

Автоматизированная генерация предметно-ориентированных языков: Обзор нейро-символических подходов, ко-эволюции метамodelей и архитектурных паттернов

1. Введение

Современный этап развития информационных технологий характеризуется активным внедрением языковых инструментов в самые разнообразные сферы человеческой деятельности, выходя далеко за пределы традиционной разработки программного обеспечения. Парадигма предметно-ориентированных языков (Domain-Specific Languages, DSL) стала ключевым элементом в стратегии управления сложностью современных цифровых систем. В отличие от языков общего назначения (General Purpose Languages, GPL), таких как Java, Python или C++, которые разработаны для решения широкого класса вычислительных задач, DSL проектируются с учетом специфики конкретной предметной области, будь то моделирование бизнес-процессов, конфигурация инфраструктуры, обработка сигналов или автоматизация тестирования.¹

Ключевым преимуществом DSL является повышение уровня абстракции, позволяющее экспертам предметной области (domain experts) — врачам, инженерам, финансовым аналитикам — формулировать требования и логику системы в терминах, приближенных к их профессиональному языку, минимизируя разрыв между концептуальным замыслом и технической реализацией.² Однако традиционный процесс разработки и поддержки DSL сопряжен с высокими издержками. Создание качественного DSL требует глубокой экспертизы в области языковой инженерии: разработки формальной грамматики, реализации парсеров, проектирования абстрактного синтаксического дерева (AST) и создания инструментов поддержки (IDE, подсветка синтаксиса, автодополнение).¹ Более того, эволюция предметной области неизбежно влечет за собой необходимость адаптации языка, что порождает сложные задачи ко-эволюции метамodelей и существующей кодовой базы.⁵

Появление больших языковых моделей (Large Language Models, LLM) и методов генеративного искусственного интеллекта (GenAI) открывает принципиально новые возможности для автоматизации жизненного цикла DSL. Способность LLM генерировать синтаксически корректный код и интерпретировать естественный язык позволяет рассматривать их как универсальные механизмы трансляции намерений пользователя в

формальные спецификации.² Тем не менее, стохастическая природа нейронных сетей вступает в противоречие со строгими требованиями к корректности, предъявляемыми компиляторами и интерпретаторами. Проблема "галлюцинаций", когда модель генерирует правдоподобный, но невалидный код, требует разработки новых нейро-символических архитектур, объединяющих гибкость нейросетей с надежностью символьных вычислений.³

Настоящий отчет представляет собой исчерпывающий обзор литературы и текущих исследований в области автоматизированной генерации DSL. Мы подробно рассмотрим методы интеграции LLM в процесс разработки языков, проанализируем архитектурные паттерны, такие как паттерн "Посетитель" (Visitor), обеспечивающие связность генерируемого кода, и исследуем новейшие подходы к обеспечению семантической и синтаксической корректности через грамматические ограничения и эволюционные алгоритмы.

2. Теоретические основы и методология генерации DSL

Для глубокого понимания механизмов автоматизированной генерации DSL необходимо рассмотреть фундаментальные концепции, лежащие в основе языковой инженерии в контексте модельно-ориентированного проектирования (Model-Driven Engineering, MDE).

2.1 Дихотомия GPL и DSL в эпоху ИИ

Языки программирования традиционно делятся на языки общего назначения и предметно-ориентированные. Если первые стремятся к тьюринговской полноте и универсальности, то вторые сознательно ограничивают выразительную мощь ради повышения семантической плотности в конкретном домене.² Примером классического DSL является SQL: он идеально подходит для манипуляции данными, но непригоден для написания операционных систем.

В контексте использования LLM это различие становится критическим. LLM обучаются на огромных корпусах кода GPL, доступных в открытом доступе (GitHub, StackOverflow), что позволяет им демонстрировать высокие результаты в генерации Python или Java кода. Напротив, DSL часто являются проприетарными, малоресурсными (low-resource) языками, примеров которых в обучающей выборке ничтожно мало.⁸ Это создает "проблему холодного старта" при генерации кода на DSL: модель не "знает" синтаксиса специфического языка конфигурации промышленного робота или внутренней системы биллинга.

Исследования показывают, что применение методов few-shot learning (обучение на

нескольких примерах) и тонкой настройки (fine-tuning) может частично решить эту проблему, однако более перспективным направлением является использование метамodelей и формальных грамматик в качестве контекстных ограничений при генерации.⁹

2.2 Роль метамоделирования

В MDE центральным артефактом является метамодель — формальное описание структуры языка, определяющее его абстрактный синтаксис. Метамодель задает классы, атрибуты и отношения между сущностями домена.⁵ При автоматизированной генерации DSL метамодель выполняет роль "схемы правды" (ground truth).

