

Software Engineering: Code Quality and Review

Atharva Naik

11-891: Neural Code Generation

<https://cmu-codegen.github.io/f2025/>



Language
Technologies
Institute

Agenda

[CRScore](#): Reference free code review evaluation based on best practices

[CRScore++](#): Generating better code reviews to incorporate best practices

[MetaLint](#): Keeping up with evolving best practices

Code Quality is a Growing Concern ...

In the era of vibe/LLM coding, quality of the written code is a growing concern

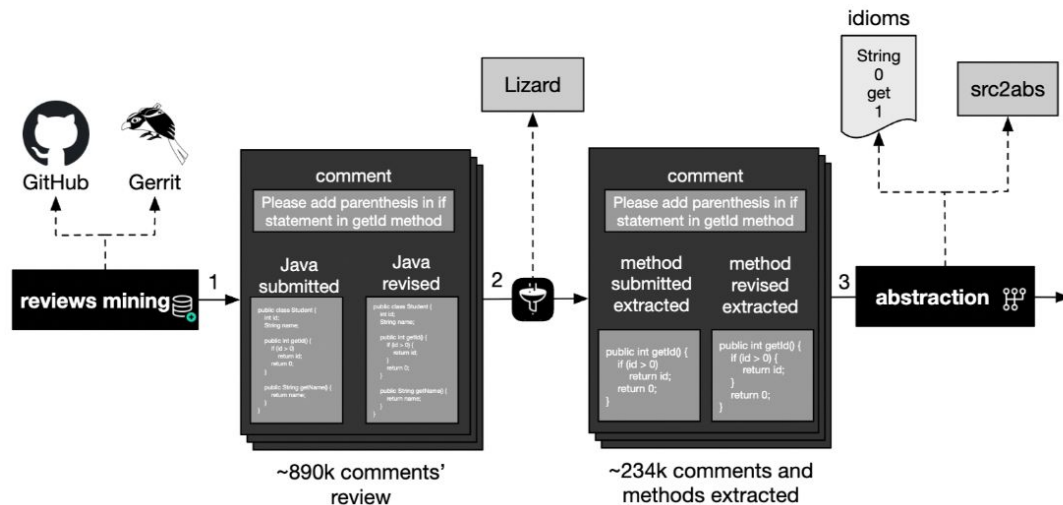
We want code to keep up with

1. Latest code quality standards (e.g. security, efficiency, maintainability)
2. Best practices of the existing codebase.

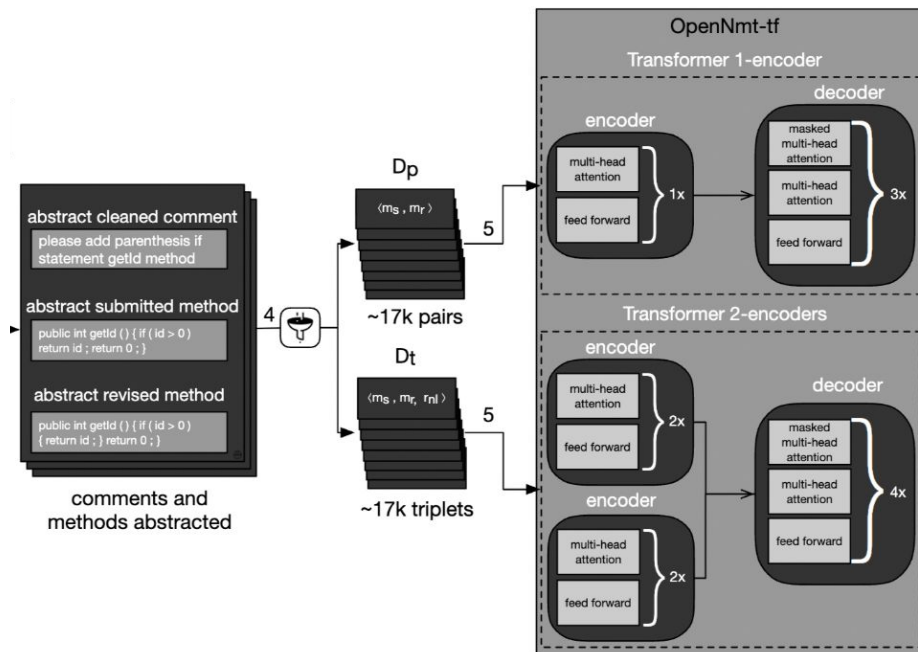


STEPHANIE ARNETT/MIT TECHNOLOGY REVIEW | ADOBE STOCK

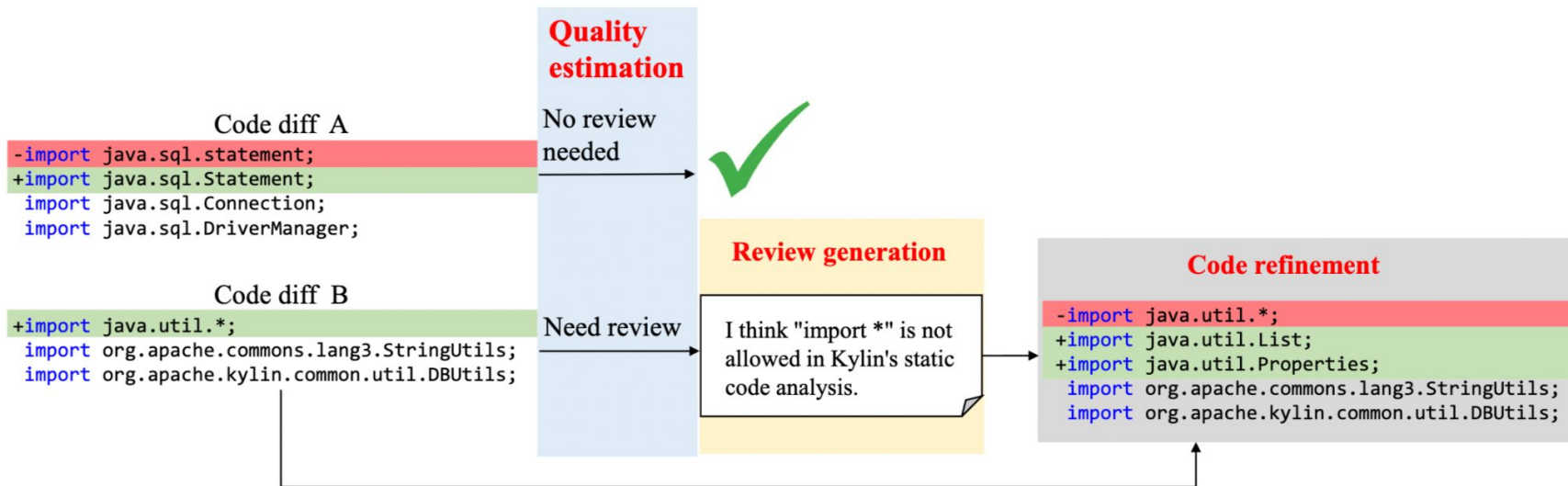
Enforcing Best Practices through Review



Enforcing Best Practices through Review



Enforcing Best Practices through Review



Issues with Training on Mined Reviews

Developer 1 (Atharva) who cares about performance is asked to review code change



Developer 1

```
@@ -133,7 +133,7 @@ namespace OpenTelemetry.Trace
    private void RunGetRequestedDataOtherSampler
    (Activity activity)
    {
        ActivityContext parentContext;
        - if (string.IsNullOrEmpty(activity.ParentId))
        + if (string.IsNullOrEmpty(activity.ParentId)
        || activity.ParentSpanId.ToHexString().Equals(
        "0000000000000000"))
        {
            parentContext = default;
        }
    }
}
```

Code Change

This maynot be a perf issue, if **ToHexString()** is not actually allocating a string, but returns the cached string value. To be confirmed.

Review Comment

Issues with Training on Mined Reviews

Developer 2 (Marcus) who cares about readability is asked to review code change



Developer 2

```
@@ -133,7 +133,7 @@ namespace OpenTelemetry.Trace
    private void RunGetRequestedDataOtherSampler
    (Activity activity)
    {
        ActivityContext parentContext;
        - if (string.IsNullOrEmpty(activity.ParentId))
        + if (string.IsNullOrEmpty(activity.ParentId)
        || activity.ParentSpanId.ToHexString().Equals(
        "0000000000000000"))
        {
            parentContext = default;
        }
    }
}
```

Code Change

The check for `activity.ParentSpanId.ToHexString().Equals("0000000000000000")` seems a bit odd. It might be better to check if the parent id is null or empty.

Review Comment

Issues with Training on Mined Reviews

So who is correct?



Developer 1

This maynot be a perf issue, if ToHexString() is not actually allocating a string, but returns the cached string value. To be confirmed.

Review Comment
(Performance)

VS

The check for **activity.ParentSpanId.ToHexString().Equals("00000000000000000000")** seems a bit odd. It might be better to check if the parent id is null or empty.

Review Comment
(Readability)



Developer 2

Issues with Training on Mined Reviews

So who is correct? **(BOTH!)**



Developer 1

This maynot be a perf issue, if `ToHexString()` is not actually allocating a string, but returns the cached string value. To be confirmed.

Review Comment
(Performance)



VS

The check for `activity.ParentSpanId.ToHexString().Equals("00000000000000000000")` seems a bit odd. It might be better to check if the parent id is null or empty.

Review Comment
(Readability)



Developer 2

Issues with Training on Mined Reviews

But their reviews are very different!



Developer 1

This maynot be a perf issue, if ToHexString() is not actually allocating a string, but returns the cached string value. To be confirmed.

Review Comment
(Performance)



VS

The check for `activity.ParentSpanId.ToHexString().Equals("00000000000000000000")` seems a bit odd. It might be better to check if the parent id is null or empty.

Review Comment
(Readability)



Developer 2

Low BLEU/BERT score with each other!

Issues with Training on Mined Reviews

Now imagine if Developer 1 = “CodeReviewer dataset¹” and Developer 2 = “LLM”



**CodeReviewer
Dataset**

This maynot be a perf issue, if `ToHexString()` is not actually allocating a string, but returns the cached string value. To be confirmed.

**Review Comment
(Performance)**

VS

The check for `activity.ParentSpanId.ToHexString().Equals("0000000000000000")` seems a bit odd. It might be better to check if the parent id is null or empty.

**Review Comment
(Readability)**



LLM

Low BLEU/BERT score with each other!

1: Li, Zhiyu, et al. "Automating code review activities by large-scale pre-training." Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2022.

How can we balance various code quality perspectives?

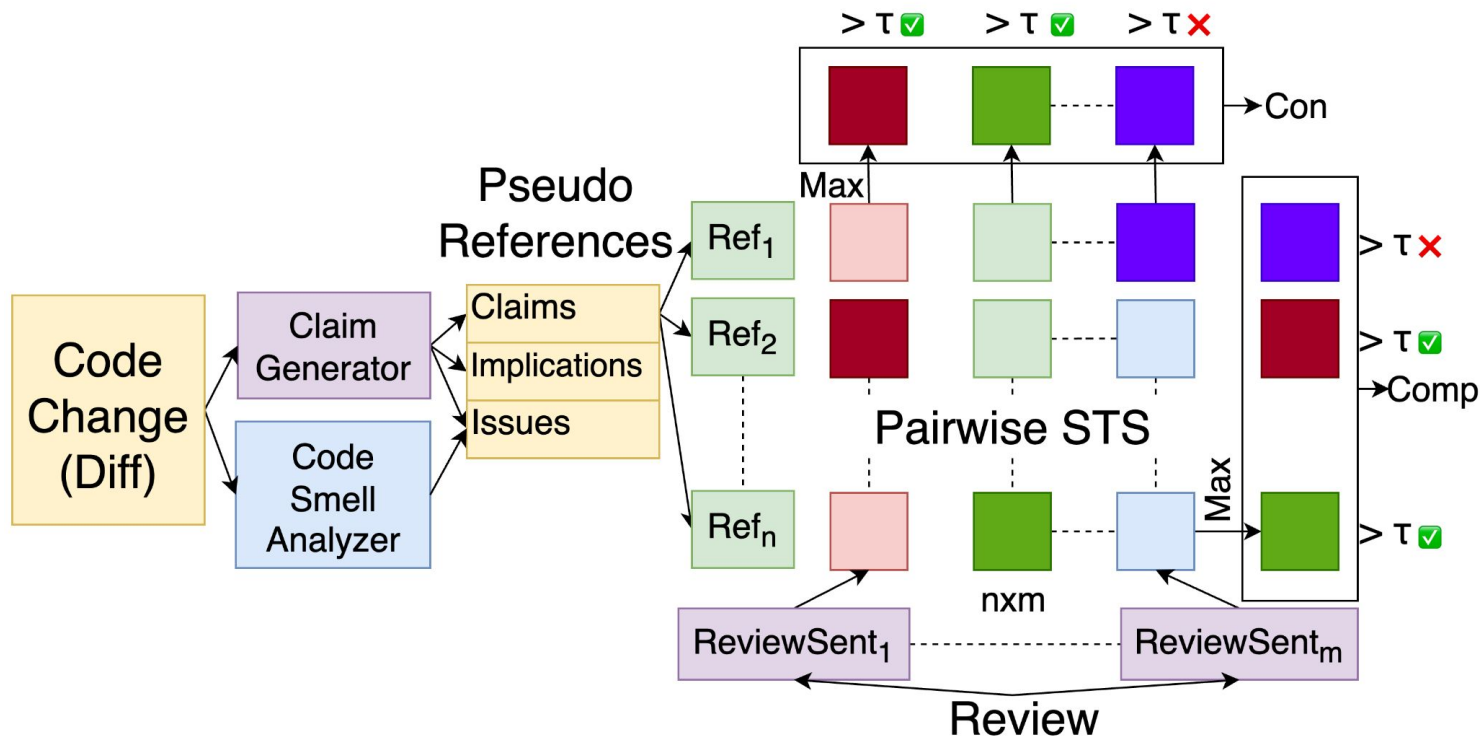
Question to the class

Our Solution: CRScore

A reference free evaluation metric with 3 components:

- **Pseudo-References:** LLM + static analysis tools (code smell detectors) to exhaustively span code quality issues (tackle incomplete references)
- **Quality Dimensions** (based on past work):
 - **Conciseness (Con)** - Does the review only include essential details
 - **Comprehensiveness (Comp)** - Does the review cover all pseudo-references
 - **Relevance (Rel)** - Balances both **Con** & **Comp**
- **Semantic Textual Similarity (STS):** Use semantic embeddings to compute quality dimension scores grounded in pseudo-references

Our Solution: CRScore



Validating CRScore: Systems & Metrics

We evaluated a wide variety of systems and metrics:

1. **Systems:** Simple baselines (retrieval, LSTM), [CodeReviewer-T5](#), LMs (5 open LLMs, GPT-3.5), References from CodeReviewer¹ dataset
2. **Metrics:** 6 Reference based & 2 Reference free (LLM-as-a-judge or LaaJ)

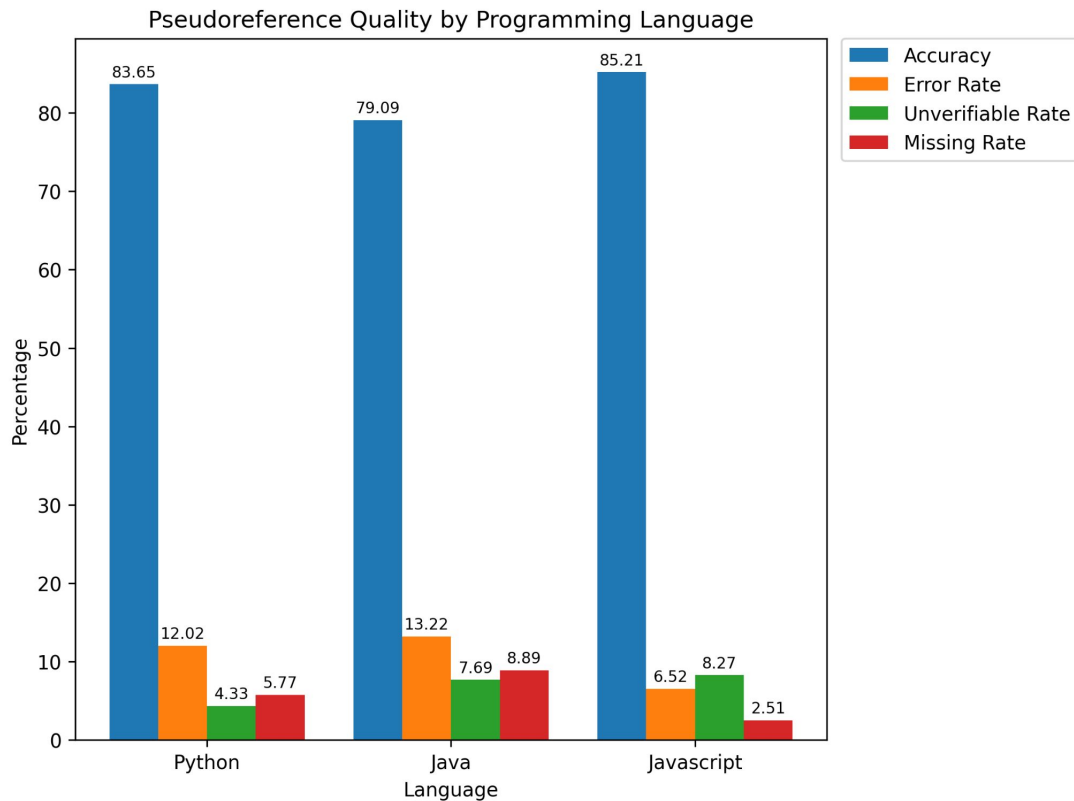
1: Li, Zhiyu, et al. "Automating code review activities by large-scale pre-training." Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2022.

CRScore Results: Pseudo Reference Quality

Asked human annotators to classify pseudo-references as:

- Correct Claims (N_c) - Accuracy
- Incorrect Claims (N_i) - Error Rate
- Unverifiable Claims (N_u) - Unverifiable Rate
- Missing claims (N_a): Annotators were asked to write them - Missing Rate

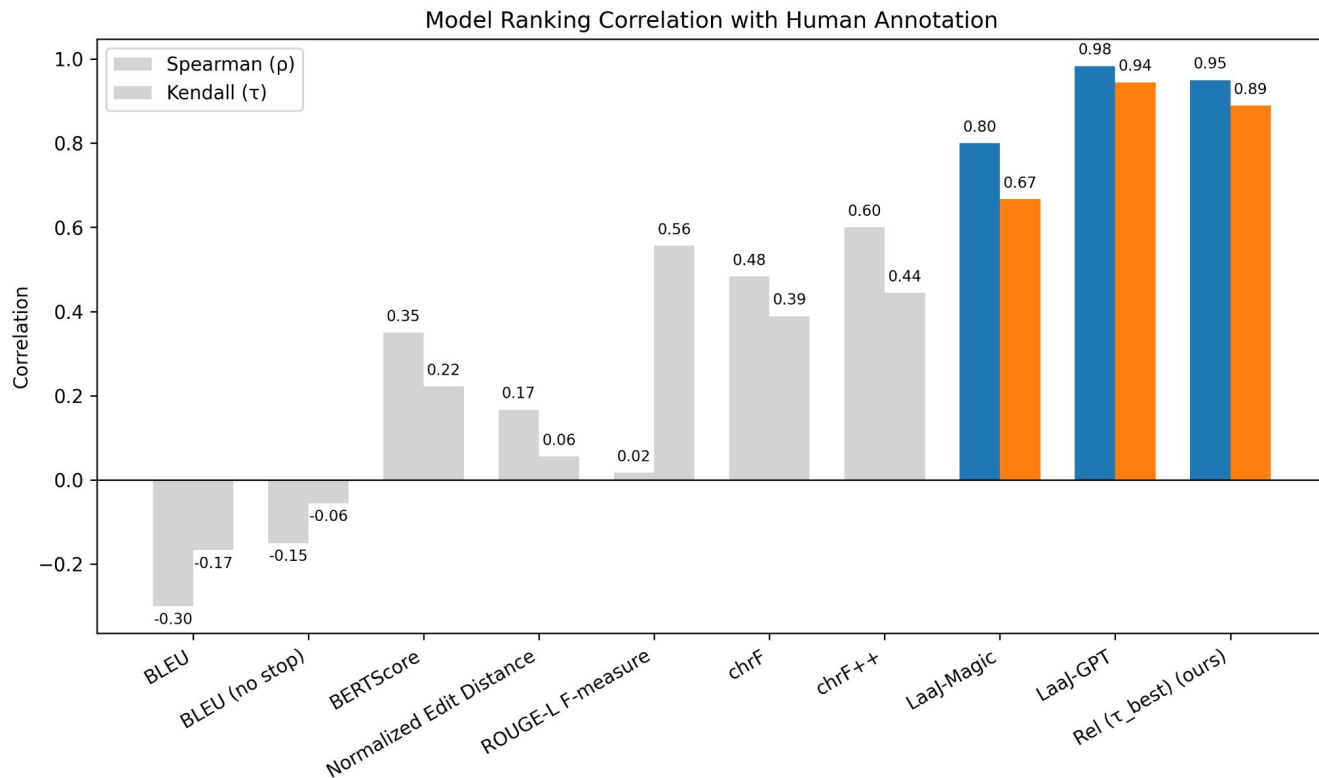
CRScore Results: Pseudo Reference Quality



CRScore Results: Agreement with Human Judgment

- Human annotators are given the corrected pseudo-references from the previous stage
- Annotate reviews generated by 9 models + CodeReviewer “ground truth” reviews for 300 code changes (100 each for Python, Java & Javascript)
- 5 point likert scale for each quality dimension (Con, Comp & Rel)
- Also asked to annotate pseudo-references covered by each review

CRScore Results: Agreement with Human Judgment



If CRSScore is not the best, why should we use it?

Question to the class

CRScore Pros

- Reference free & content based
- **Validity:** 2nd best correlation with human judgment (behind LaaJ-GPT). Performs much better than LaaJ-Magic
- **Interpretable:** Matching between pseudo-references & review sentences to understand scores
- **Scalable:** $O(d)$ LLM calls compared to $O(md)$ for LaaJ (m: #models, d: #datapoints)
- **Efficient:** Less expensive + more environment friendly than LaaJ
- **Reproducibility:** Uses open-weight models only

CRScore Cons

- Only tackles evaluation, but we want to improve review quality!
- Worse than closed source LLM judges like GPT-4o
- Limitations of STS models (like low precision)
- Limited and static set of best practices covered by linters (also limited to specific programming languages)

Discussion Questions

Are there ways to fix the underestimation/overestimation issues mentioned in the Failure Case section? Potentially, we could scale the score by the number of p-ref claims in some way. (James Kim)

What would a temporal variant of CRScore look like? eg. tracking whether a review predicts or prevents future bug-fixes and can such longitudinal outcomes be folded into evaluation? (Nachiket Kotalwar)

Why do the authors extract token embeddings and then pool them? Have they tried other similarity methods, such as embedding entire sentences? (Xiaochuan Li)

CRScore++: Addressing Limitations

- Training to improve code review
- Use closed source models (e.g. GPT-4o-mini) for pseudo-references
- Replace STS models with LLM judges
- Evaluate generalization across programming languages (training on Python, evaluation on Java/Javascript)

CRScore++: Challenges

- How to best combine feedback from LLMs and tools (like CRScore)?
 - Tool feedback is RLVR¹
 - LLM feedback is RLAIIF²
 - Can we unify RLVR and RLAIIF?
- How to adapt to languages for which we may not have tools?
- For RL/DPO how to combine reward/score from LLM and tools (replacing STS with judge)

1: Wang, Yiping, et al. "Reinforcement learning for reasoning in large language models with one training example." *arXiv preprint arXiv:2504.20571* (2025).

2: Lee, Harrison, et al. "Rlaif vs. rlhf: Scaling reinforcement learning from human feedback with ai feedback." *arXiv preprint arXiv:2309.00267* (2023).

