Where's the Bug? Attention Probing for Scalable Fault Localization

Adam Stein[*]
University of Pennsylvania
steinad@seas.upenn.edu

Arthur Wayne[*]
University of Pennsylvania
artwayne@seas.upenn.edu

Aaditya Naik
University of Pennsylvania
asnaik@seas.upenn.edu

Mayur Naik
University of Pennsylvania
mhnaik@seas.upenn.edu

Eric Wong
University of Pennsylvania
exwong@seas.upenn.edu

Abstract

Ensuring code correctness remains a challenging problem even as large language models (LLMs) become increasingly capable at code-related tasks. While LLM-based program repair systems can propose bug fixes using only a user's bug report, their effectiveness is fundamentally limited by their ability to perform fault localization (FL), a challenging problem for both humans and LLMs. Existing FL approaches rely on executable test cases, require training on costly and often noisy line-level annotations, or demand resource-intensive LLMs. In this paper, we present *Bug Attention Probe* (*BAP*), a method which learns state-of-the-art fault localization without any direct localization labels, outperforming traditional FL baselines and prompting of large-scale LLMs. We evaluate our approach across a variety of code settings, including real-world Java bugs from the standard Defects4J dataset as well as seven other datasets which span a diverse set of bug types and languages. Averaged across all eight datasets, *BAP* improves by 34.6% top-1 accuracy compared to the strongest baseline and 93.4% over zero-shot prompting GPT-4o. *BAP* is also significantly more efficient than prompting, outperforming large open-weight models at a small fraction of the computational cost.[2]

1 Introduction

Correctness is a fundamental desirable property of code. Both human-written and LLM-generated code are prone to bugs [1, 2] including syntax errors that prevent the execution of code, semantic mistakes that cause incorrect or unintended behaviors, and vulnerabilities that compromise security in otherwise correct code. While there are various methods for detecting bugs (e.g. failed tests, user bug reports, program crashes, etc.), identifying its root cause, or *localizing* the bug, is still costly [3, 4].

Automated software fault localization (FL) aims to help a programmer answer the question, "Where's the bug?", ideally pointing to specific lines of buggy code. The traditional FL approaches rely on executable tests to determine the buggy lines [5]. Without relying on tests, FL is even more challenging since such a system must reason about what is buggy without external feedback. But recently, supervised training of models on large, labeled datasets [6, 7], and prompting of the largest LLMs such as GPT-4o has shown promise at FL without tests [8]. On a single method context, large-scale supervised training and prompting approaches can significantly surpass the traditional techniques which need tests [6, 8].

[*]Equal contribution.

[2]*BAP* is open-sourced here: https://github.com/adaminsky/BAP

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Mechanistic Interpretability.

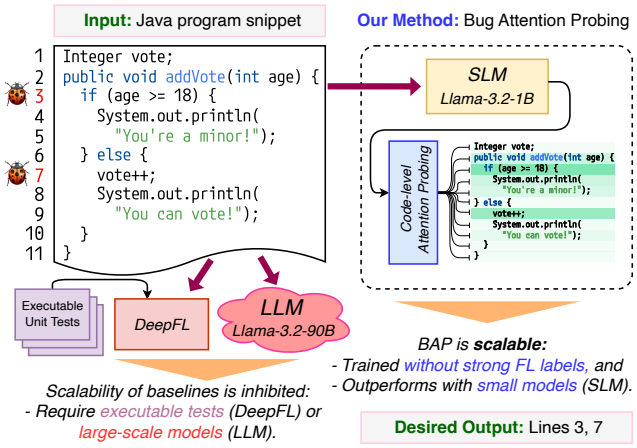


Figure 1: Comparison of our approach *Bug Attention Probe* (*BAP*) with baselines DeepFL and LLM prompting on a Java program snippet. The program has two bugs: the age condition on line 3 is reversed and line 6 throws a null pointer exception. *BAP* correctly localizes both bugs. Here, our method is trained on Llama-3.2-1B, a "small" language model (SLM), with only weak supervision *i.e.* binary bug presence labels. Obtaining comparable accuracy via prompting demands a significantly more resource-intensive LLM, such as Llama-3.2-90B, or larger. Previous approaches like DeepFL require executable test cases before they can attempt to provide useful information.

State-of-the-art FL methods, however, are still limited in *scalability*, or the ability to leverage cheaply available supervision to reach strong performance, even with small models. This lack of scalability leads to the impracticality of many FL techniques. For example, we show a simple buggy Java code snippet in Figure 1 which we want to run FL on, but we encounter several issues: traditional methods require executable tests and LLM prompting methods are only effective for the largest of models. On large codebases, LLMs must be called at least once per function, which quickly becomes expensive. Further, training-based approaches assume extensive amounts of strong supervision for FL, which is rarely available in practice [9].

This leads to our central question: *How do we achieve strong fault localization performance without relying on executable tests, costly large-scale LLMs, or strong supervision?*

We answer this question by proposing the Bug Attention Probe (*BAP*), a scalable LLM probing technique for FL, scaling to use available bug related data without strong FL supervision and scaling with base model size while still achieving strong performance with small models. *BAP* exhibits three desirable properties: (1) **lightweight**, (2) **code-level localization**, and (3) localization of **multi-line bugs**. First, *lightweight* refers to the limited requirement for supervision (we use an attention mechanism to learn from binary bug presence rather than fine-grained location supervision), the test-free nature, and the model size (*BAP* can probe small language models (SLMs) to elicit performance significantly stronger than the underlying SLM). Second, *BAP* localizes bugs in a human-interpretable manner to expressions, statements, or lines in code, even though LLMs operate on the token level. Third, *BAP* localizes multi-line bugs, or multiple bugs in one method, better than existing approaches. Multi-line bugs are practically relevant since the majority of real bugs are multi-line [10].