Современные подходы, такие как *DSL Assistant*, используют метамодели (например, в формате Ecore или JSON Schema) для управления генерацией. Процесс строится итеративно:

1. Пользователь описывает домен на естественном языке.
2. LLM генерирует проект метамодели.
3. На основе метамодели синтезируется конкретная грамматика (например, в нотации Xtext или ANTLR).
4. Генерируются примеры моделей для валидации выразительности языка.¹

Важно отметить, что метамодели не статичны. Они эволюционируют вместе с требованиями бизнеса. Исследования в области ко-эволюции демонстрируют, что LLM способны автоматизировать миграцию кода при изменении метамодели, анализируя разницу (diff) между версиями и предлагая соответствующие трансформации для существующей кодовой базы.⁵

2.3 Нейро-символический разрыв

Одной из главных теоретических проблем является фундаментальное различие в природе нейронных сетей и формальных языков. Нейронные сети оперируют в непрерывном векторном пространстве и генерируют токены на основе вероятностных распределений. Формальные языки дискретны, строго структурированы и не допускают малейших отклонений от синтаксических правил.¹²

Этот конфликт порождает так называемый "синтаксический разрыв" (syntax gap). Код, сгенерированный LLM, может быть семантически близким к решению, но синтаксически некорректным (например, пропущена скобка или использован неверный идентификатор). Для преодоления этого разрыва разрабатываются нейро-символические архитектуры, которые внедряют символьные ограничения (грамматики, логические правила) непосредственно в процесс декодирования нейросети.¹⁴

3. Фреймворки автоматизированного синтеза DSL

В последние годы (2023–2025 гг.) наблюдается взрывной рост количества инструментов и фреймворков, нацеленных на автоматизацию создания DSL. Рассмотрим наиболее значимые из них, выделенные в ходе анализа литературы.

3.1 DSL Assistant: Итеративное прототипирование

Фреймворк *DSL Assistant* представляет собой инструмент, интегрирующий генеративные модели (в частности, GPT-4o) для поддержки полного цикла разработки внешних DSL. Основная идея заключается в снижении порога входа для экспертов предметной области. Инструмент поддерживает несколько режимов взаимодействия, включая генерацию грамматик из текстового описания и автоматическое исправление ошибок (auto-repair).¹

Архитектура *DSL Assistant* построена на модульном принципе. Пользователь взаимодействует с системой через чат-интерфейс, описывая сценарии использования (например, "Я хочу язык для описания схем оригами"). Система транслирует этот запрос в промпты для LLM, получая на выходе спецификацию грамматики. Критически важным компонентом является модуль валидации: сгенерированная грамматика немедленно компилируется, и в случае ошибки компиляции сообщение об ошибке возвращается в LLM для регенерации. Этот цикл обратной связи (feedback loop) позволяет достичь высокого процента синтаксически корректных результатов.¹

3.2 Text2Scenario: DSL для автономных систем

В индустрии беспилотного транспорта создание сценариев тестирования (simulation scenarios) является критической задачей. Ручное написание сценариев на сложных языках типа OpenSCENARIO трудоемко и подвержено ошибкам. Фреймворк *Text2Scenario* предлагает решение через генерацию промежуточного DSL из естественного языка.¹⁶

Процесс включает пять этапов:

1. **Декомпозиция:** Сложное описание дорожной ситуации разбивается на атомарные элементы (погодные условия, типы транспортных средств, маневры).
2. **Иерархический репозиторий:** Элементы сопоставляются с иерархической библиотекой сценариев.
3. **DSL Корпус:** Формируется специализированный корпус DSL, описывающий параметры и события.
4. **Генерация:** LLM синтезирует код сценария, связывая параметры с событиями.
5. **Валидация:** Полученный сценарий проверяется на выполнимость в симуляторе.¹⁶

Этот подход демонстрирует эффективность использования "промежуточных

представлений" (Intermediate Representations, IR). Вместо того чтобы генерировать финальный низкоуровневый код симулятора, LLM генерирует высокоуровневый DSL, который затем детерминировано транслируется в исполняемый формат. Это снижает вероятность логических ошибок.¹⁶