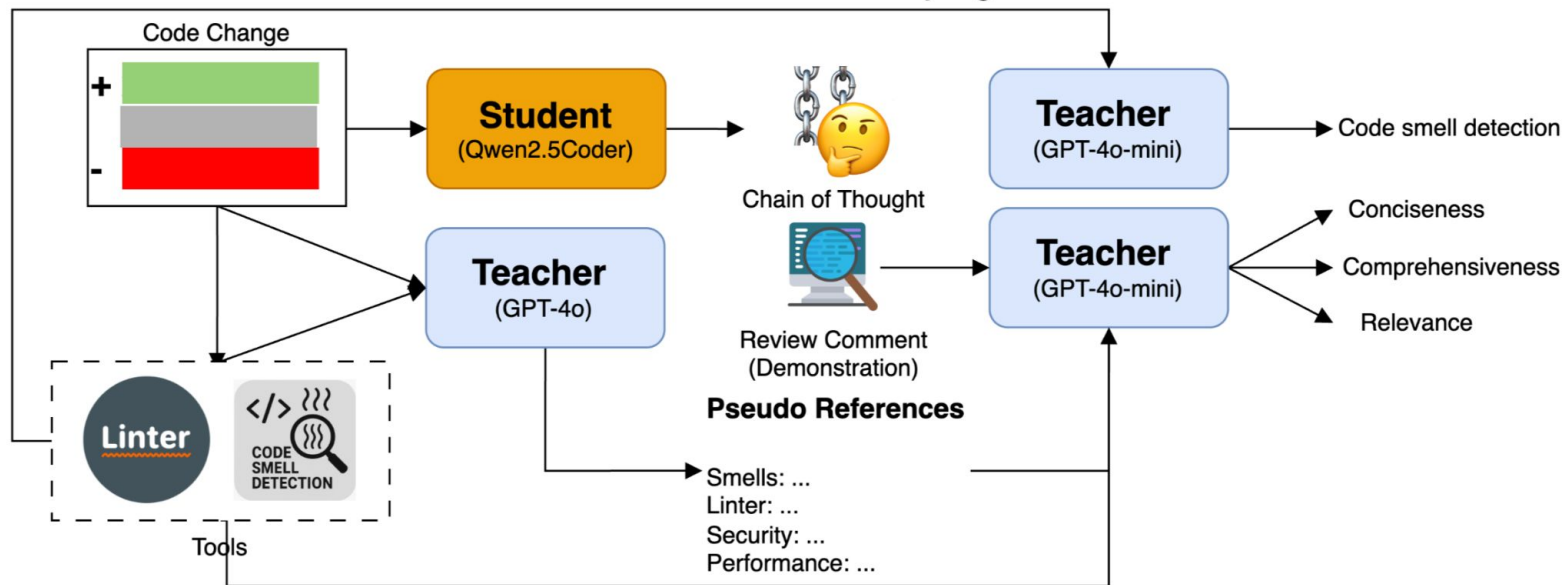
How can we incorporate verifiers for text generation?

Question to the class

CRScore++: Method

CRScore++: Evaluation

Evaluation: CRScore + LLM-as-a-judge



CRScore++: Experimental Setup

Zero Shot: Instruct model is directly asked to review code change

CR Dataset (Python only): SFT on mined **Python** code review comments

Tool Guided: Tool feedback given in context before writing review

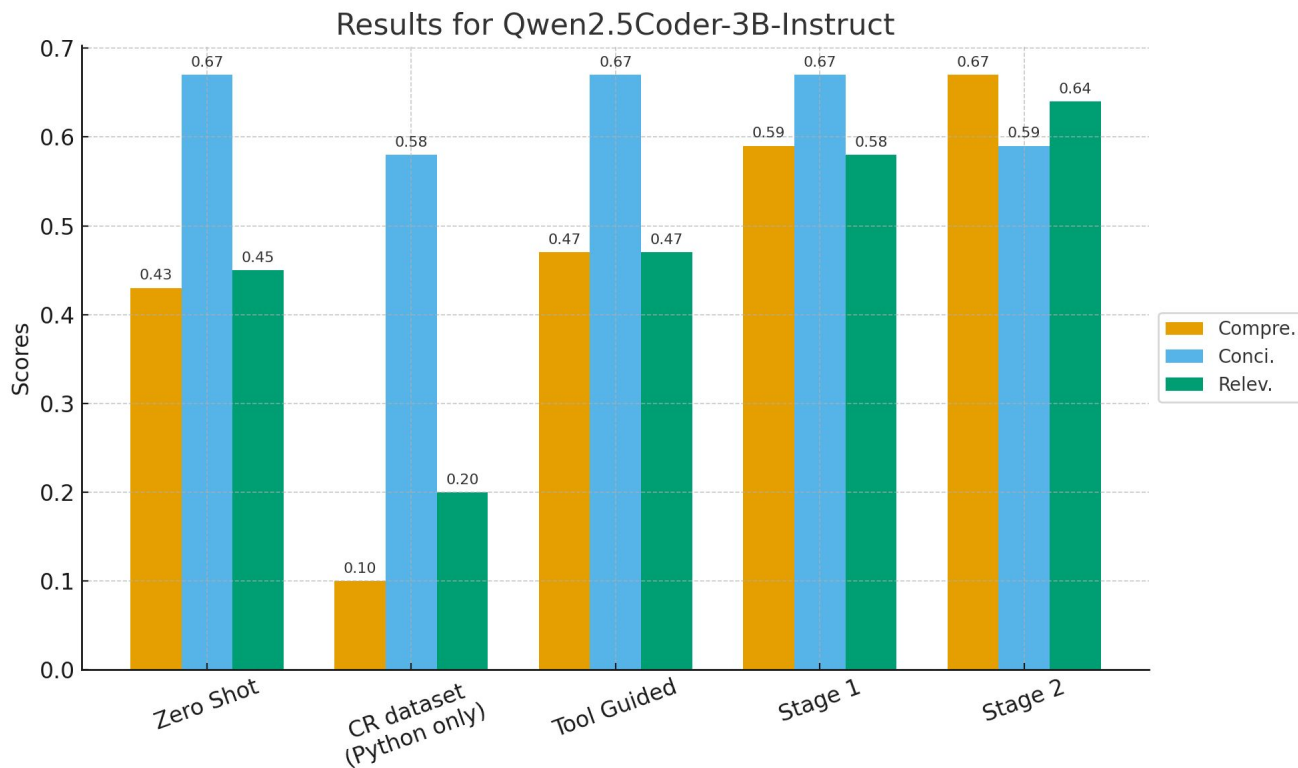
Stage 1: SFT on teacher (GPT-4o-mini) generated CoT + response incorporating tool feedback on **Python**

Stage 2: Generate multiple CoT + responses (**Python**) from Stage 1 model, evaluate with teacher (GPT-4o-mini) and train on contrast pairs (created based on relative reward differences – RS-DPO)

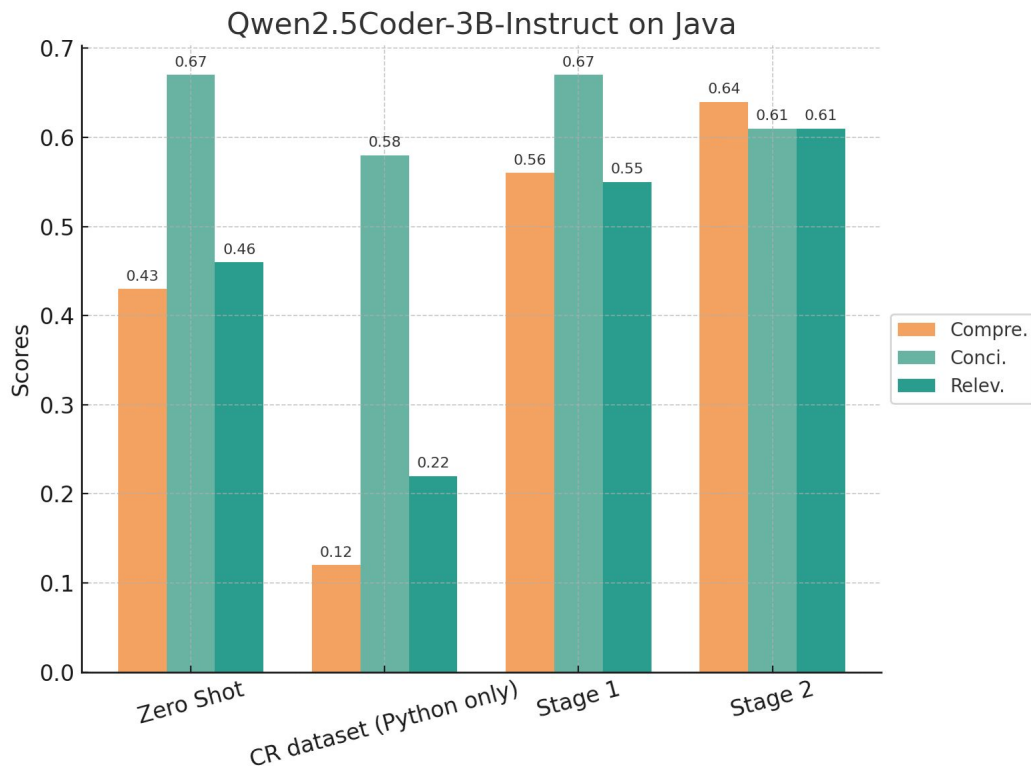
What other baselines could be interesting to cover?

Question to the class

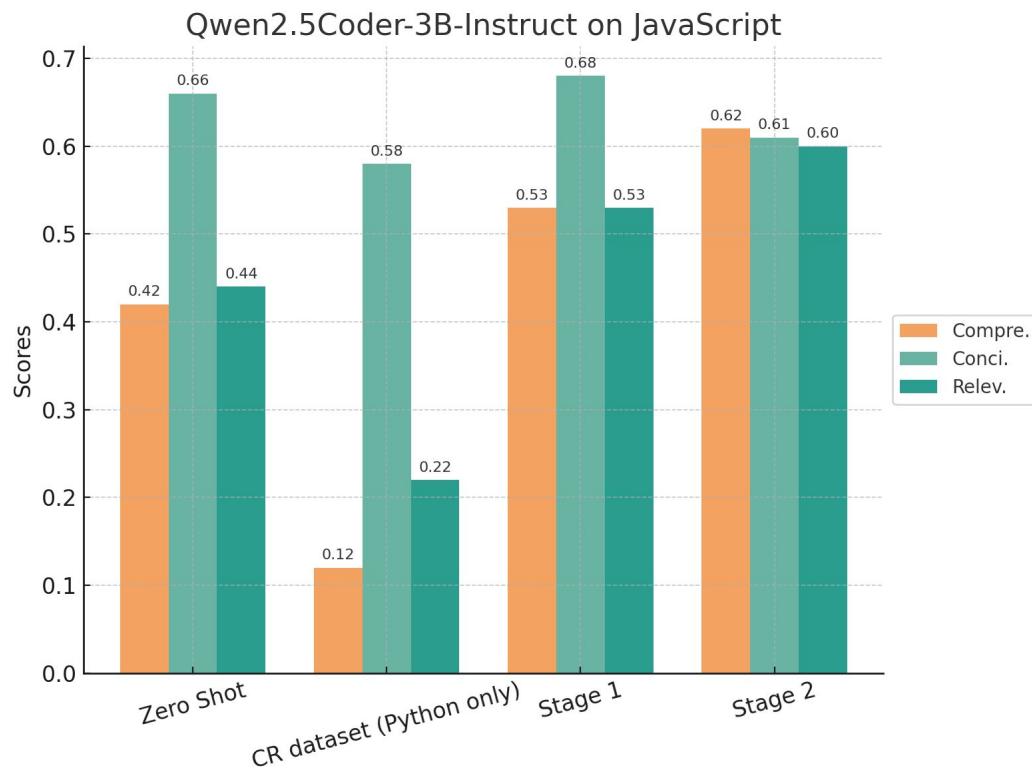
CRScore++: Results (ID)



CRScore++: Results (OOD) Java



CRScore++: Results (OOD) JavaScript



CRScore++: Human Eval

Model	Python			Java			JavaScript		
	Comp.	Conc.	Rel.	Comp.	Conc.	Rel.	Comp.	Conc.	Rel.
3B (Zero Shot)	2.79	3.85	2.77	2.90	3.84	2.99	3.00	3.73	3.18
7B (Zero Shot)	-	-	-	2.97	3.59	2.81	3.21	3.92	3.21
3B (Tool Guided)	3.15	3.53	3.08	2.94	3.61	2.83	2.82	3.52	2.79
7B (Tool Guided)	-	-	-	2.93	4.04	3.05	2.65	3.42	2.82
3B (Stage 1)	<u>3.56</u>	<u>3.71</u>	<u>3.17</u>	3.76	<u>3.88</u>	3.46	<u>3.72</u>	<u>3.81</u>	<u>3.44</u>
7B (Stage 1)	-	-	-	3.45	3.76	3.11	3.39	3.76	3.26
3B (Stage 2)	3.91	3.26	3.19	<u>3.88</u>	3.42	3.19	4.09	3.38	3.51
7B (Stage 2)	-	-	-	4.39	3.18	<u>3.41</u>	4.12	3.24	3.48

Table 4: Human evaluation results across Python, Java, and JavaScript (scores range 1-5, higher is better). Best scores are **bolded** and second-best are underlined.