To evaluate *BAP*, we use eight diverse and widely-used fault localization benchmarks, including syntax errors, semantic mistakes, and weaknesses. Our evaluation suite covers over 50,000 code samples across three languages: Python, Java, and C. This notably includes Defects4J [11], the most commonly used FL dataset.

We evaluate *BAP* on top of the Llama-3 family of models and compare it to state-of-the-art FL methods, including traditional test-based FL and prompting of various proprietary and open-weights LLMs. Averaged across eight datasets, *BAP* improves by 23.4% over the strongest baselines for top-1 FL accuracy which includes a 24.2% improvement for Defects4J, and a 50.5% improvement on DeepFix [12]. In addition, *BAP* achieves these performance improvements at over ten times greater efficiency in terms of model size and FLOPs for inference. *BAP* also localizes multi-line bugs better than existing methods and continues to have stronger performance than prompting for longer code sequences. While *BAP* significantly outperforms competitive baselines, it is able to achieve 35%

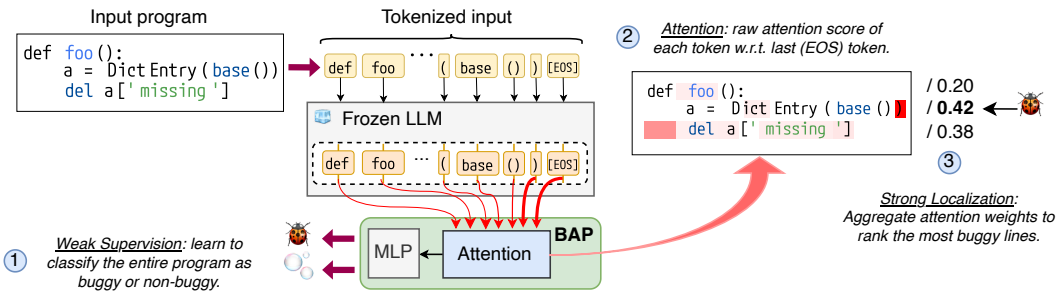


Figure 2: Illustration of *BAP* as a method to elicit line-level fault localization from a frozen LLM through weak supervision. In step one, the probe is trained as a binary classifier to distinguish buggy from non-buggy code. Then in step two, we visualize the learned attention weights on the given sequence. Finally, in step three, we sum the attention weights within each line to produce a line-level "bugginess" score. *BAP* localizes the bug to the line with the highest score, the Top-1 result.

top-1 FL accuracy on average over our datasets, and our evaluation highlights avenues for further advances in scalable FL.

In summary, our work makes the following contributions:

- We propose Bug Attention Probe (*BAP*) as a general method for scalable fault localization, requiring only coarse-grained detection supervision and eschewing the need for localization labels (Section 3).
- *BAP* significantly improves over state-of-the-art fault localization methods by 34.6% top-1 accuracy on average over eight fault localization benchmarks while using ten times less memory and FLOPs.
- *BAP* exhibits better length generalization and can predict multi-line bugs more effectively than prompting. We also identify areas for further advances.

2 Background

We introduce fault localization, LLM probing, and challenges with existing FL techniques.

2.1 Fault Localization

Before defining bug localization, we first describe the problem of bug *detection*, or determining if code is overall buggy or not. A bug is a concept $b : \mathcal{P} \to \{0, 1\}$ where $\mathcal{P}$ is the space of all programs. For a bug b, we construct a supervised detection dataset $\mathcal{D}_{\text{Det}} = \{(p, b(p)) \mid p \in \mathcal{P}\}$ consisting of programs labeled as buggy or not buggy.

Bug localization is the task of identifying the line (or lines) of code where a bug occurs. A bug localizer is a mapping $l : \mathcal{P} \to \mathbb{Z}^+$ from programs to one or more line numbers. For a program $p \in \mathcal{P}$ split into lines $[p_0, \ldots, p_k]$, where p_i is the i^{th} line of program p, we define the ground truth bug localization using the notion of a counterfactual explanation: a program has a bug localized to line i if modifying p_i would remove the bug. Formally, for program p such that $b(p) = 1$, bug b is localized to line i if there exists a modified line p'_i such that $b(p') = 0$ where $p' = [p_0, \ldots, p_{i-1}, p'_i, p_{i+1}, \ldots, p_k]$.

Some bugs require multi-line fixes, in which case changing multiple lines of the program would fix the bug. We therefore assume that the ground truth localization consists of one or more lines of code. We can now define a localization dataset containing direct line-level supervision:

$$\mathcal{D}_{\text{Loc}} = \{(p, 0, \varnothing) \mid p \in \mathcal{P} \text{ and } b(p) = 0\} \cup \{(p, 1, l(p)) \mid p \in \mathcal{P} \text{ and } b(p) = 1\}.$$

2.2 LLM Probing

LLM probing involves training a classifier on top of intermediate states from a model [13]. Since all probing techniques are trained from a dataset of model hidden representations, we start by introducing this dataset in our setting.

For a program fragment p consisting of T tokens, we call the intermediate representation from LLM layer k, $\text{LLMRep}(p, k) = z \in \mathbb{R}^{T \times d}$ where d is the hidden dimension of the LLM. The dataset we

use for probing in the rest of this paper is the following:

$$\mathcal{H}_{\text{Det}} = \{(\text{LLMRep}(p, k), y) : (p, y) \in \mathcal{D}_{\text{Det}}\}.$$

We can also define $\mathcal{H}_{\text{Loc}}$ equivalently which additionally includes the ground truth line numbers with each sample, but as we discuss later, this dataset is not ideal for training.

These datasets, $\mathcal{H}_{\text{Det}}$ and $\mathcal{H}_{\text{Loc}}$, are not directly amenable to standard probing since each sample, $\text{LLMRep}(p, k)$, is a sequence of hidden representations varying in length across samples. Producing a general-purpose fixed-length sequence representation from an autoregressive LLM is a challenging problem [14], so probing methods typically apply a simple pooling operation, $\text{POOL} : \mathbb{R}^{T \times d} \rightarrow \mathbb{R}^{1 \times d}$, such as selecting the last token or averaging all tokens. Specific to using $\mathcal{H}_{\text{Loc}}$, existing work takes the approach of converting the labels $l(p)$ into a binary mask over the buggy program lines so that a model can be trained to predict such masks [6].

