Protocode: Prototype-Driven Interpretability for Code Generation in LLMs

Anonymous AACL submission

Abstract

Since the introduction of Large Language Models (LLMs), they have been widely adopted for various tasks such as text summarization, question answering, speech-to-text translation, and more. In recent times, the use of LLMs for code generation has gained significant attention, with tools such as Cursor and Windsurf demonstrating the ability to analyze massive code repositories and recommend relevant changes. Big tech companies have also acknowledged the growing reliance on LLMs for code generation within their codebases. Although these advances significantly improve developer productivity, increasing reliance on automated code generation can proportionally increase the risk of suboptimal solutions and insecure code. We introduce a retrieval framework for code generation that connects geometry-aware prototype sampling with syntax-grounded interpretability analysis. By conditioning AST-based explanations on retrieved demonstrations, our method evaluates retrieval quality not only through functional correctness, but also through interpretability alignment. In our experiments, we show that high-quality ICL demonstrations not only make outputs easier to interpret but also yield a positive performance improvement on the P@10 metric. Conversely, poorly chosen ICL demonstrations negatively affected P@10 performance compared to the base model. Overall, our approach highlights the importance of efficient sampling strategies for ICL, which can affect the performance of the model on any given task.

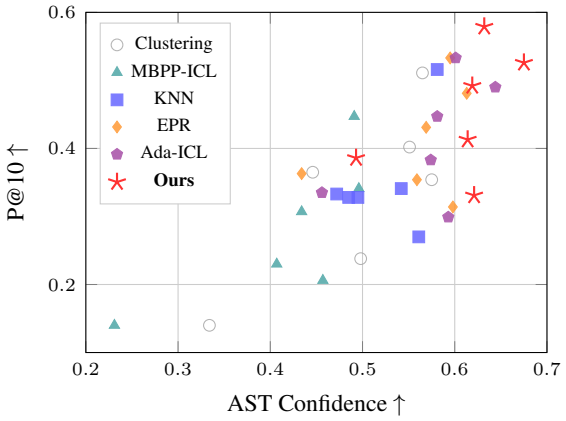


Figure 1: AST Confidence vs. P@10 across all models and ICL strategies. **Ours** (red stars) consistently occupies the upper-right region.

1 Introduction

In recent years, Large Language Models (LLMs) have gained significant traction in the fields of code completion and code filling. This growth has been fueled by the availability of large-scale open-source datasets such as The vault (Manh et al., 2023), CodeSearchNet (Husain et al., 2020), CodeXGlue (Lu et al., 2021) and many others. Alongside these datasets, we have also witnessed the emergence of open-source models designed specifically for code-related tasks, including the CodeLlama series (Rozière et al., 2024), Qwen Coder (Hui et al., 2024) series, and StarCoder series (Li et al., 2023a). In parallel, closed-source models such as GPT-4o (OpenAI et al., 2024) and Claude Code (Anthropic, 2025) have been widely adopted by various big tech companies for generating production-ready code. Despite these advancements, most of these models remain difficult to interpret in the context of code generation. While a variety of interpretability methods have been developed to interpret the outputs generated by LLMs and foster trust in their usage across domains, many of these approaches are generic and not specifically tailored for code generation tasks. Some methods, however, specifically focus on interpretability for code generation. For instance, Code-Q (Palacio et al., 2025) identifies influential tokens that guide the model's output, but it requires repeated sampling and generation, which introduces significant computational overhead during inference.

Another method, ASTrust (Palacio et al., 2024),

leverages Abstract Syntax Trees (ASTs) by using model-generated token probabilities to generate syntax-grounded explanations for code generation. Tokens are mapped to code-level subsets, which are then grouped into terminal and non-terminal AST nodes, where each non-terminal node is represented by the aggregated confidence of its associated terminal nodes. While effective, this approach requires storing probability distributions over the entire vocabulary at every decoding step, which scales poorly as the output length increases. More importantly, existing syntax-grounded interpretability methods such as ASTrust analyze generated code independently of the retrieval process, whereas most manifold learning and metric learning approaches focus primarily on representation learning and retrieval quality without considering interpretability. In contrast, our work connects these previously separate directions through a unified framework for interpretable code generation, where geometry-aware prototype retrieval is directly linked to syntax-grounded explanations of generated code. Specifically, we propose a manifold-guided prototype sampling strategy that automatically selects representative ICL demonstrations from a dataset and evaluates their quality not only through downstream functional correctness, but also through interpretability alignment. By combining prototype-conditioned attribution with AST-based analysis, our framework identifies which syntactic regions of generated code such as iterations, data structures, and conditional logic are most influenced by the retrieved demonstrations, enabling retrieval-aware and syntax-grounded interpretability for code generation in LLMs.

- **Unified Retrieval and Interpretability Framework for Code Generation** We introduce a framework that jointly connects prototype-based retrieval, manifold-aware representation learning, and syntax-grounded interpretability for LLM code generation. Unlike prior approaches that optimize retrieval quality or interpretability independently, our method links the quality of retrieved demonstrations with both functional correctness and AST-grounded explanation quality.

2 Related work

2.1 In context learning

In-context learning (ICL) (Brown et al., 2020) is a paradigm that enables language models to perform tasks using only a few demonstrations without explicit parameter updates. Since demonstrations are expressed in natural language, ICL provides an interpretable interface for interacting with large language models (LLMs). Furthermore, ICL resembles the human decision-making process of learning through analogy (Winston, 1980). Unlike supervised training, ICL is a training-free framework that allows models to generalize to new tasks without additional computational costs for fine-tuning.

Based on (Dong et al., 2024), several unsupervised strategies have been proposed to sample effective demonstrations for ICL. A simple yet effective method is to select the nearest neighbors of the input instance based on similarity measures ((Liu et al., 2022), (Tanwar et al., 2023), (Qin et al., 2024)). Common distance metrics include L2 distance and cosine similarity derived from sentence embeddings. Beyond distance-based approaches, mutual information (Sorensen et al., 2022) and perplexity (Gonen et al., 2023) have also been shown to be useful for selecting prompts without labeled data or model-specific assumptions.

Although off-the-shelf retrievers provide convenient solutions for a wide range of NLP tasks, they are often heuristic and sub-optimal due to the absence of task-specific supervision. To overcome this limitation, supervised retriever-based methods have been introduced ((Rubin et al., 2022) (Ye et al., 2023) (Wang et al., 2024) (Zhang et al., 2022)). For instance, (Rubin et al., 2022) proposed EPR, a two-stage framework for training dense retrievers to identify suitable demonstrations. Building on this, (Li et al., 2023c) developed a unified retriever capable of selecting demonstrations across diverse tasks, while (Mavromatis et al., 2023) introduced AdaICL, a model-adaptive method that leverages LLMs to predict outcomes for unlabeled data and assign uncertainty scores to guide demonstration selection.

(Rodriguez-Cardenas et al., 2023) emphasized the sensitivity of demonstration selection by comparing two different prompt groups in a controlled experiment. One group exhibited a positive causal effect, improving the Average Treatment Effect (ATE) by 5.1% on ChatGPT, while the other group showed a negative causal effect, decreasing ATE by 3.3% relative to the control group. Here, ATE quantifies the average causal influence of a treatment (i.e., the chosen prompt group) on model performance. These findings highlight the critical role of demonstration quality: poorly chosen

examples may reduce performance, sometimes performing worse than LLMs that do not use ICL at all. Throughout the paper, we use the terms demonstrations and examples interchangeably in the context of ICL.

3 Methodology

Prototype-based approaches provide an interpretable mechanism to associate each class with representative examples, commonly referred to as prototypes. A simple baseline is to define prototypes using statistics such as class means or medoids in the embedding space. However, these statistical summaries fail to capture the intrinsic geometry of the representation space: they are vulnerable to outliers, insensitive to intra-class multimodality, and often yield prototypes that are statistically central yet semantically uninformative.

To overcome these shortcomings, we turn to the manifold perspective. The manifold hypothesis Cayton (2005) posits that high-dimensional representations lie on low-dimensional manifolds. Leveraging this structure allows prototypes to be sampled from regions that capture the local geometry of the data, rather than from globally averaged or distorted positions in embedding space. While classical manifold learning techniques such as t-SNE (van der Maaten and Hinton, 2008), UMAP (McInnes et al., 2020), and LLE (Roweis and Saul, 2000) emphasize neighborhood preservation, they often distort local dependencies or fail to maintain global structure. We therefore adopt a piecewise-linear manifold learning strategy, which decomposes nonlinear manifolds into locally linear regions.

While geometry preserves structural fidelity, it does not guarantee that prototypes are discriminative across classes. To enforce both intra-class compactness and inter-class separation, we integrate metric learning objectives. Traditional formulations such as triplet or contrastive loss require prespecified prototypes and extensive mining, making them inefficient and unstable. Instead, we employ Proxy-Anchor loss, which introduces learnable class-level proxy vectors to directly optimize intra-class cohesion and inter-class margins. After training, each learned proxy vector is mapped to its nearest training instance using Euclidean distance.