OOD Generalization of Tool Feedback!

Language	Code Change	Model Type	Model's Code Change Review
Java	<pre>@@ -53,7 +53,7 @@ public class ProtocGopic PluginGeneratorTest { model.GetFiles.stream .mapProtoFile::getProto .collectCollectors.toList() // Only the file to generate .addFileToGenerate"multiple_services.proto" - .setParameter"language=java" + .setParameter"language=java,transport=grpc" .build;</pre>	Qwen 3B Stage 1	<p>I noticed that a new parameter "transport=grpc" has been added to the setParameter() method. This change introduces a potential code smell as it increases the dependency on external function calls. The test may now have a "shotgun smell" due to high coupling with external components. I recommend adding documentation for this parameter.</p>
JavaScript	<pre>@@ -124,6 +124,12 @@ function validateConfigconfig if !config.hasOwnProperty'timeout' { config.timeout = 30000; } + if !config.hasOwnProperty'retries' { + config.retries = 3; + } + if config.retries > 0 { + throw new Error'Retries cannot be negative'; + } return config; }</pre>	Qwen 3B Stage 1	<p>The PR adds retry functionality to the config validation. I see a potential code smell here with the repetitive property checking pattern. Linter would flag this as a violation of the DRY principle. Consider refactoring to use a helper function for property validation. Also, the Error message could be more descriptive for debugging purposes.</p>

Evaluating Accuracy of Tool Use in CoTs

Model Name	Python		Java		JavaScript	
	Avg Accuracy	Avg Coverage	Avg Accuracy	Avg Coverage	Avg Accuracy	Avg Coverage
Qwen 3B (Zero Shot)	2.26	1.55	2.63	1.47	2.68	1.11
Qwen 3B (Stage 1)	3.38	1.90	3.29	1.88	3.72	2.35
Qwen 3B (Stage 2)	2.97	1.62	3.07	1.72	3.02	1.42
Qwen 7B (Zero Shot)	2.61	1.61	3.04	1.60	3.03	1.14
Qwen 7B (Stage 1)	3.59	1.93	3.50	1.84	3.71	2.03
Qwen 7B (Stage 2)	3.29	2.01	3.12	1.44	3.20	1.80

Table 3: Tool usage accuracy and coverage evaluation on Python, Java, and JavaScript using GPT4o-mini as judge.

CRScore++: Takeaways

- CRScore++ improves comprehensiveness and relevance of reviews including coverage of code quality issues
- LLM and Human Eval point to CRScore++ training generalizing across programming languages
 - Tool feedback for one language can teach code smell detection/linting for other languages
- However suffers from length bias
 - Conciseness doesn't improve with stage 1 and drops with stage 2
 - Also too use worsens from stage 1 to stage 2

Discussion Questions

What is the optimal interaction loop between LLMs and humans? Will code review models still be relevant as LLMs becoming better coders? (Andre He)

Flagging Best Practice Violations

```
def test_file_url_not_allowed(self):
    fake_file_url = "fake_image.png"
    with self.assertRaises(ValueError) as ve:
        with self.assertRaisesRegex(
            ValueError, f"File URL not explicitly allowed: {fake_file_url}"
        ):
            web_utils.get_url_to_bytes(
                fake_file_url, colab_debug=False, allow_file_url=False
            )
    self.assertEqual(
        str(ve.exception),
        "File URL not explicitly allowed: " + str(fake_file_url),
    )
```

Reviewer anonymized

Mar 6, 9:01 PM

Consider using `self.assertRaisesRegex` here instead of `self.assertRaises + self.assertEqual` on the exception, as it has the same effect. Note that you can also use partial string matching here if desired.

[python-style-advice#common_exception_message](#)

Author anonymized

Mar 7, 10:53 PM

Done. Good idea!

Limitations of Existing Tools

Traditionally linters or static analyzers have been used to enforce best practices

Can capture formulaic stuff like naming, formatting etc.

Struggle with fuzzier concepts like **code idioms**

Need to be **updated** as languages/best practices evolve (or across different teams)

Focus of our work

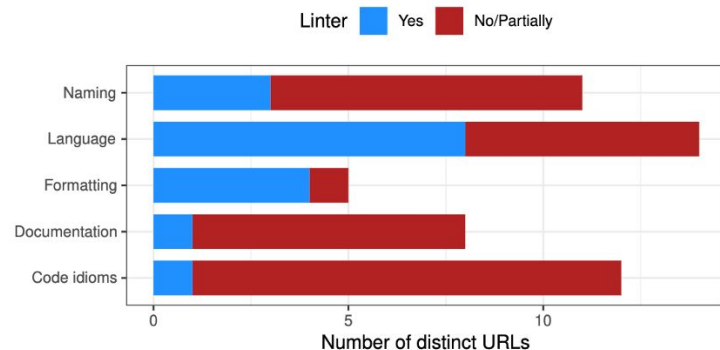


Figure 6: The top-50 most frequently predicted URLs categorized into types. Linter indicates whether a linter that detects a violation exists or can easily be built.

Can we build the next generation of linters w LLMs?

Desiderata:

1. Address complex code quality concerns (e.g. code idioms)
2. Keep up with evolving best practices
3. Use LLM's natural language understanding for low/no code specification of best practices
4. Train on existing linters to improve handling of harder, more complicated issues

What is a “code idiom”?

Programming language specific
conventions/best practices about how to
do “XYZ”



I Am Developer 
@iamdeveloper · [Follow](#)



I bought some milk over the weekend and also picked a new JavaScript framework to use.

At least one of these will be out of date before the week's up.

12:50 PM · Feb 5, 2018



5.7K



Reply



Copy link

[Read 41 replies](#)

Sources of Code Idioms - PEPs

Python Enhancement Proposals (or PEPs) are used to propose:

1. New language features (e.g. modules, syntax, methods, etc.)
2. Updates to language features
3. Changes in behavior of existing features

PEP 506

Code Idiom for password generation with random module: **(BAD/INSECURE)**

```
password = "".join(random.sample(string, length))
```

Code Idiom for password generation with secrets module: **(GOOD/SECURE)**

```
password = "".join(secrets.choice(string) for _ in  
                    range(length))
```

Sources of Code Idioms - JEPs

Java Enhancement Proposals (or JEPs) are Java equivalents of PEPs:

Old Idiom

```
class Point {  
    private final int x;  
    private final int y;  
  
    Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    int x() { return x; }  
    int y() { return y; }  
  
    public boolean equals(Object o) {  
        if (!(o instanceof Point)) return false;  
        Point other = (Point) o;  
        return other.x == x && other.y == y;  
    }  
  
    public int hashCode() {  
        return Objects.hash(x, y);  
    }  
  
    public String toString() {  
        return String.format("Point[x=%d, y=%d]", x, y);  
    }  
}
```

JEP 395: Records

New Idiom

→ record Point(int x, int y) { }

“Transparent carrier of data”:

- Accessor + final field per component
- Canonical constructor
- **equals()** / **hashCode()** by component values
- **toString()** with component names & values

Idioms and Code Quality

Idioms enforce common code quality standards (shared values/culture across dev communities)

Target aspects like security, efficiency, readability, maintainability ...

But standards evolve over time/vary across orgs:

- e.g. **PEP 506**: greater security with **secrets** module for Python ≥ 3.6
- Enforcing latest, community specific standards/values is essential for smooth collaboration!
- We use PEPs/JEPs as proxies for nuanced idioms/best practices over time or across teams

Easy vs Hard Code Idioms


Easy (Rule Based) Idioms

E.g. PEP 563 (Postponed Evaluation of Annotations)

```
from __future__ import annotations
from typing import Optional

class A:
    def __init__(self, b: B) -> None:
        self.b = b

    def get_sibling(self) -> Optional[A]:
        # Even though A isn't fully defined
        # at this point,
        # the annotation is stored as a
        # string, not evaluated.
        return None
```



Always flagged using rule-based analysis

Hard (Nuanced) Idioms

E.g. PEP 506 (Secrets Module)

Idiom Violation poses security risk

```
import random, string
alphabet = string.ascii_letters + string.digits
token = ''.join(random.choice(alphabet) for _ in
range(16))
```

Refactored to follow idiom and mitigate security risk

```
import secrets, string
alphabet = string.ascii_letters + string.digits
secure_token = ''.join(secrets.choice(alphabet)
for _ in range(16))
```

Not an idiom violation (not a security critical scenario)

```
import random
options = ['rock', 'paper', 'scissors']
play = random.choice(options)
```

MetaLint: Method

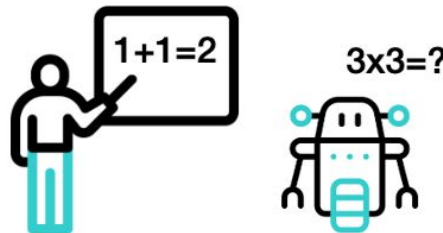
Generalization through Instruction Following:

- Treat each idiom to be analyzed as an instruction-following “meta-task”
- Keep up with evolving conventions by updating instructions at test time
- Based on instruction tuning for cross-task (idiom) generalization¹

Easy-to-Hard Generalization:

- Limit human (linter) supervision to training on easy meta-tasks (idioms)²
- Achieve performance improvement on hard meta-tasks (idioms)²

Our Analogy on Easy-to-Hard Generalization



humans reliably supervise strong models on **easy** tasks and evaluate them on **hard** tasks

1: Wang, Yizhong, et al. "Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks." *arXiv preprint arXiv:2204.07705* (2022).

2: Sun, Zhiqing, et al. "Easy-to-hard generalization: Scalable alignment beyond human supervision." *Advances in Neural Information Processing Systems* 37 (2024): 51118-51168.

MetaLint: Generalization Through Instruction Following

MetaLint: Easy-to-Hard Generalization

How should we evaluate code idiom flagging?

Question to the class

What kind of transfer/generalization should we evaluate?

Question to the class

MetaLint: Experiments and Metrics

2 tasks:

Detection (Coarse): Binary prediction if there is idiom violation (macro P, R, F)

Localization (Fine): Flagging specific lines as violations (set based P, R, F)

Experiments:

Generalization through Instruction Following:

Python synthetic data (Ruff Linter): In Domain, Near Transfer, Far Transfer

Java synthetic data (PMD, JEP Tree-Sitter): In Domain (e.g. PMD→PMD), Transfer (e.g. PMD→JEP)

Easy-to-Hard Generalization:

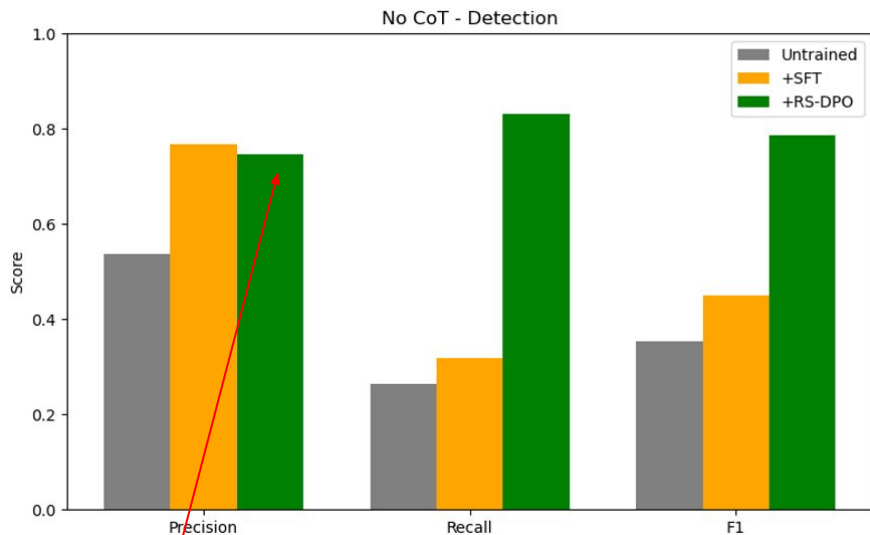
Ruff (Python synthetic data) → PEP (Human curated hard idiom benchmark)