2.3 Challenges

We highlight three main challenges for FL.

Need for Strong Supervision. High-quality FL datasets are rare due to the heavy manual effort required: a failing test may reveal a bug but not its cause. As a result, real-world datasets are either small (e.g., Defects4J with 395 samples [11]) or noisy (e.g., ManySStuBs4J [1]). In our experiments, methods trained on such data perform worse than those avoiding strong supervision.

Localizing Multi-line Bugs. Most real-world bugs span multiple lines [10], yet existing FL methods and datasets mainly target single-line bugs [15, 1, 16].

Resource Efficiency. State-of-the-art FL performance requires large LLMs on method-level context, but these models are resource-intensive, API-restricted, and costly when used repeatedly. By contrast, traditional execution-based FL methods can run locally in an IDE with modest resources.

3 Attention Probing for Fault Localization

In this section, we introduce our approach for addressing the previous challenges.

We propose the Bug Attention Probe (*BAP*), an LLM probing technique for performing FL which provides scalability through lightweight requirements (both in terms of training supervision and model size), interpretable code-level localization, and handling of multi-line bugs. We take inspiration from attention probing [17, 18], a technique from the interpretability literature for studying linguistic phenomena such as the attention placed on verb tokens, but we use such an approach for code-level FL, as in localizing bugs to lines within code.

We illustrate *BAP*, its process of training from bug detection data, $\mathcal{H}_{\text{Det}}$, and the computation of line-level bug localization from attention weights in Figure 2.

3.1 Why Attention Probing for Fault Localization?

Our main motivation for using an attention mechanism is that the attention pooling operation trained for the task of bug detection encourages the probe to learn to attend to the location of bugs, without strong FL supervision. Intuitively, the probe attends to parts of the code which are more informative for bug detection, and we hypothesize that these locations often correspond to the location of bugs.

For bugs which are localized to multiple lines, which is the majority of real world faults, an attention mechanism is also helpful since it operates on the token-level, making no distinction between a single and multi-line bug location.

Finally, an attention mechanism is efficient since it operates over all tokens in parallel. This means that a complete ranking of all the program lines in terms of the likelihood they contain a bug is produced by a single forward pass of attention. This is in contrast to methods such as LLM prompting which must continue to output more tokens to localize a bug to more lines.

Algorithm 1 *BAP* Line-level Fault Localization

1: **Input:** code sample p, layer k.
2: Tokenize p to get $p = [t_1, \ldots, t_T]$.
3: $z = \text{LLMRep}(p, k)$
4: $v, a = \text{MHA}(z)$ where $v \in \mathbb{R}^{T \times d}$ and $a \in \mathbb{R}^{M \times T \times T}$
5: $\bar{a} = \frac{1}{M} \sum_{m=1}^{M} a_{m,-1,:}$ {Average attention over all heads, for the last token}
6: Group attention scores into lines: $s_1 = [\bar{a}_1, \ldots, \bar{a}_i]$, $s_2 = [\bar{a}_{i+1}, \ldots, \bar{a}_{i+j}], \ldots$
7: $l_i = \sum_{t \in s_i} t$ for all i {Line-level attention score}
8: **Return:** $\text{argsort}_{i \in [1,L]}\, l_i$

3.2 From Weak Supervision to Token-level Localization

To enable learning FL from weak supervision, we use a single layer Transformer decoder block as the architecture of *BAP* to factorize the bug detection task into localization (attention) over tokens and detection (classification) on the sequence-level. The input to the probe consists of a sequence of token representations from the kth layer of the LLM, $\text{LLMRep}(p, k) = [z_1, \ldots, z_T] \in \mathbb{R}^{T \times d}$, for a program p consisting of T tokens. The standard multi-head attention mechanism takes $\text{LLMRep}(p, k)$ as input and outputs attention scores $a \in \mathbb{R}^{M \times T \times T}$ where M is the number of attention heads, and processed token representations $v \in \mathbb{R}^{T \times d}$. We compute token-level attention scores of all tokens to the final token as $\bar{a} = \frac{1}{M} \sum_{m=1}^{M} a_{m,-1,:} \in \mathbb{R}^{T}$ where we average the attention from each head. The process of producing token-level attention from input code is also shown in lines 1-5 of Algorithm 1.

To get useful token-level attention from our model, we must train it on a downstream task. To train *BAP*, we use weak supervision from a bug detection dataset, $\mathcal{H}_{\text{Det}}$ defined in Section 2. We pass the last token from the processed sequence v from the attention through a feed-forward network (MLP) to produce a scalar output representing the buggy/non-buggy prediction. The model is then optimized using gradient descent with the binary cross-entropy loss. After training the probe for bug detection, we then examine the learned attention weights $\bar{a}$ which provide token-level fault localization.

3.3 From Token-level to Code-level Localization

A major challenge with this approach is that tokens are not interpretable for programmers. Therefore, we provide a method to aggregate token-level fault localization into a code-level localization. Our approach is summarized in lines 6 to 8 of Algorithm 1. In our experiments, we focus on line-level granularity, but this method also allows us to perform statement-level and function-level localization without any significant modifications.

Line 5 of Algorithm 1 computes the token-level attention scores, and we call $\bar{a}_i$ the probe's attention score for the ith token. To produce a line-level attention score, we sum the attention scores for all tokens in the ith line, $s_i = [\bar{a}_{i_1}, \bar{a}_{i_2}, \ldots]$, to produce l_i, the probe's attention score for the ith line. The computation of line-level attention scores is shown in line 7 of Algorithm 1.

Line 8 of Algorithm 1 shows that the resulting line-level localization from *BAP* is the ranking of input lines based on their respective attention scores. In practice, we truncate to the top-k lines out of total lines L. The line with the highest attention score is thus noted as the top-1 prediction.