As highlighted in Rodriguez-Cardenas et al. (2023), in the context of ICL, the selection of demonstrations plays a crucial role in model performance. In our approach, we dedicate considerable effort to identifying the most suitable prototypes (ICL examples) for each LLM. Our method can be divided into two main components. In the first stage, we initialize a simple neural network h_θ and train it on Dataset D to jointly optimize manifold learning and metric learning objectives. Once the training is complete, the learned proxy vectors are employed to sample prototypes.

3.1 Dataset

We have used the Magicoder-OSS-Instruct-75K (Wei et al., 2024) for sampling the prototypes. This dataset consists of 75,000 synthetic instruction-following examples generated using OSS-Instruct; it contains 9 programming languages. For every query, it has a programming language id, the query, and the code solution. For every sample in the dataset, we have used the following prompt structure to format all the samples in the dataset.

Prompt Structure: *"This is the query being assigned:"+" "+ [/Q]+" "+"The following is the code solution to the query"+" "+[/S]"*. Where the placeholders *[/Q]* and *[/S]* are for query and code solution respectively. After formatting the prompts, we use the respective Large Language Model (M) to encode the final prompts into the latent representations (z). We simultaneously label encode the programming language ID for using them as class labels; this method gives us 9 different classes, and for each sample in the dataset, we will be storing the encoded label (l) and the latent representation (z) as pairs in dataset D.

3.2 Training overview

As mentioned in Section 3, our method consists of two stages. In the first stage of our method, we initialize a simple neural network h_θ (Table 3) and train it on Dataset D to jointly optimize manifold learning (Eq. 2) and metric learning objectives (Eq. 1). The neural network h_θ learns to map the high-dimensional encoded representations into lower dimensions. Before the training process, we initialize the proxies θ_q and θ_m. Here both the proxies are unique for each class and initialized randomly with $\theta_q = \theta_m$. The proxy vector θ_q is updated via back-propagation, and the proxy vector θ_m is updated via the Momentum update (He et al., 2020) where γ is the momentum constant, $[\theta_k \leftarrow \gamma\theta_k + (1-\gamma)\theta_q]$

During training, for every mini-batch B we build linear piecewise manifolds as outlined in Section

3.3. For every point in B, we then compute the manifold-based similarity following the procedure in Section 3.4. This similarity measure is used to compute the manifold point-to-point loss $\mathcal{L}_{\text{manifold}}$. At the same time, we compute the Proxy Anchor loss $\mathcal{L}_{\text{PA}}$ using randomly initialized class proxies θ_q and latent representations z in batch B. The final loss is computed as, $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{PA}} + \mathcal{L}_{\text{manifold}}$.

While the manifold loss preserves local geometric structure, the Proxy-Anchor loss promotes intra-class compactness and inter-class separation, thereby facilitating the discriminative learning of prototypes. Across epochs, the network parameters are updated via backpropagation. After training, the momentum-updated proxies θ_m are used to select the nearest training instance as the prototype for each class, yielding a single prototype per class for the subsequent stage.

After the completion of the training process, we proceed to the second stage, where we generate the code completions using the prototypes. After that, we utilize the encoded latent representations of the prototypes to calculate the confidence score per token for AST analysis. For each code completion, we compute a prototype attribution-based score to quantify the influence of the prototypes on the generated code. Specifically, influence is measured by the attribution between the sampled demonstrations and the code completions. Finally, we perform an AST analysis to analyze how the prototypes impact the syntactic structure of the generated code.

3.3 Manifold Construction

Based on the manifold hypothesis, we can assume that the encoded latent representations z, which are inherently complex and non-linear, can be locally approximated into smaller chunks of linear regions. Our approach leverages this structural assumption to automatically identify representative prototypes that capture the essential characteristics of each action class.

To efficiently approximate the structure of the linear data manifolds, we adopt a piecewise linear manifold learning method which constructs localized m-dimensional linear submanifolds around selected anchor points. Given a mini batch B containing N data points, we randomly select n of them to serve as anchors. For each anchor point $h_\theta(z_i)$, we initially collect its $m-1$ nearest neighbors in the encoded representation space based on Euclidean distance to form the neighborhood set X_i.

The manifold expansion process proceeds iteratively by attempting to add the m-th nearest neighbor to X_i. After each addition, we recompute the best-fit m-dimensional submanifold using PCA and assess whether all points in X_i can be reconstructed with a quality above a threshold $T\%$. If the reconstruction quality remains acceptable, the new point is retained in X_i; otherwise, it is excluded. This evaluation is repeated for subsequent neighbors $N(h_\theta(x_i))_j$ for $j \in \{m+1, \ldots, k\}$, gradually constructing a local linear approximation of the manifold.

The final set X_i comprises all points in the anchor's neighborhood that lie well within an m-dimensional linear submanifold. A basis for this submanifold is computed by applying PCA to X_i and extracting the top m eigenvectors. We choose PCA for this task as it can effectively construct the lower-dimensional manifolds for the locally linear regions.

3.4 Training objectives

Proxy Anchor Loss: We use a modified version of proxy anchor loss with Euclidean distance instead of cosine similarity:

$$\mathcal{L}_{\text{PA}} = \frac{1}{|\Theta_+|} \sum_{\theta_q \in \Theta_+} \log\Bigg(1 + \sum_{Z' \in \mathcal{Z}'^{+}_{\theta_q}} \exp\Big(-\alpha\big(\|h_\theta(Z) - \theta_q\|_2 - \epsilon\big)\Big)\Bigg) + \frac{1}{|\Theta|} \sum_{\theta_q \in \Theta} \log\Bigg(1 + \sum_{Z' \in \mathcal{Z}'^{-}_{\theta_q}} \exp\Big(\alpha\big(\|h_\theta(Z) - \theta_q\|_2 - \epsilon\big)\Big)\Bigg) \tag{1}$$

Here, Θ denotes the set of all proxies, where each proxy $\theta_q \in \Theta$ serves as a representative vector for a class. The subset $\Theta_+ \subseteq \Theta$ includes only those proxies that have at least one positive embedding in the current batch B. For a given proxy θ_q, the latent representations $\mathcal{Z}$ in B (where $z \in \mathcal{Z}$) are partitioned into two sets: $\mathcal{Z}^+_{\theta_q}$, the positive embeddings belonging to the same class as θ_q, and $\mathcal{Z}^-_{\theta_q} = \mathcal{Z} \setminus \mathcal{Z}^+_{\theta_q}$, the negative embeddings. The scaling factor α controls the sharpness of optimization by amplifying hard examples when large (focusing gradients on difficult pairs) or smoothing training when small (spreading weight across all pairs). The margin ϵ enforces a buffer zone between positives and negatives by requiring positives to be closer to their proxies and negatives to be sufficiently farther away.

Manifold Point-to-Point Loss: This loss helps in estimating the point to point similarities preserv-

ing the geometric structure:

$$\mathcal{L}_{\text{manifold}} = \sum_{i,j} (\delta \cdot (1 - s(z_i, z_j)) - \|h_\theta(z_i) - h_\theta(z_j)\|_2)^2 \tag{2}$$

where $s(z_i, z_j)$ is the manifold similarity computed as:

$$s(z_i, z_j) = \frac{s'(z_i, z_j) + s'(z_j, z_i)}{2}$$

with $s'(z_i, z_j) = \alpha(z_i, z_j) \cdot \beta(z_i, z_j)$, where:

$$\alpha(z_i, z_j) = \frac{1}{(1 + o(z_i, z_j)^2)^{N_\alpha}}$$
$$\beta(z_i, z_j) = \frac{1}{(1 + p(z_i, z_j))^{N_\beta}}$$

In Equations (1) & (2), h_θ is a simple neural network with a structure specified in Table 3 and δ is a scaling factor that determines the maximum separation between dissimilar points. The loss encourages Euclidean distances in the embedding space to match manifold-based dissimilarities $1 - s(z_i, z_j)$, ensuring that the learned metric space respects the underlying manifold structure. $o(z_i, z_j)$ is the orthogonal distance from point z_i to the manifold of point z_j, and $p(z_i, z_j)$ is the projected distance between point z_j and the projection of z_i on the manifold. The parameters N_α and N_β control how rapidly similarity decays with distance, with $N_\alpha > N_\beta$ ensuring that similarity decreases more rapidly for points lying off the manifold than for points on the same manifold.

Distance Calculation. For each point pair (z_i, z_j), the distances $o(z_i, z_j)$ and $p(z_i, z_j)$ are calculated using the manifold basis vectors P_j associated with point z_j. The projection of z_i onto P_j is computed as $\text{proj}_{P_j}(z_i) = z_j + \sum_k \langle z_i - z_j, v_k \rangle v_k$, where v_k are the basis vectors of P_j. The orthogonal distance is then $o(z_i, z_j) = \|z_i - \text{proj}_{P_j}(z_i)\|_2$, and the projected distance is $p(z_i, z_j) = \|\text{proj}_{P_j}(z_i) - z_j\|_2$. This process is repeated for all point pairs, capturing the full geometric structure of the data manifold.

4 Results

We evaluated the effectiveness of different sampling methods by applying them as in-context learning (ICL) examples on the MBPP test set (Austin et al., 2021). To ensure fair evaluation across retrieval strategies, all ICL methods sample exactly one demonstration per class and use an identical class-wise ordering based on class IDs within the prompt. This produces the same number of demonstrations and a consistent prompt structure across all methods. When the retrieved demonstrations exceed the context limit (set as respective model's context length), truncation is applied uniformly across all methods. Therefore, performance differences primarily reflect the quality of the retrieved demonstrations rather than variations in prompt ordering, formatting, or demonstration count. For methods that retrieve multiple examples from the same class, only the highest-ranked instance from each class is retained to maintain a consistent class-balanced prompt across all methods. To demonstrate the effectiveness of our method we have used 2 sets of models for experimentation, the first set consisting of generic models of Qwen3 (Yang et al., 2025), Llama-3.2 (AI, 2024), Falcon-3 (Team, 2024) and for the second set we have used code-specialized pretrained models. StarCoder-base (Li et al., 2023b), Qwen2.5-Coder (Hui et al., 2024), CodeLlama (Rozière et al., 2024).

The results are reported on a scale from $[0, 1]$, where 0 is the lowest and 1 is the highest (for instance, 0.1 can be interpreted as 10%). The Qwen2.5-Coder model, despite having fewer parameters than CodeLlama, achieves comparable performance on both the $P@1$ and $P@10$ metrics across the MBPP and MBPP+ test sets. As noted by (Rodriguez-Cardenas et al., 2023), within the ICL setting, the quality of selected demonstrations can also negatively affect model performance.

For the Qwen3 and Qwen2.5-Coder models, using demonstrations sampled from distance based approaches leads to a decline in performance on both MBPP and MBPP+. A similar trend is observed for the StarCoder and CodeLlama models. These results suggest that the Qwen family of models, as well as code-specialized pretrained models in general, are particularly sensitive to the choice of ICL demonstrations. An unsuitable set of demonstrations can reduce performance compared to the base model, underscoring the importance of effective sampling strategies for ICL. To assess scalability, we evaluated our method on two larger models: Qwen2.5-Coder-14B and CodeLlama-13B. The results (Table 5) demonstrate that our approach scales effectively to larger architectures, maintaining consistent performance gains over both heuristic and learned retriever baselines. For each target LLM, we train a separate prototype selector h_θ using embeddings extracted from that specific model. Consequently, the retriever becomes model-specific,

Model	Metric	Zero shot	Clustering	MBPP-ICL	Ours	KNN	EPR	Ada-ICL
CodeLlama-7B	MBPP P@1	0.159	0.038	0.071	**0.228**	0.114	0.205	0.183
	MBPP P@10	0.497	0.140	0.341	**0.526**	0.328	0.481	0.490
	MBPP+ P@1	0.129	0.033	0.053	**0.190**	0.094	0.154	0.167
	MBPP+ P@10	0.407	0.119	0.278	**0.450**	0.278	0.432	0.426
Falcon3-1B	MBPP P@1	0.112	0.119	0.025	**0.155**	0.116	0.131	0.125
	MBPP P@10	0.342	0.365	0.140	**0.386**	0.333	0.363	0.335
	MBPP+ P@1	0.097	0.097	0.018	**0.134**	0.097	0.114	0.097
	MBPP+ P@10	0.288	0.288	0.101	**0.318**	0.267	0.288	0.296
LLaMA-3.2-1B	MBPP P@1	0.111	0.153	0.043	**0.157**	0.111	0.126	0.131
	MBPP P@10	0.354	0.354	0.230	**0.413**	0.341	0.354	0.383
	MBPP+ P@1	0.097	0.125	0.029	**0.135**	0.097	0.125	0.097
	MBPP+ P@10	0.310	0.310	0.167	**0.344**	0.275	0.325	0.325
Qwen2.5-Coder-0.5B	MBPP P@1	0.233	0.208	0.115	**0.308**	0.229	0.247	0.256
	MBPP P@10	0.521	0.511	0.447	**0.579**	0.516	0.533	0.533
	MBPP+ P@1	0.188	0.167	0.095	**0.267**	0.193	0.216	0.225
	MBPP+ P@10	0.437	0.410	0.378	**0.492**	0.434	0.451	0.458
Qwen3-0.6B	MBPP P@1	0.101	0.144	0.055	**0.203**	0.112	0.153	0.161
	MBPP P@10	0.373	0.402	0.307	**0.492**	0.328	0.431	0.447
	MBPP+ P@1	0.084	0.115	0.044	**0.167**	0.087	0.137	0.128
	MBPP+ P@10	0.323	0.344	0.257	**0.418**	0.270	0.344	0.366
StarCoderBase-1B	MBPP P@1	0.098	0.056	0.043	**0.113**	0.098	0.098	0.106
	MBPP P@10	0.299	0.238	0.206	**0.331**	0.270	0.314	0.299
	MBPP+ P@1	0.079	0.047	0.029	**0.091**	0.079	0.065	0.079
	MBPP+ P@10	0.230	0.183	0.148	**0.272**	0.230	0.242	0.249

Table 1: Performance on MBPP and MBPP+ datasets with different ICL selection strategies

allowing the learned prototypes to adapt to the latent representation geometry of each LLM independently. The prototype-selection stage is performed offline once per model and does not introduce additional training during inference-time code generation. Although MBPP is a Python benchmark, cross-language prototypes can still encode transferable algorithmic and structural reasoning patterns (e.g., iteration, recursion, data-structure manipulation). Our method therefore treats language-specific manifolds as carriers of broader program semantics rather than syntax alone.

5 AST analysis

To analyze the influence of the sampled ICL demonstrations on the generated code we perform an AST analysis. In the ASTrust framework, the authors employ token-level probabilities produced by the model M as the confidence scores in the token set. For a sequence of tokens $w_1, w_2, \ldots, w_i$, the probability of generating the next token w_{i+1} is defined as Eq. (3) where M denotes the Large Language Model and $M(w_{1:i})$ represents the non-normalized log probabilities output by the model for the given context.

$$P(w_{i+1} \mid w_{1:i}) = \mathrm{Softmax}\big(M(w_{1:i})\big), \quad (3)$$

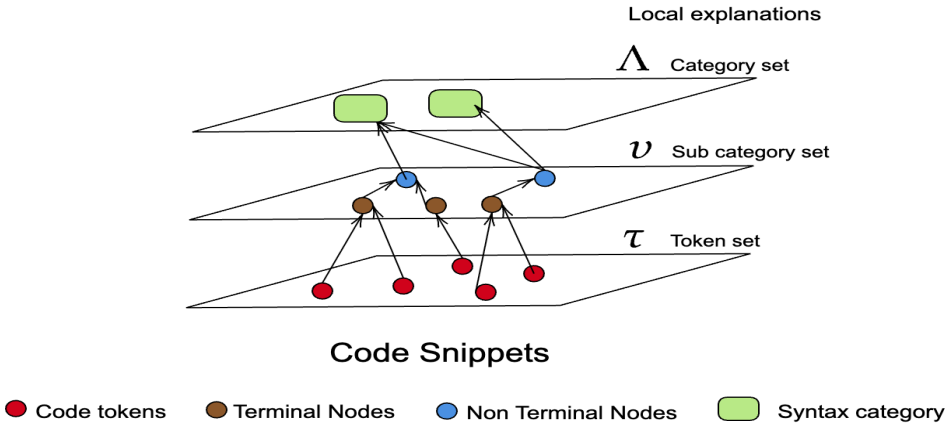
While ASTrust analyzes token confidence distributions produced during decoding, our method introduces prototype-conditioned attribution analysis that links retrieval behavior with syntax-grounded interpretability; we replace token probabilities with attribution-based prototype influence scores derived from the relationship between the sampled demonstrations and the generated code. Concretely, given a model-generated code snippet C, we first extract the generated tokens w_i and their contextual latent representations z_{w_i}. Let z_p denote the latent representation of a sampled prototype p. We compute the average prototype representation as

$$z_a = \frac{1}{|P|} \sum_{i \in P} z_i,$$

where P denotes the set of sampled prototypes. We then compute the cosine similarity between z_a and each token representation z_{w_i}, and calculate its gradient with respect to z_{w_i}. The normalized gradient magnitudes $\nabla_{z_{w_i}}$ (Eq. (4)) are subsequently used as token-level confidence scores for AST analysis.

$$\nabla_{z_{w_i}} = \left\| \frac{\partial \cos(z_{w_i}, z_a)}{\partial z_{w_i}} \right\|_2 \quad (4)$$

Figure 2: Conceptual working of AST analysis ((Palacio et al., 2024))



5.1 Syntax Grounded explanations

AST analysis involves using the prototype-based attribution scores as token confidence scores as discussed in Section 5. We then compute the average confidence over tokens corresponding to each AST node, and report these averages as performance values grouped by manually defined syntax categories. The process follows three steps, illustrated in Fig. 2. In Step 1, for every generated code snippet, the tokenizer splits the code into tokens w_i (forming the token set τ D.1), and the model assigns a confidence score to each token as described in Section 5. In Step 2, the token-level predictions are aligned with the respective Abstract Syntax Tree (AST) terminal nodes. Terminal nodes retain the raw confidences, whereas non-terminal nodes hierarchically store aggregated values. Together, terminal and non-terminal nodes form the subcategory set υ D.1. For instance, the token 'if_' from the token set aligns with a terminal AST node but is grouped under the non-terminal node 'if_statement'. Finally, in Step 3, the analysis introduces eight syntax categories to summarize model predictions. These categories aggregate subcategories into broader, human-interpretable groups. The Syntax Categories form a fixed Category Set Λ D.1, providing more intuitive elements for interpretation.

For example, the sub-categories 'if_statement' and 'if' are grouped under the syntax category 'Decisions' Fig. 3. Ultimately, ASTrust outputs an averaged score for each category to provide global explanations, along with an AST tree visualization that embeds confidence scores at every node for local explanations. In essence, we argue that syntax elements encode semantic information that contextualizes token-level confidence scores, though this semantic value differs depending on the granularity of the elements. For instance, tokens alone convey less interpretable meaning compared to higher-level categories. AST analysis thus serves as a post-hoc explanation framework at both local and global levels. Local explanations focus on breaking down a single code snippet into AST elements to interpret its generation, while global explanations rely on multiple generated snippets to provide a holistic view of the model through Syntax Categories (SCs) D.1.

5.2 Retrieval Quality as Interpretability Alignment

Existing retrieval strategies for in-context learning are typically evaluated using downstream accuracy alone. In contrast, we investigate whether retrieval quality can additionally be assessed through interpretability alignment. Specifically, we analyze whether demonstrations that improve functional correctness also produce more coherent syntax-grounded attribution patterns within generated code. For each method, we compute attribution scores following Section 5 and aggregate them per syntax category across all generated snippets in the MBPP test set. Table 2 shows that our prototype-based method consistently achieves the highest mean AST confidence across all models, mirroring the P@10 rankings in Table 1. In addition, Section D.2 presents a syntax-aware analysis of prototype influence across different syntactic regions of the generated code.

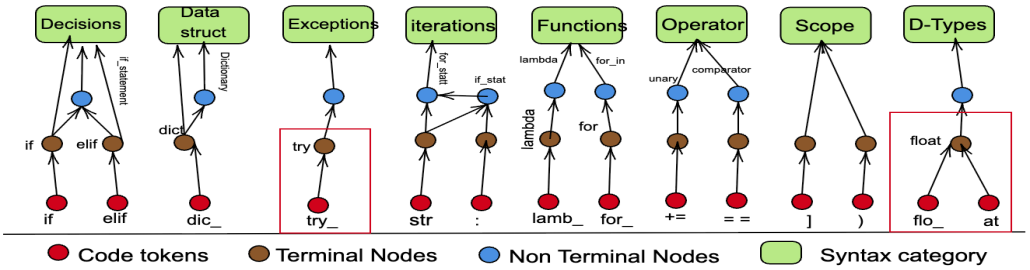


Figure 3: Alignment & Clustering Interactions. The δ function aligns tokens w_i to terminal nodes λ. Terminal and Non-terminal nodes λ, $\alpha \in \upsilon$ are clustered by Syntax Categories Λ ((Palacio et al., 2024))

Model	ICL Strategy	Functional Correctness		AST Alignment
		P@1 ↑	P@10 ↑	AST Conf. ↑
CodeLlama-7B	Clustering	0.038	0.140	0.334
	MBPP-ICL	0.071	0.341	0.496
	KNN	0.114	0.328	0.485
	EPR	0.205	0.481	0.613
	Ada-ICL	0.183	0.490	0.644
	Ours	**0.228**	**0.526**	**0.675**
Falcon3-1B	Clustering	0.119	0.365	0.446
	MBPP-ICL	0.025	0.140	0.231
	KNN	0.116	0.333	0.472
	EPR	0.131	0.363	0.434
	Ada-ICL	0.125	0.335	0.456
	Ours	**0.155**	**0.386**	**0.493**
LLaMA-3.2-1B	Clustering	0.153	0.354	0.575
	MBPP-ICL	0.043	0.230	0.407
	KNN	0.111	0.341	0.542
	EPR	0.126	0.354	0.559
	Ada-ICL	0.131	0.383	0.574
	Ours	**0.157**	**0.413**	**0.614**
Qwen2.5-Coder-0.5B	Clustering	0.208	0.511	0.565
	MBPP-ICL	0.115	0.447	0.491
	KNN	0.229	0.516	0.581
	EPR	0.247	0.533	0.595
	Ada-ICL	0.256	0.533	0.601
	Ours	**0.308**	**0.579**	**0.632**
Qwen3-0.6B	Clustering	0.144	0.402	0.551
	MBPP-ICL	0.055	0.307	0.434
	KNN	0.112	0.328	0.495
	EPR	0.153	0.431	0.569
	Ada-ICL	0.161	0.447	0.581
	Ours	**0.203**	**0.492**	**0.619**
StarCoderBase-1B	Clustering	0.056	0.238	0.498
	MBPP-ICL	0.043	0.206	0.457
	KNN	0.098	0.270	0.561
	EPR	0.098	0.314	0.598
	Ada-ICL	0.106	0.299	0.593
	Ours	**0.113**	**0.331**	**0.621**

Table 2: Mean AST confidence score vs P@1, P@10 metrics.

6 Limitations and Future Works

In our experiments, prototypes were sampled exclusively from the Magicoder dataset. While this choice provided a consistent basis for evaluation, extending the analysis to additional datasets could offer a broader understanding of prototype quality. In fact, our method can naturally be applied as a global metric for ranking datasets with respect to

their ability to yield effective prototypes. Another limitation arises from differences in model stability. For example, Llama3.2 (Fig. 5) exhibited high sensitivity to changes in nearly all hyperparameters, which led to inconsistent results on the $P@k$ metric. In contrast, the Qwen2.5 Coder model (Fig. 5) displayed only marginal sensitivity, with the exception of the α parameter, resulting in more stable and reliable performance. Finally, while our current approach uses sampled prototypes as in-context learning demonstrations, the framework can be extended toward pre-hoc interpretability by design. More broadly, our framework opens the possibility of interpretability-aware retrieval systems, where demonstration selection is optimized not only for downstream performance but also for controllability.

References

Meta AI. 2024. Llama 3.2 connect 2024: Vision for edge mobile devices.

Anthropic. 2025. Claude code. https://claude.com/product/claude-code. Accessed: 2025-09-24.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Ahsan Bilal, David Ebert, and Beiyu Lin. 2025. Llms for explainable ai: A comprehensive survey. *Preprint*, arXiv:2504.00125.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

Lawrence Cayton. 2005. Algorithms for manifold learning.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *Preprint*, arXiv:2110.14168.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, Baobao Chang, Xu Sun, Lei Li, and Zhifang Sui. 2024. A survey on in-context learning. *Preprint*, arXiv:2301.00234.

Hila Gonen, Srini Iyer, Terra Blevins, Noah Smith, and Luke Zettlemoyer. 2023. Demystifying prompts in language models via perplexity estimation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 10136–10148, Singapore. Association for Computational Linguistics.

Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. Momentum contrast for unsupervised visual representation learning. *Preprint*, arXiv:1911.05722.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: Evaluating the state of semantic code search. *Preprint*, arXiv:1909.09436.

Pang Wei Koh, Thao Nguyen, Yew Siang Tang, Stephen Mussmann, Emma Pierson, Been Kim, and Percy Liang. 2020. Concept bottleneck models. *Preprint*, arXiv:2007.04612.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023a. Starcoder: may the source be with you! *Preprint*, arXiv:2305.06161.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023b. Starcoder: may the source be with you!

Xiaonan Li, Kai Lv, Hang Yan, Tianyang Lin, Wei Zhu, Yuan Ni, Guotong Xie, Xiaoling Wang, and Xipeng Qiu. 2023c. Unified demonstration retriever for in-context learning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4644–4668, Toronto, Canada. Association for Computational Linguistics.

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2022. What makes good in-context examples for GPT-3? In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pages 100–114, Dublin, Ireland and Online. Association for Computational Linguistics.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, and 3 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664.

Scott Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. *Preprint*, arXiv:1705.07874.

Wei Ma, Shangqing Liu, Zhihao Lin, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, Li Li, and Yang Liu. 2024. Lms: Understanding code syntax and semantics for code analysis. *Preprint*, arXiv:2305.12138.

Dung Nguyen Manh, Nam Le Hai, Anh TV Dau, Anh Minh Nguyen, Khanh Nghiem, Jin Guo, and Nghi DQ Bui. 2023. The vault: A comprehensive multilingual dataset for advancing code understanding and generation. *arXiv preprint arXiv:2305.06156*.

Costas Mavromatis, Balasubramaniam Srinivasan, Zhengyuan Shen, Jiani Zhang, Huzefa Rangwala, Christos Faloutsos, and George Karypis. 2023. Which examples to annotate for in-context learning? towards effective and efficient selection. *Preprint*, arXiv:2310.20046.

Leland McInnes, John Healy, and James Melville. 2020. Umap: Uniform manifold approximation and projection for dimension reduction. *Preprint*, arXiv:1802.03426.

OpenAI, :, Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, Aleksander Mądry, Alex Baker-Whitcomb, Alex Beutel, Alex Borzunov, Alex Carney, Alex Chow, Alex Kirillov, and 401 others. 2024. Gpt-4o system card. *Preprint*, arXiv:2410.21276.

David N. Palacio, Dipin Khati, Daniel Rodriguez-Cardenas, Alejandro Velasco, and Denys Poshyvanyk. 2025. On explaining (large) language models for code using global code-based explanations. *Preprint*, arXiv:2503.16771.

David N. Palacio, Daniel Rodriguez-Cardenas, Alejandro Velasco, Dipin Khati, Kevin Moran, and Denys Poshyvanyk. 2024. Towards more trustworthy and interpretable llms for code through syntax-grounded explanations. *Preprint*, arXiv:2407.08983.

Chengwei Qin, Aston Zhang, Chen Chen, Anirudh Dagar, and Wenming Ye. 2024. In-context learning with iterative demonstration selection. *Preprint*, arXiv:2310.09881.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *Preprint*, arXiv:2009.10297.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "why should i trust you?": Explaining the predictions of any classifier. *Preprint*, arXiv:1602.04938.

Daniel Rodriguez-Cardenas, David N. Palacio, Dipin Khati, Henry Burke, and Denys Poshyvanyk. 2023. Benchmarking causal study to interpret large language models for source code. *Preprint*, arXiv:2308.12415.

Sam T. Roweis and Lawrence K. Saul. 2000. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Ohad Rubin, Jonathan Herzig, and Jonathan Berant. 2022. Learning to retrieve prompts for in-context learning. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2671, Seattle, United States. Association for Computational Linguistics.

Chufan Shi, Haoran Yang, Deng Cai, Zhisong Zhang, Yifan Wang, Yujiu Yang, and Wai Lam. 2024. A thorough examination of decoding methods in the era of llms. *Preprint*, arXiv:2402.06925.

Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg. 2017. Smoothgrad: removing noise by adding noise. *Preprint*, arXiv:1706.03825.

Taylor Sorensen, Joshua Robinson, Christopher Rytting, Alexander Shaw, Kyle Rogers, Alexia Delorey, Mahmoud Khalil, Nancy Fulda, and David Wingate. 2022. An information-theoretic approach to prompt engineering without ground truth labels. In *Proceedings of the 60th Annual Meeting of the Association*

for Computational Linguistics (Volume 1: Long Papers), pages 819–862, Dublin, Ireland. Association for Computational Linguistics.

Chung-En Sun, Tuomas Oikarinen, Berk Ustun, and Tsui-Wei Weng. 2025. Concept bottleneck large language models. *Preprint*, arXiv:2412.07992.

Mukund Sundararajan, Ankur Taly, and Qiqi Yan. 2017. Axiomatic attribution for deep networks. *Preprint*, arXiv:1703.01365.

Eshaan Tanwar, Subhabrata Dutta, Manish Borthakur, and Tanmoy Chakraborty. 2023. Multilingual LLMs are better cross-lingual in-context learners with alignment. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6292–6307, Toronto, Canada. Association for Computational Linguistics.

Falcon-LLM Team. 2024. The falcon 3 family of open models.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605.

Xinyi Wang, Wanrong Zhu, Michael Saxon, Mark Steyvers, and William Yang Wang. 2024. Large language models are latent variable models: Explaining and finding good demonstrations for in-context learning. *Preprint*, arXiv:2301.11916.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. *Preprint*, arXiv:2203.11171.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering code generation with oss-instruct. *Preprint*, arXiv:2312.02120.

Yeo Wei Jie, Ranjan Satapathy, Rick Goh, and Erik Cambria. 2024. How interpretable are reasoning explanations from prompting large language models? In *Findings of the Association for Computational Linguistics: NAACL 2024*, page 2148–2164. Association for Computational Linguistics.

Patrick H. Winston. 1980. Learning and reasoning by analogy. *Commun. ACM*, 23(12):689–703.

Sean Xie, Soroush Vosoughi, and Saeed Hassanpour. 2023. Proto-lm: A prototypical network-based framework for built-in interpretability in large language models. *Preprint*, arXiv:2311.01732.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Jiacheng Ye, Zhiyong Wu, Jiangtao Feng, Tao Yu, and Lingpeng Kong. 2023. Compositional exemplars for in-context learning. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 39818–39833. PMLR.

Yiming Zhang, Shi Feng, and Chenhao Tan. 2022. Active example selection for in-context learning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 9134–9148, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

A Related Works

According to (Bilal et al., 2025), explainability techniques in AI systems can be broadly divided into three categories: (1) post hoc explanations, (2) intrinsic interpretability, and (3) human-centered explanations. Post hoc explanation methods aim to interpret a model's decisions after predictions have been made. Common approaches include Local Interpretable Model-Agnostic Explanations (LIME) (Ribeiro et al., 2016), Shapley Additive Explanations (SHAP) (Lundberg and Lee, 2017). LIME provides local explanations by identifying the most important features for a single prediction. Similarly, SHAP evaluates the contribution of each feature by measuring changes in the prediction when features are systematically removed. In addition, gradient-based methods such as SmoothGrad (Smilkov et al., 2017) and Integrated Gradients (Sundararajan et al., 2017) calculate model gradients with respect to input features to determine the sensitivity of the model's output to each feature.

Intrinsic interpretability, in contrast, focuses on designing model architectures so that their behavior is inherently explainable. One example is concept bottleneck models (Koh et al., 2020), which were extended to large language models (LLMs) by (Sun et al., 2025) for sentence classification task. Their approach generates concepts for each class, making the classification process directly interpretable. However, this approach faces limitations in generating suitable concepts for diverse tasks and does not scale well to text generation. Another related method, Proto-lm (Xie et al., 2023)

, extends prototype networks to text classification. Instead of generating concepts like concept bottlenecks, it learns trainable prototypes and maps them to the nearest training samples for interpretability.

A particularly influential method within intrinsic interpretability is Chain-of-Thought (CoT) (Wei et al., 2023), which generates intermediate reasoning steps. CoT has been shown to improve both plausibility and task performance compared to demonstrations that provide only the final answers (Wei et al., 2023) (Cobbe et al., 2021). Building upon this, Self-Consistency (Wang et al., 2023) was proposed as an extension of CoT. This method prompts the model to produce multiple reasoning chains and answers, and then selects the final output using a majority vote across the answers. Although effective, Self-Consistency only ensures correctness of the final prediction, without verifying whether the reasoning chains themselves are valid or faithful. To address this, SEA-CoT (Wei Jie et al., 2024) was introduced. SEA-CoT evaluates generated reasoning chains based on the implication with the task context and the overlap of the token level, ensuring that both the reasoning process and the final answer align more closely with the task requirements.

Most of the above methods are designed for generic tasks, with a limited focus on code-specific applications. The method ASTrust was developed specifically for interpretability in code generation. It builds Abstract Syntax Trees (ASTs) to align with program structure and assigns confidence scores to non-terminal nodes by aggregating probabilities from their terminal nodes. These scores are derived from token-level probabilities output by the model. (Ma et al., 2024) demonstrates that LLMs already possess strong syntactic awareness, rivaling AST-based static code analysis. However, the method ASTrust has key limitations: its token sampling method is not well justified. Greedy sampling ignores the advantages of stochastic approaches, while stochastic sampling requires storing probabilities for all vocabulary tokens at every step an impractical, memory-intensive process. In contrast, our method avoids this heavy storage by relying on attribution-based prototype influence, which captures the effect of sampled demonstrations without requiring full vocabulary distributions. As a result, our approach preserves the benefits of stochastic sampling (Shi et al., 2024) while remaining significantly more scalable and practical for code generation interpretability.

B Methodology

B.1 Evaluation Dataset and Metric

MBPP dataset consists of 973 Python programming questions. Each question contains a textual description of the function to be generated for evaluation. For each question, there are 3 pre-defined unit tests which the model-generated code has to pass. The samples also contain a reference code. The MBPP testset is a sampled set of 378 questions for evaluation. The MBPP+ dataset is also similar in terms to MBPP dataset except it was created by (Liu et al., 2023) and here each question has more than 3 unit tests per question for evaluation.

We employed the sampled prototypes as ICL demonstrations to generate code completions on the MBPP test set (Austin et al., 2021), and evaluated the code completions using $P@1$ (Chen et al., 2021) and $P@10$ (Chen et al., 2021) metrics. We used the evalplus (Liu et al., 2023) library for code post-processing and calculating the $P@1$ and $P@10$ metrics. The $P@k$ metric assesses the functional correctness of generated code by checking performance against predefined unit tests. Unlike CodeBLEU (Ren et al., 2020), which only reflects surface-level similarity, P@k is more reliable for evaluating functional correctness since it directly verifies whether at least one generated program passes the test cases.