Generalization through Instruction Following

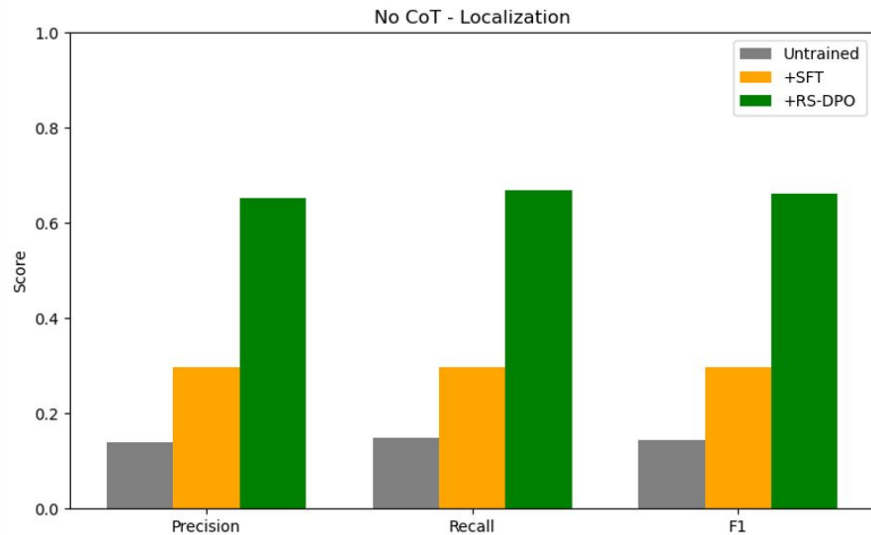
All results on synthetic datasets

MetaLint: Generalization through Instruction Following (Ruff Idioms)

Qwen3-4B trained with MetaLint (non-reasoning setting)



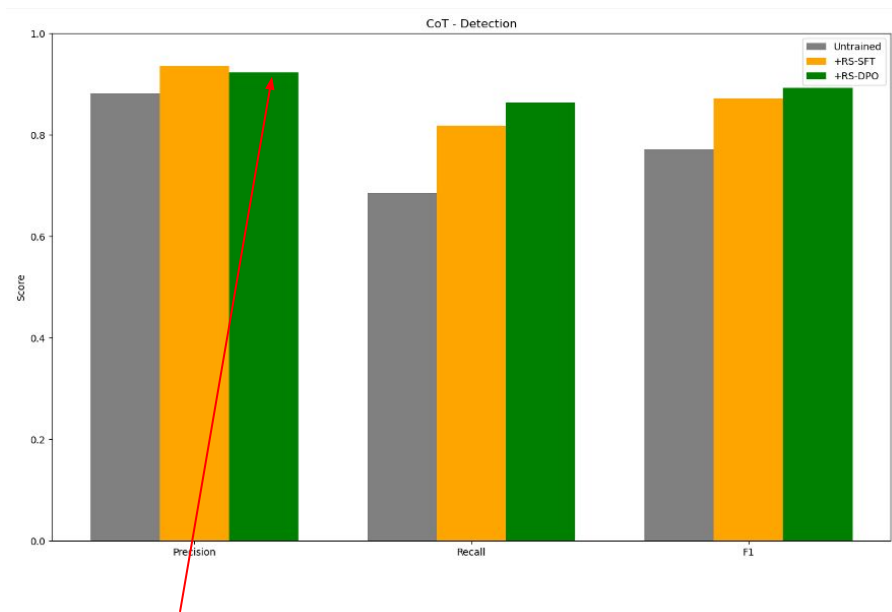
Slight Drop in Precision



Drastic improvements over base model for both det. & loc.

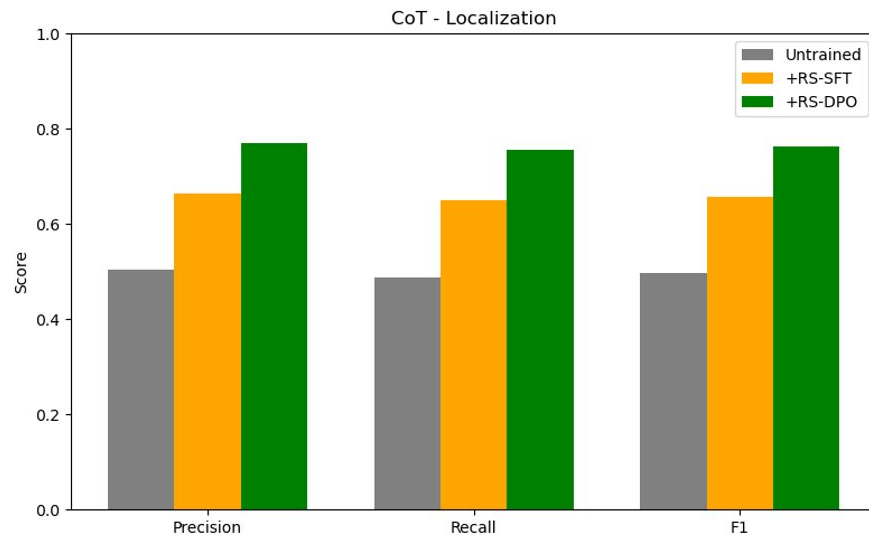
MetaLint: Generalization through Instruction Following (Ruff Idioms)

Qwen3-4B trained with MetaLint (reasoning setting)



Slight Drop in Precision

Base model has strong det. performance



Minor improvements in det., major improvements in loc.

MetaLint: Generalization through Instruction Following (Ruff Idioms)

Model	In-Domain			Near Transfer			Far Transfer		
	P_{Det}	R_{Det}	F_{Det}	P_{Det}	R_{Det}	F_{Det}	P_{Det}	R_{Det}	F_{Det}
Qwen3-4B	0.45	0.14	0.22	0.58	0.24	0.34	0.54	0.29	0.38
+SFT	0.93 (+0.48)	0.74 (+0.6)	0.83 (+0.61)	0.89 (+0.31)	0.24 (+0)	0.38 (+0.04)	0.72 (+0.18)	0.27 (-0.02)	0.39 (+0.01)
+RS-DPO	0.72 (+0.27)	1 (+0.86)	0.83 (+0.61)	0.76 (+0.18)	0.8 (+0.56)	0.78 (+0.44)	0.75 (+0.21)	0.81 (+0.52)	0.78 (+0.4)
Qwen3-4B w CoT	0.87	0.5	0.63	0.95	0.88	0.91	0.87	0.68	0.76
+RS-SFT	0.87 (+0)	0.73 (+0.23)	0.8 (+0.17)	0.97 (+0.02)	0.86 (-0.02)	0.91 (+0)	0.94 (+0.07)	0.82 (+0.14)	0.88 (+0.12)
+RS-DPO	0.86 (-0.1)	0.85 (+0.35)	0.85 (+0.22)	0.97 (+0.02)	0.92 (+0.04)	0.94 (+0.03)	0.92 (+0.05)	0.86 (+0.18)	0.89 (+0.13)
Llama3.2-3B-Instruct	0.54	0.43	0.48	0.69	0.68	0.69	0.47	0.51	0.49
+SFT	0.88 (+0.34)	0.87 (+0.44)	0.88 (+0.4)	0.89 (+0.2)	0.44 (-0.24)	0.59 (-0.1)	0.61 (+0.14)	0.27 (-0.24)	0.37 (-0.12)
+RS-DPO	0.75 (+0.21)	0.92 (+0.49)	0.83 (+0.35)	0.81 (+0.12)	0.71 (+0.03)	0.76 (+0.07)	0.61 (+0.14)	0.59 (+0.08)	0.60 (+0.11)

SFT helps more for “in-domain” performance (can hurt transfer)

DPO helps more for “transfer” performance (can marginally hurt in-domain)

MetaLint: DPO NV Fraction Ablations

- NO VIOLATION (NV) data
prevents false positives (PEP violations are rare in natural code distributions)
- Proportion of NV instances in DPO heavily affects det. & loc.
- **High fraction of NV:** high P_{Det} , low R_{Det} , low loc. metrics
- **Low fraction of NV:** lower P_{Det} , high R_{Det} , high loc. metrics (more favorable)

Fraction of NV data	Detection			Localization		
	P_{Det}	R_{Det}	F_{Det}	P_{Loc}	R_{Loc}	F_{Loc}
0%	0.6268	0.9577	0.7577	0.6777	0.6932	0.6854
2%	0.671	0.9128	0.7734	0.6681	0.6812	0.6746
5%	0.7469	0.8315	0.7869	0.6527	0.6696	0.6611
10%	0.7584	0.8114	0.784	0.6263	0.6474	0.6367
20%	0.8382	0.7227	0.7762	0.5721	0.5815	0.5768
40%	0.8683	0.5618	0.6822	0.4683	0.4735	0.4709
100%	0.8565	0.4152	0.5593	0.4041	0.4056	0.4048

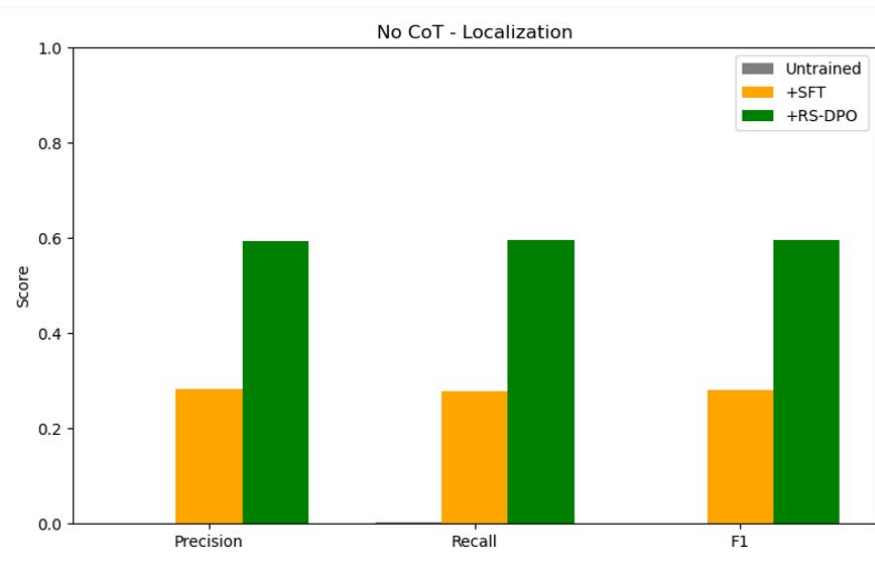
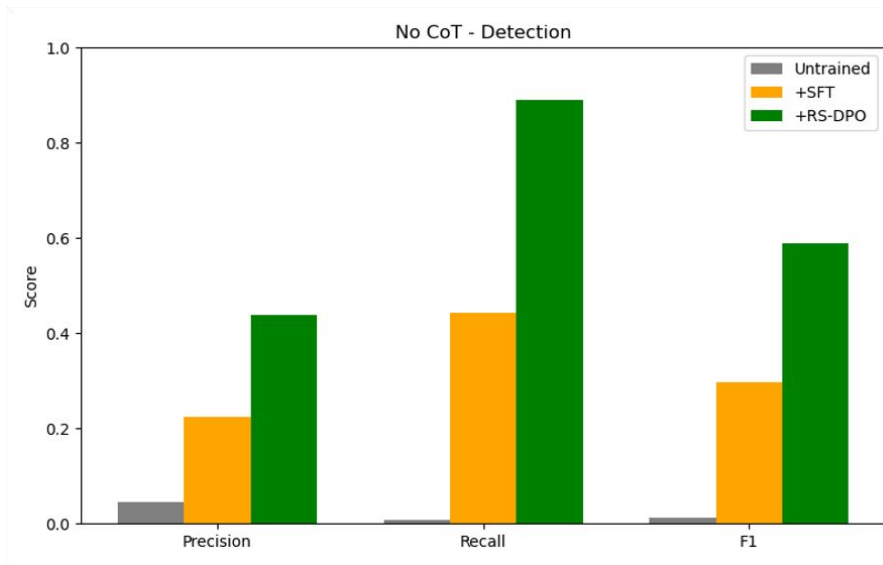
Why do you think NV instances reduce performance?