4 Experiments

We evaluate *BAP* over a diverse suite of eight fault localization benchmarks. This includes Defects4J [11], the most popular fault localization benchmark, as well as seven additional benchmarks covering three general bug types. In the rest of this section, we first introduce the datasets, then the baseline methods, and finally the results of each experiment. We answer the following research questions in this section: **RQ1**: How effective is *BAP* at FL in diverse scenarios? **RQ2**: How does the efficiency of *BAP* compare to baselines? **RQ3**: Can *BAP* effectively localize multi-line bugs? **RQ4**: How does the generalization ability of *BAP* compare to zero-shot prompting of the base model?

Table 1: A summary of the datasets we use for our evaluation. For a further breakdown of buggy samples into categories, see our discussion in Appendix A.1.

	# Train		# Test		
Dataset	Bug	Clean	Bug	Clean	$\mathbb{E}$[LoC]
Defects4J v1.2.0	368	368	90	90	35.8
Defects4J v3.0.1	N/A	N/A	437	N/A	46.7
GitHub-Py	1323	1323	400	400	9.3
GitHub-J	1370	1370	460	460	19.3
DeepFix	1475	1475	365	365	26.3
TSSB	4085	3745	1104	1080	24.8
ManySStuBs4J	3821	3821	1093	1093	15.5
Juliet-J	4039	3061	1011	989	62.5
Juliet-C	3718	3697	966	939	44.7

Table 2: Comparison of *BAP* with existing FL methods across eight benchmarks. We evaluate line-level localization performance on a method-level context, measured by top-1 accuracy. From left-to-right: Defects4J v1.2.0, GitHub-Python, GitHub-Java, DeepFix, TSSB-3M, ManySStuBs4J, Juliet-Java, and Juliet-C. Error bars are provided in Appendix B.

Method	**D4J**	**G-Py**	**G-J**	**DF**	**TSSB**	**MS4J**	**J-J**	**J-C**	**Avg.**
Random	0.144	0.100	0.134	0.038	0.069	0.124	0.025	0.058	0.087
DeepFL	0.144	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.144
SmartFL	0.158	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.158
TRANSFER-FL	0.218	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.218
CodeLlama-70B	0.212	0.145	0.316	0.084	0.077	0.169	0.038	0.095	0.142
Llama-3.3-70B	0.269	0.225	0.272	0.092	0.114	0.211	0.072	0.040	0.162
Qwen2.5-72B	0.157	0.333	0.289	0.124	0.088	0.194	0.061	0.040	0.161
DeepSeek-R1-Llama-70B	0.221	0.188	0.218	0.035	0.138	0.185	0.041	0.025	0.131
GPT-4o	0.249	0.375	0.365	0.097	0.089	0.240	0.009	0.026	0.181
Linear Probe Llama-3.2-11B	0.279	0.373	0.300	0.140	0.202	0.235	0.048	0.043	0.202
LLMAO-Llama-3.2-11B	0.144	0.190	0.188	0.078	0.118	0.116	0.063	0.113	0.126
WELL-CodeBERT	0.090	**0.575**	0.532	0.129	0.094	0.111	**0.216**	0.059	0.226
WELL-Llama-3.2-11B	0.236	0.028	0.139	0.000	0.054	0.081	0.000	0.000	0.067
GridLoc-Llama-3.2-11B	0.291	0.498	0.206	0.332	**0.262**	**0.339**	0.158	0.039	0.266
BAP-Llama-3.2-11B	**0.334**	**0.575**	**0.568**	**0.481**	0.237	0.291	0.096	**0.217**	**0.350**

4.1 Datasets

We evaluate on eight datasets summarized below. The dataset summary is in Table 1 and a detailed breakdown is in Appendix A.1.

- **Defects4J**: Bugs along with commits to fix the bug. We use two versions of the dataset: the standard Defects4J v1.2.0 [11] containing 395 bugs, and the additional 543 bugs from Defects4J v3.0.1 released in November 2024, which we use as a stronger evaluation of generalization. In both versions, we split each buggy file into a set of buggy methods and evaluate on the method level following Wu et al. [8].
- **GitHub-Python** [19] and **GitHub-Java** [20]: Code mined from GitHub with syntax errors in Python and Java, respectively.
- **DeepFix** [12]: Real C programs written by students, some containing beginner syntax mistakes.
- **TSSB-3M** [16]: "Simple, stupid bugs" (SStuBs) in Python mined from GitHub and categorized. Despite their name, these bugs are extremely challenging for humans to localize.
- **ManySStuBs4J** [1]: Java SStuBs.
- **Juliet-Java** [21] and **Juliet-C** [22]: Synthetic code corresponding to Common Weakness Enumerations (CWEs). We rename variables and function names to remove indicators of the vulnerability, remove comments, and rename imports that refer to the dataset names. In total, we consider 89 CWEs across both Juliet datasets.

Table 3: Resource efficiency across methods for localizing Defects4J bugs. We measure GPU overhead (GB) and expected inference cost (FLOPs). *BAP* models shown are trained on Llama-3.2 models of their respective sizes

Model	Top-1	GPU (GB)	$\mathbb{E}$[FLOPs]
Llama-3.2-90B	0.269	170.0	3.4e13
CodeLlama-70B	0.212	131.7	7.9e12
Qwen2.5-72B	0.157	138.9	4.2e12
Llama-3.3-70B	0.269	154.0	3.4e13
DeepSeek-R1	0.221	141.2	1.4e14
BAP-1B	0.282	**6.2**	**2.0e9**
BAP-11B	**0.334**	24.2	2.2e10

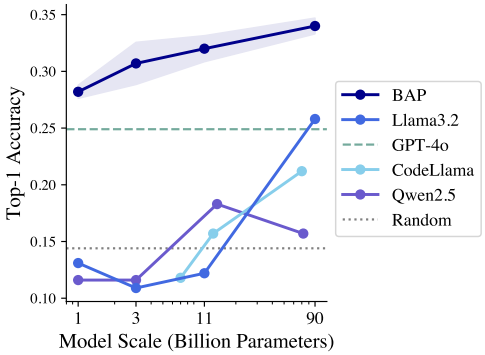


Figure 3: Model scale versus Top-1 on D4J

4.2 Baselines

We group the baselines into three types: traditional FL methods that require code execution, LLM prompting of different models, and LLM probing/training.

