DOI: https://doi.org/10.15276/aait.09.2026.08
UDC 004.8:004.4

A system internals modeling and annotation language for large language model-driven software engineering

Mykyta V. Syromiatnikov[1)]
ORCID: https://orcid.org/0000-0002-0610-3639; nik.syromyatnikov@gmail.com. Scopus Author ID: 59533584100
Victoria M. Ruvinska[1)]
ORCID: https://orcid.org/0000-0002-7243-5535; ruvinska@op.edu.ua. Scopus Author ID: 57188870062
[1)] Odesa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044, Ukraine

ABSTRACT

Rapid scaling of large language models has significantly disrupted traditional software engineering paradigms with unprecedented capabilities for code understanding, generation, and automated review. However, practical repository-level deployment is constrained by context limits and token cost. While retrieval-augmented generation is widely used to address this limit, it often splits a codebase into disconnected semantic chunks, omitting high-level structural dependencies. In contrast, alternative approaches that attempt to feed the entire repository structure into the context window typically rely on generic formats such as JSON. While widely recognized, these formats introduce redundant syntactic units that significantly influence the token budget, effectively reintroducing the bottleneck. This work introduces SiMAL (System internals Modeling and Annotation Language), a domain-specific language designed specifically for language-model-driven software engineering workflows, with the primary objective of providing a compact, human-readable, error-tolerant, yet highly structured representation of a software system, optimized for iterative interaction with language models. It combines both static and dynamic aspects of a software system, unifying architectural views, component and endpoint definitions, runtime deployment metadata, and other development artifacts into a single textual schema that can be converted to and from a normalized machine representation. This effort includes the definition of the language's syntax and grammar, an open-source parser, and a visualizer that renders schemas into nested system diagrams. The proposed language is validated through a comprehensive protocol that assesses token efficiency alongside schema validity and semantic faithfulness. This includes deterministic structural checks (parsing integrity and annotation consistency), schema-to-code correspondence analysis, and an "LLM-as-a-judge" evaluation for repository coverage. The results indicate that prompt-efficient schema modeling reduces token overhead without systematically degrading structural usability or quality, making it a practical backbone for scalable autonomous software engineering.

Keywords: System modeling; software architecture; language models; token efficiency; code summarization; software engineering

***For citation*: Syromiatnikov M. V., Ruvinska V. M. "A system internals modeling and annotation language for large language model-driven software engineering". *Applied Aspects of Information Technology*. 2026; Vol.9 No.1: 103–121. DOI: https://doi.org/10.15276/aait.09.2026.08**

INTRODUCTION

Led by the decade of ascension from simple text autocomplete to gold-medal level performance in international math and programming competitions [1], [2], the evolution of language models has shifted the focus of AI-assisted software engineering from localized code completion to project-level reasoning and execution [3]. Modern agents can refactor legacy systems, implement features in large, distributed codebases, or even perform automatic code review with multi-repo context [4]. These directions align with the broader landscape of machine learning applied to source code [5].

As these capabilities scale, the primary performance limitation identified is the context bottleneck. While modern large language models (LLMs) support larger context windows up to hundreds of thousands of tokens [6], feeding massive system internals required to understand system completely (e.g., service component definitions, APIs, data models, dependencies, runtime configuration, and cross-service interactions) can degrade reasoning in non-obvious ways, especially when the important knowledge is distributed across a large prompt and input exceeds the effective reasoning window [7]. Moreover, large input prompts for iterative processes like feature implementation directly lead to costs blowing up for API interaction or increased latency for local model inference. As a result, prompt efficiency remains important even for larger context windows.

In the field of natural language processing (NLP) there are generally two main tactics for mitigating the context bottleneck: retrieval-based and summarization-based approaches. Retrieval-based approaches reduce prompt size by selecting semantically relevant data chunks from pre-filled storage at query time. However, research indicates that retrieval approaches, splitting the codebase into

independent semantic chunks, may unintentionally drop structural dependencies needed for system-level tasks, such as interface contracts, dependencies, inheritance, and call relationships [8], [9]. This may fragment system understanding and miss critical connected evidence [7]. Summarization approaches, in turn, provide a distilled representation of the system. They often rely on generic serialization formats (e.g., JSON) that introduce significant syntactic overhead, or on diagram-centric architecture notations that prioritize human visualization over machine-checkable schema constraints [10].

To address these challenges, this paper investigates compact, machine-readable representations of repository structure and evaluates their suitability for iterative LLM-driven software engineering. The proposed approach targets prompt efficiency while preserving machine-checkable structure and developer readability, placing it between retrieval-centric pipelines and verbose generic serializations. The remainder of this paper reviews prior context mitigation strategies, describes the proposed domain-specific language and toolchain, while also presenting an empirical evaluation of token efficiency and schema validity.

RELATED WORKS

Context Bottleneck. Over the last few years, LLMs have increased the hard cap on input length from thousands to hundreds of thousands of tokens. Moreover, recent versions of the Gemini and Claude systems even offer a million-token context window [11], [12].

This enhancement allows LLM-powered systems to process large inputs, such as books or codebases with multiple files, modules, and services, within a single request. In general, an increased context window enables high-precision reasoning and improves conversation memory. However, various research works highlight that simply increasing the size of the context window does not guarantee reliable utilization of long inputs [13], [14]. For instance, adding irrelevant context can substantially degrade retrieval-task performance (13.9 %-85 %) [13]. Even if the relevant information is present in the input prompt, the model's long-sequence processing capability may still be limited by position sensitivity. Research indicates that retrieval accuracy can drop by over 20% when critical data is located in the middle of the context window compared to the start or end [7].

Retrieval-based methods for context bottleneck mitigation. Nowadays, many search systems leverage retrieval-augmented generation (RAG) to inject relevant information from a large corpus into the prompt [15]. Retrieval augmentation has also been applied specifically to repository-scale code completion [16]. While effective for text, standard embedding-based retrieval struggles with software repositories' complex dependencies (inheritance, API contracts). Since code chunks often fail to preserve these relations [9], [17], retrieving a function without its interface contract strips critical context. This forces the model to reason over incomplete artifacts or be overwhelmed by noisy candidates [13], [14].

GraphRAG, a recent effort to improve retrieval by incorporating graph structure, aims to combine data points through a relational structure rather than separate chunks [18]. However, using these representations in LLM prompting often requires textual serialization of the graph, which, again, introduces token overhead. Therefore, there is still a practical need for representations that preserve structural relations and remain prompt-efficient.

Structured summarization- and compression-based methods. A parallel workstream focuses on summarization and structured distillation of code repositories rather than dynamic retrieval of relevant information. Early approaches often relied on abstract syntax trees (ASTs) to generate natural language summaries of functions and classes. Graph-structured neural approaches have also been explored for source code summarization [19]. While such summaries are valuable for documentation, they are often much larger and deeper than the corresponding source code and sometimes lack strict signature precision and schema-level constraints needed for code generation and verification tasks.

Recent works such as RepoGraph [17] and CodexGraph [9] highlight the value of representing code dependencies and relations rather than treating repositories as flat text corpora. However, these approaches often rely on verbose serialization formats that can drastically increase prompt token usage.

The sweet spot between retrieval and full summarization could be the repository map approach. Tools such as Aider propose building a compact, symbol-level overview using syntax-aware extraction (e.g., tree-sitter), ranking definitions to fit a fixed token budget, and presenting the resulting structure as a stable context for repeated interactions [20]. Repository maps reduce dependence on top-k semantic retrieval and provide predictable global orientation, which helps tasks that require navigation across many files. However, repository maps are

ISSN 2617-4316 (Print)
ISSN 2663-7723 (Online)

typically read-only structural sketches: they describe code entities and signatures but often omit runtime configuration, deployment constraints, cross-service protocol contracts, and operational dependencies. That is why, for project-level tasks that require both high-level architecture and low-level component context, a richer schema representation is still needed.

System Description Languages. Software architects commonly use high-level design notations to describe the structure and behavior of software systems. One of the well-established tools is the unified modeling language (UML). This visual language was introduced to represent object-oriented designs via static class diagrams, component diagrams, and dynamic sequence views [21]. Simultaneously, architecture description languages (ADLs) were proposed as a textual format for modeling software architectures with explicit semantics [22]. Both these approaches can represent a system at a higher level than raw source code. Related work on domain analysis and variability management also relies on structured, machine-checkable representations to capture system-level variability and constraints, but it is not optimized for prompt-efficient iterative interaction with LLMs [23]. However, standard UML and formal ADLs are not optimized for LLM consumption. UML's visual-first nature requires verbose textual explanations, while compact ADLs like PlantUML prioritize human-readable visualization over machine-checkable specifications. Moreover, PlantUML permits inconsistent diagrams and therefore behaves "more as a drawing tool rather than a modeling tool" [10], thus limiting its suitability as a strict source of truth for automated pipelines. As a result, relationship-first ADLs are effective for communicating architectural topology but often insufficient for LLM-driven code generation and verification tasks that require explicit typing, strict contracts, and unambiguous schema-level constraints.

Other architecture documentation practices, such as the 4+1 view model [24] and stakeholder-centric documentation templates [25], remain valuable for engineering communication while being focused on clarity and simplicity for human readability, but they are not focused on prompt token economy or structured LLM input requirements. The same applies to industry models, such as C4 [26], and enterprise-oriented frameworks, such as ArchiMate [27].