Question to the class

MetaLint: Generalization through Instruction Following (PMD Idioms)

Llama3.2-3B (non-reasoning model) trained with MetaLint-Java (PMD)

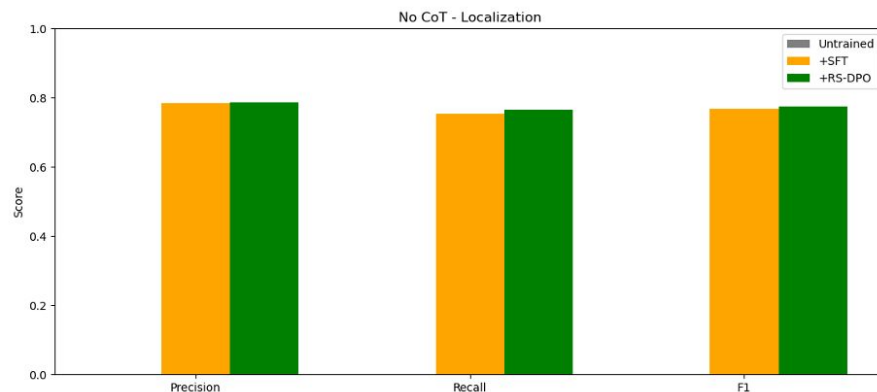
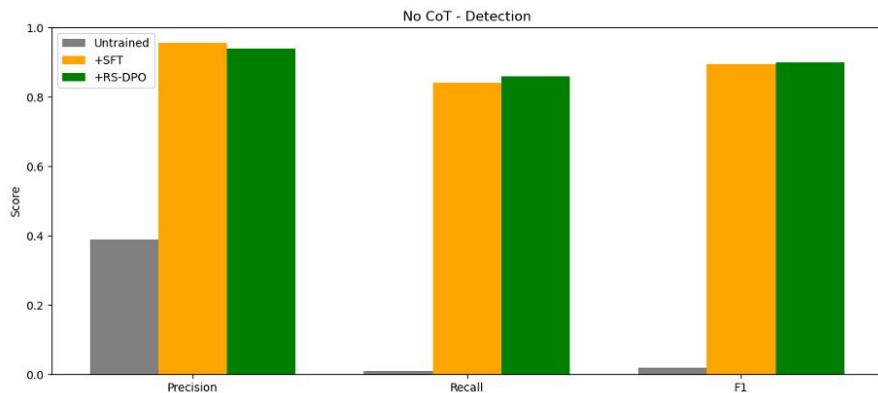
In Domain training and testing on PMD synthetic data



MetaLint: Generalization through Instruction Following (JEP Idioms)

Llama3.2-3B (non-reasoning model) trained with MetaLint-Java (JEP)

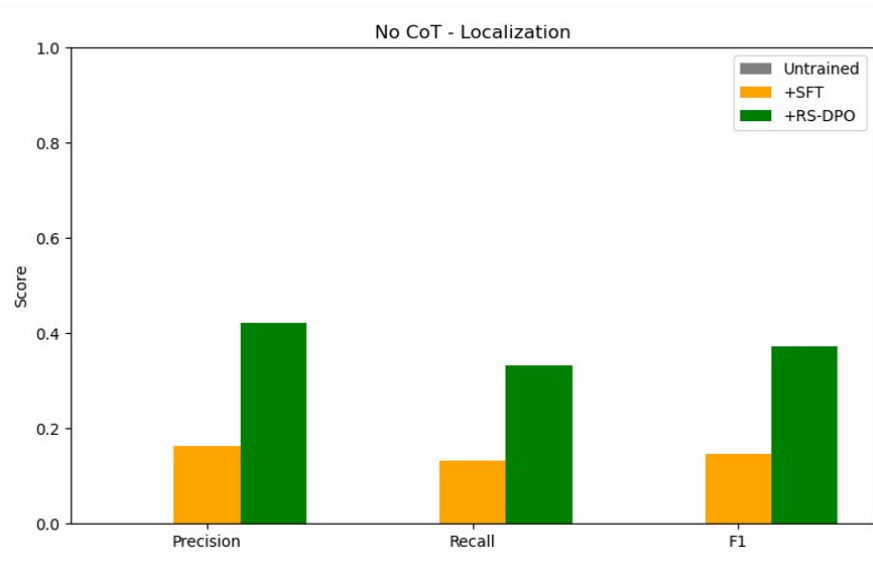
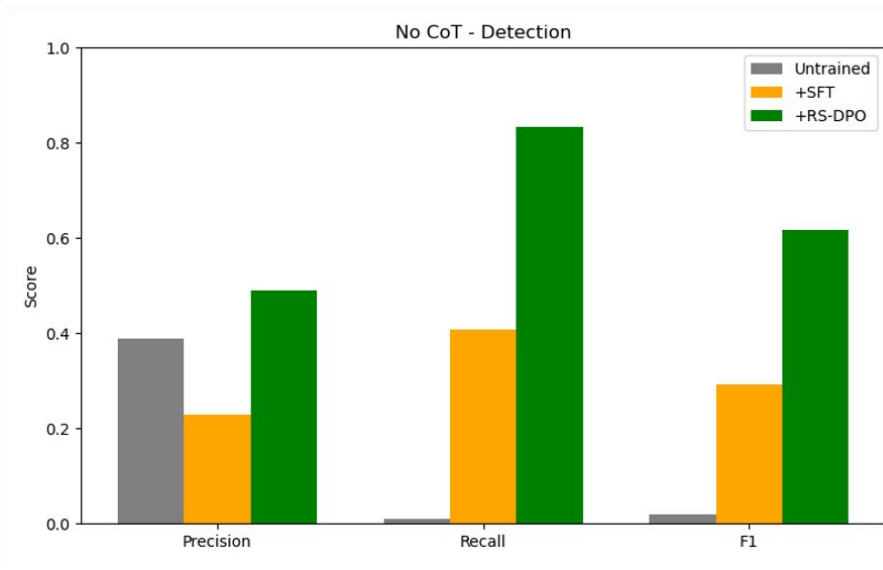
In Domain training and testing on JEP synthetic data



MetaLint: Generalization through Instruction Following (PMD→JEP)

Llama3.2-3B (non-reasoning model) trained with MetaLint-Java (JEP)

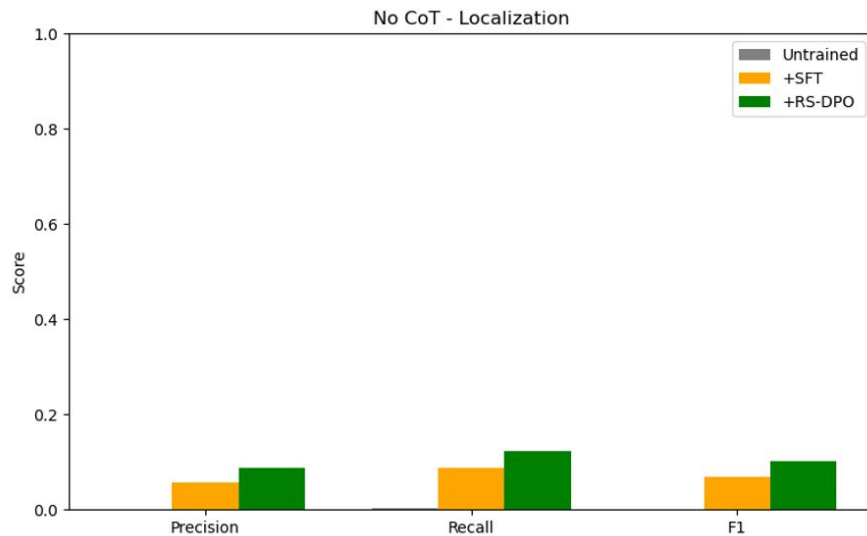
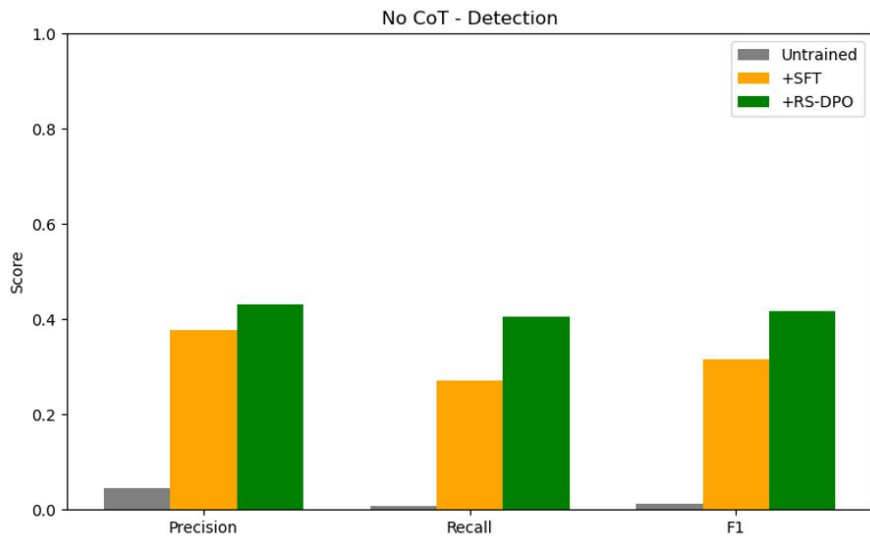
Far transfer from PMD training to JEP testing



MetaLint: Generalization through Instruction Following (JEP→PMD)

Llama3.2-3B (non-reasoning model) trained with MetaLint-Java (JEP)

Far transfer from JEP training to PMD testing

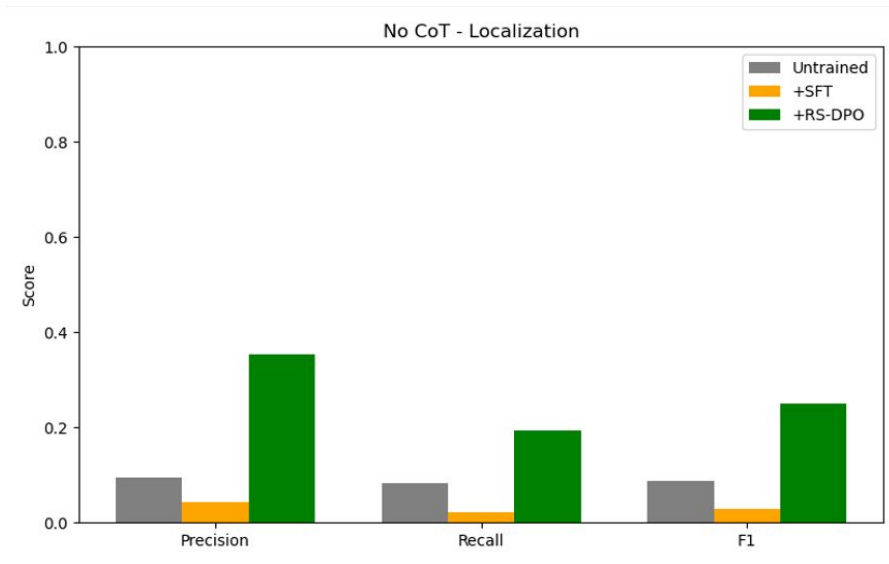
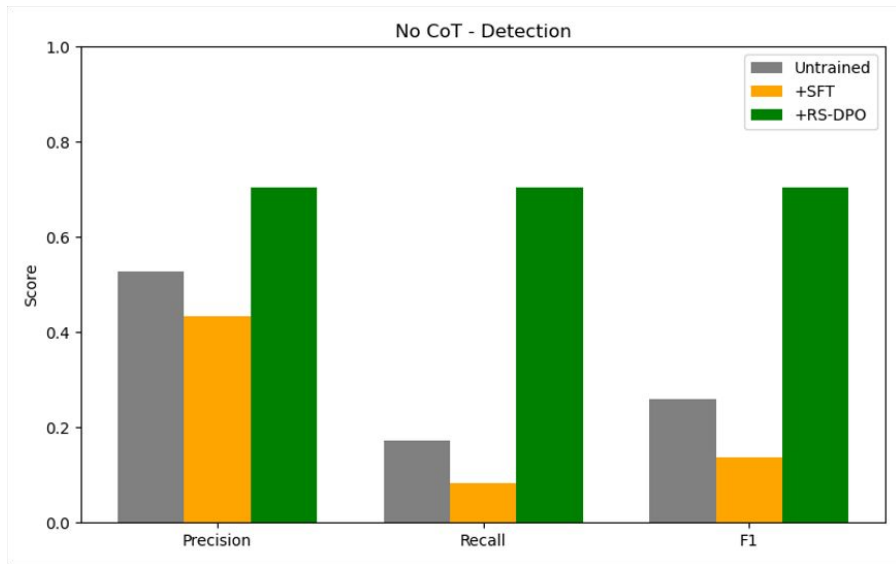