Traditional FL Methods For methods requiring code execution, we compare with DeepFL [23], SmartFL [24], and TRANSFER-FL [7] on Defects4J since these are the best performing traditional FL methods. Results for DeepFL and Transfer-FL are from Yang et al. [6], and we cite SmartFL results directly from Zeng et al. [24]. We can not evaluate these benchmarks on the other datasets since they do not provide tests.

Prompting Methods For models, we use a diverse set of four open-weights LLMs of size $\sim$70B as well as a proprietary model. We consider Llama 3.3 70B [25], Qwen 2.5 72B [26], CodeLlama 70B [27], and DeepSeek-R1-Distill-Llama-70B [28] as the main open-weights LLMs and we use GPT-4o [29] as the representative proprietary LLM for prompting experiments. Llama 3.3 70B and Qwen 2.5 72B are LLMs pretrained on a diverse dataset of natural language and code while CodeLlama 70B additionally trained on primarily code data [27]. DeepSeek-R1-Distill-Llama-70B is a "reasoning" model in that it can use longer chain-of-thought output to solve more complex problems [28].

We experimented with several prompts based on that used by Wu et al. [8]. Our prompt asks for the buggy line text as well as the line number in case the LLM cannot count lines. We use this prompt with temperature 0 sampling for all models. The prompt is provided in Appendix A.2.

Probing Methods We evaluate the following LLM probing baselines:

- Linear Probing: Logistic regression on the last-token representation to predict bugs. Following Zou et al. [30], we extend predictions to token-level scores and aggregate to line rankings as in *BAP*.
- GridLoc [18]: An RNN-based attention probing method. We train it for bug detection on $\mathcal{H}_{\text{Det}}$ and interpret its attention weights with our line-aggregation method.
- LLMAO [6]: Adapter trained with strong FL supervision, originally for CodeGen [31]. We use the largest CodeGen model and adapt their code to Llama-3.2-11B.
- WELL [9]: Learns FL without strong supervision by finetuning CodeBERT [32] (a bidirectional attention LLM) for bug detection and interpreting last-layer attention. We also report results on Llama-3.2-11B despite its causal attention.

4.3 RQ1: FL Performance

For FL, given a program fragment with bugs, each method ranks the lines of the input program based on the likeliness of the bug being located to that line. We note that our method was not trained with direct localization information and instead makes use of weak supervision even though baselines (such as LLMAO) use strong FL supervision. WELL is the only baseline which uses weak supervision, similar to our method [9].

We compare the size of the model which *BAP* is trained on to the model's zero-shot FL capabilities in Figure 3. We focus on top-1 accuracy for Defects4J since it is the most widely used FL benchmark.

FL performance of *BAP* and baselines on eight datasets is shown in Table 2. For comparison with prior work on Defects4J, we use 10-fold cross validation with method-level context following Wu et al. [8]. Results for DeepFL, TRANSFER-FL, and Smart-FL are taken from Yang et al. [6], Zeng et al. [24]. "N/A" indicates baselines requiring tests only available for Defects4J. *BAP* improves over the strongest baseline by 34.6% top-1 accuracy on average, including gains of 19.7% on Defects4J v1.2.0 and 128% on Juliet-C. The next best method is GridLoc, a probing method we adapted.

Training classifiers with weak supervision from fault detection is insufficient for localization: WELL with Llama-3.2-11B performs worse than random guessing, nearly always predicting the first line, which is rarely buggy. This issue does not occur with CodeBERT, which uses bidirectional attention.

Prompting $\sim$70B-parameter LLMs outperforms traditional FL methods requiring code execution. GPT-4o (rumored $>$200B) performs comparably to Llama-3.3-70B, while DeepSeek-R1-Distill-Llama-70B underperforms its base model, producing overly cautious $\sim$1000-token chain-of-thoughts that flag benign lines as buggy. Section 4.4 discusses scaling trends of prompting and probing.

4.4 RQ2: Efficiency

Figure 3 compares *BAP* trained on a 1B LLM with zero-shot prompting of larger models, focusing on Defects4J top-1 accuracy. *BAP* consistently outperforms zero-shot prompting—even for 70B+ models—though the margin narrows as model size increases.

While *BAP* scales linearly with log model size, prompting shows "emergent behavior" [33]. To test limits, we scaled as far as resources allowed, evaluated GPT-4o via API, and tested DeepSeek-R1-Distill-Llama-70B with test-time scaling; none exceeded *BAP*. Surprisingly, the DeepSeek reasoning model performed worse than its base Llama-3.3-70B.

4.5 RQ3: Localizing Multi-line Bugs

Since our method produces a ranking for every line of the input, our method is better suited for directly finding multiple bugs at once, or multi-line bugs. Since the top-k accuracy metric only cares if at least one of the top k predictions are correct, we additionally use the precision at k (P@k) metric which measures the percent of true buggy lines in the top k predictions. The exact formula is provided in Appendix C. We compare *BAP* and baselines on Defects4J v1.2.0 in terms of P@k in Table 4.

4.6 RQ4: New Bug and Length Generalization

Apart from efficiency as discussed above, we investigate several differences between *BAP* and zero-shot prompting of the underlying model.

New Bug Generalization We evaluate *BAP* compared to LLM prompting and probing on 543 new bugs from Defects4J v3.0.1 in Table 5. Our method and probing baselines are only trained on Defects4J v1.2.0, so this serves as an unbiased evaluation on the ability of these methods to generalize to new bugs. Our method, as well as the baselines, drop in performance for the new bugs, but *BAP* maintains the highest top-k accuracy (14.4% increase in top-1 over the strongest baseline).

