RAP-Gen: Retrieval-Augmented Patch Generation with CodeT5 for Automatic Program Repair

Weishi Wang[*12], ***Yue Wang****[*1], *Shafiq Joty*[12], *Steven Hoi*[1] (* equal contribution)

[1]Salesforce AI Research [2]Nanyang Technological University




Motivation

Debugging has been one of the most time-consuming in software development. **Automatic Program Repair (APR)** is crucial to improve developers' productivity!

You face a bug

Error msg: Expected an object to be thrown.

Buggy code:
```
if (_.isUndefined(exp))
    {throw "expected a value, got undefined instead";}
```

```
if (_.isUndefined(exp))
    {throw new Error("expected a value, got undefined instead");}
```

Retrieve

Fix

One bug fix example in codebase

Before fix:
```
evaluate: function() {
    throw "undefined for looping expressions";
}
```

After fix:
```
evaluate: function() {
    throw new Error("undefined for looping expressions");
}
```

In debugging, developers often search for similar bugs faced by others before (e.g., at StackOverflow), and learn from how they fixed the bugs

Motivation

Comparison with prior work

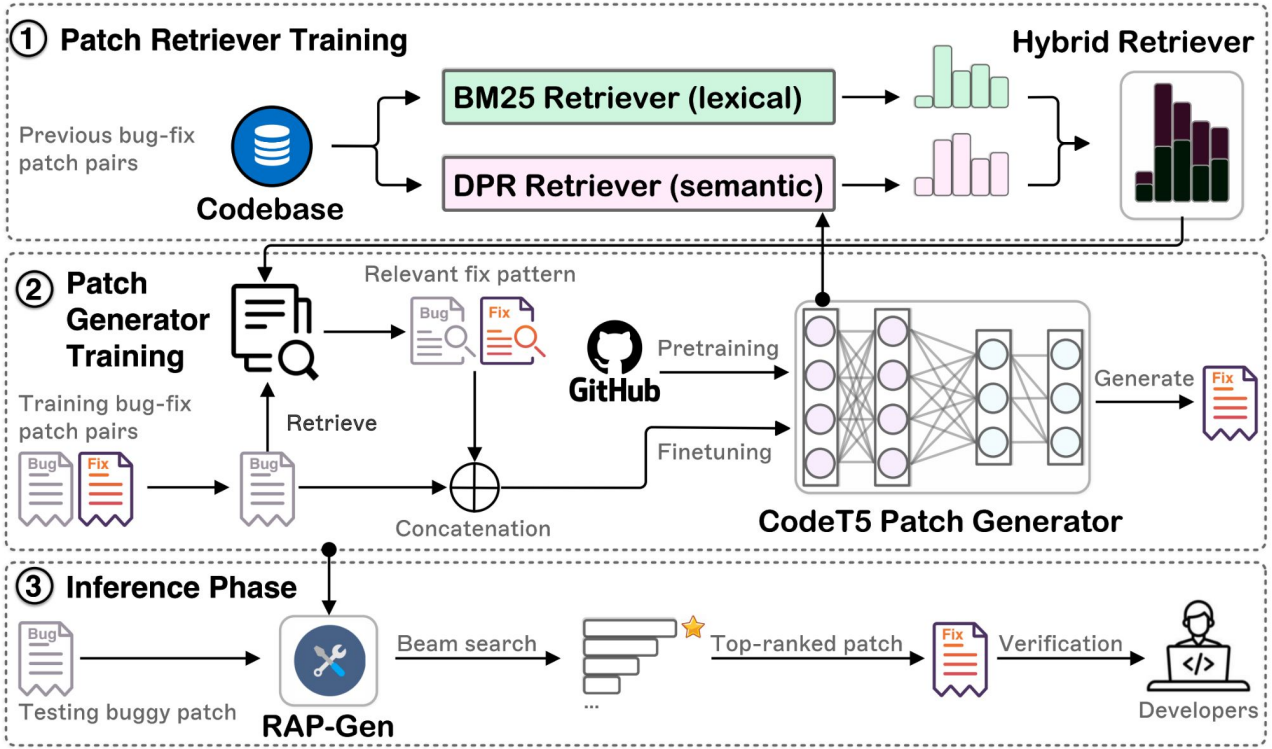
- **Prior Work**
  - Search-based techniques: based on *fix patterns* mined from existing codebase

    ⇒ Limitation: require lots of efforts to handcraft various fix pattern mining strategies
  - Learning-based techniques: automate APR as a sequence-to-sequence generation task in a pure data-driven manner with neural networks

    ⇒ Limitation: rely on a fixed set of parameters to model the complex search space of APR

- **Our Work**: we propose to ease the burden of neural APR systems by explicitly incorporating the fix pattern with a **R**etrieval-**A**ugmented **P**atch **Gen**eration framework (**RAP-Gen**)
  - It retrieves *one relevant bug-fix pair* in the existing codebase as the fix pattern

RAP-Gen Framework



RAP-Gen Components

Hybrid Patch Retriever $f_\phi(X_i, B_j) = DPR(X_i, B_j) + \lambda\,\mathrm{BM25}(X_i, B_j)$

- How to construct the training data for DPR?
  - Leverage the bug-fix pairs in the codebase as the pair often shares similar identifiers and functionalities
- Trained using an InfoNCE loss to contrast the positive pair with in-batch negatives

$$\mathcal{L}_{\text{infoNCE}} = \frac{1}{N}\sum_{i=1}^{N} -\log \frac{\exp(sim(B_i, F_i))}{\exp(sim(B_i, F_i)) + \sum_{j\in\mathcal{M}, j\neq i}\exp(sim(B_i, F_j))}$$

CodeT5 Patch Generator

- Adapt a code-aware encoder-decoder LLM CodeT5 as the foundation model
- Trained to generate the fix based on the bug and the top-1 retrieved bug-fix pair
  - Source input format:

$$\hat{X}_i = \texttt{[CLS]}\ \mathrm{X}_i\ \texttt{[BUG]}\ B_j\ \texttt{[FIX]}\ F_j$$

Benchmarks

- **TFix** (*snippet-level* in JavaScript)
  - Coding errors validated by a static analyzer EsLint
- **Code Refinement** (*function-level* in Java)
  - Detected by checking if the commit message matches certain patterns
    - (“fix” or “solve”) and (“bug” or “issue” or “problem” or “error”)
- **Defects4J** (*file-level* in Java)
  - Validated by running against test cases

Table 1: Statistics of three program repair benchmarks.

Benchmark	Version	Train	Valid	Test
TFix	Original	84,846	9,454	10,504
TFix	Deduplicated	84,673	9,423	10,465
Code Refinement	Small	46,680	5,835	5,835
Code Refinement	Medium	52,364	6,545	6,545
Defects4J	v1.2	-	-	388
Defects4J	v2.0	-	-	430

Research Questions

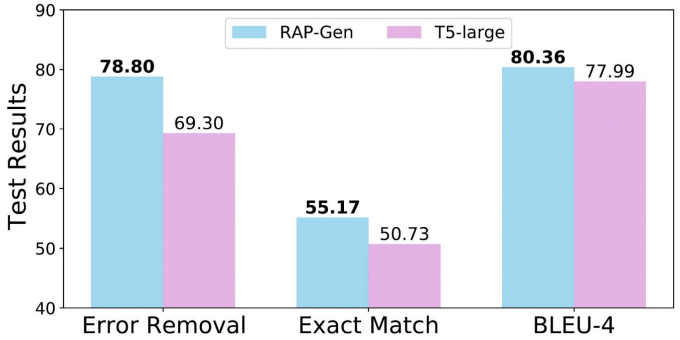
- **RQ1: Comparative study with other APR models on TFix.**
  - How does RAP-Gen perform to repair Linter-flagged JavaScript coding errors?
- **RQ2: Analysis of RAP-Gen predictions on TFix.**
  - How does RAP-Gen perform for different error types?
  - What fix operations does RAP-Gen adopt in repairing bugs?
- **RQ3: Comparative study with other APR models on Code Refinement.**
  - How does RAP-Gen perform to repair commit-related Java bugs?
- **RQ4: Analysis of our hybrid patch retriever.**
  - Can our hybrid patch retriever find relevant fix pattern to guide APR?
- **RQ5: Comparative study with other APR models on Defects4J.**
  - How does RAP-Gen perform to repair real Java bugs in open-source projects?

RQ1: Comparative Study on TFix

- RAP-Gen achieves the best performance in both Exact Match (EM) and BLEU-4, i.e. repairing 478+ bugs than T5-large

Model	EM	BLEU-4
T5-large (TFix)	49.58	76.96
CodeT5-base	53.46	78.92
RAP-Gen	**54.15**	**79.66**

- Error removal: Correct if the error is removed and no new error is introduced
- RAP-Gen achieves a much larger gain on error removal than other metrics
- Error removal is aligned with EM and BLEU-4



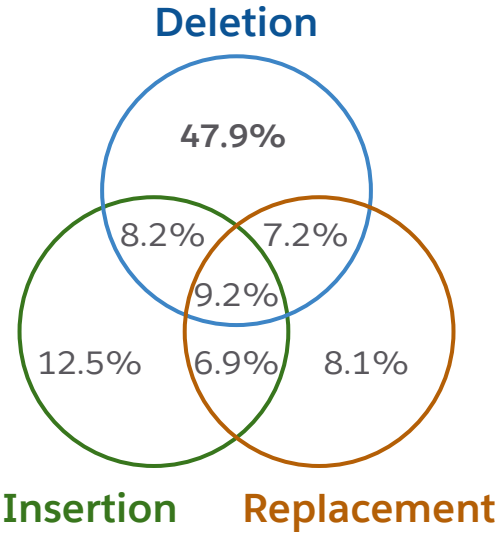
RQ2: Analysis of RAP-Gen Predictions on TFix

Table 4: Performance breakdown on 52 error types on TFix.

Error Type	#Samples	T5-large	RAP-Gen	Error Type	#Samples	T5-large	RAP-Gen
no-new-symbol	10	100.00	100.00	no-extra-bind	674	70.59	73.53
no-compare-neg-zero	13	0.00	0.00	no-case-declarations	723	58.90	67.12
no-ex-assign	40	25.00	25.00	no-fallthrough	743	76.00	77.33
for-direction	50	40.00	60.00	no-inner-declarations	830	38.10	46.43
no-unsafe-finally	63	42.86	14.29	no-array-constructor	980	86.73	85.71
use-isnan	71	37.50	25.00	no-constant-condition	1,251	48.78	54.47
no-class-assign	111	41.67	50.00	generator-star-spacing	1,396	67.86	72.86
no-dupe-class-members	117	8.33	8.33	no-extra-boolean-cast	1,458	54.11	58.22
no-func-assign	147	46.67	60.00	no-cond-assign	1,472	45.21	47.95
no-empty-pattern	178	27.78	44.44	no-process-exit	1,514	32.89	37.50
no-unused-labels	187	52.63	63.16	no-empty	2,055	26.70	31.55
no-duplicate-case	195	65.00	60.00	no-dupe-keys	2,181	53.42	55.25
getter-return	203	52.38	61.90	prefer-spread	2,466	45.08	45.49
no-sparse-arrays	237	25.00	45.83	no-useless-escape	2,920	35.15	40.61
no-const-assign	277	35.71	42.86	no-console	3,067	73.62	73.94
no-global-assign	318	59.38	68.75	guard-for-in	3,231	41.98	45.37
no-new-wrappers	360	27.78	38.89	no-throw-literal	4,075	72.06	74.51
no-this-before-super	413	47.62	69.05	no-debugger	4,164	94.48	94.24
no-unsafe-negation	423	72.09	76.74	prefer-rest-params	4,534	35.68	43.61
require-yield	429	72.09	76.74	no-unreachable	4,725	63.85	64.69
no-extend-native	443	31.11	26.67	no-extra-semi	5,969	82.61	83.61
no-new-object	446	71.11	66.67	no-redeclare	6,381	49.45	59.78
no-caller	446	20.00	22.22	comma-style	6,395	46.48	52.11
constructor-super	464	59.57	70.21	no-unused-vars	7,765	51.87	56.11
valid-typeof	539	51.85	51.85	no-undef	10,636	22.65	27.35
no-self-assign	610	34.43	44.26	no-invalid-this	16,166	37.48	44.13
				Sum/W. Avg.	104,561	49.58	**54.15**

RAP-Gen outperforms T5-large in 40/52 error types

What fix operations are performed by our RAP-Gen?



RQ3: Comparative Study on Code Refinement

- "Naive Copy" gives a high BLEU-4 score but with a zero exact match (EM)
  ⇒ EM as the primary metric
- RAP-Gen achieves new SoTA results with significant improvements over CodeT5
  ⇒ Retrieved fix patterns provide helpful signals to guide APR

Table 6: Performance of RAP-Gen on the Code Refinement.

Model	Small		Medium	
	EM	BLEU-4	EM	BLEU-4
Naive Copy	0.00	78.06	0.00	90.91
LSTM	10.00	76.76	2.50	72.08
Transformer	14.70	77.21	3.70	89.25
RoBERTa (code)	15.90	77.30	4.10	90.07
CodeBERT	16.40	77.42	5.16	91.07
GraphCodeBERT	17.30	80.02	9.10	91.31
PLBART	19.21	77.02	8.98	88.50
CoTexT	21.58	77.28	13.11	88.40
NSEdit	24.04	71.06	13.87	85.72
CodeT5	21.61	77.43	13.96	87.64
RAP-Gen	**24.80**	78.28	**15.84**	90.01

RQ4: Analysis of Hybrid Patch Retriever

Which retriever is the best for RAP-Gen?

- Randomly retrieving a bug-fix example for augmentation does not help
- CodeT5 is better than CodeBERT, while our hybrid retriever is the best

Can our retriever retrieve lexically and semantically relevant patches?

- Hybrid retriever is able to balance both lexical and semantic matching

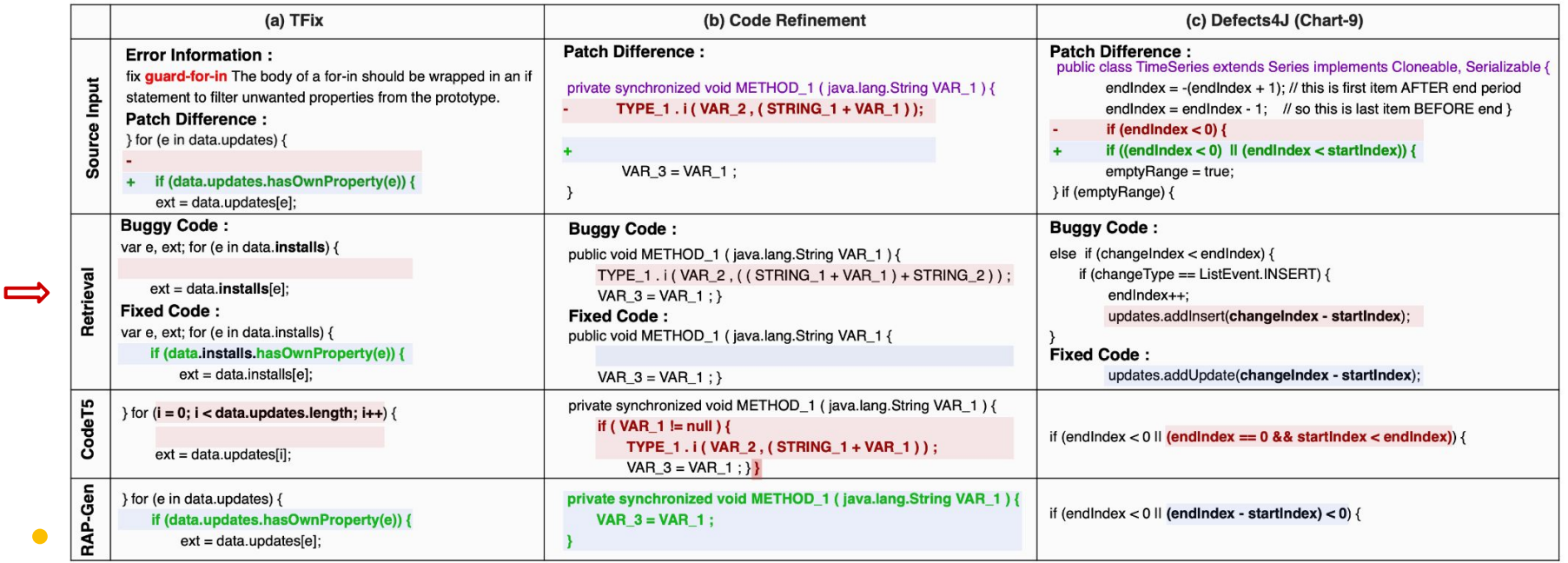
Table 7: Effects of retriever modules in RAP-Gen.

Retriever	TFix	Refine-Small	Refine-Medium
No Retriever	53.46	21.61	13.96
Random	52.98	21.25	13.53
BM25	53.88	23.82	15.37
CodeBERT	52.96	22.28	15.42
CodeT5	53.93	24.37	15.60
Hybrid (BM25+CodeT5)	54.15	24.80	15.84

Table 8: Lexical (BLEU-4) and semantic (CosSim) retrieval matching results on TFix and Code Refinement benchmarks.

Retriever	TFix		Refine-Small		Refine-Medium	
	BLEU-4	CosSim	BLEU-4	CosSim	BLEU-4	CosSim
Random	0.1	35.5	14.6	35.4	14.6	30.6
BM25	23.7	70.9	41.5	68.5	39.0	66.6
DPR	21.7	**75.4**	54.4	**84.9**	44.3	**81.3**
Hybrid	**24.4**	73.4	**57.4**	84.2	**45.0**	80.9

RQ4: Analysis of Hybrid Patch Retriever



	(a) TFix	(b) Code Refinement	(c) Defects4J (Chart-9)

Source Input	**Error Information :**
	fix **guard-for-in** The body of a for-in should be wrapped in an if statement to filter unwanted properties from the prototype.
	Patch Difference :
	} for (e in data.updates) {
	-
	+ if (data.updates.hasOwnProperty(e)) {
	ext = data.updates[e];
		Patch Difference :
		private synchronized void METHOD_1 (java.lang.String VAR_1) {
		- TYPE_1 . i (VAR_2 , (STRING_1 + VAR_1));
		+
		VAR_3 = VAR_1 ;
		}
			Patch Difference :
			public class TimeSeries extends Series implements Cloneable, Serializable {
			endIndex = -(endIndex + 1); // this is first item AFTER end period
			endIndex = endIndex - 1; // so this is last item BEFORE end }
			- if (endIndex < 0) {
			+ if ((endIndex < 0) || (endIndex < startIndex)) {
			emptyRange = true;
			} if (emptyRange) {

Retrieval	**Buggy Code :**
	var e, ext; for (e in data.**installs**) {
	ext = data.**installs**[e];
	Fixed Code :
	var e, ext; for (e in data.installs) {
	if (data.**installs**.hasOwnProperty(e)) {
	ext = data.installs[e];
		Buggy Code :
		public void METHOD_1 (java.lang.String VAR_1) {
		TYPE_1 . i (VAR_2 , ((STRING_1 + VAR_1) + STRING_2)) ;
		VAR_3 = VAR_1 ; }
		Fixed Code :
		public void METHOD_1 (java.lang.String VAR_1 {
		VAR_3 = VAR_1 ; }
			Buggy Code :
			else if (changeIndex < endIndex) {
			if (changeType == ListEvent.INSERT) {
			endIndex++;
			updates.addInsert(**changeIndex - startIndex**);
			}
			Fixed Code :
			updates.addUpdate(**changeIndex - startIndex**);

CodeT5	} for (**i = 0; i < data.updates.length; i++**) {
	ext = data.updates[i];
		private synchronized void METHOD_1 (java.lang.String VAR_1) {
		if (VAR_1 != null) {
		TYPE_1 . i (VAR_2 , (STRING_1 + VAR_1)) ;
		VAR_3 = VAR_1 ; } }
			if (endIndex < 0 || (**endIndex == 0 && startIndex < endIndex**)) {

RAP-Gen	} for (e in data.updates) {
	if (data.updates.hasOwnProperty(e)) {
	ext = data.updates[e];
		private synchronized void METHOD_1 (java.lang.String VAR_1) {
		VAR_3 = VAR_1 ;
		}
			if (endIndex < 0 || (**endIndex - startIndex) < 0**) {

Figure 3: Bug fix examples on three APR benchmarks, where RAP-Gen successfully fix bugs while CodeT5 fails to do so.

RQ5: Comparative Study on Defects4J

- RAP-Gen repairs the largest number of bugs in both spectrum-based Fault Localization (FL) and perfect FL settings
- It complements other existing APR models (i.e. RewardRepair, Recoder, and SelfAPR) by repairing 13 and 12 unique bugs for v1.2 and v2.0, respectively

Table 9: Performance of RAP-Gen on Defects4J v1.2 and v2.0.

Model	Spectrum-based FL		Perfect FL	
	v1.2	v2.0	v1.2	v2.0
SequenceR [7]	-	-	14	-
BugLab [2]	-	-	17	6
DLFix [31]	30	-	40	-
CoCoNuT [40]	-	-	43	-
RewardRepair [76]	27	24	44	43
DEAR [32]	47	-	53	-
CURE [20]	-	-	55	-
Recoder [79]	**49**	19	64	-
SelfAPR [75]	39	**28**	65	45
CodeT5	27	13	58	28
RAP-Gen	48	26	**72**	**53**

In the table cells, it represents the number of correct patches. '-' indicates data unavailability.

Conclusion

In this work, we

- present a novel retrieval-augmented generation framework for APR
- show that retrieved bug-fix pairs can serve as a good guiding fix pattern for APR
- comprehensively evaluate RAP-gen on 3 APR benchmarks with different types of bugs and demonstrate its superiority over other learning-based models

In future work, we'll

- explore more end-to-end framework to connect retriever and generator like RAG
- explore the use of larger code LLMs for APR tasks


Thank You
Q&A