Easy-to-Hard Generalization

All results on hard PEP idiom
benchmark

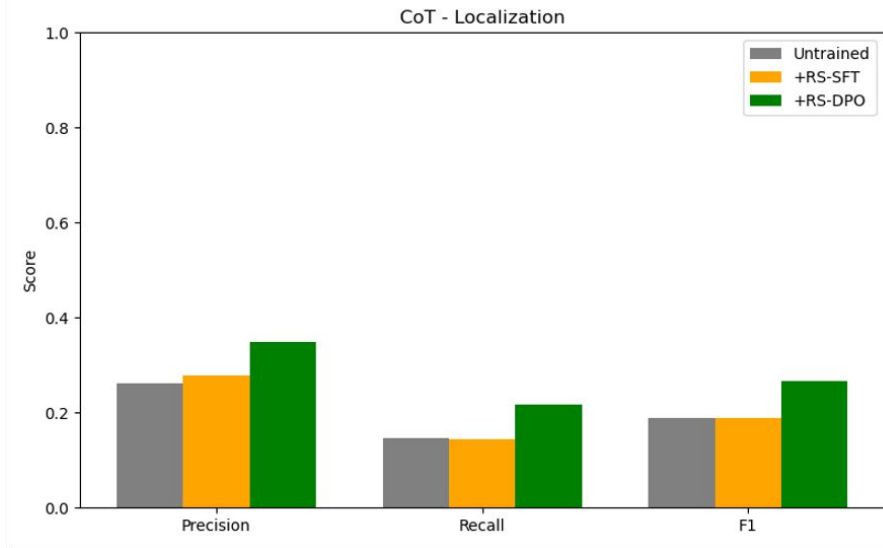
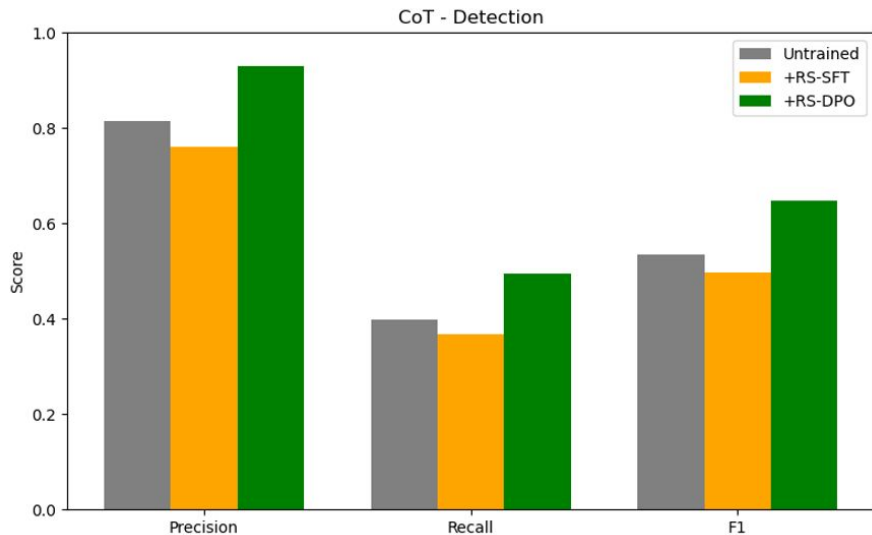
MetaLint: Easy-to-Hard Generalization (PEP Idioms)

Qwen3-4B trained with MetaLint (non-reasoning setting)



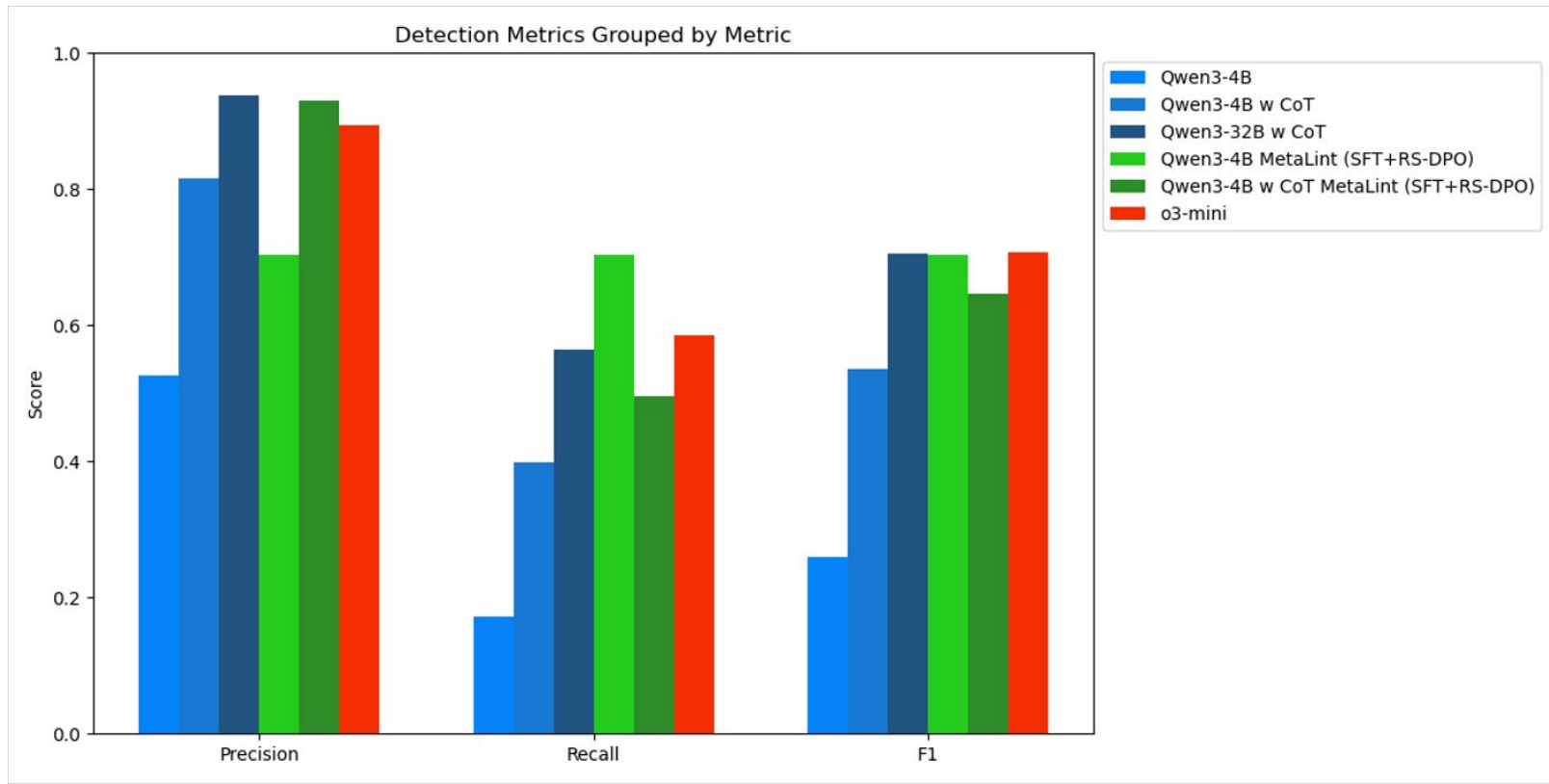
MetaLint: Easy-to-Hard Generalization (PEP Idioms)

Qwen3-4B trained with MetaLint (reasoning setting)



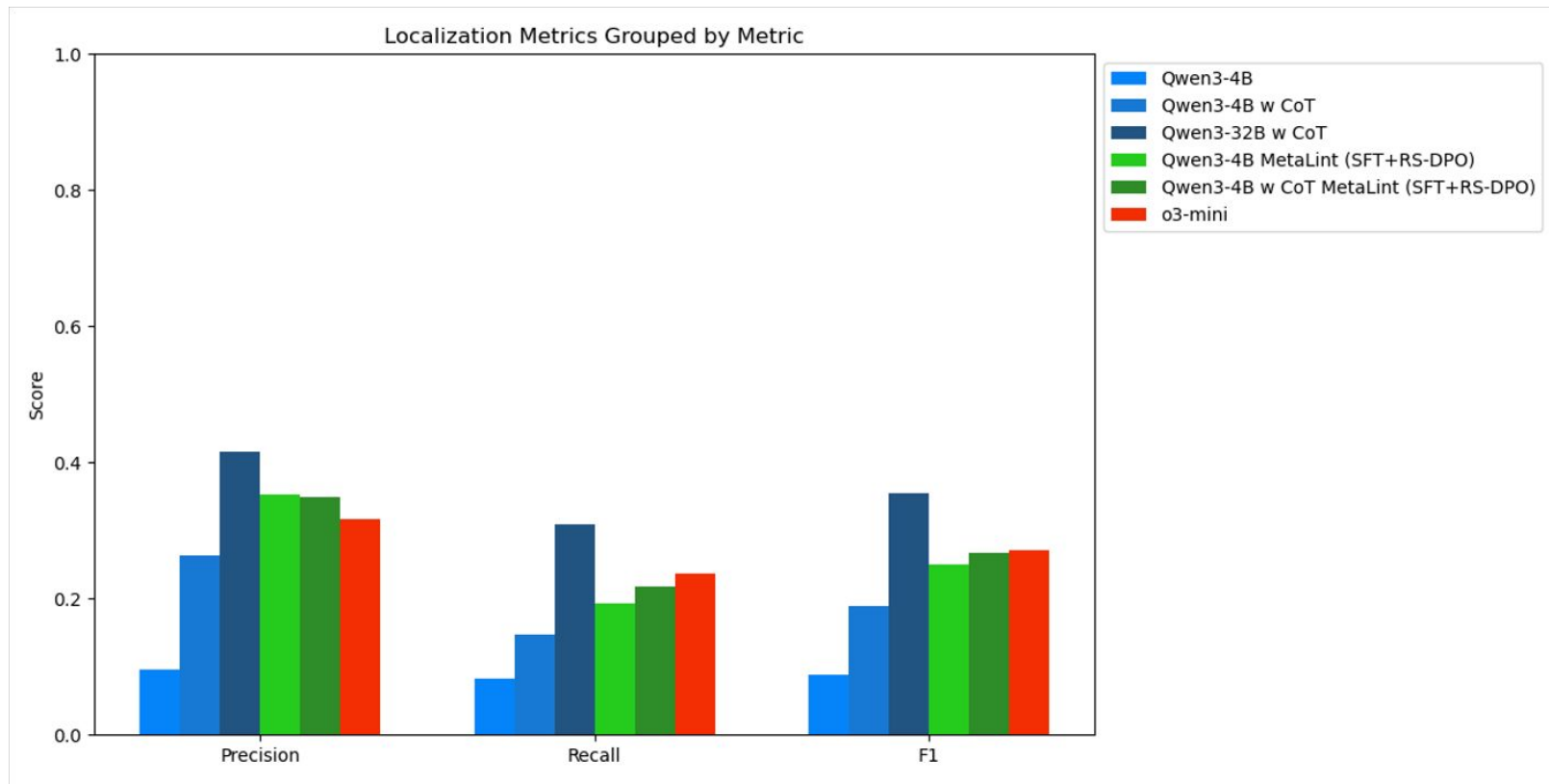
MetaLint: Easy-to-Hard Generalization (PEP Idioms)

Detection performance of MetaLint (Qwen3-4B) vs top open-source models & o3-mini



MetaLint: Easy-to-Hard Generalization (PEP Idioms)

Localization performance of MetaLint (Qwen3-4B) vs top open-source models & o3-mini



Discussion Questions

Can the choice of linters limit the generalization ability of LLMs trained with MetaLint?
(Emily Guo)

MetaLint: Discussion of Results

- MetaLint helps LLMs adapt to idioms at test time through instruction-following
- “Easy” idioms can be used to generate synthetic data at scale
- LLMs trained on “easy” idioms perform better on “hard” idioms
- Level of generalization depends on the base model
- SFT leads to memorization, but primes the model for RS-DPO (generalization)
- Can achieve comparable generalization without CoT reasoning (save on test time compute)
- Qwen3-4B (non-CoT setting) gets best in class detection recall and (CoT, non-CoT setting) matches o3-mini for localization!