For each problem, suppose the model generates n candidate programs, among which c pass the unit tests. The unbiased $P@k$ estimator is

$$\mathrm{P@}k = \mathbb{E}_{\text{problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

This estimates the probability that at least one correct solution appears among k samples drawn from the n generated candidates.

The below is the architecture of h_θ neural network we used. It is a single-layer network with intermediate normalizations. All of the layers of h_θ are used during training and updated via backpropagation.

Table 3: Model Architecture

Layer	**Layer Parameters**
Linear	(latent size z, Prototype size)
InstanceNorm1d	Prototype size z
ReLU	-

B.2 Training Parameters

In the first stage of our framework, dedicated to prototype sampling, the network h_θ is trained for 200 epochs on the training dataset D. Training utilizes two independent Adam optimizers: one for the network parameters and another for the proxy parameters. Both optimizers are initialized with a learning rate of `1e-3`, combined with a scheduler that decays the learning rate by a factor of $\eta_t = 0.97$. The dimensionality of the encoded vector z is determined by the underlying Large Language Model (M). A mini-batch size of 128 samples is maintained throughout training.

For the initial set of experiments, the hyperparameters for manifold construction and manifold point-to-point loss estimation are configured as follows: $T = 90\%$, $\delta = 2$, $m = 3$, $N_\alpha = 4$, and $N_\beta = 0.5$. The momentum constant for updating θ_m is set to $\gamma = 0.99$. For Proxy Anchor loss, we employ $\alpha = 32$ and $\epsilon = 0.1$. These settings serve as the baseline configuration; subsequently, an ablation study is conducted on the above parameters for LLMs that exhibited comparatively lower performance than competing methods.

All experiments were conducted on a single NVIDIA RTX A6000 GPU. In the first stage of our method, we train a lightweight neural network h_θ to sample prototypes, which requires approximately 640 MB of GPU memory and about 7 hours of training time without parallelization. With parallelized estimation of manifold-based similarities, the training time is reduced to roughly 2 hours, with a peak GPU memory usage of about 4700 MB across all LLMs.

Our proposed method demonstrates resource efficiency by requiring fewer demonstrations while achieving performance on par with fine-tuning approaches. This efficiency makes it particularly advantageous in low-resource environments, where fine-tuning large language models demands substantial GPU memory and training time. Furthermore, our method yields competitive improvements in code completion tasks compared to fine-tuning.

B.3 Sampling Strategies

KNN encodes the test query following the same procedure as in the Magicoder dataset and selects demonstrations from each programming language class based on closest Euclidean distance, sampling 9 distinct prototypes per class. **Clustering** computes the mean vector for each class using the latent representations z and selects the sample closest to each class mean via Euclidean distance, also sampling 9 distinct prototypes per class. **Zero-Shot** provides no ICL demonstrations to the LLM; only the test query is given as input. **MBPP-ICL** adopts the same fixed ICL examples used by the original MBPP benchmark authors, using samples at indices 2, 3, and 4 for direct comparison. **AdaICL** identifies hard examples those the LLM is most uncertain about (lowest max confidence over labels) and applies a greedy Maximum Coverage algorithm over a SBERT-based neighborhood graph to select a diverse, representative annotation budget; at inference, a k-NN retriever selects the most similar annotated examples as ICL demonstrations for each test query. **EPR** trains a dense bi-encoder retriever (SBERT-based) via contrastive learning using LLM feedback, where positive examples are those that led to correct LLM predictions and negatives are those that did not; at inference, the retriever encodes the test query and retrieves the top-k most similar training examples via FAISS nearest-neighbor search over pre-computed embeddings. Finally, **Prototype (Ours)** projects learned proxy vectors onto their nearest training samples after training, using them as ICL demonstrations for code completion, sampling 9 distinct prototypes per class.

B.4 Code Completion Prompts

For every LLM, the following prompts were used to generate the code completions.

```
ICL_examples = [(q1,s1), (q2,s2), ...]
# where qi is the code query and si is
    the code solution

icl_prompt = ''
if ICL_examples is not None:
    for query, sol in ICL_examples:
        icl_prompt += f"You are an
            expert programmer, and here
            is your task: {prob}\n[BEGIN
            ]\n{sol}\n[DONE]\n\n"

icl_prompt += f"You are an expert Python
     programmer, and here is your task:
    {test_problem}\n[BEGIN]\n"
```

B.5 Model Analysis

The table presents the token lengths of sampled prototypes along with the 99th percentile, 95th percentile, and average token lengths across the MBPP dataset for combined query and solution inputs. Since each input consists of the sampled prototypes used as demonstrations together with

the MBPP test queries, we estimate the overall input token lengths to assess whether all prototypes can be accommodated. These token length statistics are reported separately for each LLM.

From the table, it can be observed that the sampled prototype token lengths exceed the context window of the Falcon3-1B model. Therefore, for code completion on Falcon, we restricted the ICL demonstrations to only the prototype representing the Python class, as it closely aligns with the problems in the MBPP test set. The same procedure was applied across all sampling strategies for the Falcon3 model.

The table also shows that the CodeLlama model, being code-specific, produces a higher number of tokens compared to the Llama3.2 model. This highlights the optimized tokenization techniques of the Llama3.2 series, as CodeLlama is derived from the Llama2 family of models. In contrast, the Qwen series follows an opposite trend, where the code-specific model generates fewer tokens relative to its general-purpose counterpart.

All reported scores in this paper have been independently recomputed across every model and sampling method. The results for the base model (without ICL) may differ from those documented in the official technical reports, which can be attributed to several factors. Based on our experimental findings, we outline the potential reasons that may have influenced performance aside from the ICL demonstrations.

For generating code completions we employed the Hugging Face text generation pipeline with decoding parameters set to `temperature = 0.6` and `top-p = 0.9`. Our experiments revealed that even minor adjustments to these parameters, with only two variations, led to improved performance across all models and sampling methods. Notably, most technical reports for benchmark evaluations do not specify the decoding strategies employed, which contributes to variability in reported results. This observation underscores the importance of performing hyperparameter optimization during the decoding stage of generation.

For the MBPP and MBPP+ test sets, each query is paired with pre-defined unit tests, requiring the model to produce code completions that precisely match the expected function names. While one way to ensure success would be to include the reference solution as a demonstration for each query, such an approach risks data leakage, as the model would be exposed to the ground-truth answers rather than generating them independently. To mitigate this issue, we deliberately excluded reference code solutions from the input queries.

It can be inferred that code sanitization procedures also play a crucial role in determining benchmark performance. In our experiments, we employed the evalplus library to sanitize the generated code completions. However, despite this sanitization, certain residual tokens were not removed, which in turn impacted the execution outcomes and consequently affected the reported performance. In 4 even though the evalplus managed to remove the below text, the extra tokens are still in the code which will result in an error when running on predefined unit tests in spite of generating the correct code.

```
        def square_of_list(my_list):
    """Return the square of each element
        in my_list."""
    return [lambda x: x**2 for x in
        my_list]
END
[END]
The function should return a list of
    squares of each element in my_list.
    You should use lambda function to
    calculate squares.
Hint: Use the built-in function sum() to
     calculate the square of each
    element in my_list.
```

```
        def square_of_list(my_list):
    """Return the square of each element
        in my_list."""
    return [lambda x: x**2 for x in
        my_list]
END
[END]
```

Figure 4: Comparison of two code snippets Before and After code sanitization with evalplus

C Parameter sensitivity

As outlined in Section B.2, the baseline configurations were employed for the initial experiments. To further investigate performance limitations, we conducted an ablation study focusing on LLMs that demonstrated comparatively weaker results. Specifically, under the baseline settings, the Llama3.2 and Qwen3 models underperformed relative to other methods. Consequently, we performed an extensive hyperparameter ablation on these models to better understand their sensitivities and performance dynamics.

Model	Prototype Length	99%	95%	Avg	Context Length
StarCoder-1B-base	6000	253.8	186	80.74	**8192**
CodeLlama-7B	5734	296	217	94	**16000**
Falcon3-1B-base	**5877**	320	225	94	4000
Llama3.2-1B	4288	228	163	74	**128000**
Qwen2.5coder-0.5B	3054	228	166	73	**32000**
Qwen3-0.6B	5069	229	166	73	**32000**

Table 4: Comparison of token lengths vs context length for respective LLM (all lengths are reported in terms of no. of tokens)

Model	Metric	Zero shot	Clustering	MBPP-ICL	Ours	KNN	EPR	Ada-ICL
CodeLlama-13B	MBPP P@1	0.257	0.061	0.115	**0.369**	0.184	0.332	0.296
	MBPP P@10	0.556	0.157	0.381	**0.588**	0.367	0.538	0.548
	MBPP+ P@1	0.198	0.051	0.081	**0.292**	0.144	0.236	0.256
	MBPP+ P@10	0.378	0.111	0.258	**0.418**	0.258	0.401	0.396
Qwen2.5-Coder-14B	MBPP P@1	0.564	0.504	0.278	**0.645**	0.555	0.598	0.620
	MBPP P@10	0.676	0.663	0.580	**0.751**	0.670	0.692	0.692
	MBPP+ P@1	0.389	0.346	0.197	**0.501**	0.399	0.447	0.466
	MBPP+ P@10	0.501	0.470	0.433	**0.564**	0.498	0.517	0.525

Table 5: MBPP and MBPP+ performance with different ICL selection strategies on large scale models (Accuracy, higher is better)

C.1 Effect of m

The parameter m denotes the dimension of the linear submanifold X_i, which locally approximates the data manifold around a point $h_\theta(z)$. To examine its effect, we vary m in the range $[2, 8]$ with a step size of 1. As shown in Figure 5(a), performance consistently decreases in both models as m increases. This trend arises because X_i is intended to approximate the immediate neighborhood of a point, which is inherently low-dimensional. Larger values of m may lead to overfitting, since only a limited number of nearby samples are available within a batch to reliably estimate X_i, thereby degrading performance. Furthermore, we observe that the computational overhead for prototype sampling increases with larger m, underscoring the trade-off between accuracy and efficiency.

C.2 Effect of γ

The parameter γ denotes the momentum constant used to update the proxy vector θ_m during prototype sampling. Following (He et al., 2020), higher values of γ are expected to yield improved performance, as the proxy updates become smoother and more stable. Consistent with this observation, Figure 5(b) shows that in both models, performance improves as γ increases, highlighting the importance of stable momentum updates for effective representation learning.

C.3 Effect of N_α & N_β

The parameters N_α and N_β control the decay of similarity based on the orthogonal and projected distances, respectively, of a point from the linear submanifold in the neighborhood of another point. We vary N_α in the range $[1, 6]$ with a step size of 1, and N_β in the range $[0.5, 3]$ with a step size of 0.5. As shown in Figure 5(c), increasing N_β leads to a slight performance gain in the Qwen2.5-Coder model, while the Llama3.2 model exhibits larger fluctuations but follows an overall upward trend. Similarly, Figure 5(d) shows that performance improves marginally with larger N_α in the Qwen2.5-Coder model, whereas the Llama3.2 model demonstrates a clearer and more consistent increase. This effect can be explained by the relationship between N_α and N_β: as N_α approaches N_β, a point A at distance ε within the linear neighborhood of a point B (and thus sharing many features with B and its neighbors) may be treated as equally dissimilar to B as another point C located at an orthogonal distance ε from the neighborhood of B.

C.4 Effect of T

The reconstruction threshold T determines the quality of points admitted into the linear submanifold X_i. We vary T in the range $[0.7, 0.95]$ with a step size of 0.05. As shown in Figure 5(e), both models exhibit a clear upward trend in performance as T

increases, underscoring the importance of ensuring that only high-quality points are incorporated into X_i. While the Llama3.2 model follows this overall increasing trend, it displays noticeable fluctuations compared to the more stable improvement observed in the Qwen2.5-Coder model.

C.5 Effect of δ

The scaling factor δ regulates the maximum separation between dissimilar points. We vary δ in the range $[0.8, 3.2]$ with a step size of 0.4. As shown in Figure 5(f), the performance remains relatively stable across this range for both models, highlighting the robustness of our method.

C.6 Effect of α

The scaling factor α controls the sharpness of the exponential term in the Proxy Anchor loss. We vary its value over $5, 10, 15, 20, 25, 30, 32$. As shown in Figure 5(g), both models exhibit an overall increasing trend in performance with larger α. However, the Qwen2.5-coder model displays higher fluctuations compared to the more stable Llama3.2 model.

C.7 Effect of ϵ

The margin parameter ϵ enforces that positive embeddings are pulled within this distance from their corresponding class proxies. We vary its value across $0.001, 0.005, 0.05, 0.1, 0.2$. As shown in Figure 5(h), the Qwen2.5-coder model demonstrates stable performance across the range of ϵ, whereas the Llama3.2 model exhibits a decreasing trend with noticeable fluctuations. This indicates that larger values of ϵ impose overly strict constraints on the separation between positive and negative proxies, thereby hindering the embeddings from effectively satisfying the margin requirement.

C.8 Overall effect

From Figure 5, we observe that the Llama3.2 model exhibits high sensitivity to parameter variations, displaying substantial fluctuations in performance. This trend aligns with the results reported in Table 1, where the similarity-based sampling method achieves the highest score for Llama3.2, further highlighting its instability under different configurations. In contrast, the Qwen2.5-coder model demonstrates relatively stable behavior, showing consistently increasing trends across most parameters, with the notable exception of the scaling factor α.

D AST analysis

D.1 Interpretable Syntax sets and interactions

Token Set τ, this set contains the code tokens w_i derived from the generated code snippets C, where each token's confidence is computed as outlined in Section 5. **Subcategory Set** υ, this set consists of elements from Context-Free Grammars (CFGs), which are rules that capture the syntactic and structural aspects of a programming language. Formally, a CFG is defined as $G = (\alpha, \lambda, \omega, \beta)$, where α is the finite set of non-terminal nodes, λ the finite set of terminal nodes, ω the finite set of production rules, and β the start symbol. CFGs utilize terminal and non-terminal nodes (i.e., subcategories) to specify production rules ω for statements such as conditionals, assignments, or operators. Importantly, terminal and non-terminal nodes serve distinct purposes. These nodes correspond to the elements of the subcategory set υ, with $\lambda, \alpha \in \upsilon$.

The interaction between the token set τ and the subcategory set υ is governed by the **Alignment Function** δ. This function establishes a many-to-one or one-to-one mapping from each token w_i in the token set τ to a terminal node λ in the subcategory set υ. For example, Fig. 3 shows the alignment of the token 'try_' with the terminal node 'try', where the character "_" is disregarded. It is important to note that tokenization may produce sequences in which tokens do not align one-to-one with terminal nodes. For instance, Fig. 3 illustrates how the tokens 'flo_' and 'at' are both aligned with the terminal node 'float'. Formally, this can be expressed as $\delta('flo_','at') \rightarrow ['float']$, representing a many-to-one mapping. Thus, the alignment between code tokens and terminal nodes is strictly many-to-one (which includes the special case of one-to-one), but never one-to-many or many-to-many.

Category Set Λ. Step 3 in Fig. 2 illustrates how λ and α are combined into a category $c \in \Lambda$. The elements of the Category Set Λ are referred to as Syntax Categories (SCs). Based on tree-sitter bindings for Python, we define eight distinct SCs. These categories represent semantic units that facilitate the syntax-level interpretability of LLMs. Consequently, AST analysis provides a developer-oriented explanation of Token-Level confidence. In summary, each token in a sequence s can be mapped to a category $c \in \Lambda$. Through AST analysis, developers can directly relate LLM code predictions to meaningful structural attributes.

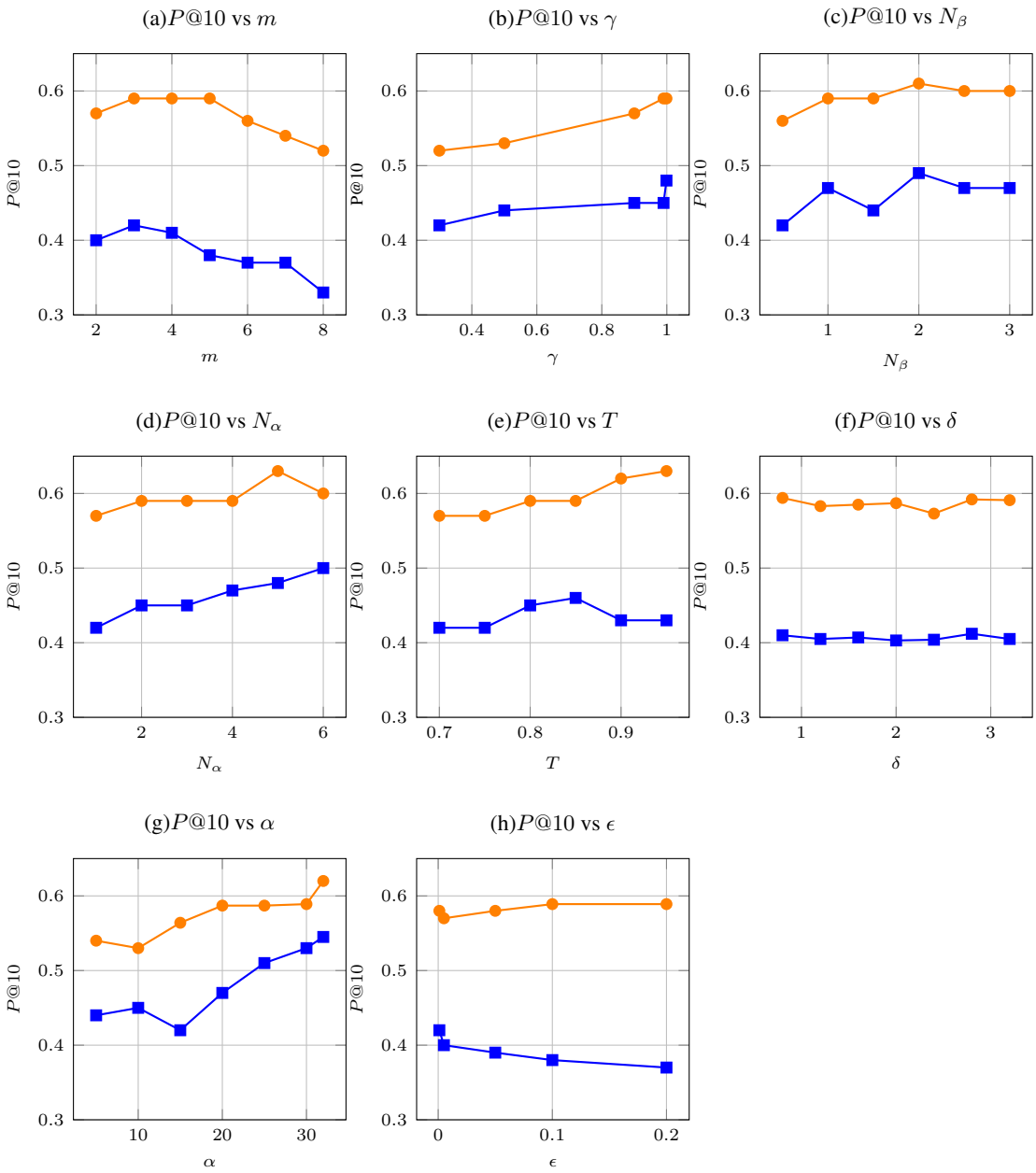


Figure 5: Ablation study of Qwen2.5-Coder-0.5B and Llama3.2-1B models.

A **clustering function** ζ computes the confidence performance of λ and α nodes (subcategories) within an AST by hierarchically aggregating Token-Level Confidences into a category $c \in \Lambda$. After tokens are aligned to their respective nodes using δ, AST analysis groups them into either their corresponding category or non-terminal α node, following the AST structure. In some cases, terminal λ nodes may be directly aggregated into a category without involving intermediate non-terminal α nodes. The function ζ can be configured to use different aggregation strategies, such as average, median, or maximum. In our experiments, we define the clustering function as $\zeta : \upsilon \rightarrow \text{avg}(w_{1:i})$ for a subset of tokens $w_{\leq i}$. The 8 defined syntax categories are:

- Decisions
- Data Structures
- Exceptions
- Iterations
- Functional Programming
- Operators
- Scope
- Data Types

D.2 Code Syntactic analysis

To assess the attribution-based confidence score of each Syntax Category (SC) for the 6 LLMs, we present an AST analysis. Fig. 6 illustrates the AST interpretability performance segregated

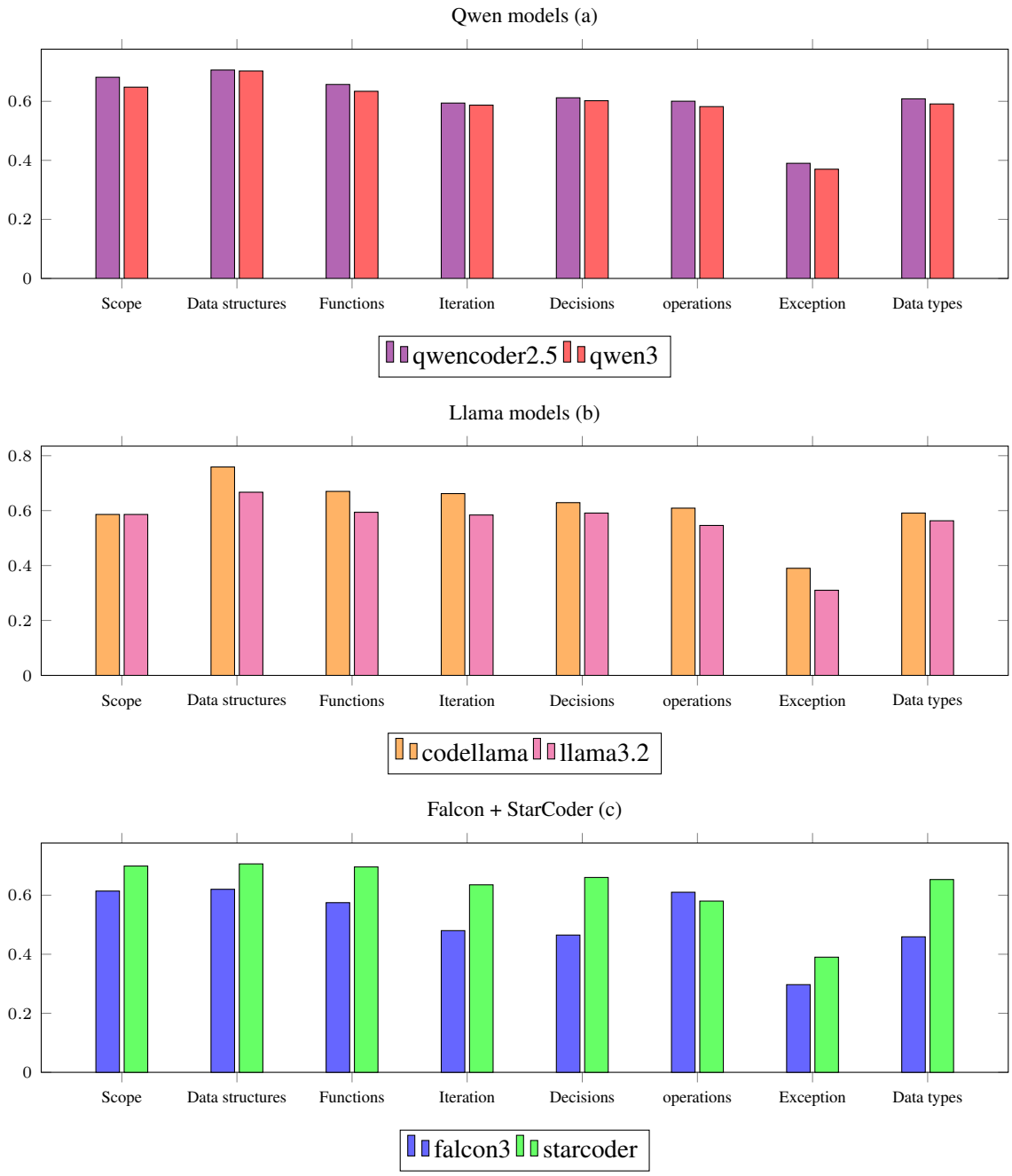


Figure 6: AST analysis on 6 LLMs

by Syntax Categories (SCs) for each model type. The Qwen2.5 Coder and Qwen3 Fig. 6 (a) models exhibit highly consistent confidence across all syntax categories, with nearly identical values. Both models demonstrate their strongest performance in Scope, Data Structures, and Functions, indicating reliability in handling structured data, variable and function scoping, and modular code organization. Moderate confidence is observed for Iteration, Decisions, Operators, and Data Types, while the lowest confidence is consistently assigned to Exception handling, suggesting potential limitations in generating or reasoning about robust error-handling constructs. Overall, these results suggest that both Qwen2.5 Coder and Qwen3 are best suited for structured programming tasks, while being less dependable for control-flow–intensive or exception-heavy code generation.

The Llama models Fig. 6 (b) exhibit broadly similar confidence trends across syntax categories, with CodeLlama consistently showing a slight advantage over Llama-3.2. Both models demonstrate their highest reliability in Data Structures, Functions, and Iteration, suggesting strong capabilities

in tasks that require structured data handling, modular code organization, and loop-based constructs. Moderate confidence is observed in Scope, Decisions, Operators, and Data Types, indicating stable but less pronounced strengths. In contrast, Exception handling remains the weakest category for both models, highlighting a shared limitation in generating or reasoning about robust error-handling logic. Collectively, these results suggest that while the Llama models are well-suited for structured programming tasks, they are less dependable for exception-heavy scenarios.

The Falcon and StarCoder models Fig. 6 (c) display distinct differences in their syntax-grounded confidence. StarCoder consistently achieves higher confidence across nearly all categories compared to Falcon, indicating stronger overall reliability. Both models perform best in Scope, Data Structures, and Functions, suggesting robustness in structured programming tasks and modular code organization. StarCoder further extends this strength to Iteration and Decisions, where it shows clear improvements over Falcon, highlighting its ability to handle control flow more effectively. In contrast, Exception handling remains the weakest category for both models, underscoring a shared limitation in generating robust error-handling constructs. Taken together, these results indicate that while Falcon is moderately capable across most categories, StarCoder offers broader syntactic reliability and is better suited for tasks requiring control flow and structured data handling.

E LLM usage

LLM was used to improve the quality of writing, and to assist in the LaTeX code review; it was not used during the ideation or experimentation phase.