3.3 DSL-Xpert 2.0: Адаптация через Few-Shot Learning

Инструмент *DSL-Xpert 2.0* фокусируется на проблеме генерации кода для проприетарных и малоизвестных DSL. Поскольку такие языки отсутствуют в обучающих выборках публичных моделей, *DSL-Xpert* использует стратегии Few-Shot Learning (обучение на нескольких примерах) и RAG (Retrieval-Augmented Generation).⁹

Пользователь предоставляет инструменту определение грамматики и несколько примеров пар "естественный язык — код". Система автоматически формирует динамический промпт, включающий эти примеры и структурные ограничения грамматики. Кроме того, *DSL-Xpert* интегрирует механизмы автоматической валидации ввода/вывода, что позволяет использовать его как "умный" редактор кода для DSL, способный исправлять ошибки пользователя в реальном времени.⁹

3.4 Microsoft Automation DSL: Роль определений функций

Исследователи из Microsoft представили фреймворк для автоматизации рабочих процессов через API, использующий DSL, где имена API представлены как функции. В ходе экспериментов было выявлено, что качество генерации критически зависит от наличия определений функций (Function Definitions, FD) в контексте модели.¹⁷

Сравнительный анализ показал, что добавление FD в промпт снижает уровень галлюцинаций (выдуманных имен API) более эффективно, чем простое дообучение (fine-tuning) модели без предоставления контекста.

Таблица 1. Влияние определений функций (FD) на метрики качества генерации DSL ¹⁷

Метрика	Pre-trained + FD	TST (Task-Specific Tuning) без FD	TST + FD
Среднее сходство (LCSS)	Базовый уровень	+0.02	+0.02
% Нераспарсенных потоков	+2.75% (Ухудшение)	-0.61% (Улучшение)	+0.68%

% Выдуманных API	-4.30% (Улучшение)	-3.53% (Улучшение)	-6.29% (Лучший результат)
% Выдуманных параметров	-20.16% (Улучшение)	-1.04% (Улучшение)	-19.99% (Лучший результат)

Эти данные подчеркивают, что для DSL, опирающихся на обширные библиотеки API, контекстная инъекция метаданных (сигнатур функций) является более приоритетной стратегией, чем попытка "выучить" API через веса модели.¹⁷

4. Грамматически-управляемая генерация (Grammar-Guided Generation)

Одной из наиболее значимых инноваций в области надежной генерации кода является переход от свободной генерации текста к генерации, ограниченной грамматикой (Constrained Decoding). Этот метод гарантирует, что выход модели будет строго соответствовать заданному синтаксису.

4.1 Механизм маскирования токенов

В стандартном режиме LLM предсказывает следующий токен, выбирая его из всего словаря (vocabulary) на основе вероятности. В режиме грамматически-управляемой генерации процесс изменяется:

1. На каждом шаге генерации специальный модуль (парсер) анализирует текущее состояние сгенерированного префикса.
2. На основе формальной грамматики (обычно EBNF или Context-Free Grammar) определяется множество допустимых следующих токенов.
3. Вероятности (логиты) всех недопустимых токенов зануляются (маскируются, устанавливаются в $-\infty$).
4. Модель вынуждена выбирать токен только из разрешенного множества.¹⁴

4.2 Инструментарий: Guidance, SynCode, XGrammar

Существует несколько реализаций данного подхода:

- **Guidance:** Разработан Microsoft, использует Earley-парсер для обработки любых контекстно-свободных грамматик. Позволяет чередовать свободную генерацию (например, комментариев) с жестко структурированной (код DSL). Guidance преобразует шаблон промпта в дерево, где узлы представляют правила грамматики.¹⁴
- **SynCode:** Использует детерминированные конечные автоматы (DFA), построенные

оффлайн на основе терминалов грамматики. Это позволяет существенно ускорить проверку допустимости токенов по сравнению с полным парсингом на каждом шаге. SynCode продемонстрировал сокращение синтаксических ошибок на 96% при генерации Python и Go кода.¹⁵

- **XGrammar:** Интегрируется непосредственно в движки инференса (такие как vLLM), обеспечивая низколатентную структурную генерацию. Поддерживает EBNF и JSON Schema.¹⁹

4.3 Преимущества и ограничения

Главное преимущество — гарантия синтаксической корректности. Модель физически не может сгенерировать код с незакрытой скобкой или ключевым словом, написанным с ошибкой. Однако этот метод не гарантирует *семантической* корректности (например, использование необъявленной переменной или нарушение системы типов).⁷ Кроме того, сложные грамматики могут замедлять инференс из-за накладных расходов на парсинг.²¹

Для решения проблемы семантической корректности применяются гибридные подходы, такие как атрибутные грамматики, которые мы рассмотрим в разделе о нейро-символических методах.

5. Паттерн "Посетитель" (Visitor) в архитектуре генерации кода

Архитектурная целостность генерируемого кода не менее важна, чем его синтаксическая правильность. Паттерн "Посетитель" (Visitor) является де-факто стандартом при разработке компиляторов, интерпретаторов и инструментов анализа DSL, обеспечивая разделение структур данных (AST) и алгоритмов их обработки.

5.1 Теоретическое обоснование и проблема двойной диспетчеризации

Паттерн Visitor решает фундаментальную проблему расширяемости в объектно-ориентированном программировании, часто называемую "Expression Problem". Задача состоит в том, чтобы иметь возможность добавлять новые операции над иерархией классов без изменения самих классов. В языках с одиночной диспетчеризацией (Java, C#, C++) выбор метода зависит только от типа объекта, у которого он вызывается. Visitor реализует механизм двойной диспетчеризации (double dispatch), где выбор метода зависит от типов двух объектов: самого посетителя и элемента, который он посещает.²²

Формально, каждый узел AST (например, ExpressionNode, StatementNode) реализует

метод `accept(Visitor v)`, который вызывает `v.visit(this)`. Интерфейс `Visitor` содержит методы `visit` для каждого типа узла. Это позволяет группировать логику обработки (например, генерацию байт-кода, проверку типов, `pretty-printing`) в отдельных классах-посетителях, оставляя классы узлов чистыми контейнерами данных.²⁴

5.2 Автоматизация генерации Visitor

Ручная реализация паттерна `Visitor` для больших грамматик утомительна и подвержена ошибкам. Если в языке 100 типов выражений, интерфейс посетителя должен содержать 100 методов.

Инструменты автоматизации, такие как `AdaGOOP`, исторически решали эту проблему, генерируя классы посетителей (например, `PrintVisitor`, `TypeCheckVisitor`) на основе спецификации парсера.²⁶

В современных фреймворках на базе LLM этот процесс выходит на новый уровень. При так называемом "Vibe Coding" (интуитивном программировании с помощью ИИ) разработчик может попросить модель: "Сгенерируй иерархию классов AST для этой грамматики и реализуй паттерн `Visitor`". LLM, обученная на миллионах строк кода компиляторов, способна мгновенно сгенерировать весь необходимый "бойлерплейт" (шаблонный код), включая интерфейсы, абстрактные классы и методы `accept`.⁶

5.3 Visitor как связующее звено в нейро-символических системах

Паттерн `Visitor` играет критическую роль в надежности кода, сгенерированного ИИ:

1. **Промежуточное представление (IR):** Инструменты, такие как *Workflow Composer* от *Diagrid*, используют граф-ориентированное промежуточное представление. Сгенерированная LLM структура преобразуется в граф, а затем *Graph Visitor* обходит этот граф для генерации типобезопасного кода на целевом языке (Go, Java). Это изолирует нестабильность вывода LLM от финальной генерации кода.²⁷
2. **Анализ и трансформация:** При внедрении водяных знаков (watermarking) в сгенерированный код используются посетители AST для поиска мест, где возможны семантически эквивалентные трансформации (например, перестановка независимых операторов), что позволяет скрыть метаданные в структуре кода.²⁸
3. **Стабильность API:** В отличие от сопоставления с образцом (Pattern Matching), который может "молча" пропустить новый тип узла, интерфейс `Visitor` заставляет разработчика (или LLM) реализовать метод `visit` для каждого типа узла, иначе код не скомпилируется. Это обеспечивает строгую типизацию и полноту обработки.²⁹

6. Нейро-символический синтез программ

Нейро-символический подход (Neuro-Symbolic AI) представляет собой синтез интуитивных способностей нейронных сетей и логической строгости символьных

систем. В контексте DSL это направление решает задачи, недоступные для чистых LLM.

6.1 Гибридный синтез (HySynth)

Фреймворк *HySynth* демонстрирует эффективность гибридного подхода. Вместо того чтобы заставлять LLM генерировать финальный код, система использует LLM для генерации вероятностной контекстно-свободной грамматики (PCFG), которая описывает пространство поиска решений. Эта грамматика затем передается в символьный синтезатор (bottom-up synthesizer), который выполняет комбинаторный поиск программы, удовлетворяющей примерам ввода-вывода.⁷

Этот метод особенно эффективен в задачах, требующих сложной логики и точных вычислений, таких как задачи корпуса ARC (Abstraction and Reasoning Corpus), где чистые LLM часто терпят неудачу.¹³

6.2 Атрибутные грамматики и семантическая валидация

Для научных и инженерных DSL критически важна не только синтаксическая, но и размерная корректность (dimensional consistency). Использование атрибутных грамматик позволяет внедрить проверки физических размерностей (метры, секунды, килограммы) в процесс генерации.

Фреймворк *Tahr* и подходы на основе вероятностных атрибутных грамматик позволяют генерировать уравнения, которые гарантированно имеют физический смысл. LLM предлагает структуру формулы, а атрибутная грамматика проверяет, что нельзя сложить "скорость" и "массу".³⁰

6.3 Семантические интерпретаторы

Система *Semantic Interpreter* от Microsoft использует специализированный язык ODSL (Office Domain Specific Language) для управления приложениями Office. Вместо прямого выполнения команд, LLM транслирует запрос пользователя в программу на ODSL. Символьный интерпретатор затем выполняет эту программу в "песочнице", проверяя права доступа, наличие сущностей и логическую связность перед реальным воздействием на документы. Это предотвращает деструктивные действия, вызванные галлюцинациями модели.³²

7. Ко-эволюция метамodelей и поддержка целостности

В мире MDE изменения неизбежны. Эволюция метамодели (изменение схемы данных, правил валидации) требует синхронного обновления всех зависящих артефактов: грамматик, редакторов, генераторов кода и существующих моделей.

7.1 LLM как агент миграции

Исследования показывают, что LLM могут выступать в роли интеллектуальных агентов миграции. При изменении метамодели разработчик может предоставить модели описание изменений (diff) и фрагменты устаревшего кода. LLM, используя знания о паттернах рефакторинга, способна предложить обновленный код, соответствующий новой версии метамодели.⁵

Эксперименты с ко-эволюцией демонстрируют, что структурированные промпты, содержащие контекст изменения и информацию об "абстракционном разрыве" (abstraction gap) между метамоделью и кодом, позволяют достичь высокой точности автоматического рефакторинга.⁵

7.2 Интеграция в платформы моделирования

Платформы метамоделирования, такие как *MM-AR*, интегрируют LLM непосредственно в редакторы моделей. Это позволяет пользователям выполнять операции моделирования (создание классов, связей) через команды на естественном языке. Архитектура таких систем скрывает сложность метамодели от LLM, предоставляя ей набор высокоуровневых функций API. Это не только упрощает работу пользователя, но и гарантирует, что все изменения модели будут проходить через штатные механизмы валидации платформы.³³

8. Инструментарий, IDE и опыт разработчика (DX)

Успех технологий генерации DSL зависит от их интеграции в рабочую среду разработчика.

8.1 Language Workbenches и LLM

Инструментальные средства создания языков (Language Workbenches), такие как **Langium** или **Xtext**, начинают включать нативную поддержку LLM. Расширение *langium-llm* позволяет разработчику описывать желаемое поведение DSL, а среда автоматически генерирует соответствующие грамматики и обработчики. Важной особенностью является возможность выбора режима общения с LLM: через конкретный синтаксис (текст кода) или абстрактный (JSON-дерево). Использование JSON часто дает лучшие результаты, так как LLM лучше обучены на структурированных данных формата JSON.³⁴

8.2 Горячая перезагрузка (Hot Reloading) и динамические грамматики

Для эффективной разработки DSL необходим быстрый цикл обратной связи. Технологии

Hot Reloading позволяют изменять грамматику или интерпретатор языка "на лету", без перезапуска всего приложения.

- **Интерпретируемые DSL:** Инструменты типа *LiveSharp* используют собственные интерпретаторы для подмены тел методов в работающем приложении, что идеально подходит для быстрой итерации над DSL.³⁵
- **Динамическое сокращение грамматик:** В эволюционных подходах (Genetic Programming) применяется динамическое сокращение грамматики (Dynamic Grammar Pruning). В процессе эволюции удаляются правила, которые редко используются или ведут к тупиковым ветвям, что оптимизирует пространство поиска и ускоряет синтез программ.³⁶

8.3 "Vibe Coding"

Новый феномен, получивший название "Vibe Coding", описывает процесс, где разработчик задает "настроение" (vibe) или стиль языка, а LLM берет на себя рутинную реализацию. Разработчик выступает в роли архитектора, проверяющего логику и семантику, в то время как LLM генерирует парсеры, классы AST и посетителей. Это радикально меняет экономику создания DSL, делая возможным создание микро-языков (micro-DSLs) для решения узких задач в рамках одного проекта за считанные часы.⁶

9. Оценка качества, бенчмарки и безопасность

9.1 Метрики корректности

Для оценки качества генерации DSL используются специфические метрики:

- **Pass@k:** Вероятность того, что среди \$k\$ сгенерированных вариантов хотя бы один пройдет тесты.
- **LCSS (Longest Common Subsequence):** Используется для оценки сходства последовательностей API-вызовов. Эта метрика более устойчива к незначительным вариациям кода, чем точное совпадение строк.¹⁷
- **Уровень галлюцинаций (Hallucination Rate):** Процент сгенерированных программ, содержащих несуществующие имена функций или параметры. Использование RAG и определений функций позволяет снизить этот показатель практически до нуля.¹⁸

9.2 Бенчмарки

Существующие бенчмарки, такие как **HumanEval** и **MBPP**, ориентированы на Python и алгоритмические задачи, что делает их недостаточно репрезентативными для DSL.

- **ClassEval:** Оценивает генерацию на уровне классов и зависимостей, что ближе к задачам генерации инфраструктуры DSL.³⁷
- **BigCodeBench:** Содержит сложные задачи с использованием библиотек, выявляя

слабости LLM в точном вызове функций (точность 60% против 97% у человека).³⁸

- **DS-1000:** Специализированный бенчмарк для Data Science, полезный для оценки DSL обработки данных.³⁹

9.3 Безопасность и Genetic Improvement

Проблемы безопасности сгенерированного кода решаются через специализированные RAG-фреймворки, такие как RESCUE, которые обогащают контекст знаниями о уязвимостях (CWE) и безопасных паттернах кодирования.⁴⁰

Методы Genetic Improvement (GI) применяются для "лечения" кода, сгенерированного LLM. Если код почти верен ("near-miss"), эволюционный алгоритм мутирует AST программы, используя тесты как функцию приспособленности, что позволяет исправить мелкие ошибки без повторного обращения к модели.⁴¹

10. Заключение

Автоматизированная генерация предметно-ориентированных языков прошла путь от теоретических экспериментов до мощных промышленных инструментов. Интеграция больших языковых моделей с методами формальной верификации, грамматического вывода и архитектурными паттернами MDE создала фундамент для новой эры программной инженерии.

Ключевые выводы исследования:

1. **Необходимость ограничений:** Чистая генерация текста LLM недостаточна для создания надежных DSL. Использование грамматического маскирования (*SynCode*, *Guidance*) и атрибутивных грамматик является обязательным условием для получения компилируемого кода.
2. **Ренессанс паттерна Visitor:** Старый добрый паттерн Visitor обрел новую жизнь как идеальный контракт между генерируемой структурой данных и рукописной логикой, обеспечивая модульность и расширяемость синтезируемых систем.
3. **Гибридные архитектуры:** Наиболее успешные системы (*HySynth*, *DSL-Xpert*) являются гибридными, объединяя интуицию нейросетей с точностью символьных алгоритмов.
4. **Ко-эволюция:** LLM эффективно решают одну из самых болезненных проблем MDE — синхронизацию метамodelей и кода, выступая в роли интеллектуальных агентов рефакторинга.

Будущее области лежит в развитии "LLM-Hardened" языков, спроектированных с учетом вероятностной природы их создателей, и в дальнейшем совершенствовании нейро-символических интерфейсов, делающих создание собственного языка таким же простым, как написание промпта.

Таблица 2. Сравнительный анализ подходов к генерации DSL

Характеристика	Чистый промптинг (Pure Prompting)	Грамматически-управляемая (SynCode/Guidance)	Нейро-символический синтез (HySynth)	Метамодельный подход (DSL Assistant)
Входные данные	Естественный язык (NL)	NL + Грамматика (EBNF/CFG)	NL + Примеры + Логика	NL + Метамодель (Ecore)
Механизм ограничений	Отсутствует (статистический)	Маскирование токенов (Парсер)	Атрибутные грамматики / SMT-решатели	Валидация по метамодели
Синтаксическая корректность	Низкая (<40%)	Гарантирована (100%)	Высокая (Верифицируема)	Высокая (Correct-by-construction)
Семантическая корректность	Низкая (Галлюцинации)	Средняя (Только синтаксис)	Высокая (Логические проверки)	Средняя (Зависит от валидатора)
Латентность (Задержка)	Низкая	Средняя (Накладные расходы парсера)	Высокая (Время поиска/решения)	Средняя
Типовые сценарии	Прототипирование, Vibe Coding	Автодополнение, JSON API	Научные вычисления, Критичные системы	MDE, Создание инструментов

Источники

1. From a Natural to a Formal Language with DSL Assistant - arXiv, дата последнего обращения: декабря 28, 2025, <https://arxiv.org/pdf/2408.09766>
2. Large Language Models for Domain-Specific Language Generation, дата последнего обращения: декабря 28, 2025, <https://medium.com/itemis/large-language-models-for-domain-specific-language-generation-how-to-train-your-dragon-0b5360e8ed76>
3. LLM-Hardened DSLs for Probabilistic Code Generation ... - Dean Mai, дата последнего обращения: декабря 28, 2025, <https://deanm.ai/blog/2025/5/24/toward-data-driven-multi-model-enterprise-ai-7e545-sw6c2>
4. Evolution of Textual Domain-Specific Languages in the Context of ..., дата последнего обращения: декабря 28, 2025, https://gupea.ub.gu.se/bitstream/handle/2077/89382/PhD_Thesis_Weixing_2025%20%281%29.pdf?sequence=1&isAllowed=y
5. (PDF) An Empirical Study on Leveraging LLMs for Metamodels and ..., дата последнего обращения: декабря 28, 2025, https://www.researchgate.net/publication/383450322_An_Empirical_Study_on_Leveraging_LLMs_for_Metamodels_and_Code_Co-evolution
6. Vibe Coding a Compiler: From Natural Language to LLVM IR, дата последнего обращения: декабря 28, 2025, <https://medium.com/@fhinkel/vibe-coding-a-compiler-from-natural-language-to-llvm-ir-f4e192a49260>
7. Vtool: Context-Free LLM Approximation for Guiding Program Synthesis, дата последнего обращения: декабря 28, 2025, <https://arxiv.org/html/2405.15880v2>
8. A Survey on LLM-based Code Generation for Low-Resource ... - arXiv, дата последнего обращения: декабря 28, 2025, <https://arxiv.org/html/2410.03981v3>
9. DSL-Xpert 2.0: Enhancing LLM-Driven code generation for domain ..., дата последнего обращения: декабря 28, 2025, https://www.researchgate.net/publication/397111652_DSL-Xpert_20_Enhancing_LLM-Driven_code_generation_for_domain-specific_languages
10. Domain Specialization as the Key to Make Large Language Models ..., дата последнего обращения: декабря 28, 2025, <https://arxiv.org/html/2305.18703v7>
11. Exploring the use of large language models in domain-specific ..., дата последнего обращения: декабря 28, 2025, <https://ceur-ws.org/Vol-4122/paper6.pdf>
12. Context-Free LLM Approximation for Guiding Program Synthesis, дата последнего обращения: декабря 28, 2025, <https://openreview.net/pdf?id=5jt0ZSA6Co>
13. NSA: Neuro-symbolic ARC Challenge - arXiv, дата последнего обращения: декабря 28, 2025, <https://arxiv.org/html/2501.04424v1>
14. Using Grammar Masking to Ensure Syntactic Validity in LLM-based ..., дата последнего обращения: декабря 28, 2025, <https://arxiv.org/html/2407.06146v1>
15. Improving LLM Code Generation with Grammar Augmentation - arXiv, дата последнего обращения: декабря 28, 2025, <https://arxiv.org/html/2403.01632v1>
16. Text-Driven Scenario Generation for Autonomous Driving Test - arXiv, дата

- последнего обращения: декабря 28, 2025, <https://arxiv.org/html/2503.02911v1>
17. A Comparative Study of DSL Code Generation: Fine-Tuning vs ..., дата последнего обращения: декабря 28, 2025, <https://arxiv.org/html/2407.02742v1>
 18. Comparing approaches for robust NL to DSL generation - arXiv, дата последнего обращения: декабря 28, 2025, <https://arxiv.org/html/2408.08335v1>
 19. EBNF-Guided Generation — XGrammar 0.1.29 documentation, дата последнего обращения: декабря 28, 2025, https://xgrammar.mlc.ai/docs/tutorials/ebnf_guided_generation.html
 20. Structured Outputs - vLLM, дата последнего обращения: декабря 28, 2025, https://docs.vllm.ai/en/latest/features/structured_outputs/
 21. Using grammars to constrain llama.cpp output - Ian Maurer's Notes, дата последнего обращения: декабря 28, 2025, <https://www.imaurer.com/blog/posts/2023-09-06-llama-cpp-grammars/>
 22. Visitor pattern - Wikipedia, дата последнего обращения: декабря 28, 2025, https://en.wikipedia.org/wiki/Visitor_pattern
 23. The Visitor Pattern - 'Revisited' using Data Oriented Programming ..., дата последнего обращения: декабря 28, 2025, https://www.reddit.com/r/java/comments/1k6lwpu/the_visitor_pattern_revisited_using_data_oriented/
 24. Visitor Design Pattern - Medium, дата последнего обращения: декабря 28, 2025, <https://medium.com/@robinsrivastava737/visitor-design-pattern-669831e2d4f9>
 25. Visitor pattern | ODP - Slideshare, дата последнего обращения: декабря 28, 2025, <https://www.slideshare.net/slideshow/visitor-pattern-69003229/69003229>
 26. An automatic "visitor" generator for ada - SciSpace, дата последнего обращения: декабря 28, 2025, <https://scispace.com/pdf/an-automatic-visitor-generator-for-ada-51jq1ah4vj.pdf>
 27. Go with the (Work)Flow: AI-Driven Code Generation for Workflows, дата последнего обращения: декабря 28, 2025, <https://www.diagrid.io/blog/go-with-the-work-flow>
 28. Practical and Effective Code Watermarking for Large Language ..., дата последнего обращения: декабря 28, 2025, <https://openreview.net/pdf/e28031c958fa8b0115bf14d0fcd0a2c33c8d8826.pdf>
 29. Implementing APIs with the visitor pattern - Davide Lettieri, дата последнего обращения: декабря 28, 2025, <https://davidelietteri.it/2023/06/24/implementing-apis-with-the-visitor-pattern>
 30. Dimensionally-consistent equation discovery through probabilistic ..., дата последнего обращения: декабря 28, 2025, https://www.researchgate.net/publication/369173899_Dimensionally-consistent_equation_discovery_through_probabilistic_attribute_grammars
 31. Tahr: The Generative Attribute Grammar Framework - arXiv, дата последнего обращения: декабря 28, 2025, <https://arxiv.org/html/2512.01872v1>
 32. Natural Language Commanding via Program Synthesis - arXiv, дата последнего обращения: декабря 28, 2025, <https://arxiv.org/pdf/2306.03460>
 33. (PDF) An Architecture for Integrating Large Language Models into ..., дата

последнего обращения: декабря 28, 2025,

https://www.researchgate.net/publication/386332431_An_Architecture_for_Integrating_Large_Language_Models_into_Metamodeling_Platforms_The_Example_of_MM-AR

34. A language workbench extension to generate conversational ..., дата последнего обращения: декабря 28, 2025, <https://ceur-ws.org/Vol-4122/paper16.pdf>
35. Hot Reloading Declarative C# UI for Xamarin.Forms with ..., дата последнего обращения: декабря 28, 2025, <https://vincenth.net/blog/2019/07/16/hot-reloading-declarative-c-ui-for-xamarin-forms-with-csharpformmarkup-and-livesharp>
36. Dynamic Grammar Pruning for Program Size Reduction in Symbolic ..., дата последнего обращения: декабря 28, 2025, https://www.researchgate.net/publication/370841191_Dynamic_Grammar_Pruning_for_Program_Size_Reduction_in_Symbolic_Regression
37. Evaluating Large Language Models in Class-Level Code Generation, дата последнего обращения: декабря 28, 2025, <https://mingwei-liu.github.io/assets/pdf/ICSE2024ClassEval-V2.pdf>
38. BigCodeBench: Benchmarking Code Generation with Diverse ..., дата последнего обращения: декабря 28, 2025, <https://openreview.net/forum?id=YrycTjllLO>
39. Which AI model writes the best Pandas code? - Posit, дата последнего обращения: декабря 28, 2025, <https://posit.co/blog/python-llm-evaluation/>
40. RESCUE: Retrieval Augmented Secure Code Generation - arXiv, дата последнего обращения: декабря 28, 2025, <https://www.arxiv.org/pdf/2510.18204>
41. (PDF) Exploring the Effect of Genetic Improvement for Large ..., дата последнего обращения: декабря 28, 2025, https://www.researchgate.net/publication/394856671_Exploring_the_Effect_of_Genetic_Improvement_for_Large_Language_Models-Generated_Code
42. Improving LLM-Generated Code via Genetic Improvement, дата последнего обращения: декабря 28, 2025, https://ceur-ws.org/Vol-4121/Ital-IA_2025_paper_111.pdf