MetaLint: Conclusion

Instruction Tuning + Preference Optimization (PO) can build generalizable linters:

1. Data distributions & Idioms: JEP \rightarrow PMD, PMD \rightarrow JEP, Ruff \rightarrow PEP
2. Programming Languages: Python, Java
3. Model families: Qwen, Llama

SFT leads to memorization but is crucial to learn task format

DPO leads to generalization (can slightly impact in-domain performance)

Have we built the “next generation of linters?”

No! ...

MetaLint trained models are more expensive and less accurate than Ruff (for easy idioms)

But ...

It expands the scope of what is “lintable” (for hard idioms)

Translates natural language idiom specs (e.g. docs) into linters

Allows personalization of code quality values by keeping LLMs adaptable (can keep up with evolving or org specific best practices)

Discussion Questions

Can allowing model to make refactoring decisions improve idiom understanding? **(Sathwik Acharya)**

How can we extend MetaLint into dynamic, online learning? **(Jiseung Hong)**

Larger, tool using models can be evaluated in different ways: **(Saujas)**

Easy idioms, no tools

Easy idioms with tools (the linters that detect them)

Hard idioms with tools (do models inappropriately try to use tools?)

It would be interesting to see if performance varies across these categories, and any patterns that might be surfaced.

Further Reading

[\[2509.21170\] Fine-Tuning LLMs to Analyze Multiple Dimensions of Code Review: A Maximum Entropy Regulated Long Chain-of-Thought Approach](#)

[Leveraging Reward Models for Guiding Code Review Comment Generation](#)

[Rethinking Code Review Workflows with LLM Assistance: An Empirical Study](#)

[\[2502.06633\] Combining Large Language Models with Static Analyzers for Code Review Generation](#)

[AI-Assisted Fixes to Code Review Comments at Scale](#)

[\[2412.18531\] Automated Code Review In Practice](#)

[\[2405.13565\] AI-Assisted Assessment of Coding Practices in Modern Code Review](#)

[\[2404.18496\] AI-powered Code Review with LLMs: Early Results](#)

[\[2402.02172\] CodeAgent: Autonomous Communicative Agents for Code Review](#)

[Automated Refactoring of Non-Idiomatic Python Code: A Differentiated Replication with LLMs](#)

[\[2406.03660\] Refactoring to Pythonic Idioms: A Hybrid Knowledge-Driven Approach Leveraging Large Language Models](#)

[CoUpJava: A Dataset of Code Upgrade Histories in Open-Source Java Repositories](#)

[\[2508.14419\] Static Analysis as a Feedback Loop: Enhancing LLM-Generated Code Beyond Correctness](#)

[GitHub - gptlint/gptlint: A linter with superpowers! 🔥 Use LLMs to enforce best practices across your codebase.](#)

[\[2506.10322\] Minimizing False Positives in Static Bug Detection via LLM-Enhanced Path Feasibility Analysis](#)

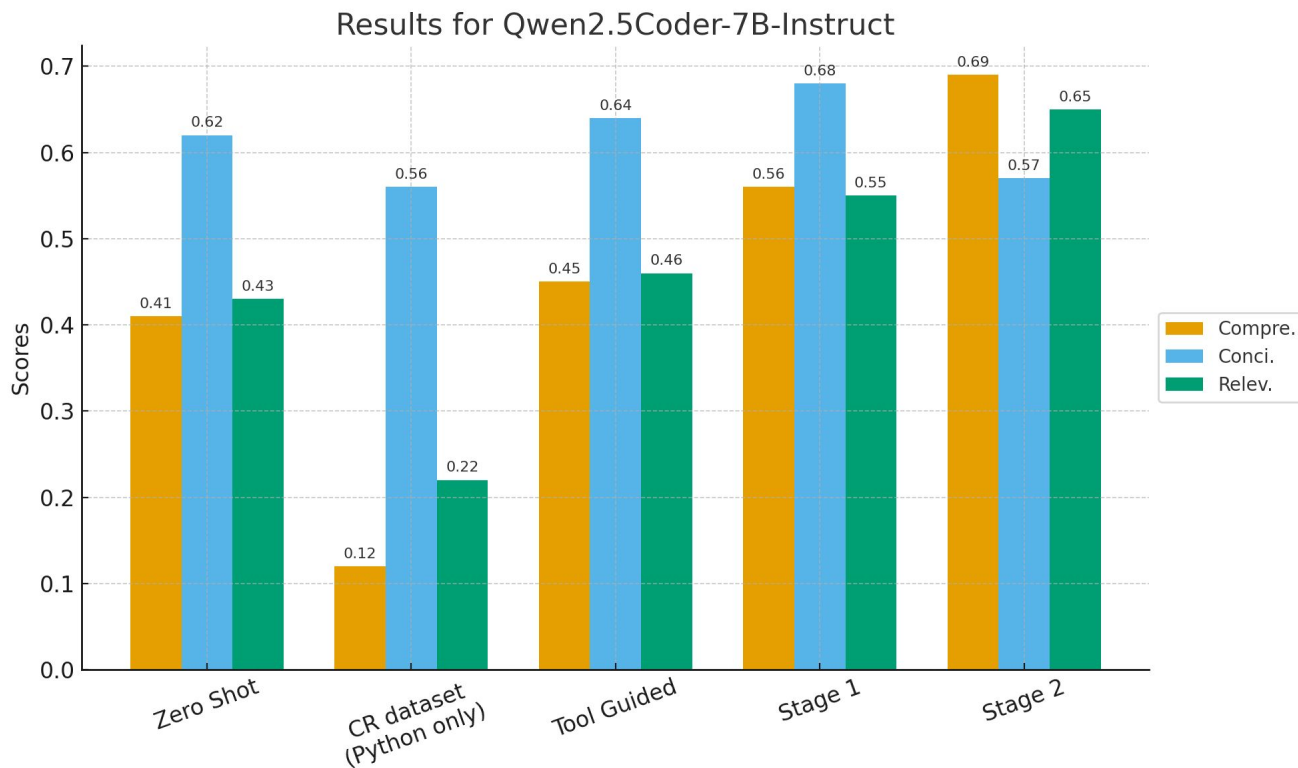
[\[2502.10815\] LintLLM: An Open-Source Verilog Linting Framework Based on Large Language Models](#)

[\[2504.05204\] Quantum Program Linting with LLMs: Emerging Results from a Comparative Study](#)

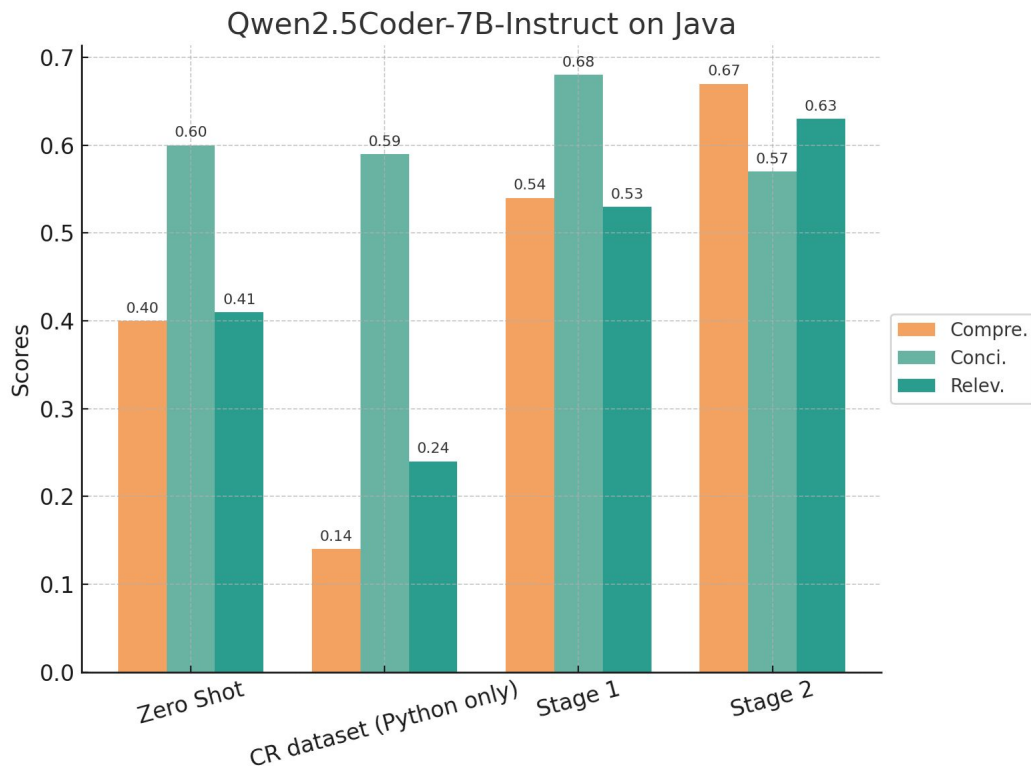
[Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities | IEEE Conference Publication](#)

Thank You!

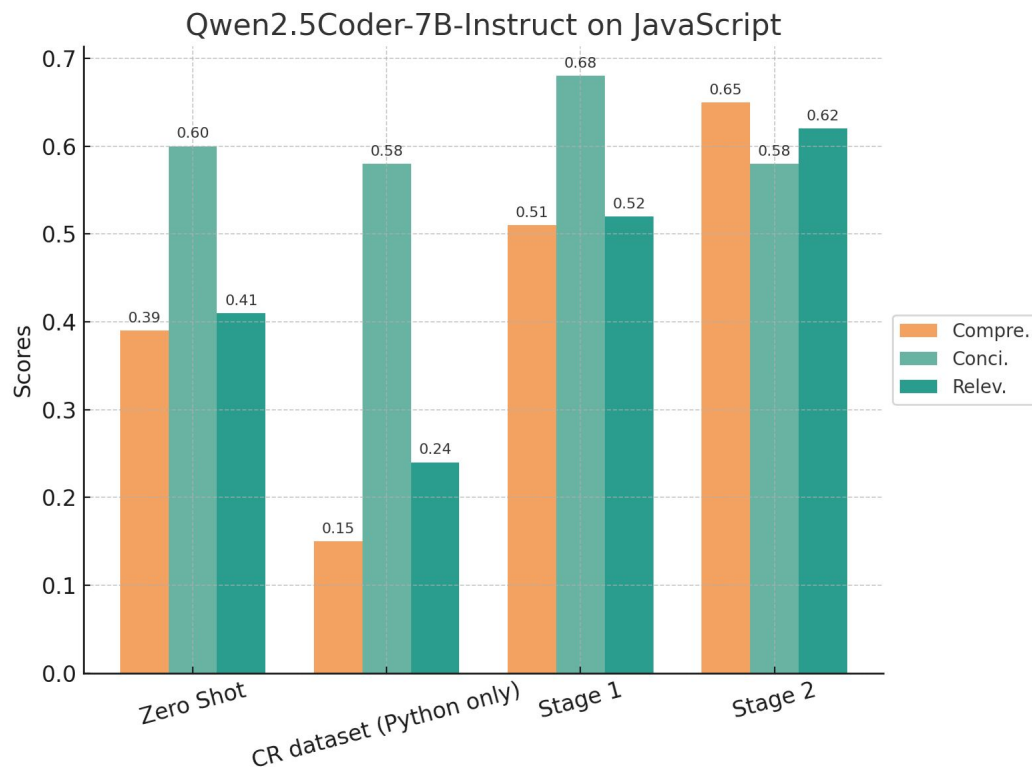
CRScore++: Results (ID)



CRScore++: Results (OOD) Java



CRScore++: Results (OOD) JavaScript



MetaLint: Meta-Task Prompt

Each meta-task targets 1 idiom at a time

METALINT Instruction Following Prompt

Look at the following list of code idiom specifications with definitions and examples:
{LIST_OF_IDIOM_SPECS}

Given these idioms, your task is to look at a code file and detect violations of the above idioms, and flag them like a linter. You should also suggest a fix if possible. Report the results per idiom specification mentioned above and just say NO VIOLATIONS FOUND if no violations are found for a given idiom. Do not detect any idioms not specified above.

Code file: {CODE_FILE}

Violations per idiom: