iterative method. This process consists of four phases: analysis, decision-making, design, and implementation (Borum & Seidl, 2022), (Kouzel, 2024) as following:

- **Analysis Phase**

Identifying the problem domain and compiling specific DSL requirements are the objectives of this phase. Working with domain experts to comprehend the problem domain is the first step. Determine the basic concepts and extract terms unique to the domain. Lastly, identify the DSL's limitations, including what it should and shouldn't support. Use cases that are clearly specified in the issue domain and the first DSL goals in written form are the results of this phase (Ouaddi et al., 2025).

- **Design Phase**

Determining the DSL's syntax, semantics, and general structure is the aim of this phase. Decide whether users will build programs in the DSL using graphical or textual syntax to define the concrete syntax. Describe the abstract syntax by Make a metamodel that illustrates the language's fundamental ideas and how they relate to one another. Determine the meaning of linguistic constructs to define semantics. This phase produces the syntax and semantics specifications as well as DSL grammar (if textual) or metamodel (if graphical) (Panayiotou et al., 2024).

- **Implementation Phase**

This phase's objective is to construct the DSL's fundamental infrastructure, which includes interpreters, compilers, and parsers. For textual DSLs, implement parsers using programs like ANTLR, Xtext, or Yacc. Regarding graphical DSLs: Use technologies such as Eclipse GMF or Sirius to implement a visual editor. Establish guidelines in a GPPL for code generation and converting DSL constructs into executable code. Lastly, provide users the tools they need to develop DSL applications, like syntax highlighting, auto-completion, and support for error detection and debugging. A DSL compiler/interpreter with a DSL editor is the result of this phase (Qassir, 2025).

- **Testing Phase**

Making sure the DSL functions well and satisfies the criteria is the aim of this phase. This is accomplished by evaluating distinct linguistic structures, verify that generated code or interpreted DSL programs operate well, test the DSL in the context of full use cases, and then collaborate with subject matter experts to make sure the DSL is user-friendly and intuitive. This phase results in a validated DSL along with test reports (Gómez-Abajo et al., 2025), (Diallo & Adda, 2024).

5 RQ4. What are the current trends in DSL development?

5.1 Low-code and No-code Platforms

With the help of this new kind of platform, which has caused a paradigm shift in software development, users of all skill levels from non-technical domain experts to experienced developers can construct apps with little to no manual coding. Business automation, data analysis, machine learning, and the Internet of Things are just a few of the domains where these platforms have found use. This strategy is distinguished by three features: Initially, the visual development environment relies on pre-built components, visual workflow designers, and drag-and-drop interfaces. Second, users can create desktop, mobile, and online applications with a single development effort because to this platform type's cross-platform capability. Lastly, declarative logic allows users to specify application behavior without writing code by using visual models and graphical interfaces. Mendix, OutSystems, Microsoft Power Apps, Bubble, Adalo, Thunkable, and Appian are a few of the platforms (Brandon & Margaria, 2023), (Hagel et al., 2024). DSLs align naturally with this composability trend, especially when expressed as model-based or graphical notations. Yet, obstacles arise in the form of vendor lock-in, limited portability, and reduced expressiveness. The simplicity that makes low/no-code DSLs accessible also restricts their ability to handle complex scenarios, and managing large-scale applications through visual DSLs can quickly become unwieldy (Naimi et al., 2025).

5.2 DSLs for Machine Learning and AI

Artificial intelligence (AI) and machine learning (ML) have benefited greatly from DSLs. These DSLs enable effective ML/AI system design, training, and deployment by offering abstractions and tools that are suited to the requirements of data scientists, researchers, and engineers. It offers declarative syntax, domain-specific focus, performance optimization, and high-level abstractions. OptiML, DLVM, and Anamika are a few prominent DSLs and frameworks that have been created for machine learning and artificial intelligence (Morales et al., 2022), (Wang et al., 2024). The necessity to interact with heterogeneous hardware accelerators (GPUs, TPUs, and distributed systems) complicates DSLs' capacity to strike a balance between expressiveness and efficiency, and the quick speed of AI invention makes it challenging for them to be adaptable and extendable (Macià et al., 2023).

5.3 DSLs for Cyber-Physical Systems

In the modeling, analysis, and development of Cyber-Physical Systems (CPS), DSLs are essential. They offer customized abstractions that improve system design and verification by bridging the gap between computational and physical components. Among the notable DSLs and frameworks in this field are: Triton is a DSL that offers high-level domain-specific capabilities in an effort to promote abstraction in CPS development. By offering constructs that directly map to domain concepts, it seeks to streamline the CPS engineering process, lowering complexity and increasing productivity. Formal system design is the main focus of ForSyDe-Atom, a shallowly embedded DSL for CPS modeling. It enables the representation of both discrete and continuous behaviors inside a single framework by using innovative modeling principles that are helpful in the design of CPS (Carvalho et al., 2023), (Gerhold et al., 2024), (Giner-Miguelez et al., 2023).

6 RQ5. Which evaluation metrics best capture the effectiveness of DSLs?

Both qualitative and quantitative indicators are used to assess a DSL's efficacy. Figure 5 explains the six fundamental metrics that are used to evaluate how well the DSL supports its users, achieves its objectives, and fits within its intended domain. Table 3 shows the sub-metrics and explanations for each related metric (Mehmed, 2024), (Mertz & Nunes, 2021).

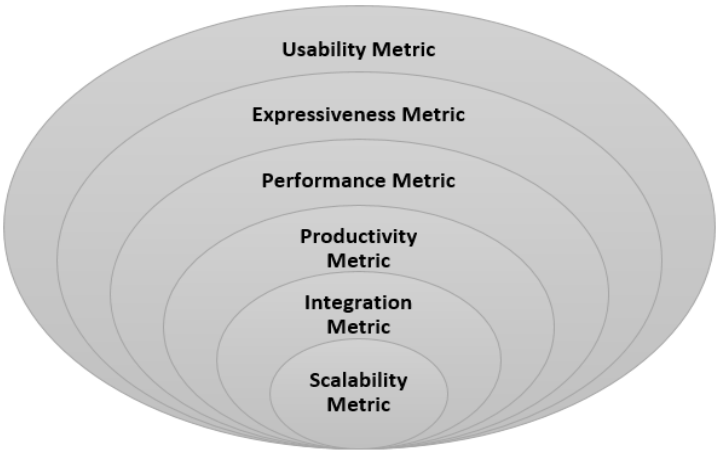

Fig. 5. The Six Essential Metrics for DSL Evaluations (Qassir, 2024), (Giner-Miguelez et al., 2023)

Table 3 - Examples of DSL with its specific name and problem domain

Metric Name	Its mean and sub-metrics	
Usability	This measure assesses the DSL's usability and ease of learning for developers and domain specialists.	
	Learnability	The amount of time needed to master the DSL and the quantity of training sessions or references to documentation that are required.
	Syntax Simplicity	The DSL's keyword or construct count and a measure of the syntax's clarity and intuitiveness.
	Error Rate	Error types and frequency that users make when using the DSL, as well as the average time it takes to find and fix faults.
	User Satisfaction	Usability test results and input from developers and subject matter experts (such as surveys or interviews).
Expressiveness	This statistic assesses how well the DSL can solve issues and convey domain notions.	
	Coverage	The proportion of domain concepts that can be modeled by the DSL and Conformity of the DSL to the domain's needs and terminology.
	Conciseness	When comparing the DSL to a general-purpose language (GPL), the number of lines of code (LOC) needed to represent a typical solution and Boilerplate code reduction.
	Extensibility	Adding new features or constructs to the DSL is simple and doesn't interfere with already-running apps.
Performance	This measure evaluates the DSL's efficiency and runtime performance.	
	Execution Speed	DSL applications' runtime efficiency in comparison to comparable programs written in GPPL.
	Compilation or Interpretation Overhead	The amount of time needed to parse, compile, or interpret DSL programs.
	Memory Usage	Resource usage of DSL programs while they are running.
Productivity	This indicator assesses how the DSL affects developer productivity.	
	Development Time	Implementation time for a DSL solution as opposed to a GPPL.
	Reusability	The availability of libraries or pre-built components and the frequency of DSL program or component reuse.
	Automation	The DSL's level of automation like automatic code creation or configuration.

Integration	This measure assesses how well the DSL integrates with other systems, tools, and languages.	
	Interoperability	Support for importing and exporting data or artifacts to and from other systems, as well as ease of integration of DSL programs with other languages or platforms.
	Toolchain Support	IDEs, debuggers, profilers, and other DSL tools are available.
Scalability	This statistic evaluates the DSL's performance as the size and complexity of the challenge increase.	
	Scalability of Language Constructs	the capacity to manage intricate domain situations without suffering appreciable performance deterioration.
	Tooling Scalability	Performance of tools (such as editors and interpreters) that are particular to DSL when working with big codebases.

7 RQ6. What obstacles currently hinder the scalability, interoperability?

Although DSL has many advantages, like abstraction, productivity, and simplicity of use in particular fields, they also have a number of drawbacks and restrictions that limit its wider use and efficacy. These difficulties fall into three categories: interoperability, usability, and technological. Regarding the technological side of things, scalability is a major obstacle in the development and implementation of DSLs, particularly when use cases and systems get more sophisticated. The linguistic complexity of a DSL that was first created for a particular problem domain, such as the Internet of Things, is a crucial factor indicating scalability concerns; it could find it difficult to accommodate new, more general requirements, such as integrating AI-driven IoT applications, without undergoing a major redesign. Additionally, the lack of adequate tooling and IDE support can make maintaining sizable DSL-based codebases difficult. Compared to GPPLs, many DSLs lack strong, scalable toolchains, such as editors, debuggers, and profilers. As projects get bigger and more complex, this reduces developer productivity. Users are forced to work within the limitations of the language since DSLs frequently lack the flexibility of GPPLs when it comes to usability issues. In addition, many DSLs lack comprehensive documentation, tutorials, and a robust user community, making adoption difficult. Finally, interoperability limitations; the lack of universal standards for DSL development can lead to fragmentation and incompatibility within and across domains (Bragança et al., 2021), (Haindl & Plösch, 2022).

8 Discussion

The review's conclusions show that domain-specific languages (DSLs) have developed into vital instruments for handling the complexity of contemporary software systems, especially in fields like DevOps automation, low-code/no-code development, and artificial intelligence. This study broadens the viewpoint by connecting the evolution of DSLs to contemporary technological changes, in contrast to previous surveys like (Mernik et al., 2005) and (Kosar et al., 2010), which focused on the lifecycle of DSL development and the cognitive benefits of DSLs over general-purpose programming languages. Similarly, more recent studies that examined DSLs for visual computing (Shen et al., 2021) and DSL assessment frameworks (Nimm et al., 2019) offered domain-specific insights but neglected to address cross-cutting trends like DevOps integration and low-code acceptance. According to our findings, previous research mostly focused on the theoretical underpinnings and usability advantages of DSLs, but current trends indicate a growing dependence on DSLs as the unseen infrastructure underlying AI model specification, automation platforms, and user-friendly development environments. The comparison shows that even while DSLs have been consistently acknowledged as tools that increase productivity, there are still persistent issues that have remained across decades of study, most notably interoperability, tooling constraints, and scalability. DSLs' entry into new fields, however, suggests that their function is growing beyond specialized acceptance and becoming ingrained in standard software procedures. Overall, the review's findings support previous research's findings about the expressiveness and usability advantages of DSLs while also broadening the discussion to take into account contemporary developments that are changing how they are used. When considered

collectively, the data indicates that DSLs are continuing to be relevant and adjusting to new software engineering paradigms. The broad conclusion that comes out of this is that DSLs have evolved from specialized research artifacts to essential elements of modern software ecosystems, and that their further development will be crucial to closing the gap between technical implementation and domain expertise.

8 Conclusion

This study has looked at how domain-specific language (DSL) approaches have changed over time, emphasizing new developments, assessment criteria, challenges, and potential applications. The study's key findings show that DSLs are still essential to contemporary software engineering because they increase productivity, expressiveness, and domain alignment. However, they still face enduring issues such inadequate tools support, scalability, and interoperability. Current software developments, specifically DevOps pipelines, low-code/no-code platforms, and artificial intelligence, have opened up new avenues for DSL adoption while also posing new challenges in terms of maintainability, flexibility, and integration. To fully realize the promise of DSLs, a more methodical and controlled approach to their creation and assessment is required, according to the research's findings. at order to ensure that DSLs stay flexible and significant at a time of fast technological advancement, future DSL research may address the gaps and issues that have been discovered. This will help to advance both domain-specific solutions and more general software engineering methods.

References

Attiogbé, C., & Rocheteau, J. (2023). Correctness of IoT-based systems: From a DSL to a mechanised analysis. *Journal of Computer Languages, 77*, 101239. https://doi.org/10.1016/j.cola.2023.101239

Borum, H. S., & Seidl, C. (2022). Survey of established practices in the life cycle of domain-specific languages. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems* (pp. 266-277). https://doi.org/10.1145/3550355.3552413

Boutot, P., Tabassum, M. R., Abedin, A., & Mustafiz, S. (2024). Requirements development for IoT systems with UCM4IoT. *Journal of Computer Languages, 78*, 101251. https://doi.org/10.1016/j.cola.2023.101251

Bragança, A., Azevedo, I., Bettencourt, N., Morais, C., Teixeira, D., & Caetano, D. (2021, October). *Towards supporting SPL engineering in low-code platforms using a DSL approach*. In Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (pp. 16-28). https://doi.org/10.1145/3486609.3487196

Brandon, C., & Margaria, T. (2023). Low-Code/No-Code Artificial Intelligence Platforms for the Health Informatics Domain. *Electronic Communications of the EASST, 82*. https://doi.org/10.14279/tuj.eceasst.82.1221

Bucchiarone, A., Cicchetti, A., Ciccozzi, F., & Pierantonio, A. (Eds.). (2021). *Domain-specific languages in practice: With JetBrains MPS* (Springer Nature). https://doi.org/10.1007/978-3-030-73758-0

Cardozo, N., & Mens, K. (2022). Programming language implementations for context-oriented self-adaptive systems. *Information and Software Technology, 143*, 106789. https://doi.org/10.1016/j.infsof.2021.106789

Carvalho, T., Bispo, J., Pinto, P., & Cardoso, J. M. P. (2023). A DSL-based runtime adaptivity framework for Java. *SoftwareX, 23*, 101496. https://doi.org/10.1016/j.softx.2023.101496

Chavarriaga, E., Jurado, F., & Rodríguez, F. D. (2023). An approach to build JSON-based domain-specific languages solutions for web applications. *Journal of Computer Languages, 75*, 101203. https://doi.org/10.1016/j.cola.2023.101203

Chaudhary, H. A. A., & Margaria, T. (2022). DSL-based interoperability and integration in the smart manufacturing digital thread. *Electronic Communications of the EASST*, 81. https://doi.org/10.14279/tuj.eceasst.81.1198

Diallo, M. M., & Adda, M. (2024). HoBACDSL: HoBAC-focused Access Control Domain Specific Language. *Procedia Computer Science, 241*, 40-47. https://doi.org/10.1016/j.procs.2024.08.008

Gabbrielli, M., Martini, S., & Giallorenzo, S. (2010). *Programming Languages: Principles and Paradigms* (Vol. 8). London: Springer. https://doi.org/10.1007/978-3-031-34144-1

Gerhold, M., Kouzel, A., Mangal, H., Mehmed, S., & Zaytsev, V. (2024, September). Modelling of Cyber-Physical Systems through Domain-Specific Languages: Decision, Analysis, Design. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems* (pp. 1170-1179). Association for Computing Machinery. https://doi.org/10.1145/3652620.3688348

Chaudhary, H. A. A., & Margaria, T. (2022). DSL-based interoperability and integration in the smart manufacturing digital thread. *Electronic Communications of the EASST, 81*. https://doi.org/10.14279/tuj.eceasst.81.1198

Diallo, M. M., & Adda, M. (2024). HoBACDSL: HoBAC-focused Access Control Domain Specific Language. *Procedia Computer Science, 241*, 40–47. https://doi.org/10.1016/j.procs.2024.08.008

Gabbrielli, M., Martini, S., & Giallorenzo, S. (2010). *Programming Languages: Principles and Paradigms* (Vol. 8). London: Springer.

Gerhold, M., Kouzel, A., Mangal, H., Mehmed, S., & Zaytsev, V. (2024, September). Modelling of Cyber-Physical Systems through domain-specific languages: Decision, analysis, design. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems* (pp. 1170–1179). https://doi.org/10.1145/3652620.3688348

Johanson, A. N., & Hasselbring, W. (2017). Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: A controlled experiment. *Empirical Software Engineering, 22*, 2206-2236. https://doi.org/10.1007/s10664-016-9483-z

Katsikeas, S., Ling, E. R., Johnsson, P., & Ekstedt, M. (2024). Empirical evaluation of a threat modeling language as a cybersecurity assessment tool. *Computers & Security, 140*, 103743. https://doi.org/10.1016/j.cose.2024.103743

Kosar, T., Oliveira, N., Mernik, M., Pereira, V. J. M., Črepinšek, M., Da Cruz, D., & Henriques, R. P. (2010). Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems, 7*(2), 247-264. https://doi.org/10.2298/CSIS1002247K

Kouzel, A. (2024). *Developing a DSL design methodology for CPS diagnostics* (Bachelor's thesis, University of Twente). https://essay.utwente.nl/100776/1/Kouzel_BA_EEMCS.pdf

Lutalo, J. W. (2024). *Programming language engineering—a review of text processing language design, implementation and evaluation methods.* Preprints.org. https://doi.org/10.20944/preprints202410.0636.v2

Macià, S., Martínez-Ferrer, P. J., Ayguadé, E., & Beltran, V. (2023). Assessing Saiph, a task-based DSL for high-performance computational fluid dynamics. *Future Generation Computer Systems, 147*, 235-250. https://doi.org/10.1016/j.future.2023.04.035

Mehmed, S. (2024). *Domain-Specific Languages for Cyber-Physical Systems: A survey* (Bachelor's thesis, University of Twente, Enschede, The Netherlands). https://essay.utwente.nl/100883/

Méndez-Acuña, D., Galindo, J. A., Degueule, T., Combemale, B., & Baudry, B. (2016). Leveraging software product lines engineering in the development of external DSLs: A systematic literature review. *Computer Languages, Systems & Structures, 46*, 206-235. https://doi.org/10.1016/j.cl.2016.03.003

Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR), 37*(4), 316-344. https://doi.org/10.1145/1118890.1118892

Mertz, J., & Nunes, I. (2021). Tigris: A DSL and framework for monitoring software systems at runtime. *Journal of Systems and Software, 177*, 110963. https://doi.org/10.1016/j.jss.2021.110963

Morales, S., Clarisó, R., & Cabot, J. (2022). *Towards a DSL for AI engineering process modeling*. In P. Kuhrmann, J. Klünder, J. Taibi & T. Mikkonen (Eds.), *International Conference on Product-Focused Software Process Improvement (PROFES 2022)* (Lecture Notes in Computer Science, Vol. 13709, pp. 53-60). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-031-21388-5_4

Naimi, L., Abdelmalek, H., & Jakimi, A. (2024). A DSL-based approach for code generation and navigation process management in a single page application. *Procedia Computer Science, 231,* 299-304. https://doi.org/10.1016/j.procs.2023.12.207

Negm, E., Makady, S., & Salah, A. (2019). Survey on domain specific languages implementation aspects. *International Journal of Advanced Computer Science and Applications, 10*(11). https://doi.org/10.14569/IJACSA.2019.0101183

Ouaddi, C., Benaddi, L., & Jakimi, A. (2024). Architecture, tools, and DSLs for developing conversational agents: An overview. *Procedia Computer Science, 231*, 293-298. https://doi.org/10.1016/j.procs.2023.12.206

Panayiotou, K., Doumanidis, C., Tsardoulias, E., & Symeonidis, A. L. (2024). SmAuto: A domain-specific-language for application development in smart environments. *Pervasive and Mobile Computing, 101,* 101931. https://doi.org/10.1016/j.pmcj.2024.101931

Predoaia, I., Kolovos, D., Lenk, M., & García-Domínguez, A. (2023). Streamlining the development of hybrid graphical-textual model editors for domain-specific languages. *Journal of Object Technology, 22*(2). https://doi.org/10.5381/jot.2023.22.2.a8

Qassir, S. A. (2024). Building a graphical modelling language for efficient homomorphic encryption schema configuration: HomoLang. *TEM Journal, 13*(3), 2285-2296. https://doi.org/10.18421/TEM133-56

Qassir, S. A. (2025). MyDSL: Front-End compiler design for a user-friendly language supporting hybrid meta-heuristics. *TEM Journal, 14*(3). https://doi.org/10.18421/TEM143-11

Qassir, S. A., Gaata, M. T., & Sadiq, A. T. (2023). SCLang: Graphical domain-specific modelling language for stream cipher. *Cybernetics and Information Technologies, 23*(2), 54-71. https://doi.org/10.2478/cait-2023-0013

Qassir, S. A., Gaata, M. T., Sadiq, A. T., & Al Alawy, F. (2023). Designing a graphical domain-specific modelling language for efficient block cipher configuration: BCLang. *TEM Journal, 12*(4), 2038-2049. https://doi.org/10.18421/TEM124-14

Qassir, S. A., Gaata, M. T., Sadiq, A. T., & Taha, I. F. (2024). *Developing a graphical domain-specific modeling language for efficient lightweight block cipher schemas configuration: LWBCLang. Iraqi Journal of Science, 65*(10), 5819-5836. https://doi.org/10.24996/ijs.2024.65.10.39

Shen, L., Chen, X., Liu, R., Wang, H., & Ji, G. (2021). Domain-specific language techniques for visual computing: A comprehensive study. *Archives of Computational Methods in Engineering,* 28, 3113-3134. https://doi.org/10.1007/s11831-020-09492-4

Staroletov, S. M. (2022, September). *Grammar-based testing a process-oriented extension of the IEC 61131-3 Structured Text language*. In *Proceedings of the 2022 International Russian Automation Conference (RusAutoCon)* (pp. 863-869). IEEE. https://doi.org/10.1109/RusAutoCon54946.2022.9896346

Sufi, F. (2023). Algorithms in low-code-no-code for research applications: A practical review. *Algorithms*, 16(2), 108. https://doi.org/10.3390/a16020108

Van Roy, P. (2009). Programming paradigms for dummies: What every programmer should know. In *New Computational Paradigms for Computer Music* (Vol. 104, pp. 616-621).

Vierhauser, M., Wohlrab, R., Stadler, M., & Cleland-Huang, J. (2023). AMon: A domain-specific language and framework for adaptive monitoring of Cyber-Physical Systems. *Journal of Systems and Software*, 195, 111507. https://doi.org/10.1016/j.jss.2022.111507

Wang, R., Lu, M., Yu, C. H., Lai, Y. H., & Zhang, T. (2024). *Automated deep learning optimization via DSL-based source code transformation.* In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 479-490). https://doi.org/10.1145/3650212.3652143