Context Length Generalization We compare the behavior of *BAP* to prompting in terms of length generalization in Figure 4. We see that *BAP* outperforms zero-shot prompting of various LLMs across context lengths from 10 to 50 lines. On code fragments of 60 lines and longer, all methods perform near random, making fault localization on code over 50 lines a challenging, but valuable task for future work. We visualize the output of *BAP* on two examples in Figure 5.

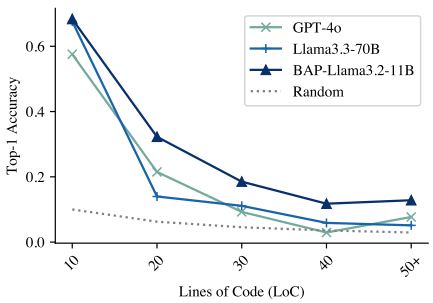


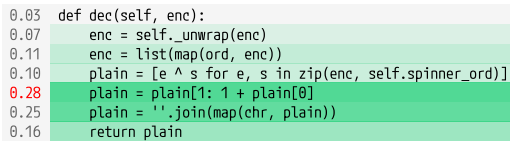
Figure 4: Top-1 accuracy versus context length, measured by lines of code (LOC) on Defects4J. We compare *BAP*-Llama3.2-11B against models at least six times larger.

Table 4: Precision@K for multi-line bugs. Evaluated on a subset of Defects4J where functions contain two or more buggy lines.

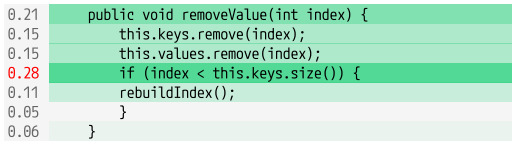
Method	P@2	P@3	P@5
Random	0.201	0.231	0.297
CodeLlama-70B	0.250	0.284	0.351
Llama-3.3-70B	0.240	0.266	0.355
Qwen2.5-72B	0.221	0.271	0.347
DeepSeek-R1DL-70B	0.245	0.283	0.336
GPT-4o	0.218	0.288	0.359
BAP-Llama-3.2-11B	**0.289**	**0.298**	**0.367**

Table 5: Comparison with LLMAO on 437 new bugs from Defects4J v3.0.1.

	Defects4J v3.0.1		
Method	Top-1	Top-3	Top-5
Random	0.166	0.377	0.512
CodeLlama-70B	0.152	0.276	0.326
Llama-3.3-70B	0.215	0.416	0.528
Qwen-2.5-72B	0.161	0.395	0.515
GPT-4o	0.181	0.451	0.579
Linear Probe Llama-3.2-11B	0.156	0.409	0.557
LLMAO-Llama-3.2-11B	0.174	0.389	0.515
GridLoc-Llama-3.2-11B	0.165	0.377	0.512
BAP-Llama-3.2-11B	**0.246**	**0.459**	**0.583**



(a) Python syntax error



(b) Defects4J bug

Figure 5: Examples of bug localization with *BAP* on two evaluation set samples. We visualize the line-level weights from *BAP* above such that lines highlighted in a darker color have higher weights. *BAP* correctly identifies bug locations at Top-1.

5 Related Work

We survey related work in FL techniques and LLM probing.

Automated Fault Localization. Methods for FL include the traditional spectrum-based (SBFL) and mutation-based (MBFL) methods which require executable code and deep-learning based approaches [34]. SBFL methods are simple but have low accuracy while MBFL and deep learning approaches have higher accuracy at larger computational cost [34]. Various deep learning approaches combine SBFL and MBFL with semantic features from deep models [23, 7, 24]. Recently, LLMs have significantly outperformed SBFL and MBFL approaches on FL on the method level [8]. Prompting and agent-based systems can even perform repository-level FL [35, 36], but they must reduce the problem to method-level FL [35]. LLMAO [6] trains an adapter on an LLM from strong FL supervision to perform FL without executable tests, and WELL [9] finetunes an LLM on bug detection supervision and interprets the attention for FL. Unlike these approaches, our method uses LLM probing, and we leverage bug detection supervision to scale to more available data.

Probing LLMs. Probing is useful tool in LLM interpretability. There is extensive work on probing LLMs, most notably BERT [37], to understand what linguistic knowledge it encodes. Hewitt and Manning [38] design a probe for eliciting natural language syntax parse trees from BERT, and Hernández López et al. [39] probe for the code abstract syntax trees. These probes are usually trained on a fixed size input [30], but pooling sequence representations using global weights [17] and sample-conditional weights [18] have been studied. Unlike these approaches, we adopt a traditional Transformer layer as our probe where the attention module learns to pool the input tokens.

6 Conclusion

In this paper, we approach the problem of scalable FL, and propose a method for achieving state-of-the-art FL performance at a fraction of the model inference cost by training on available data without strong FL supervision. Existing methods for fault localization either require executable code, test cases, finegrained line-level supervision, or resource intensive LLMs. To this end, we propose Bug Attention Probe (*BAP*), an LLM probing method which uses an attention mechanism to learn to

localize bugs from only coarse-grained bug detection supervision. Using a suite of eight diverse FL benchmarks, we demonstrate that *BAP* significantly outperforms existing fault localization techniques as well as LLM prompting of models over ten times larger than that used by our probe. We also identify avenues for future research including FL on long-code samples (over 50 lines), creation of more bug detection datasets by running existing bug detectors on large repositories of code, and execution-free FL on the file and project-level.

Acknowledgements

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-2236662 and the Google Research Fellowship.

References

[1] Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 573–577, 2020.

[2] Kevin Jesse, Toufique Ahmed, Premkumar T Devanbu, and Emily Morgan. Large language models and simple, stupid bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 563–575. IEEE, 2023.

