Beacon-Guided Generation: A Structured Semantic Framework for Repository-Grounded Code Synthesis

Anonymous ACL submission

Abstract

Pragmatic code generation requires models to produce implementations consistent not only with local task descriptions but also with repository-level semantics distributed across functions, files, and project context. Existing large language model pipelines typically expose such context as flat prompt text, leaving models to infer semantic centrality implicitly, which often causes missing dependencies and structurally inconsistent outputs on context-intensive tasks.

We propose *Beacon Logic*, a conditioned formal system for extracting task-relevant semantic anchors from repository context, and *Beacon-Guided Generation*, a representation-centered framework that turns these anchors into explicit semantic artifacts for generation, verification, and reuse. Rather than repeatedly reasoning over raw context with large models, our approach externalizes semantic structure through local reasoning, global propagation, and structured reduction, using the resulting Beacon-derived representation to guide downstream code generation.

We evaluate our method on `CoderEval` in a controlled same-model setting, using **Qwen3-80B** as the shared backbone for both a direct-prompt baseline and our system. Results show that Beacon-guided generation yields substantial gains, especially on Python and repository-grounded runnable levels with strong contextual dependencies. These findings suggest that improving pragmatic code generation requires not only larger models or longer contexts, but also explicit semantic centrality through a formal and executable semantic layer.

1 Introduction

Large language models (LLMs) have achieved strong results on code generation, from benchmark-style function completion to increasingly agentic software-engineering workflows (4; 11; 10; 1). However, their performance remains limited when generation must be grounded in realistic repository context rather than isolated specifications. In project-level settings, correct code is often determined not by a docstring alone, but by repository-specific semantics such as helper-function contracts, validation logic, call dependencies, state propagation, and API usage constraints (12; 13; 3). Recent work suggests that simply exposing more context is insufficient: additional code may introduce noise, dilute critical dependencies, and make it harder for the model to identify the structures that actually govern the target behavior (3; 13).

This issue is especially salient in repository-grounded benchmarks, where models must produce code that is not only syntactically plausible but also behaviorally consistent with an existing project (12). Failures in such settings are often structural: models may call incorrect helper functions, omit required validation paths, hallucinate unavailable symbols, or generate locally reasonable code that violates project-level assumptions. These errors indicate that the central challenge is not merely language modeling, but semantic selection: given a large and heterogeneous project context, the model must identify the sparse subset of code elements that is decisive for the target implementation.

Our motivation is also informed by research on program comprehension. Prior studies show that programmers do not inspect code uniformly; instead, they rely on selective reading strategies, structurally meaningful cues, and focused reasoning paths when understanding or debugging programs (2; 9; 5; 6). Wiedenbeck introduced the notion of *beacons*: semantically indicative code cues that help programmers recognize important structures and infer behavior efficiently (8). This perspective suggests that repository-grounded generation may benefit from making such decisive semantic cues explicit, rather than leaving them implicit in raw prompt context.

To address this challenge, we propose **Beacon Logic**, a structured framework for extracting task-relevant semantic anchors from project context. Instead of treating surrounding code as flat prompt text, Beacon Logic identifies code elements that are central to the target behavior and organizes them into a structured intermediate representation, **Beacon IR**. Beacon IR captures relevant symbols, dependency paths, call relations, validation constraints, and provenance information, providing an explicit semantic layer that can guide generation and support downstream checking.

Building on this representation, we introduce a **Beacon-guided code generation framework**. The framework first derives a task-specific Beacon IR from repository context, then uses this representation to guide code generation under explicit semantic constraints. After generation, the produced code is re-analyzed to rebuild its Beacon structure, which is compared against the original Beacon IR to detect missing or inconsistent semantic anchors. In this way, repository context is transformed from passive prompt material into a compact and checkable semantic artifact.

Our contributions are threefold. First, we formulate repository-grounded code generation as a problem of task-conditioned semantic-core extraction. Second, we operationalize the cognitive notion of beacons as structured semantic anchors for generation and verification. Third, experiments on `CoderEval` show that, under the same **Qwen3-80B** backbone, Beacon-guided generation improves over direct prompting, especially in context-intensive runnable levels. These results suggest that improving pragmatic code generation requires not only larger models or longer contexts, but also explicit representations of semantic centrality.

2 Motivation and Problem Statement

2.1 Motivation

Pragmatic code generation requires models to produce code that is consistent with repository-level behavior, rather than only locally plausible with respect to a task description (12; 13; 3). In repository-grounded settings, the relevant information is rarely distributed uniformly across all available files. Instead, correctness often depends on a compact set of semantically decisive structures, such as helper-function contracts, validation branches, dependency paths, state updates, and project-specific API conventions (12; 3). However, existing LLM-based pipelines commonly expose repository context as flat prompt text or retrieved code snippets, leaving the model to infer which parts are truly outcome-relevant (13; 3).

This creates a mismatch between the structure of repository context and the needs of generation. Repository context is large, heterogeneous, and only partially relevant, while successful generation depends on identifying a sparse semantic core. Simply adding more code to the prompt can therefore increase ambiguity: models may attend to nearby but irrelevant fragments, miss critical dependencies, or rely on surface-level similarity rather than behavioral constraints (3). For project-grounded code generation, the key issue is thus not merely how to provide more context, but how to isolate and operationalize the context that matters.

Our motivation is inspired by the notion of *beacons* in program comprehension (8). Prior work suggests that programmers do not read code as a flat token sequence; they selectively attend to salient elements that help them infer intent, trace behavior, and verify consistency (2; 9; 5; 6). Such beacons may include function calls, validation logic, state transitions, return paths, or other structures that carry disproportionate interpretive value (8). We adapt this cognitive insight to repository-level code generation: within a large project, some elements are important not because they are textually nearby, but because they are semantically central to the target behavior.

Recent software-engineering agents have made substantial progress through repository search, tool use, patch generation, execution feedback, and workflow orchestration (11; 10; 1). Benchmarks and systems such as SWE-agent, ContextBench, and ProjectGen illustrate this trend by improving how agents retrieve context, interact with environments, and structure development (10; 3; 13). Yet most existing approaches still lack an explicit intermediate layer that represents task-conditioned semantic centrality. They can often retrieve or use context, but they rarely provide a principled mechanism for distinguishing behaviorally decisive structures from incidental surrounding code, or for carrying such structures into generation and verification.

We therefore reinterpret beacons as extractable semantic anchors in repository context. Rather than modeling the full cognitive process of human programmers, we use the beacon concept as an en-

gineering abstraction: correct repository-grounded generation often depends on a small number of semantically central structures, and these structures should be explicitly identified, organized, and used to guide downstream generation.

2.2 Problem Statement

Given a natural-language code generation task and its repository context, our objective is to generate code that is both locally plausible and behaviorally consistent with the surrounding project (12). The challenge is that only a sparse subset of the available context is decisive for the target implementation (3). These decisive elements may include key symbols, helper contracts, validation logic, dependency paths, call relations, and state-relevant structures.

We define the problem as identifying this task-relevant semantic core, representing it in a structured form, and using it as an explicit basis for generation and verification. In contrast to flat-context prompting, the goal is not to expose more repository code to the model, but to make semantic centrality explicit so that generation becomes more focused, checkable, and consistent with project-level behavior.

3 Beacon Logic

3.1 Scope and Program Model

Beacon Logic is an operational inference framework for constructing a compact, provenance-aware semantic intermediate representation from repository context. It extracts the output-relevant semantic scaffold of a target program fragment and exposes it to downstream generation and verification. We do not use Beacon Logic as a full formal semantics of the program, nor do we claim semantic equivalence between the extracted beacon set and the original repository. Instead, Beacon Logic formalizes a practical construction procedure: starting from observable outputs, tracing relevant dependencies, abstracting validation-dominated paths, and projecting the result into a structured Beacon IR.

The raw extraction layer is intentionally slicing-like: it begins from observable output nodes and expands backward along approximated data and control dependencies. However, Beacon Logic is not merely a backward slice. Classical slicing preserves statements relevant to a slicing criterion, whereas Beacon Logic builds an LLM-facing scaffold under context-budget and verification constraints. Its contribution lies in the complete pipeline: semantic role filtering, reduction and normalization, interprocedural propagation, provenance annotation, and verifier-facing constraints. Compared with heuristic retrieval, Beacon Logic provides explicit dependency-based reasoning; compared with full program analysis, it remains lightweight enough for repository-grounded generation where precise whole-program semantics, complete alias analysis, and exhaustive interprocedural analysis are often unavailable.

Let a program $\mathcal{P}$ contain a finite set of functions $\mathcal{F} = \{f_1, \ldots, f_n\}$. Each function f is represented by an abstract syntax structure $T_f = (N_f, E_f)$, where N_f contains statements, expressions, branch conditions, call sites, returns, side-effect operations, and symbol-binding nodes such as imports, constants, and helper definitions. Let $N_{\mathcal{P}} = \bigcup_{f \in \mathcal{F}} N_f$. For each function f, $O(f) \subseteq N_f$ denotes observable output nodes, including returns, yields, output calls, and externally visible side effects such as file, database, or API writes when they constitute observable behavior.

The central predicate is $Dep(n, n')$, meaning that node n may influence node n' under the implemented dependence approximation:

$$\begin{aligned} Dep(n, n') = {} & Dep_{\text{data}}(n, n') \vee Dep_{\text{ctrl}}(n, n') \\ & \vee Dep_{\text{call}}(n, n') \vee Dep_{\text{ret}}(n, n') \\ & \vee Dep_{\text{state}}(n, n') \vee Dep_{\text{sym}}(n, n'). \end{aligned} \tag{1}$$

Here, data dependence covers definitions used by later expressions; control dependence covers branch or loop conditions controlling execution; call and return dependence link call sites, callee behavior, and return-value flow; state dependence covers attributes, fields, globals, and shared mutable state; and symbol dependence records imports, constants, helper functions, classes, and project-level bindings required to resolve a node. Since repository-level generation often operates under incomplete static information, Dep is conservative in the raw stage: uncertain but plausible dependencies may be included and later compacted by filtering or reduction.

3.2 Local Beacon Construction

Local construction extracts the function-level scaffold through raw closure, validation filtering, and reduction. For compact notation, write $\vdash_f^{\ell}$ for $\mathcal{P}, f \vdash_{\text{loc}}$, $\vdash_f^{g}$ for $\mathcal{P}, f \vdash_{\text{glob}}$, and $\vdash_{\mathcal{P}}$ for $\mathcal{P} \vdash$.

Raw closure. Given $O(f)$ and Dep, the raw local beacon set is the least fixed point:

$$B^{\text{raw}}_{\text{loc}}(f) = \mu X.\big(O(f) \cup \{n \mid \exists n' \in X,\ Dep(n, n')\}\big). \tag{2}$$

Equivalently, it is generated by:

$$\frac{n \in O(f)}{\vdash^{\ell}_{f}\ n \in B^{\text{raw}}}\ \text{[L-OUT]} \tag{3}$$

$$\frac{\begin{array}{c}\vdash^{\ell}_{f}\ n' \in B^{\text{raw}} \\ Dep(n, n')\end{array}}{\vdash^{\ell}_{f}\ n \in B^{\text{raw}}}\ \text{[L-DEP]}. \tag{4}$$

This stage is conservative: every node that may influence an observable output under Dep is admitted before abstraction.

Validation filtering. Let $V(f) \subseteq N_f$ be validation-dominated nodes, i.e., nodes whose main role is to enforce preconditions, reject invalid inputs, or terminate exceptional paths without constructing the primary output. In implementation, $V(f)$ is detected by finite patterns such as null checks, type or range checks, exception-only branches, constant early returns, and guard clauses whose body does not define values consumed by the normal output path. The filtered set is:

$$B^{\text{flt}}_{\text{loc}}(f) = B^{\text{raw}}_{\text{loc}}(f) \setminus V(f). \tag{5}$$

The corresponding rules are:

$$\frac{\begin{array}{c}\vdash^{\ell}_{f}\ n \in B^{\text{raw}} \\ n \notin V(f)\end{array}}{\vdash^{\ell}_{f}\ n \in B^{\text{flt}}}\ \text{[L-VAL-KEEP]} \tag{6}$$

$$\frac{\begin{array}{c}\vdash^{\ell}_{f}\ n \in B^{\text{raw}} \\ n \in V(f)\end{array}}{\nvdash^{\ell}_{f}\ n \in B^{\text{flt}}}\ \text{[L-VAL-DROP]}. \tag{7}$$

Filtering is an abstraction step: it focuses the beacon set on primary computation and does not claim to preserve all edge-case behavior.

Reduction. After filtering, Beacon Logic applies a deterministic operator:

$$B^{*}_{\text{loc}}(f) = Reduce(B^{\text{flt}}_{\text{loc}}(f)). \tag{8}$$

$Reduce : 2^{N_f} \to 2^{N_f}$ removes duplicates, merges equivalent expressions, prioritizes nodes closer to outputs, preserves call and return boundaries, retains dependency-critical intermediates, and removes low-value peripheral nodes when the context budget is exceeded. Retained nodes are annotated with provenance:

$$\frac{\begin{array}{c}\vdash^{\ell}_{f}\ n \in B^{\text{flt}} \\ Reduce(B^{\text{flt}}_{\text{loc}}(f)) = B^{*}_{\text{loc}} \\ n \in B^{*}_{\text{loc}}\end{array}}{\vdash^{\ell}_{f}\ n \in B^{*}}\ \text{[L-RED]}. \tag{9}$$

Thus,

$$B^{*}_{\text{loc}}(f) = \{n \mid \vdash^{\ell}_{f}\ n \in B^{*}\}. \tag{10}$$

3.3 Global Construction and Beacon IR

Global construction lifts local beacons across function boundaries without expanding the whole repository. Local beacons seed global reasoning:

$$\frac{\vdash^{\ell}_{f}\ n \in B^{*}}{\vdash^{g}_{f}\ n \in B}\ \text{[G-BASE]}. \tag{11}$$

If a beacon-relevant call site c in f_i invokes f_j, callee beacons are selectively propagated:

$$\frac{\begin{array}{c}\vdash^{g}_{f_i}\ c \in B \\ call(c, f_j) \\ \vdash^{g}_{f_j}\ n \in B\end{array}}{\vdash^{g}_{f_i}\ n \in B}\ \text{[G-CALL]}. \tag{12}$$

Return-value propagation connects callee computation to caller use:

$$\frac{\begin{array}{c}RetFlow(f_j, f_i) \\ \vdash^{g}_{f_j}\ n \in B\end{array}}{\vdash^{g}_{f_i}\ n \in B}\ \text{[G-RET]}. \tag{13}$$

$RetFlow$ is detected from assignment targets, expression uses, direct returns, and nested call arguments involving the result of a call. For shared state g, with readers $Read(g)$ and writers $Write(g)$, Beacon Logic conservatively admits related reads and writes when one side is beacon-relevant:

$$\frac{\begin{array}{c}\vdash_{\mathcal{P}}\ n_r \in B(\mathcal{P}) \\ n_r \in Read(g) \\ n_w \in Write(g)\end{array}}{\vdash_{\mathcal{P}}\ n_w \in B(\mathcal{P})}\ \text{[G-GLOB]}. \tag{14}$$

$$\frac{\begin{array}{c}\vdash_{\mathcal{P}}\ n_w \in B(\mathcal{P}) \\ n_w \in Write(g) \\ n_r \in Read(g)\end{array}}{\vdash_{\mathcal{P}}\ n_r \in B(\mathcal{P})}\ \text{[G-GLOB-2]}. \tag{15}$$

This conservative treatment is used when precise alias or state analysis is unavailable and remains subject to reduction and provenance tracking.

Given a task-relevant entry function f_{entry}, the program-level beacon set is:

$$B(\mathcal{P}) = \{n \mid \vdash^{g}_{f_{\text{entry}}} \; n \in B\}. \tag{16}$$

The final Beacon IR is:

$$\mathcal{B} = \langle N_B, E_B, S_B, F_B, \Pi_B \rangle. \tag{17}$$

Here, $N_B \subseteq B(\mathcal{P})$ contains retained beacon nodes; E_B stores typed dependency edges; S_B records required symbols such as imports, constants, helpers, classes, and project bindings; F_B records filtered validation paths and forbidden patterns; and Π_B maps each retained node or edge to the rule or operation that introduced or preserved it. The generator uses $\mathcal{B}$ as a constrained implementation scaffold, while the verifier checks beacon coverage, dependency alignment, symbol grounding, and forbidden-pattern compliance.

3.4 Properties, Non-claims, and Trace

Beacon Logic states properties of its construction procedure under the implemented dependency approximation; it does not assert full program equivalence.

Termination. For every function f, raw closure terminates because N_f is finite and L-OUT/L-DEP only add nodes from N_f. The fixed point stabilizes after at most $|N_f|$ additions. The same argument applies globally when $N_{\mathcal{P}}$ is finite and repeated nodes are not re-added.

Raw monotonicity. If $Dep_1 \subseteq Dep_2$ and $O_1(f) \subseteq O_2(f)$, then

$$B^{\text{raw}}_{\text{loc},1}(f) \subseteq B^{\text{raw}}_{\text{loc},2}(f). \tag{18}$$

The second fixed-point operator contains all roots and expansion edges of the first, so adding outputs or dependency edges cannot remove raw beacons.

Conservative raw coverage. If Dep over-approximates true output-relevant dependence, then $B^{\text{raw}}_{\text{loc}}(f)$ contains all dependency ancestors of $O(f)$ reachable through Dep. This follows directly from L-OUT and repeated application of L-DEP; it requires only that relevant dependencies are not omitted.