Serialization formats for LLM. Generic serialization formats such as JSON [28] and YAML [29] are widely used for structured data interchange and configuration. In LLM-based workflows, these formats are widely used because they are familiar to models, given their broad presence in training corpora and simple parsing. Moreover, JSON is the only option when an LLM must produce a valid structured output [30].

Despite being widely used for LLM prompting, JSON requires repeated structural tokens (quotes, braces, commas) and deep nesting, consuming a significant portion of the context window. Evaluations show pretty-printed JSON requires ~30-40% more tokens than equivalent domain-specific language (DSL) representations [31].

This introduces not only a cost issue but also a usability problem, as schemas must be frequently read, edited, and iteratively refined by humans and machines. YAML can improve readability by relying on indentation and allowing unquoted scalars. However, it remains a generic data format rather than an architecture-oriented representation and can introduce parsing ambiguity under inconsistent indentation.

To sum it all up, existing mitigation strategies for repository-scale context either retrieve small semantic chunks that can omit cross-file structural dependencies required for system-level reasoning (e.g., contracts and call relationships) [8], [9], or rely on verbose structured formats and diagram-centric notations that are not optimized for iterative machine consumption and prompt token economy [10], [28], [29], [30]. Therefore, there is a need for a prompt-oriented, machine-checkable representation that preserves repository structure and interface-level relations while reducing syntactic overhead relative to common baselines such as JSON.

RESEARCH AIM AND OBJECTIVES

The aim of this research is to develop and empirically validate a compact, prompt-oriented modeling language for representing repository-level software system context in LLM-driven software engineering workflows, while preserving machine-checkable structure and reducing token overhead relative to widely used structured formats.

To achieve this aim, the following objectives are set:

1) define a compact language for system modeling that represents repository structure, interfaces, and key metadata in a prompt-efficient and human-readable form;

2) implement a deterministic parser and an open-source toolchain enabling syntax validation, structured extraction, and bidirectional SiMAL-

JSON conversion, including prompt-oriented compact variants;

3) conduct an empirical comparison of token footprint across a multi-project schema corpus under multiple tokenizers;

4) evaluate schema validity and faithfulness using complementary signals, including deterministic non-LLM metrics and a controlled LLM-as-a-judge protocol.

METHODOLOGY

This work introduces SiMAL (System internals Modeling and Annotation Language) – a lightweight DSL that prioritizes prompt efficiency while maintaining architectural structure (services, components, APIs), operational context (runtime configuration, dependencies), and evidence linking in a single compact yet complete representation of the system. Compared to diagram-centric notations, proposed DSL is focused on machine-checkable schema content rather than visual layout directives. Compared to JSON/YAML, it reduces syntactic overhead by using a compact block-and-attribute grammar, while also supporting deterministic parsing with JSON conversion for downstream tools.

Design Principles. DSL was designed with four key principles.

1. Readability is defined as simplicity for developers when reading, writing, and editing SiMAL schemas.

2. Token efficiency is reflected in minimizing redundant characters.

3. Error-tolerance is achieved by ensuring minor formatting mistakes do not invalidate the full schema. Minor mistakes are defined as local formatting deviations that do not alter the nesting structure, such as missing commas, unquoted keys, or unformatted text lines inside a map. In contrast, structural violations such as unbalanced brackets or ambiguous nesting boundaries are considered critical errors that render the schema invalid.

4. Lossless convertibility supports conversion to standard JSON and back without losing information.

Syntax Choices. SiMAL's syntax is deliberately similar to a combination of YAML and programming-language code blocks:

– it uses curly braces ({ }) to denote blocks (like JSON objects) but without the need for quoting keys or adding commas;

– it represents lists with "[ ]" (like JSON arrays) but does not require quotes or commas between items;

– it represents key-value pairs with a colon (:) similar to JSON/YAML, but keys are unquoted bare identifiers (unless they contain special characters) and values are usually unquoted if they are alphanumeric or symbols;

– it introduces a heredoc string syntax "<<TEXT ... TEXT" for embedding multi-line text, such as long descriptions, code snippets, or algorithms, as a single string token.

Overall, these choices aim at making SiMAL schemas not only visually shorter but also reduce the number of tokens compared to JSON.

Basic Structure. A SiMAL file represents a complete software system. It always starts with a "system { ... }" block at the top level. Within the system, one can declare simple attributes (key-value pairs), complex attributes (such as lists or dicts), and service blocks (e.g., "service Auth { … }"). Each service represents a major subsystem or module (for example, a microservice in a microservice-based system, or a major component in a monolithic architecture). Services themselves contain attributes and possibly nested component blocks like "table Messages { … }". A parsing-oriented excerpt of the SiMAL grammar in Extended Backus-Naur Form (EBNF) is provided in Fig. 1. The complete grammar specification is available at github.com/NLPForUA/SiMAL/blob/main/ebnf.md.

Annotations. This feature allows encoding metadata without adding full sections in the schema. They are also useful for selection, filtering, or steering LLMs (e.g., an annotation might tell the LLM to ignore certain parts or to note a particular design decision). For example, one can annotate a service with "@DEPRECATED" or "@VERSION(1.2)" or specify cross-links like "@CALLS(other_service)" to indicate inter-service communication. Annotations can be attached to services, components, or even individual attributes to add meta-information. Representing the same thing in JSON might require a nested object.

Components and Fields. Within a service block, one can define arbitrary components such as databases, caches, data structures, etc., using a generic "components" list. Each component in the list has a kind and a name. For example, "database Users { … }" or "struct Address { … }" define two components of kinds "database" and "struct". In a structured JSON representation, this would likely be an array of objects, each with a "kind" and "name" key. In addition, a component might have a "fields" list where each field ("name: type" item) can optionally start with + (public), - (private), or # (protected) to describe visibility.

ISSN 2617-4316 (Print)
ISSN 2663-7723 (Online)

```
Token set: tokens are IDENT/STRING plus punctuation { } [ ] ( ) : , @ ->, NL
(NEWLINE), EOF; numbers are IDENT; strings support quoted literals and heredoc.
Meta-rule: ScalarText(T) = "read token values until a terminator in T occurs at
nesting depth 0, where nesting counts {…}, […], (…); join into a string".
Extended Backus–Naur form excerpt (core syntax):
Schema = NL* "system{" SystemBlock "}" NL* EOF ;
SystemBlock = { NL* SystemItem } ;
SystemItem = AnnotatedService | AnnotatedAttribute ;

Annotations = { Annotation NL* } ;
Annotation = "@" Ident [ "(" AnnArgList? ")" ] ;
AnnArgList = AnnArg { "," AnnArg } ;
AnnArg = AnnArgText ; (* tokens captured inside (...) split by commas *)

Service = "service" Ident "{" { NL* Attribute } NL* "}" NL* ;

AnnotatedAttribute = Annotations? Attribute ;
Attribute = Ident AttributeRhs ;
AttributeRhs = ( ":" NL* Value )
             | ( NL* Map ) (* missing ":" allowed when next token is "{" *)
             | ( NL* List ) (* missing ":" allowed when next token is "[" and
not bracket-literal *)
             | ( NL* ComponentBlockValue ) (* missing ":" allowed when lookahead
is: Ident "{" after a name *) ;
ComponentBlockValue = (* only when there was no ":" and lookahead matches: Ident
"{" after a name *) ComponentBlock ;
Value = Map | List | String | Scalar ;
Scalar = ScalarText({ NL, "}", "]" })
ComponentBlock = Ident Ident "{" Annotations? { NL* Attribute } NL* "}" NL* ;

Map = "{" NL* { MapItem } NL* "}" NL* ;
MapItem = NL* ( MapEntry | RawMapLine ) ;
MapEntry = Annotations? MapKey [ ":" NL* MapValue ] NL* [ "," NL* ] ;
MapKey = Ident | String ;
MapValue = Map | List | String | ScalarText({ ",", NL, "}", "]" }) ;
RawMapLine = MapRawText NL* ; (* any line that cannot be parsed as MapEntry
key/value, or a line starting with "@" not followed by Ident *)

List = "[" NL* [ ListItems ] NL* "]" NL* ;
ListItems = ListItem { ( "," NL* | NL+ ) ListItem } [ "," NL* | NL+ ] ;
ListItem = Annotations? ( Map | ScalarText({",", NEWLINE, "]"}) | ContextItem );
(* certain list keys change item parsing ("methods", "fields", "endpoints",
"components"); details in full grammar *)
```

Fig 1. **EBNF excerpt of the SiMAL grammar**
***Source*: compiled by the authors**

Methods and Endpoints. A unique aspect of SiMAL is how it represents interface methods and API endpoints. Methods of a class/service are listed in a "methods" list, and each method is written in a signature-like syntax with or without an optional body of attributes (Fig. 1).

Fig. 2 demonstrates a public method createUser with Go-like parameter definition and return type. In SiMAL, the method body is allowed to contain any attributes (description, notes, potential issues, etc.). In JSON, a method would likely be an object with multiple fields (name, params, returns, description, etc.).

```
+CreateUser(name, email, password string) -> (user User, err error) {
  description: Registers new user and sends verification email.
  algo: <<TEXT
    1. Validate basic fields and password strength with validateUserI
    2. Hash password with sha256
    3. Insert into users table
    4. Call VerifyUser to send verification email
  TEXT
  analysis: {
    security: [PASSWORD_COMPLEXITY_WEAK]
    linter: [UNHANDLED_ERROR]
  }
}
```

Fig. 2. **Method definition in SiMAL**
***Source*: compiled by the authors**

For external interfaces like API endpoints, SiMAL uses a method-like syntax (Fig. 3).

```
# HTTP
GET /users/{id}
-> JSON{
    user: User{name: str, email: str, verified: bool}?,
    error: str
} [auth: token, cache: 5m, idempotent: true]
# gRPC
CreateUser(CreateUserRequest{email: str, password: str})
-> (id: int?, error: str) [auth: token, rate_limit: 3/m]
```

Fig. 3. **Endpoint definition in SiMAL**

***Source*: compiled by the authors**

SiMAL supports two key endpoint definition styles: HTTP REST and gRPC, which can be used to describe other styles as well. Fig. 3 defines an HTTP GET endpoint with a path parameter {id}, no request body, a JSON response with an error string and optional User object, and an optional list with endpoint metadata. In contrast, gRPC-style endpoints use a function name and message types.

Table 1 compares JSON and SiMAL to illustrate the token savings systematically. Table 2 demonstrates a simplified SiMAL schema for a microservices application.

Table 1. **JSON vs SiMAL: side-by-side comparison**

Concept	JSON	SiMAL	Comment
Key-value attributes	1) "name": "database" 2) "name": "user database"	1) name: database 2) name: user database	Attribute syntax is key: value. Quotes are not required.
Lists/arrays	1) "text": ["one", "one two"] 3) "text": ["one", "one two"]	1) text: [one, "one two"] 3) text: [one one two]	In lists, commas are optional. Newlines also separate items. Trailing commas are tolerated.

Maps/objects	1) "textObject": {
	"text1": "single word",
	“text2”: "two words",
	"text 3": "more words"
	}
	2) "textObject": {
	"single word",
	} # triggers SyntaxError
		1) textObject: {
		text1: single word,
		text2: "two words"
		"text 3": more words
		}
		2) textObject: {
		"single word",
		} # collapses to string
			Commas after map entries are optional. Map keys can be bare identifiers or quoted strings.
			Inside SiMAL maps, non “key: value” lines (sample 2) are captured under “_raw_“ key. If a SiMAL map contains only raw lines it collapses to a string.

Component blocks	{"kind": "database",
	"name": "UserRepo",
	"engine": "postgres-12"}
		database UserRepo {
		engine: postgres-12
		}
			Components are blocks with kind, name, annotations, attributes, etc. Kind and name could be arbitrary.

Annotations	{"kind": "class",
	"name": "User",
	"annotations": [
	{
	"name": "PATH",
	"args": ["path-to-user"]
	},
	{"name": "IGNORE"}
	]}
		@PATH(path-to-user)
		@IGNORE
		class User {
		…
		}
			Syntax: @ + arbitrary name + (optional arguments). Args are split on commas at the top level of the parentheses. Annotations are attached to the following declaration without nesting, improving readability and reducing structural tokens.

Methods	{"name":"sumInts",
	"visibility": "private",
	"params": [
	{"a": "int"}, "b": "int"]
	"returns": "int",
	"attrs": {"desc": "sum ints"}}
		-sumInts(a, b int) -> int {
		desc: sums 2 integers
		}
			Compact Go-like method syntax allows for expressive signatures and optional metadata without heavy nesting.

Endpoints	{"method":"GET", "path":"/users/{id}", "inputs": [{"name":"id", "type":"str"}], "outputs": [{"name": "user", "type": "User", "optional": true}, {"name": "error", "type": "str"}], "attributes": {"cache": "15m"}}	GET /users/{id} -> (user: User?, error: str) [cache: 15m]	Unified endpoint string captures method, path, response shape, and metadata. Strong token efficiency and inline clarity (no need for nested inputs/outputs trees).

***Source*: compiled by the authors**

ISSN 2617-4316 (Print)
ISSN 2663-7723 (Online)

Table 2. **Line-by-line annotation of SiMAL syntax for a sample microservice schema**

Schema	Explanation
system {	Root container for all declarations.
type: microservices description: A fictional ecommerce system context.	Key: value declarations. Keys are arbitrary identifiers; values can be simple or complex types.
@PATH(github.com/org/ecommerce/user-service/)	Annotation binding a declaration to source location
service user_service { description: Handles user authentication and management.	System contains one or more service blocks starting with the service keyword followed by the name.
runtime: { development: {	Service block can contain complex attributes like raw/nested maps, lists, or other complex types.
backend: [k8s, "Azure Container Registry"] replicas: 1 }	Lists could be inline or multi-line. Commas between elements in multi-line maps or lists are optional. Same for element quoting.
@PATH(github.com/org/ecommerce/user-service/deployment/prod/) @BASE(system.user_service.runtime.development)	Multiple annotations could be applied to the same element. Here PATH and BASE are both applied to "production" map.

production: {
replicas: 3-10
}
	Inheritance annotation: production config inherits keys (backend, replicas) and their values from development config unless overridden.

}	End of runtime map definition.
api: [{ type: http endpoints: [	Reserved "api" keyword specifies the API endpoints. Each item in the api list is a map describing an API type (e.g., HTTP, GRPC, GraphQL, Queue/Topic-based, etc.) and list of its endpoints.
GET /users/{id} -> JSON{user: User{name: str, email: str, verified: bool}?, error: str?} POST /users JSON{name: str, email: str, password: str} -> JSON{uuid: str?, error: str?} [auth: none, cache: 5m]] }	HTTP-style endpoint: verb + path + optional request + response + optional attributes in brackets. "?" sign denotes optional parameter. Similarly, one could specify GRPC endpoint: GetUser(GetUserRequest{uuid: str}) -> (user: User{name: str, email: str, verified: bool}?, error: str?).
components: [struct UserService { **fields**: [-database: UserRepo verification: VerificationClient]	Reserved "components" keyword specifies a list of service components of arbitrary kind (e.g., database, class). Component block may contain simple and complex attributes, or even nested components. Reserved keyword "fields" lists the struct fields with optional UML-style visibility quantifier.
methods: [+GetUser(uuid string) -> *User +CreateUser(name, email, password string) -> (user User, err error) {	Methods list specified with reserved keyword follows Golang-like function signatures with types after argument names and named return values. In addition, output could be a simple type or tuple.
algo: <<TEXT 1. Store new user in db via UserRepo.Add(). 2. Send verification email via VerificationClient. TEXT	Method description and other arbitrary metadata could be provided as nested attributes. For instance, CreateUser() algorithm is specified as multi-line string value using heredoc syntax.
}	End of CreateUser() method.
]	End of methods list.
}	End of UserService component.
struct User { fields: [	Definition of User datatype component used by UserService component.
+ID: UUID based on RFC 4122 +Name: string	Field types could be specified with standard list of types (int, str, list, etc.) or using natural language.
]	End of User component fields.
}	End of User component definition.
]	End of components list.
}	End of user_service service definition.
}	End of system definition.

***Source*: compiled by the authors**

From the table above, it can be seen that, overall, the SiMAL schema resembles a simplified configuration file with embedded code-structure elements. SiMAL is a higher-level abstraction than source code in the sense that it abstracts away low-level implementation details (e.g., local helper logic). At the same time, it is designed to preserve machine-checkable structural information, including component boundaries, relationships, and interface-level contracts (e.g., API endpoints and method signatures) together with key metadata.

Implementation-wise, SiMAL schema parsing is supported by an open-source parser built with Python. In addition to simple parsing and syntax-checking scenarios, the developed tool also supports conversion to and from JSON for downstream processing, providing additional flexibility and extending the list of supported use cases. One such use case is the tool created for SiMAL schema visualization. This tool first parses the input schema, converts it to JSON, and then renders a nested system diagram using the Graphviz package [32]. Fig. 4 shows a diagram of the sample microservice schema from Table 2 generated by a visualization tool.

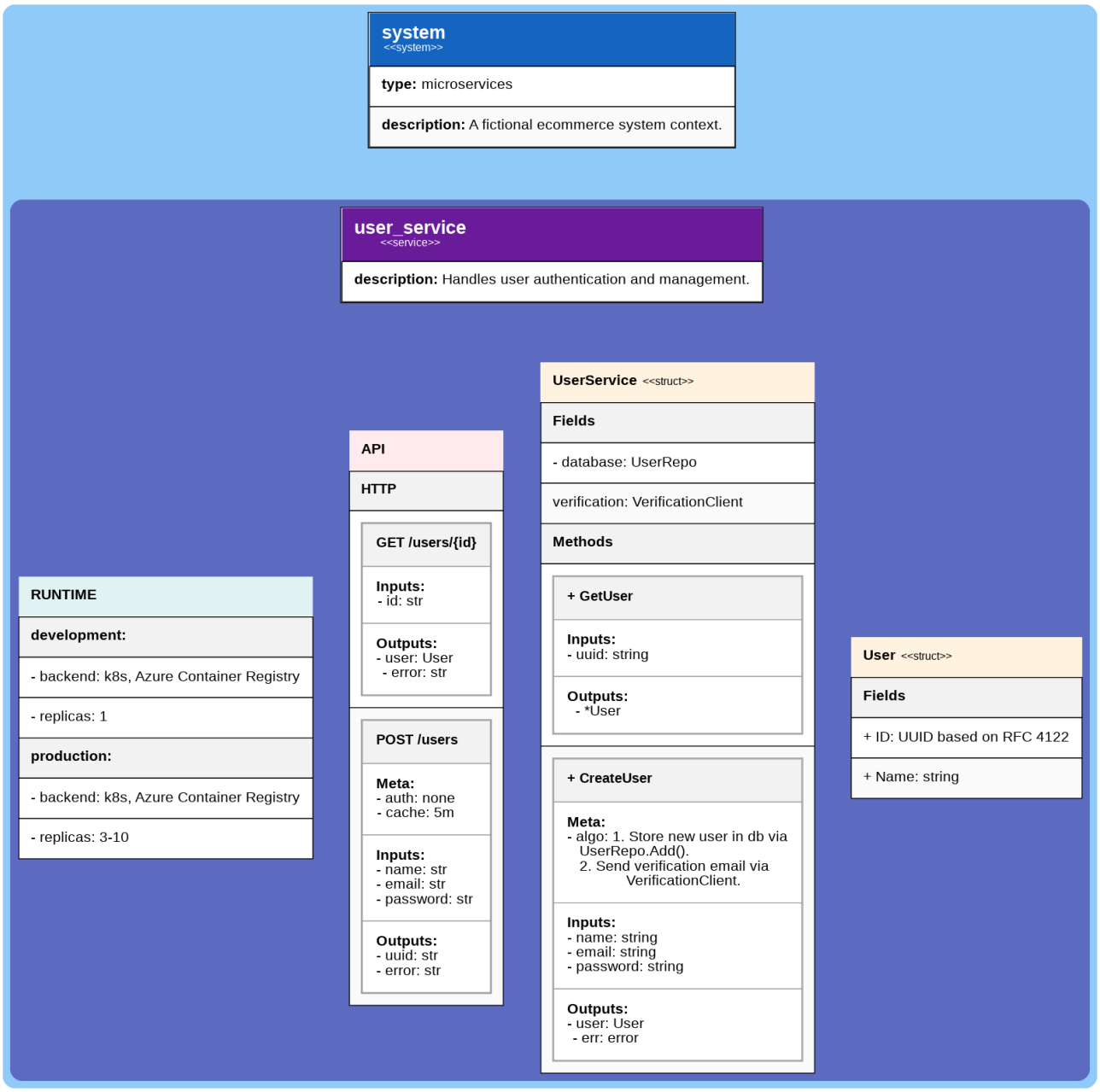

Fig. 4. **Sample microservice schema rendered by the SiMAL visualization tool**
***Source*: compiled by the authors**

This hierarchical representation highlights the separation between service-level declarations and nested substructures such as runtime configuration, API endpoints, and internal components. This visualization also makes the schema easier to inspect for completeness and contradictions (e.g., missing dependency targets or inconsistent endpoint signatures). Consistent with approaches using visual dashboards for data quality evaluation [33], such a visual representation serves as an additional "sanity check" layer rather than an independent, potentially inconsistent artifact.

In LLM-driven workflows, this diagram view accelerates onboarding by revealing major components compactly, while the schema text remains the primary format for prompting. This multimodality enables human-centric verification without introducing additional specification formats.

However, compact visual and textual schema formats, accompanied by token-efficient representations, are not sufficient for integrating DSL into LLM-driven software engineering workflows. The key aspect is how accurately the language model understands, edits, or generates the SiMAL schema from scratch for projects of different scales and technology stacks.

EXPERIMENTAL SETUP

SiMAL evaluation prioritizes two criteria: token efficiency and schema quality. Token efficiency compares the number of tokens required to represent the same content in SiMAL and JSON, while schema quality evaluates whether the schema accurately captures source data. JSON was selected as the primary baseline because it is the dominant structured format in LLM-oriented pipelines: tool/function calling, structured outputs, and machine-checkable responses are typically defined as JSON objects, and the LLM tooling ecosystem is optimized for producing syntactically valid JSON [30]. JSON is also widely used in software engineering as a canonical data interchange format, making it the most relevant comparison point for a prompt-oriented structured representation.

Baseline selection was constrained by reproducibility and task alignment. A candidate baseline must:

- be commonly used in LLM prompting;
- provide comparable expressiveness for system schemas (typed components, interfaces/endpoints, cross-component relations);
- have a canonical form that can be compared directly without additional semantic resolution.

YAML and unconstrained free-form outputs were tested in early experiments, but incremental updates of large schemas across many iterations frequently produced syntactic corruption (e.g., indentation drift and unterminated structures), motivating the use of JSON with "structured output" mode. Compact configuration languages such as TOML or HOCON were excluded since they are primarily configuration authoring formats rather than repository modeling languages, and their permissive syntax and resolution mechanisms (e.g., substitutions, merges, concatenation) make the evaluation target ambiguous. Broader comparisons to additional DSLs are therefore left for future work.

For evaluation, 10 diverse repositories (Table 3), spanning web applications, libraries, and services, were selected from GitHub.

Table 3. **Repositories selected for evaluation**

GitHub Project	Project tokens	Description	Tech stack
JuniorTest	35905	News web app + REST API	PHP (Laravel backend), Vue.js frontend
OHLCFormer	29452	Stock forecasting toolkit	Python, ML (Pytorch and Transformers)
Dashboard-reactjs	22003	UI dashboard template	JavaScript/TypeScript (React SPA)
full-stack-fastapi-template	143499	Full-stack app scaffold	FastAPI, SQLModel, Postgres; Python; React, TypeScript, Vite; Docker
microservice-app-example	21368	Polyglot microservices demo	Vue frontend; Go auth API; Node TODOs API; Java Spring Boot users API; Python worker; Redis/Zipkin
otel-python-cloud-run	13014	Observability demo services	Python microservices instrumented with OpenTelemetry; Google Cloud Run
spring-food-delivery-microservices	200762	Food delivery microservices	Java Spring Boot; DDD; RabbitMQ; event-driven
wild-workouts-go-ddd-example	140366	Domain-driven design (DDD) reference app	Go backend; gRPC/OpenAPI; Cloud Run, Firebase; Terraform; web frontend
sqlmodel	240793	Database ORM library	Python (Pydantic, SQLAlchemy)
tokenizers	542769	ML tokenization library	Rust core; Python and Node bindings

***Source*: compiled by the authors**

Selected repositories include diverse front-end and back-end projects. The project sizes (total codebase tokens) range from 13 thousand to 542 thousand, covering small utilities and large-scale systems. This variety evaluates how SiMAL performs across different scales and structures.

Multiple schema iterations were generated for each project to simulate an incremental documentation process. A GPT-5 series model (specifically gpt-5-2025-08-07), demonstrating leading coding capabilities [34], was prompted to read chunks of the project codebase (folder by folder) along with the SiMAL schema produced in the previous iteration to generate an updated schema. This yields a sequence of schema snapshots per project, as if the model is progressively documenting more details of the project. The reason for multiple iterations is to ensure the schema covers the entire project in stages and to obtain more data points for token analysis. In total, 394 schema files were produced (each project had between 12 and 78 iterations depending on size). Similarly, JSON schemas were generated for selected projects under identical conditions (using the same model, context window, and iterative prompting procedure) to ensure a fair comparison. Later, every SiMAL schema file was converted to pretty-printed JSON (with 2-space indentation) and to a maximally compact single-line JSON for fairness. These variants also preserve endpoints and functions as strings rather than as objects, thereby retaining the original SiMAL content.

Two tokenizers were selected to evaluate token efficiency: the OpenAI GPT tokenizer (o200k_base, used by the GPT-5 series) [35] and the Gemma 3 tokenizer [36]. The rationale is that different models have different tokenization. For instance, GPT's BPE-based algorithm might split certain strings more finely than Gemma's SentencePiece-based tokenizer. Token counts for each file were obtained by direct library calls and verified for consistency.

To assess the quality of generated schemas, an "LLM-as-a-Judge" technique, widely used to assess LLM-generated content [37], was employed. Three state-of-the-art models were used as "judges": Google Gemini 3 Pro [11], Anthropic Claude Sonnet 4.5 [38], and OpenAI GPT-5.2 [39]. Within a single evaluation request, each judge was provided with the full project context (the contents of all project files) and either the final SiMAL or the final JSON schema for that project. The judges were asked to output a structured JSON report containing six category scores on a discrete 0-5 scale (or -1 for "Not Applicable"): schema coverage, schema accuracy, API accuracy, internal structure accuracy, annotation quality, and non-hallucination. Scores follow a consistent grading where 5 indicates near-complete correctness, 4 indicates minor issues, 3 indicates partial usability, and lower values represent significant mismatch. "Not Applicable" is allowed only when the corresponding aspect is absent from the repository and not claimed by the schema (e.g., no runtime/deployment configs when the repo is a library). To minimize judge scoring variance, the final quality score is recalculated deterministically from the judge report. First, category scores are converted into a weighted base score using fixed weights: coverage (20), accuracy (20), API (20), structure (15), annotations (10), and non-hallucination (15), totaling 100. If a category is marked as "Not Applicable" (-1), its weight is excluded from normalization, and the remaining categories are scaled to preserve a 0-100 range. The weighted base is then reduced by explicit penalty counts reported by the judge and corresponding severity multipliers: missed major components (7), missed minor components (2), spurious components (4), incorrect API (3), incorrect definitions (3), and incorrect visibility flags (1). The resulting score is then clamped to [0, 100].

To mitigate judge uncertainty, each schema was evaluated in 3 independent runs [40] per model with mean score indicating schema quality. GPT judge skipped the 2 largest projects due to input length limits. Issues detected by judges were manually reviewed to avoid anomalies or hallucinations. Overall, while being subjective, these scores provide a comparative signal: if SiMAL schemas score on par with JSON, then it means SiMAL most likely did not omit crucial details.

To strengthen reliability beyond subjective LLM-based judging, a set of deterministic non-LLM metrics was computed. First, structural validity was measured across all schema iterations: whether each schema is parsable and satisfies minimal well-formedness constraints (exactly one system and at least one service), along with parsability statistics for key elements such as method signatures and argument lists. Second, annotation integrity was evaluated: "PATH" and "CALLS" scores were computed as annotation presence rate multiplied by annotation correctness, where correctness validates "PATH" links against the repository directory structure and validates "CALLS" by checking that referenced targets exist in the schema. Third, a simplified schema-to-code correspondence metric was computed on the final schema of each project by extracting entities from source code and matching

them to schema components and methods by name, producing precision and recall.

Since intermediate schemas are intentionally incomplete during iterative construction, correspondence was computed only for the latest schema per project. The correspondence check is conservative and simplified: the source entity extractor is lightweight and may miss entities or misclassify kinds, while schemas may include higher-level abstractions that do not map one-to-one to code entities. So common non-code categories were filtered to reduce systematic false positives.

RESULTS

Across all 394 files, SiMAL schemas consistently used fewer tokens than equivalent JSON representations. Table 4 presents the corpus-level metrics for pretty-printed JSON with an indent of 2 spaces, single-line dedented JSON, and multi-line SiMAL with and without indentation.

Compared to indented JSON, indented SiMAL reduced total GPT tokens from 7.92 million to 5.90 million (-25.5%). However, the most compact variant, dedented SiMAL, further reduced the total to 5.20 million GPT tokens, yielding a 34.3% reduction compared to indented JSON. A similar trend is observed for the Gemma tokenizer: dedented SiMAL required 6.37 million tokens versus 9.66 million for indented JSON (-34.1%), while also outperforming single-line JSON by 10.2%. These stats support the claim that SiMAL meets its key purpose of packing more information per token.

Token usage evaluation results per project are demonstrated in Table 5 (indented multi-line SiMAL vs. indented multi-line JSON) and Table 6 (dedented multi-line SiMAL vs. dedented single-line JSON).

Table 4. **Token usage comparison (across 394 files/iterations)**

Format	Bytes	GPT tokens	Gemma tokens	Average GPT tokens per file	Average Gemma tokens per file
JSON (single-lined and dedented)	23,575,158	6,459,374	7,090,717	16,394	17,997
JSON (multi-lined with indent of 2)	37,379,088	7,919,025	9,656,851	20,099	24,510
SiMAL (multi-lined with indent)	26,075,093	5,900,298	7,083,775	14,975	17,979
SiMAL (multi-lined and dedented)	**20,392,002**	**5,204,402**	**6,368,362**	**13,209**	**16,163**

***Source*: compiled by the authors**

Table 5. **Token usage per project: JSON (multi-line, indented) vs SiMAL (multi-line, indented) (Δ = SiMAL – JSON tokens; negative means SiMAL is more efficient)**

GitHub Project	Files / Iteration	GPT tokens (JSON → SiMAL)	GPT Δ	GPT Δ%	Gemma tokens (JSON → SiMAL)	Gemma Δ	Gemma Δ%
JuniorTest	31	246,345 → 172,507	-73,838	-30.0%	299,688 → 203,471	-96,217	-32.1%
OHLCFormer	12	62,733 → 47,374	-15,359	-24.5%	76,965 → 56,520	-20,445	-26.6%
dashboard-reactjs	26	196,564 → 166,818	-29,746	-15.1%	238,590 → 203,388	-35,202	-14.8%
full-stack-fastapi-template	39	793,822 → 613,405	-180,417	-22.7%	962,485 → 735,031	-227,454	-23.6%
microservice-app-example	29	175,694 → 127,267	-48,427	-27.6%	217,023 → 153,607	-63,416	-29.2%
otel-python-cloud-run	14	61,681 → 44,746	-16,935	-27.5%	76,291 → 54,636	-21,655	-28.4%
spring-food-delivery-microservices	63	1,506,754 → 1,186,694	-320,060	-21.2%	1,857,938 → 1,433,990	-423,948	-22.8%
wild-workouts-go-ddd-example	61	1,573,627 → 1,140,315	-433,312	-27.5%	1,914,980 → 1,372,588	-542,392	-28.3%
Average (8 projects)	**34.4**	**577,153 → 437,391**	**-139,762**	**-24.2%**	**705,495 → 526,654**	**-178,841**	**-25.3%**
sqlmodel	41	870,883 → 650,840	-220,043	-25.3%	1,046,047 → 768,631	-277,416	-26.5%
tokenizers	78	2,430,922 → 1,750,332	-680,590	-28.0%	2,966,844 → 2,101,913	-864,931	-29.2%
Average (10 projects)	**39.4**	**791,903 → 590,030**	**-201,873**	**-25.5%**	**965,685 → 708,378**	**-257,307**	**-26.6%**

***Source*: compiled by the authors**

Table 6. **Token usage per project: JSON (single-line, dedented) vs SiMAL (multi-line, dedented) (Δ = SiMAL – JSON tokens; negative means SiMAL is more efficient)**

GitHub Project	Files / Iteration	GPT tokens (JSON → SiMAL)	GPT Δ	GPT Δ%	Gemma tokens (JSON → SiMAL)	Gemma Δ	Gemma Δ%
JuniorTest	31	194,510 → 145,832	-48,678	-25.0%	207,888 → 176,528	-31,360	-15.1%
OHLCFormer	12	53,044 → 43,058	-9,986	-18.8%	59,809 → 52,000	-7,809	-13.1%
dashboard-reactjs	26	171,773 → 146,424	-25,349	-14.8%	194,509 → 182,603	-11,906	-6.1%
full-stack-fastapi-template	39	667,027 → 554,606	-112,421	-16.9%	734,920 → 674,488	-60,432	-8.2%
microservice-app-example	29	138,988 → 110,729	-28,259	-20.3%	153,007 → 137,091	-15,916	-10.4%
otel-python-cloud-run	14	50,591 → 39,583	-11,008	-21.8%	56,545 → 49,286	-7,259	-12.8%
spring-food-delivery-microservices	63	1,253,163 → 1,056,167	-196,996	-15.7%	1,411,450 → 1,300,907	-110,543	-7.8%
wild-workouts-go-ddd-example	61	1,280,611 → 1,002,777	-277,834	-21.7%	1,399,248 → 1,226,642	-172,606	-12.3%
Average (8 projects)	**34.4**	**476,213 → 387,397**	**-88,816**	**-18.7%**	**527,172 → 474,943**	**-52,229**	**-9.9%**
sqlmodel	41	706,327 → 567,839	-138,488	-19.6%	756,428 → 683,244	-73,184	-9.7%
tokenizers	78	1,943,340 → 1,537,387	-405,953	-20.9%	2,116,913 → 1,885,573	-231,340	-10.9%
Average (10 projects)	**39.4**	**645,937 → 520,440**	**-125,497**	**-19.5%**	**709,072 → 636,836**	**-72,236**	**-10.2%**

***Source*: compiled by the authors**

In Table 6, SiMAL is evaluated in a dedented multi-line form to provide a closer apples-to-apples comparison against single-line JSON. In the indented comparison, SiMAL achieves consistent reductions across all projects, ranging from about 15% (front-end-heavy repositories) to 30% (back-end services and structured systems). In the dedented comparison against single-line JSON (Table 6), SiMAL still remains consistently more token-efficient, reducing GPT token counts across all evaluated projects and improving the overall average substantially. This indicates that SiMAL's advantage is not limited to whitespace or formatting effects: even when JSON is aggressively compressed, SiMAL still reduces overhead by eliminating repeated structural tokens and enabling dense yet readable hierarchical representation.

The results obtained confirm that SiMAL can present system information with higher efficiency compared to JSON. By using SiMAL schemas as prompts, one can include up to ~25-30% more content within the same token limit, such as additional files or deeper contextual details that would otherwise have been omitted. What is even more important is that this gain comes without any model fine-tuning.

Still, it is important to note a few nuances. First, a highly optimized JSON representation (single-line and with shortened fields) can partially approach the size of indented SiMAL. In our experiments, the "max-simple" JSON achieves this mainly by encoding complex elements (such as endpoint or method signatures) as raw strings, effectively embedding a SiMAL-like compact syntax inside JSON. However, this optimization reduces readability and makes parts of the structure harder to parse deterministically or modify incrementally. SiMAL, in contrast, preserves readability while supporting further prompt-oriented optimizations such as dedented formatting, which provides strong savings even against minified JSON. Second, differences in tokenizers affect absolute numbers, meaning the magnitude of the gain depends on the target model family, though the overall trend remains consistent across both tokenizers.

Overall, SiMAL achieves substantial token savings, validating the initial hypothesis. However, token compaction alone is not sufficient without validating schema quality. Table 7 reports the averaged scores assigned by each judge model (Google Gemini 3 Pro, Claude Sonnet 4.5, OpenAI GPT-5.2) across eight small-to-medium projects, as well as two large projects (evaluated only by Gemini and Claude due to GPT-5.2 context limits). Each score is computed on a 0-100 scale by combining multiple criteria such as coverage and correctness.

Table 7. **Quality evaluation: SiMAL vs JSON (both multi-line, indented)**

GitHub Project	Project Tokens	JSON/SiMAL schema tokens	JSON			SiMAL		
			Gemini	Claude	GPT	Gemini	Claude	GPT
JuniorTest	35905	14429 / 9592	81	75.33	69	**100**	**85**	**82.33**
OHLCFormer	29452	10948 / 7545	97.33	90	**85.67**	**100**	**95.67**	80.33
dashboard-reactjs	22003	17029 / 13849	100	88	80.33	100	**100**	**86**
full-stack-fastapi-template	143499	43075 / 27086	100	**86**	**82.33**	100	65.67	63
microservice-app-example	21368	11891 / 6317	93	**77**	61.33	**97.67**	72.67	**76.67**
otel-python-cloud-run	13014	9242 / 5104	98.33	94	92.67	**100**	**97.33**	**95.33**
spring-food-delivery-microservices	200762	52097 / 41793	100	55.33	62	100	**88.33**	**67.67**
wild-workouts-go-ddd-example	140366	60046 / 32782	97.67	**95.33**	**80**	**100**	71.33	64
Average (8 projects, n=24)	75796	27345 / 18009	95.92	82.62	76.67	**99.71**	**84.50**	**76.92**
sqlmodel	240793	68411 / 18860	**74**	**73**	—	64.67	58.33	—
tokenizers	542769	83949 / 46263	**100**	**94**	—	94.67	29.67	—
Average (10 projects, n=30)	138993	37112 / 20919	94.13	**82.80**	—	**95.70**	76.40	—

***Source*: compiled by the authors**

It is also important to interpret Table 7 in the context of judge-specific scoring behavior. Based on manual inspection of evaluation results, Gemini appears less strict in its penalty assignment: it often collapses multiple issues of the same type into a single problem and prioritizes high-level correctness (overall structure, major components, key interfaces) over deep implementation details or minor schema typos. This makes Gemini scores more topology-driven and generally higher when the schema captures the main architectural picture. In contrast, Claude demonstrates a more conservative scoring style, frequently penalizing small inconsistencies and occasionally focusing way too much on source-level imperfections, which can disproportionately affect schema evaluation despite Claude's strong long-context comprehension. Finally, GPT-5.2 tends to act as a practical middle ground: it follows the grading instruction more consistently than Gemini while avoiding excessive penalties, and it reliably identifies major issues, hallucinations, and critical accuracy errors even in long inputs. To sum it all up, the observed differences reflect known reliability limitations and judge-specific biases in LLM-as-a-judge evaluation [41], so cross-model averages should be interpreted as complementary perspectives rather than absolute truth.

The main result is that SiMAL schemas performed on par with JSON schemas in quality, and in some cases slightly better. For the eight projects evaluated by all three models, SiMAL's average score was essentially comparable to JSON (within <0.3 points for GPT-5.2), while being higher by ~3.8 points with Gemini and ~1.9 points with Claude. When including the two largest projects ("sqlmodel" and "tokenizers"), Claude's average for SiMAL decreases (SiMAL ~76 vs JSON ~82), suggesting that for extremely large schemas (tens of thousands of lines), Claude may perceive JSON as more precise. This may be influenced by format familiarity (JSON being strongly represented in training data), while the large input length can also penalize recall and increase sensitivity to minor inconsistencies.

To complement the LLM-as-a-judge protocol, Tables 8 and 9 report non-LLM evaluation signals computed deterministically from schema artifacts and repository source code. Table 8 aggregates structural validity metrics across the full set of 394 schemas, including schema parsability and basic structural well-formedness (a single system and at least one service), annotation validity for "PATH" and "CALLS", and parsability statistics for function/method signatures (inputs/outputs). Table 9 evaluates schema-to-code correspondence on the final schema for each project by extracting entities from the repository source code and matching them to schema components and functions/methods by name.

Table 8. **Deterministic structural validity metrics for SiMAL vs JSON (both multi-line, indented) (Structure – binary schema validity (0 or 1); Path – "PATH" annotation validity (0-1); Calls – "CALLS" annotation validity (0-1); Req/Resp – parsable input/output argument fractions; Q – function/method count; "—" indicates that no instances of the corresponding element are present)**

GitHub Project	Structure		Annotation				Methods					
	JSON	SiMAL	JSON		SiMAL		JSON			SiMAL		
			Path	Calls	Path	Calls	Req	Resp	Q	Req	Resp	Q
JuniorTest	1	1	0.88	—	**0.89**	—	**1**	1	1594	0.54	1	1738
OHLCFormer	0	**1**	0	—	**0.95**	—	0	0	0	**0.66**	1	617
dashboard-reactjs	1	1	0.91	0	**0.92**	—	0	**1**	12	—	—	0
full-stack-fastapi-template	1	1	**0.88**	0.41	0.80	**0.50**	**1**	1	1386	0.81	1	1366
microservice-app-example	1	1	**0.93**	0.57	0.85	**0.71**	**0.78**	1	644	0.74	1	782
otel-python-cloud-run	1	1	**1**	0.5	0.88	0.36	0.69	1	26	**1**	**1**	53
spring-food-delivery-microservices	1	1	**0.89**	0	0.88	**0.12**	**1**	1	4095	0.76	1	2179
wild-workouts-go-ddd-example	1	1	**1**	**0.49**	**0.92**	0	**0.92**	0.99	4151	0.86	**1**	9595
sqlmodel	1	1	0.72	0	**0.92**	—	**1**	1	1193	0.25	1	4356
tokenizers	1	1	**0**	0.38	**0.92**	—	0.63	0.85	531	**0.70**	**1**	17707

***Source*: compiled by the authors**

Table 9. **Schema-to-code correspondence on final schemas: SiMAL vs JSON (both multi-line, indented) (P – precision; R – recall; Q/T – schema entities / source-extracted entities; evaluated on final schema per project; dashboard-reactjs project skipped due to source entity extraction limitations)**

GitHub Project	Components						Methods					
	JSON			SiMAL			JSON			SiMAL		
	P	R	Q/T	P	R	Q/T	P	R	Q/T	P	R	Q/T
JuniorTest	0.37	0.54	67/46	**0.51**	**0.89**	81/46	1	0.07	39/537	1	**0.11**	58/537
OHLCFormer	0	0	0/35	**0.54**	**0.89**	57/35	0	0	0/91	**1**	**0.95**	86/91
full-stack-fastapi-template	**0.58**	0.19	38/118	0.18	**0.26**	171/118	**1**	0.13	36/270	0.96	**0.28**	79/270
microservice-app-example	**0.5**	**0.90**	38/21	0.39	0.76	41/21	**1**	0.59	32/54	0.9	**0.67**	40/54
otel-python-cloud-run	0.67	0.43	9/14	**0.29**	**0.79**	38/14	1	0.14	3/21	1	**0.48**	10/21
spring-food-delivery-microservices	**0.79**	0.20	131/518	0.76	**0.25**	171/518	0.83	**0.24**	300/1030	**1**	0.18	184/1030
wild-workouts-go-ddd-example	0.28	**0.12**	64/151	**0.46**	**0.65**	213/151	0.96	0.06	25/387	**1**	**0.68**	266/387
Average (7 projects)	**0.46**	0.34	—	0.45	**0.64**	—	0.83	0.18	—	**0.98**	**0.48**	—
sqlmodel	0	0	8/51	**0.23**	**0.25**	56/51	0	0	0/214	**0.95**	**0.20**	44/214
tokenizers	0	0	132/394	**0.31**	**0.22**	278/394	0.3	0.01	40/1014	**0.69**	**0.14**	209/1014
Average (9 projects)	0.35	0.26	—	**0.41**	**0.55**	—	0.68	0.14	—	**0.94**	**0.41**	—

***Source*: compiled by the authors**

Table 8 shows that both formats are generally parsable, but structural failures are format- and project-dependent. In particular, JSON can become structurally invalid under iterative updates, whereas SiMAL remains structurally valid in the same setting. As for annotation integrity, "PATH" validity is consistently high, indicating that schemas usually attach file-system links that exist in the repository. The "CALLS" validity metric highlights that cross-component linkage is sensitive to schema completeness and naming consistency, and can degrade in late iterations if references drift or are omitted.

One thing to note is that component precision and recall from Table 9 are not intended to be interpreted as full repository extraction quality. First, schemas contain both code-derived entities and higher-level abstractions (e.g., configuration, deployment, and infrastructure artifacts) that do not map one-to-one to source code elements. Second, the source extractor is intentionally lightweight and can miss entities. Third, repository source code contains many low-level helpers and boilerplate elements that are intentionally omitted by instruction to keep the schema compact and task-relevant. Under these constraints, correspondence metrics are primarily informative as precision-oriented indicators of whether claimed entities are grounded in the repository, rather than as exhaustive coverage of all elements.

With these caveats, the correspondence results indicate that SiMAL is on par with JSON for component precision and typically stronger on recall, while method/function correspondence is substantially stronger for SiMAL in both precision and recall across most projects. In addition, late-iteration JSON schemas occasionally exhibit structural drift in key naming (e.g., "methods" vs. "functions"), which can degrade deterministic matching even when the content remains interpretable to an LLM. Such cases were intentionally left unnormalized to keep the non-LLM checks strictly machine-verifiable.

Taken together, the deterministic checks support the same overall conclusion as token and judge-based evaluations: SiMAL provides substantial prompt compression while remaining structurally usable under iterative schema construction, and it does not introduce systematic degradation compared to JSON under the same generation conditions.

Finally, it is important to note that both JSON and SiMAL schemas in this evaluation were generated by an LLM and may contain inaccuracies relative to the source code. Therefore, the objective here is not absolute truthfulness, but whether SiMAL causes additional loss or error relative to a well-established baseline under the same generation conditions. The key insight is that SiMAL did not introduce additional errors or loss. In practice, if a language model or a developer writes a SiMAL file manually, it should be as reliable as writing a JSON version.

CONCLUSIONS

This research introduced SiMAL, a domain-specific modeling language designed to optimize software system representation for LLM-driven software engineering.

The scientific novelty of the obtained results is that a prompt-oriented, machine-readable system modeling language is proposed and empirically validated as a token-efficient alternative to generic structured formats such as JSON, which is widely used in LLM tooling. SiMAL provides a single compact schema artifact that preserves hierarchy, evidence links, and cross-component relationships, while reducing syntactic overhead.

As a result, this work produced the following practical outcomes.

1. An open-source parsing toolchain for a designed DSL was implemented to support syntax validation, structured extraction, and bidirectional conversion between SiMAL and JSON. This makes the representation usable in multiple workflows: it can serve as an LLM-friendly prompt schema, a machine-readable intermediate artifact, or a source input for downstream tooling.

2. A token usage evaluation across a multi-project schema corpus demonstrated that SiMAL consistently reduces token usage compared to equivalent JSON representations. Across 394 files/iterations, SiMAL achieved approximately 25-30% token savings relative to pretty-printed JSON, while remaining competitive even against strongly compressed JSON baselines.

3. A schema quality evaluation protocol was introduced to assess schemas parsability and how accurately they reflect repository content (coverage, correctness, consistency, structure validity, annotation quality, hallucination resistance). Obtained results indicate that, in small- and mid-size contexts, SiMAL schemas are on par with JSON schemas in quality and, in multiple cases, slightly better, indicating that token compaction does not inherently lead to a loss of meaningful system information.

The practical significance of this work is that SiMAL provides a compact interface between software repositories and LLM reasoning. By adopting a SiMAL-based workflow, developers and AI assistants can pack substantially more system context into prompts within fixed limits, reducing cost and mitigating long-context degradation effects. Moreover, SiMAL can complement retrieval-based pipelines by acting as a structured "repository map" and evidence index that guides selective retrieval when exact source logic is required.

Limitations and applicability boundaries should be acknowledged. First, SiMAL is an LLM-first representation: its main benefit arises in workflows where repository context must be repeatedly communicated to language models under strict token budgets. For workflows not involving LLM-assisted analysis or generation, SiMAL may be less beneficial than conventional documentation or configuration formats. Second, SiMAL does not replace source code. For tasks requiring exact implementation logic (e.g., debugging a specific algorithm, verifying edge cases, or analyzing control flow), the schema alone is insufficient. The primary role of schema is to provide orientation and guide targeted retrieval of relevant code artifacts. Third, for small repositories or narrowly scoped tasks where all relevant context already fits into a prompt, the overhead of producing and maintaining a schema may not be reasonable.

From an adoption perspective, SiMAL aims to be human-readable and learnable by reusing familiar constructs: a YAML-like block structure for hierarchical data and concise Go-like function signatures. Deterministic parsing provides immediate feedback during editing, and bidirectional conversion to JSON offers an alternative for teams that prefer conventional structured formats. Nevertheless, SiMAL remains an additional notation that developers need to learn and maintain.

Finally, several evaluation-related limitations remain. The quality evaluation partially relies on the "LLM-as-a-judge" paradigm, which is known to vary across judge models, instruction style, and long-context behavior. While the grading strategy was formalized and applied consistently, judge-specific bias remains visible. Therefore, judge scores should be interpreted as comparative signals under controlled conditions rather than absolute truthfulness measures. The corpus, although diverse, is limited to 10 open-source repositories and may not fully generalize to large proprietary monorepos. In addition, both JSON and SiMAL schemas were generated iteratively by an LLM under a fixed procedure, anchoring results to a particular prompting strategy. Alternative prompting or human-authored schemas could slightly shift both token usage and quality metrics. Finally, the evaluation focuses on representation efficiency and schema quality, rather than measuring downstream task performance end-to-end. While token savings and schema correctness and completeness are necessary prerequisites, the practical impact on real workflows like code question-answering or feature implementation requires additional studies with task-specific metrics and human validation.

Future work will focus on evaluating SiMAL in end-to-end agent workflows such as guided code generation and automated code review under strict system constraints. Another promising direction is hybrid pipelines, where SiMAL provides a compact global backbone while retrieval supplies local implementation fragments only when needed.

The implementation and evaluation artifacts are available at: https://github.com/NLPForUA/SiMAL.

ACKNOWLEDGMENTS

The authors disclose limited use of AI tools during implementation. GitHub Copilot Chat (GPT-5-Codex) provided assistance in debugging the parser and resolving execution issues. GPT-5.2 assisted in developing Graphviz visualization components. Gemini 3 Pro was used to assist with reference list formatting and to translate the English abstract into Ukrainian. The authors reviewed, edited, and verified all generated suggestions and remain responsible for the final content.

REFERENCES

1. "Advanced version of Gemini with Deep Think officially achieves gold-medal standard at the International Mathematical Olympiad". *Google DeepMind Blog*. 2025. – Available from: https://deepmind.google/blog/advanced-version-of-gemini-with-deep-think-officially-achieves-gold-medal-standard-at-the-international-mathematical-olympiad. – [Accessed: Jan 2026].

2. "OpenAI". *ICPC World Finals Baku 2025*. 2025. – Available from: https://worldfinals.icpc.global/2025/openai.html. – [Accessed: Jan 2026].

ISSN 2617-4316 (Print)
ISSN 2663-7723 (Online)

3. Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O. & Narasimhan, K. “SWE-bench: Can Language Models Resolve Real-World GitHub Issues?”. In *Proc. International Conference on Learning Representations (ICLR)*. Vienna, Austria. 2024. DOI: https://doi.org/10.48550/arXiv.2310.06770.

4. “About GitHub Copilot code review”. *GitHub Docs*. 2025. – Available from: https://docs.github.com/en/copilot/concepts/agents/code-review. – [Accessed: Jan 2026].

5. Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Vats, I., Moazen, H. & Sarro, F. “A survey on machine learning techniques applied to source code”. *Journal of Systems and Software*. 2024; 207: 111864, https://scopus.com/pages/publications/85181046174. DOI: https://doi.org/10.1016/j.jss.2023.111934.

6. “Claude 1M Context”. *Claude Blog*. 2025. – Available from: https://claude.com/blog/1m-context. – [Accessed: Jan 2026].

7. Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F. & Liang, P. “Lost in the Middle: How language models use long contexts”. *Transactions of the Association for Computational Linguistics*. 2024; 12: 157–173. DOI: https://doi.org/10.1162/tacl_a_00638.

8. Nair, V. “The Context Window Problem: Scaling Agents Beyond Token Limits”. *Factory.ai Blog*. August 2025. – Available from: https://factory.ai/news/context-window-problem. – [Accessed: Jan 2026].

9. Liu, X., Lan, B., Hu, Z., Liu, Y., Zhang, Z., Wang, F., Shieh, M. & Zhou, W. “CodexGraph: Bridging Large Language Models and Code Repositories via Code Graph Databases”. In *Proc. 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics (NAACL)*. Albuquerque, NM, USA. 2025. p. 142–160. DOI: https://doi.org/10.18653/v1/2025.naacl-long.7.

10. “PlantUML Contributors. F.A.Q.” *PlantUML.com*. 2026. – Available from: https://plantuml.com/faq. – [Accessed: Jan 2026].

11. “Gemini 3 Pro”. Google DeepMind. 2025. – Available from: https://deepmind.google/models/gemini/pro. – [Accessed: Jan. 2026].

12. “Anthropic. Context Windows”. *Claude API Docs*. 2025. – Available from: https://platform.claude.com/docs/en/build-with-claude/context-windows#1-m-token-context-window. – [Accessed: Jan 2026].

13. Du, Y., Tian, M., Ronanki, S., Rongali, S., Bodapati, S., Galstyan, A., Wells, A., Schwartz, R., Huerta, E. A. & Peng, H. “Context Length Alone Hurts LLM Performance Despite Perfect Retrieval”. In *Proc. Findings of the Association for Computational Linguistics: EMNLP*. Suzhou, China. Nov. 2025. p. 23281–23298. DOI: https://doi.org/10.18653/v1/2025.findings-emnlp.1264.

14. Hong, E. et al. “Context Rot: How Increasing Input Tokens Impacts LLM Performance”. *Chroma Research Report*. 2025. – Available from: https://research.trychroma.com/context-rot. – [Accessed: Jan 2026].

15. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S. & Kiela, D. “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*. 2020; 33: 9459–9474. DOI: https://doi.org/10.48550/arXiv.2005.11401.

16. Lu, S., Duan, N., Han, H., Guo, D., Hwang, S.W. & Svyatkovskiy, A. “ReACC: A retrieval-augmented code completion framework”. *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2022. p. 6227–6240, https://scopus.com/pages/publications/85141307639. DOI: https://doi.org/10.18653/v1/2022.acl-long.431.

17. Ouyang, S., Yu, W., Ma, K., Xiao, Z., Zhang, Z., Jia, M., Han, J., Zhang, H. & Yu, D. “RepoGraph: Enhancing AI software engineering with repository-level code graph”. In *Proc. International Conference on Learning Representations (ICLR)*. Singapore. 2025. p. 30361–30384. DOI: https://doi.org/10.48550/arXiv.2410.14684.

Edge, D., Trinh, H., Truitt, S. & Larson, J. “From Local to Global: A GraphRAG Approach to Query-Focused Summarization”. *arXiv preprint*. 2024. – DOI: https://doi.org/10.48550/arXiv.2404.16130.

18. Zhang, X. & Chen, T. “Automatic source code summarization with graph attention networks”. Journal of Systems and Software. 2022; 188: 111257, https://scopus.com/pages/publications/85126290388. DOI: https://doi.org/10.1016/j.jss.2022.111257.

19. Gauthier, P. “Repository map”. *Aider Documentation*. 2025. – Available from: https://aider.chat/docs/repomap.html. – [Accessed: Jan 2026].

20. Booch, G., Rumbaugh, J. & Jacobson, I. “The Unified Modeling Language User Guide”. *Addison-Wesley*. 2005. ISBN: 978-0321267979.

21. Medvidovic, N. & Taylor, R. N. “A classification and comparison framework for software architecture description languages”. *IEEE Transactions on Software Engineering*. 2000; 26 (1): 70–93. DOI: https://doi.org/10.1109/32.825767.

22. Gamzayev, R., Tkachuk, N. & Shevkoplias, R. “Knowledge-oriented information technology to variability management at the domain analysis stage in software development”. *Advanced Information Systems*. 2020; 4 (4): 39–47. DOI: https://doi.org/10.20998/2522-9052.2020.4.06.

23. Kruchten, P. B. “The 4+1 View Model of Architecture”. *IEEE Software*. 1995; 12 (6): 42–50. DOI: https://doi.org/10.1109/52.469759.

24. Rozanski, N. & Woods, E. “Software systems architecture: Working with stakeholders using viewpoints and perspectives”. *Addison-Wesley*. 2011. ISBN: 978-0321718334.

25. Brown, S. “The C4 model for visualising software architecture”. *c4model.com*. 2026. – Available from: https://c4model.com. – [Accessed: Jan 2026].

26. “ArchiMate® 3.2 Specification”. *The Open Group*. 2023. – Available from: https://pubs.opengroup.org/architecture/archimate3-doc. – [Accessed: Jan 2026].

27. Bray, T. “The JavaScript Object Notation (JSON) data interchange format”. *IETF RFC 8259*. 2017. – Available from: https://www.rfc-editor.org/rfc/rfc8259. – [Accessed: Jan 2026].

28. Ben-Kiki, O., Evans, C. & döt Net, I. “YAML Ain’t Markup Language (YAML) Version 1.2”. *YAML.org*. 2009. – Available from: https://yaml.org/spec/1.2/spec.html. – [Accessed: Jan 2026].

29. OpenAI. “Function Calling and other API updates”. *OpenAI Blog*. 2023. – Available from: https://openai.com/index/function-calling-and-other-api-updates. – [Accessed: Jan 2026].

30. Majumder, A. “TOON vs JSON: A token-optimized data format for reducing LLM costs”. *Tensorlake Blog*. 2025. – Available from: https://www.tensorlake.ai/blog/toon-vs-json – [Accessed: Jan 2026].

31. Gansner, E. R. & North, S. C. “An open graph visualization system and its applications to software engineering”. *Software: Practice and Experience*. 2000; 30 (11): 1203–1233. DOI: https://doi.org/10.1002/1097-024X(200009)30:113.3.CO;2-E.

32. Komleva N., Liubchenko V. & Zinovatna S. “Evaluation of the quality of survey data and its visualization using dashboards”. *Computer Science and Information Technologies*. 2020; 2: 234–237, https://www.scopus.com/pages/publications/85100509851. DOI: https://doi.org/10.1109/CSIT49958.2020.9321970.

33. Syromiatnikov, M. V. & Ruvinskaya, V. M. “UA-Code-Bench: A competitive programming benchmark for evaluating large language models code generation in Ukrainian”. *Informatics. Culture. Technique*. Odesa, Ukraine. 2025; 2: 308–314. DOI: https://doi.org/10.15276/ict.02.2025.47.

34. “tiktoken: A fast BPE tokeniser for use with OpenAI’s models”. *GitHub*. 2026. – Available from: https://github.com/openai/tiktoken. – [Accessed: Jan 2026].

35. “Gemma3Tokenizer”. *Keras Hub API Documentation*. 2026. – Available from: https://keras.io/keras_hub/api/models/gemma3/gemma3_tokenizer. – [Accessed: Jan 2026].

36. Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E. & Stoica, I. “Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena”. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*. New Orleans, LA, USA. 2023; 36: 46595–46623. DOI: https://doi.org/10.48550/arXiv.2306.05685.

37. “Introducing Claude Sonnet 4.5”. *Anthropic News*. 2025. – Available from: https://www.anthropic.com/news/claude-sonnet-4-5. – [Accessed: Jan 2026].

38. “Introducing GPT-5.2”. *OpenAI*. 2025. – Available from: https://openai.com/index/introducing-gpt-5-2. – [Accessed: Jan 2026].

39. Sheng, H., Liu, X., He, H., Zhao, J. & Kang, J. “Analyzing uncertainty of LLM-as-a-Judge: Interval evaluations with conformal prediction”. In *Proc. Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Suzhou, China. 2025. p. 11286–11328. DOI: https://doi.org/10.18653/v1/2025.emnlp-main.569.

40. Gu, J., et al. “A survey on LLM-as-a-Judge”. *arXiv preprint*. 2024. – Available from: https://arxiv.org/abs/2411.15594. DOI: https://doi.org/10.48550/arXiv.2411.15594.

Conflicts of Interest: The authors declare that they have no conflict of interest regarding this study, including financial, personal, authorship or other, which could influence the research and its results presented in this article.

Received 14.01.2026
Received after revision 16.02.2026
Accepted 19.02.2026

DOI: https://doi.org/10.15276/aait.09.2026.08
УДК 004.8:004.4

Мова моделювання та анотування внутрішньої структури систем для програмної інженерії на основі великих мовних моделей

Сиром'ятніков Микита Валерійович[1)]
ORCID: https://orcid.org/0000-0002-0610-3639; nik.syromyatnikov@gmail.com. Scopus Author ID: 59533584100
Рувінська Вікторія Михайлівна[1)]
ORCID: https://orcid.org/0000-0002-7243-5535; ruvinska@op.edu.ua. Scopus Author ID: 57188870062
[1)] Національний університет «Одеська Політехніка», пр. Шевченка, 1. Одеса, 65044, Україна

АНОТАЦІЯ

Стрімке масштабування великих мовних моделей суттєво змінило традиційні парадигми програмної інженерії, надавши безпрецедентні можливості для розуміння коду, його генерації та автоматизованого рецензування. Однак практичне впровадження на рівні репозиторіїв стримується обмеженнями контексту та вартістю токенів. Хоча генерація з доповненням через пошук (retrieval-augmented generation) широко використовується для подолання цього обмеження, вона часто розбиває кодову базу на роз'єднані семантичні фрагменти, що втрачають високорівневі структурні залежності. Натомість альтернативні підходи, що намагаються подати всю структуру репозиторію у вікно контексту, зазвичай покладаються на універсальні формати, такі як JSON. Попри їх широке визнання, ці формати містять надлишкові синтаксичні елементи, які суттєво впливають на токен-бюджет. У цій роботі представлено SiMAL – предметно-орієнтовану мову, спроєктовану спеціально для робочих процесів програмної інженерії, керованих мовними моделями, з основною метою забезпечити компактне, просте для розуміння, стійке до помилок, але водночас високоструктуроване представлення програмної системи, оптимізоване для ітеративної взаємодії з мовними моделями. Мова поєднує як статичні, так і динамічні аспекти програмної системи, об'єднуючи архітектурні представлення, визначення компонентів і кінцевих точок, метадані розгортання виконуваного середовища та інші артефакти розробки в єдину текстову схему, яку можна перетворювати в нормалізоване машинне представлення. Робота включає визначення синтаксису та граматики мови, відкритий парсер і утиліту візуалізації, яка відображає схеми у вигляді вкладених системних діаграм. Запропонована мова валідується за допомогою комплексного протоколу, що оцінює ефективність використання токенів разом із точністю схеми. Протокол включає структурні перевірки (синтаксичний розбір та узгодженість анотацій), аналіз відповідності схеми коду, а також оцінювання покриття схемами репозиторіїв методом «LLM-as-a-judge». Результати свідчать: ефективне з точки зору промптингу моделювання схем зменшує витрати токенів без системного погіршення структурної зручності чи якості, що робить цей підхід практичною основою для масштабованої автономної програмної інженерії.

Ключові слова: моделювання систем; архітектура програмного забезпечення; мовні моделі; ефективність токенів; узагальнення коду; програмна інженерія

ABOUT THE AUTHORS

Mykyta V. Syromiatnikov - Postgraduate student, Software Engineering Department. Odesa Polytechnic National University. 1, Shevchenko Avenue. Odesa, 65044, Ukraine
ORCID: https://orcid.org/0000-0002-0610-3639; nik.syromyatnikov@gmail.com. Scopus Author ID: 59533584100
***Research field*:** natural language processing; large language modeling; deep learning

Сиром'ятніков Микита Валерійович - аспірант кафедри Інженерії програмного забезпечення. Національний університет «Одеська Політехніка», пр. Шевченка, 1. Одеса, 65044, Україна

Victoria M. Ruvinskaya - PhD, Professor, Software Engineering Department. Odesa Polytechnic National University. 1, Shevchenko Ave. Odesa, 65044, Ukraine
ORCID: https://orcid.org/0000-0002-7243-5535; ruvinska@op.edu.ua. Scopus Author ID: 57188870062
***Research field*:** knowledge-based systems; machine learning; algorithms; data structures learning

Рувінська Вікторія Михайлівна – кандидат технічних наук., професор кафедри Інженерії програмного забезпечення. Національний університет «Одеська Політехніка», пр. Шевченка, 1. Одеса, 65044, Україна