[3] Iris Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.

[4] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016. doi: 10.1109/TSE.2016.2521368.

[5] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013. doi: 10.1109/ICSE.2013.6606613.

[6] Aidan Z. H. Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. Large language models for test-free fault localization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3623342. URL `https://doi.org/10.1145/3597503.3623342`.

[7] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1169–1180, 2022.

[8] Yonghao Wu, Zheng Li, Jie M Zhang, Mike Papadakis, Mark Harman, and Yong Liu. Large language models in fault localisation. *arXiv preprint arXiv:2308.15276*, 2023.

[9] Huangzhao Zhang, Zhuo Li, Jia Li, Zhi Jin, and Ge Li. Well: Applying bug detectors to bug localization via weakly supervised learning. *Journal of Software: Evolution and Process*, 36(9): e2669, 2024. doi: https://doi.org/10.1002/smr.2669. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2669`.

[10] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating & improving fault localization techniques. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-16-08-03*, page 27, 2016.

[11] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.

[12] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, volume 31, 2017.

[13] Guillaume Alain and Yoshua Bengio. Understanding intermediate layers using linear classifier probes. *arXiv preprint arXiv:1610.01644*, 2016.

[14] Tian Yu Liu, Matthew Trager, Alessandro Achille, Pramuditha Perera, Luca Zancato, and Stefano Soatto. Meaning representations from trajectories in autoregressive models. *arXiv preprint arXiv:2310.18348*, 2023.

[15] Thomas Hirsch. A fault localization and debugging support framework driven by bug tracking data. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 139–142. IEEE, 2020.

[16] Cedric Richter and Heike Wehrheim. Tssb-3m: Mining single statement bugs at massive scale. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 418–422, 2022.

[17] Ian Tenney, Patrick Xia, Berlin Chen, Alex Wang, Adam Poliak, R Thomas McCoy, Najoung Kim, Benjamin Van Durme, Samuel R Bowman, Dipanjan Das, et al. What do you learn from context? probing for sentence structure in contextualized word representations. In *International Conference on Learning Representations*, 2018.

[18] Jingcheng Niu, Wenjie Lu, and Gerald Penn. Does bert rediscover a classical nlp pipeline? In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 3143–3153, 2022.

[19] Michihiro Yasunaga and Percy Liang. Break-it-fix-it: Unsupervised learning for program repair. In *International conference on machine learning*, pages 11941–11952. PMLR, 2021.

[20] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. Syntax and sensibility: Using language models to detect and correct syntax errors. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 311–322. IEEE, 2018.

[21] Juliet java 2023, 2023. URL `https://samate.nist.gov/SARD/test-suites/111`.

[22] Juliet c/c++ 2023, 2023. URL `https://samate.nist.gov/SARD/test-suites/112`.

[23] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 169–180, 2019.

[24] Muhan Zeng, Yiqian Wu, Zhentao Ye, Yingfei Xiong, Xin Zhang, and Lu Zhang. Fault localization via efficient probabilistic modeling of program semantics. In *Proceedings of the 44th International Conference on Software Engineering*, pages 958–969, 2022.

[25] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[26] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[27] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[28] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[29] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

[30] Andy Zou, Long Phan, Sarah Chen, James Campbell, Phillip Guo, Richard Ren, Alexander Pan, Xuwang Yin, Mantas Mazeika, Ann-Kathrin Dombrowski, et al. Representation engineering: A top-down approach to ai transparency. *arXiv preprint arXiv:2310.01405*, 2023.

[31] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=iaYcJKpY2B_`.

[32] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL `https://aclanthology.org/2020.findings-emnlp.139/`.

[33] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022.

[34] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[35] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

[36] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. Agentfl: Scaling llm-based fault localization to project-level context. *arXiv preprint arXiv:2403.16362*, 2024.

[37] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL `https://aclanthology.org/N19-1423`.

[38] John Hewitt and Christopher D Manning. A structural probe for finding syntax in word representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4129–4138, 2019.

[39] José Antonio Hernández López, Martin Weyssow, Jesús Sánchez Cuadrado, and Houari Sahraoui. Ast-probe: Recovering abstract syntax trees from hidden representations of pre-trained language models. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–11, 2022.

[40] Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

A Additional Experimental Details

A.1 Datasets

The detailed breakdown of each of the datasets we used, other than Defects4J. We split datasets into groups based on the three domains of syntax, single line bugs, and vulnerabilities and show the breakdown in Table 6, 7, and 8 respectively.

Table 6: Number of samples in syntax datasets.

Subtype	GitHub-Python		GitHub-Java		DeepFix	
	Train	Test	Train	Test	Train	Test
Correct syntax	1323	400	1370	460	1475	365
Mismatched parentheses	400	100	110	100	400	100
Mismatched bracket	368	100	60	60	81	31
Mismatched brace	155	100	400	100	195	34
Missing semicolon	—	—	400	100	400	100
Python-specific	400	100	—	—	—	—
Java-specific	—	—	400	100	—	—
C-specific	—	—	—	—	400	100
Total	2646	800	2740	920	2950	730

Table 7: Number of samples in SStuBs datasets.

Subtype	TSSB		ManySStuBs	
	Train	Test	Train	Test
No bug	3745	1080	3821	1093
Change identifier	400	100	400	100
Change numeral	400	100	400	100
Change binary operator	400	100	400	100
Change unary operator	202	100	301	100
Less specific if	281	100	400	100
More specific if	400	100	400	100
Same function less args	400	100	266	100
Same function more args	400	100	400	100
Swap arguments	81	80	94	93
Swap boolean literal	381	100	360	100
Python specific	400	100	—	—
Java specific	—	—	400	100
Total	7830	2184	7642	2186

A.2 Few-shot Prompts

For all prompting experiments, we use zero-shot prompting with the prompt given below:

```
Q: Please analyze the following code snippet for potential bugs.
    Return the fault
localization result in JSON format, consisting of five JSON
    objects called
"faultLocalization". These "faultLocalization" objects correspond
    to the top
five most suspicious lines of code. Each "faultLocalization"
    contains two
fields: 1) "codeContent" string which contains the line of code
    that corresponds
```

Table 8: Number of samples in security vulnerabilities datasets.

CWE Class	Juliet-Java		Juliet-C	
	Train	Test	Train	Test
Access Control	677	178	609	159
Comparison	38	11	14	4
Concurrency	21	11	172	48
Encryption	616	154	278	72
Exposed Resource	574	141	934	236
File Handling	1056	256	—	—
Implementation	104	27	144	37
Improper Check or Handling of Exceptional Conditions	37	10	604	152
Improper Input Validation	523	128	304	76
Improper Neutralization	—	—	60	16
Incorrect Calculation	40	10	99	27
Injection	2368	571	304	76
Insufficient Control Flow Management	161	49	259	69
Memory Safety	342	86	847	217
Poor Coding Practices	665	175	1820	465
Protection Mechanism Failure	—	—	28	8
Randomness	52	14	14	4
Resource Control	534	126	—	—
Resource Lifecycle Management	80	22	841	215
Sensitive Information Exposure	112	31	84	24

```
to suspicious code in the snippet and 2) "lineNumber" integer
    which indicates the
line number of this suspicious code. Output just the JSON objects
"faultLocalization" and NOTHING ELSE.
```
{code-example}
```

### A.3 Hyperparameters

For *BAP*, we trained for 30 epochs with a learning rate of $1e{-}4$, a batch size of 16, and weight decay of 1 for all datasets except for the TSSB dataset where we needed to use less training epochs to avoid overfitting. For TSSB, we trained for 5 epochs with a learning rate of $1e{-}4$, a batch size of 16, and weight decay of 1. For the architecture of *BAP*, we used grouped query attentino with 32 query heads and 8 key-value heads to match the architecture of the Llama-3.2-11B attention mechanism. For the ablation of Llama-3.2-90B, we used 64 query heads with 8 key-value heads.

For the linear probing baseline and GridLoc, we trained for 30 epochs with a learning rate of $1e{-}4$, a batch size of 16, and weight decay of 0.1 for all datasets.

Parameters were chosen by splitting the training set with an 80/20 split into train and validation samples, and selecting hyperparameters from the results on the validation set.

### A.4 Compute resources

All experiments are conducted on a server with 96 Intel Xeon Gold 6248R CPUs, each with a clock speed of 3.00 GHz, and 8 NVIDIA A100 GPUs, each with a capacity of 40GB.

The WELL baseline is the most compute intensive of the methods we explore because it requires fintuning an LLM. To finetune Llama-3.2-11B, we had to use LoRA [40] with rank 16 to make training this model accessible. GridLoc takes around twice the training time as *BAP*, and all the other baselines take 2-3 minutes for one training run on a single dataset.
```

Table 9: Comparison of *BAP* with existing fault localization methods across eight diverse bug benchmarks. We evaluate each method on line-level localization performance at the method-level, measured by top-1 localization accuracy. From left-to-right: Defects4J v1.2.0, GitHub-Python, GitHub-Java, DeepFix, TSSB-3M, ManySStuBs4J, Juliet-Java, and Juliet-C.

Method	**D4J**	**GH-Py**	**GH-J**	**DeepFix**	**TSSB**	**MS4J**	**Juliet-J**	**Juliet-C**
Random	0.144	0.100	0.134	0.038	0.069	0.124	0.025	0.058
DeepFL	0.144	N/A	N/A	N/A	N/A	N/A	N/A	N/A
SmartFL	0.158	N/A	N/A	N/A	N/A	N/A	N/A	N/A
TRANSFER-FL	0.218	N/A	N/A	N/A	N/A	N/A	N/A	N/A
CodeLlama-70B	0.212	0.145	0.316	0.084	0.077	0.169	0.038	0.095
Llama-3.3-70B	0.269	0.225	0.272	0.092	0.114	0.211	0.072	0.040
Qwen2.5-72B	0.157	0.333	0.289	0.124	0.088	0.194	0.061	0.040
DeepSeek-R1-Distill-Llama-70B	0.221	0.188	0.218	0.035	0.138	0.185	0.041	0.025
GPT-4o	0.249	0.375	0.365	0.097	0.089	0.240	0.009	0.026
Linear Probe Llama-3.2-11B	0.279±0.02	0.373±0.01	0.300±0.01	0.140±0.01	0.202±0.01	0.235±0.01	0.048±0.00	0.043±0.01
LLMAO-CodeGen	0.223	N/A	N/A	N/A	N/A	N/A	N/A	N/A
LLMAO-Llama-3.2-11B	0.144	0.190		0.078				
WELL-CodeBERT	0.090	**0.575**	0.532	0.129	0.094	0.111	**0.216**	0.059
WELL-Llama-3.2-11B	0.236	0.028	0.139	0.000	0.054	0.081	0.000	0.000
GridLoc-Llama-3.2-11B	0.291±0.02	0.498±0.08	0.206±0.08	0.332±0.03	**0.262±0.03**	**0.339±0.03**	0.158±0.04	0.039±0.01
BAP-Llama-3.2-11B	**0.334±0.02**	**0.575±0.02**	**0.568±0.01**	**0.481±0.04**	0.237±0.02	0.291±0.04	0.096±0.03	**0.217±0.00**

B Additional Results

B.1 Error Bars for FL Results

Error bars for the results in Table 2 are provided in Table 9. The prompting methods have no error bars because we use greedy decoding.

C Precision at k

The precision at k (P@k) metric which we use is calculated as:

$$\frac{\text{Correct in top } k}{\min(k, \text{Max possible correct})}.$$

We use the min in the denominator to account for the case where the number of buggy lines is much fewer than k. This is practically relevant since many bugs consist of only 2-3 buggy lines which is less than k for P@5.