Provenance traceability. Every node or edge in $\mathcal{B}$ is traceable through Π_B: output roots by L-OUT, dependency ancestors by L-DEP, kept or dropped validation paths by L-VAL-KEEP/L-VAL-DROP, retained reduced nodes by L-RED, and interprocedural/state nodes by G-CALL, G-RET, or G-GLOB. Thus the verifier can inspect not only whether a node is present, but why it was included.

Non-claims and trace. Beacon Logic does not claim that $B(\mathcal{P})$ is semantically equivalent to the original program or that the reduced beacon set is a minimal slice. Raw closure is conservative, while validation filtering and reduction are abstraction steps that may remove edge-case behavior or peripheral details. Their usefulness is evaluated empirically through ablations such as raw closure only, without validation filtering, without global propagation, without verifier, and full Beacon Logic.

For example, in an order-settlement workflow, `finalize_order` returns `final_total`. Beacon Logic roots extraction at `return final_total`, expands to definitions such as `subtotal + fee + tax`, links helper calls such as `compute_subtotal`, `compute_fee`, and `compute_tax`, records defensive early-return guards as filtered validation paths, and projects retained nodes and typed relations into Beacon IR. This illustrates that each retained element is justified by rule-level dependency expansion, while filtered paths remain visible through IR annotations.

4 Framework Overview

Figure illustrates the overall workflow of BEACON-GUIDED GENERATION. Given a task specification and its project context, the system first applies *Beacon Logic* to extract task-relevant semantic anchors from the repository. This stage combines local reasoning over candidate code elements with global reasoning over their dependencies and repository-level relations. The output is a structured Beacon intermediate representation (Beacon IR), which records semantic nodes, dependency edges, associated symbols, and provenance information.

The Beacon IR serves as the central semantic interface between context analysis and code generation. Instead of prompting the model with raw repository context alone, the Code Generator Agent receives both the task specification and Beacon-derived constraints. This allows generation to be guided by behaviorally relevant project structures,

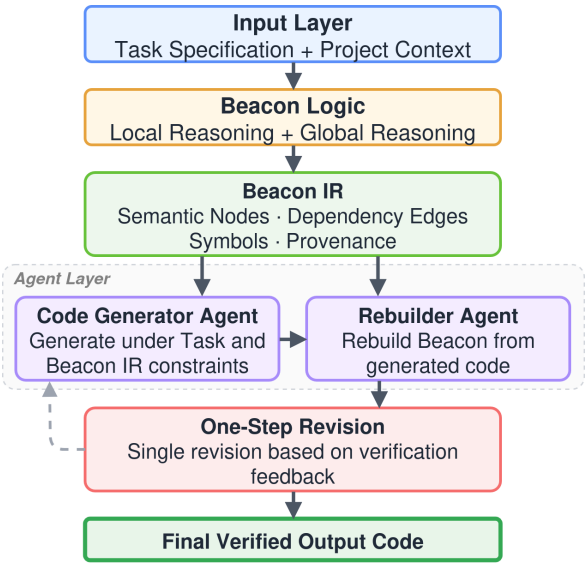

Figure 1: Beacon based genration system

such as required helper functions, validation logic, dependency paths, and state-related operations.

After code is generated, the Rebuilder Agent reconstructs a Beacon representation from the generated implementation. This reconstructed representation is compared with the original Beacon IR to check whether key semantic anchors and dependencies are preserved. The comparison provides structured verification feedback, identifying missing, inconsistent, or weakly supported semantic elements in the generated code.

Finally, the system performs a one-step revision using this verification feedback. The revised implementation is expected to better satisfy both the local task requirement and the repository-level semantic constraints encoded in the Beacon IR. In this way, the framework turns project context from flat prompt text into an explicit semantic artifact that supports generation, verification, and correction.

5 Experimental Setup

In this section, we describe the experimental setup used to evaluate the proposed Beacon-guided generation framework. Our evaluation is designed to answer a focused question: when using the same backbone model, does introducing Beacon-guided semantic extraction and verification improve pragmatic code generation compared with direct prompting? To ensure reproducibility, we follow the standard evaluation logic of CoderEval (12) and keep the underlying model fixed across all settings.

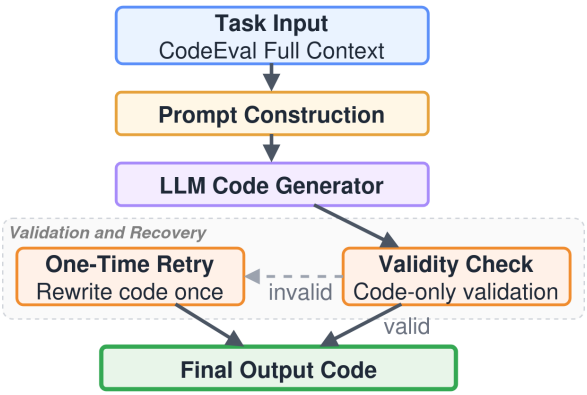

Figure 2: Baseline workflow

5.1 Benchmark: CoderEval

We conduct experiments on **CoderEval**, a benchmark curated from real-world open-source projects for pragmatic code generation. It contains 230 Python tasks and 230 Java tasks, each associated with project-grounded context and executable test infrastructure.

Unlike earlier benchmarks focused on isolated functions, CoderEval categorizes tasks into six *runnable levels* based on contextual dependency:

- **Standalone levels**: `self-contained` and `slib-runnable`.
- **Non-standalone levels**: `plib-runnable`, `class-runnable`, `file-runnable`, and `project-runnable`.

This distinction is central to our evaluation, as the core motivation of Beacon is to improve performance where implementations depend heavily on repository context.

5.2 Compared Settings

To isolate the effect of the proposed method from model scale, we use **Qwen3-80B** as the shared backbone model for all settings. The comparison is method-controlled rather than model-controlled, and both settings follow an identical outer pipeline with the same maximum number of generation attempts.

Baseline The baseline represents a standard LLM-based workflow depicted in Figure **??**. Given a task input, a prompt is constructed and passed to the model to generate candidate code, followed by a lightweight validity check. If the output is invalid, a single retry is performed. This setting does not include structured semantic reasoning, intermediate representations, or structure-level verification.

Beacon-Guided System This setting uses the same backbone model and maintains the same overall pipeline structure, but introduces a structured semantic layer. Specifically, Beacon Logic is applied to construct a Beacon IR, which is used to guide generation and enforce structural constraints. The generated code is then reconstructed and verified against the original semantic representation. If inconsistencies are detected, a single revision round is triggered based on verifier feedback.

Controlled Difference Both settings share the same model, input, and number of generation attempts. The only difference is whether generation is guided and validated by an explicit semantic representation (Beacon IR), rather than relying solely on direct prompting and syntactic validation.

5.3 Evaluation Procedure

We follow the standard execution-based evaluation process of CoderEval. For each task, the system generates candidate implementations, which are then injected into the original project environment by replacing the target function.

- **Python**: The platform sets up per-project environments and executes relevant tests.
- **Java**: The platform incrementally compiles modified files and runs the test driver.

Following CoderEval's protocol, we sample 10 candidates per task and report the unbiased $Pass@1$**,** $Pass@5$**, and** $Pass@10$ metrics.

5.4 Reported Views of Results

Our results are organized into two complementary views:

1. **Standalone vs. Non-standalone**: Comparing performance on simple vs. context-heavy tasks.
2. **Level-wise Breakdown**: Analyzing results across the six runnable levels. We expect Beacon's benefits to be more pronounced in `file-` and `project-grounded` tasks where dependencies are highest.

5.5 Design Rationale

This setup is intentionally conservative. We do not compare heterogeneous foundation models or introduce model-specific tricks. Instead, we evaluate whether adding an explicit semantic layer—Beacon extraction, structured IR, and structural verification—improves quality under a fixed model budget. Any observed performance gap can thus be attributed directly to the structured semantic guidance provided by the Beacon framework.

6 Results, Analysis, and Conclusion

Tables 1–3 report the $Pass@k$ results on `CoderEval` across Python and Java. The tables group results by runnable level and repository-dependency type, while keeping the full set of compared models visible. Since the direct-prompting baseline and **Beacon (Ours)** use the same **Qwen3-80B** backbone, their comparison isolates the effect of the proposed Beacon-guided semantic pipeline rather than differences in model scale.

Overall, Beacon yields the clearest and most consistent gains on Python, especially as tasks become more dependent on repository context. On `slib-runnable`, Beacon improves the baseline from $38.57/48.82/53.57\%$ to $52.14/60.32/60.71\%$ in $P@1/5/10$. On `plib-runnable`, the gain is larger, increasing from $26.19/28.57/28.57\%$ to $51.50/57.50/60.00\%$. The improvement remains strong in `class-runnable` and `file-runnable` settings, where Beacon improves Python $P@10$ by 20.00 and 17.53 percentage points over the direct-prompting baseline, respectively.

The advantage is most pronounced when tasks require stronger repository-level grounding. On Python `project-runnable`, Beacon improves $P@1$ from 7.27% to 43.16% and $P@10$ from 13.64% to 63.16%. This supports our central hypothesis: when correctness depends on non-local project semantics, flat-context prompting struggles to recover the decisive dependencies, while Beacon makes them explicit through a structured intermediate representation.

Java results are more mixed. Beacon improves the direct-prompting baseline in several repository-grounded settings, including `slib-runnable`, `plib-runnable`, `file-runnable`, and `project-runnable`. In `project-runnable`, Beacon improves the baseline from $29.52/41.50/42.86\%$ to $41.36/50.25/54.55\%$. However, Beacon underperforms the baseline on simpler Java settings such as `self-contained` and `class-runnable`. This suggests that the effectiveness of Beacon depends partly on the fidelity of language-specific semantic

Table 1: Performance on standalone and library-level CoderEval settings. Best results within each level-language block are bolded.

Level / Category	Model	Python (Pass@k, %)			Java (Pass@k, %)		
		P@1	P@5	P@10	P@1	P@5	P@10
Standalone	CodeGen	18.10	33.38	38.10	26.25	46.40	52.50
	PanGu-Coder	19.52	31.30	38.10	43.88	57.23	62.50
	ChatGPT	35.87	43.56	47.62	**64.88**	**68.65**	**70.00**
	Baseline	31.04	37.64	39.13	43.17	53.86	55.65
	Beacon (Ours)	**43.61**	**51.32**	**53.48**	39.61	48.56	51.30
Self-contained	CodeGen	22.57	36.75	40.00	27.27	49.37	56.36
	PanGu-Coder	23.71	34.93	40.00	48.36	57.94	61.82
	ChatGPT	52.29	60.08	**82.86**	61.82	66.21	67.27
	Baseline	**63.43**	65.71	65.71	52.36	**66.96**	**69.09**
	Beacon (Ours)	61.14	**69.13**	71.43	37.27	46.88	49.09
Slib-runnable	CodeGen	12.50	29.17	35.71	24.00	39.87	44.00
	PanGu-Coder	14.29	26.76	35.71	34.00	55.68	64.00
	ChatGPT	15.36	22.92	28.57	**71.60**	**74.00**	**76.00**
	Baseline	38.57	48.82	53.57	51.60	61.97	64.00
	Beacon (Ours)	**52.14**	**60.32**	**60.71**	58.40	67.33	68.00

Table 2: Performance on intermediate repository-grounded CoderEval settings. Best results within each level-language block are bolded.

Level / Category	Model	Python (Pass@k, %)			Java (Pass@k, %)		
		P@1	P@5	P@10	P@1	P@5	P@10
Plib-runnable	CodeGen	4.76	16.16	23.81	0.00	0.00	0.00
	PanGu-Coder	13.33	22.75	28.57	0.00	0.00	0.00
	ChatGPT	21.43	28.06	28.57	0.00	0.00	0.00
	Baseline	26.19	28.57	28.57	0.00	0.00	0.00
	Beacon (Ours)	**51.50**	**57.50**	**60.00**	**2.00**	**10.00**	**20.00**
Class-runnable	CodeGen	5.82	9.05	10.91	8.30	19.21	26.00
	PanGu-Coder	7.82	15.04	21.82	19.90	32.59	40.00
	ChatGPT	8.73	12.57	14.55	22.40	31.49	36.00
	Baseline	25.09	30.41	30.91	**46.60**	**55.80**	**57.73**
	Beacon (Ours)	**38.36**	**47.06**	**50.91**	37.20	45.46	48.00

extraction. Our current Python pipeline operates at AST-statement granularity, while the Java pipeline relies more heavily on line-level and heuristic dependency approximations.

These results lead to two main conclusions. First, the gains cannot be attributed to model scale: under the same Qwen3-80B backbone, Beacon substantially improves direct prompting in the Python setting and in several Java repository-grounded settings. Second, the benefits are strongest when contextual complexity increases, especially in `plib-`, `file-`, and `project-runnable` tasks. This pattern supports the view that repository-grounded code generation requires not only more context, but an explicit representation of which context is semantically central.

In this paper, we proposed **Beacon Logic** and **Beacon-Guided Generation** for repository-grounded pragmatic code generation. Instead of exposing project context as undifferentiated prompt text, our framework extracts task-relevant semantic anchors, organizes them into Beacon IR, and uses this representation to guide generation and verification. Empirical results show that making semantic centrality explicit can improve code generation without changing the underlying model, particularly in context-intensive settings.

Limitations

This work has several limitations. First, although Beacon externalizes part of the reasoning process through structured semantic artifacts, the final code generation step still depends on the underlying LLM. As a result, output quality may vary under sampling, and the framework cannot fully eliminate formatting errors or unstable generations.

Second, the current implementation of Beacon Logic differs in fidelity across languages. The Python pipeline uses AST-level information, while the Java pipeline relies more on heuristic and

Table 3: Performance on file- and project-level CoderEval settings. Best results within each level-language block are bolded.

Level / Category	Model	Python (Pass@k, %)			Java (Pass@k, %)		
		P@1	**P@5**	**P@10**	**P@1**	**P@5**	**P@10**
File-runnable	CodeGen	7.79	20.19	25.00	0.00	0.00	0.00
	PanGu-Coder	9.41	19.04	26.47	0.00	0.00	0.00
	ChatGPT	21.03	29.09	32.35	0.00	0.00	0.00
	Baseline	25.74	35.70	38.24	0.00	0.00	0.00
	Beacon (Ours)	**47.50**	**53.64**	**55.77**	**50.00**	**99.60**	**100.00**
Project-runnable	CodeGen	3.91	8.33	8.70	6.14	14.89	20.45
	PanGu-Coder	6.09	10.51	13.04	7.95	17.33	20.45
	ChatGPT	9.57	12.08	13.04	16.14	27.20	31.82
	Baseline	7.27	13.24	13.64	29.52	41.50	42.86
	Beacon (Ours)	**43.16**	**61.40**	**63.16**	**41.36**	**50.25**	**54.55**

line-level approximations. This may partly explain the stronger and more consistent gains observed in Python, and suggests that more precise language-specific analysis could further improve performance.

Third, our evaluation follows the `CoderEval` protocol. While this benchmark captures multiple runnable levels, it does not fully represent all complexities of real-world software engineering, such as large build systems, multi-repository dependencies, or long-term maintenance tasks. Therefore, the reported results should be interpreted as evidence within bounded repository-grounded generation settings rather than as a complete measure of real-world software development performance.

References

[1] Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. A Survey on Code Generation with LLM-based Agents. *arXiv preprint arXiv:2508.00083* (2025).

[2] Stanley Letovsky. 1987. Cognitive Processes in Program Comprehension. *Journal of Systems and Software* 7, 4 (1987), 325–339.

[3] Han Li, Letian Zhu, Bohan Zhang, Rili Feng, Jiaming Wang, Yue Pan, Earl T. Barr, Federica Sarro, Zhaoyang Chu, and He Ye. 2026. ContextBench: A Benchmark for Context Retrieval in Coding Agents. *arXiv preprint arXiv:2602.05892* (2026).

[4] Yujia Li et al. 2022. Competition-Level Code Generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097.

[5] Unaizah Obaidellah, Mohammed Al Haek, and Peter C.-H. Cheng. 2019. A Survey on the Usage of Eye-Tracking in Computer Programming. *ACM Computing Surveys* 51, 1 (2019).

[6] Norman Peitek, Janet Siegmund, and Sven Apel. 2020. What Drives the Reading Order of Programmers? An Eye Tracking Study. In *Proceedings of the IEEE/ACM International Conference on Program Comprehension (ICPC)*.

[7] Zhensu Sun, Xiaoning Du, Fu Song, Shangwen Wang, Mingze Ni, and Li Li. 2022. Learning to Prevent Profitless Neural Code Completion. *arXiv preprint arXiv:2209.05948* (2022).

[8] Susan Wiedenbeck. 1986. Beacons in computer program comprehension. *International Journal of Man-Machine Studies* 25, 6 (1986), 697–709. doi:10.1016/S0020-7373(86)80083-9.

[9] Shaochun Xu and Vaclav Rajlich. 2004. Cognitive Process During Program Debugging. In *Proceedings of the Third IEEE International Conference on Cognitive Informatics*. 176–182.

[10] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems (NeurIPS 2024)*. arXiv:2405.15793.

[11] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629* (2022).

[12] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*.

[13] Qianhui Zhao, Li Zhang, Fang Liu, Junhang Cheng, Chengru Wu, Junchen Ai, Qiaoyuanhe Meng, Lichen Zhang, Xiaoli Lian, Shubin Song, and Yuanping Guo. 2025. Towards Realistic Project-Level Code Generation via Multi-Agent Collaboration and Semantic Architecture Modeling. *arXiv preprint arXiv:2511.03404* (2025).

[14] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity Assessment of Neural Code Completion. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).*