Appendix

A DATASET

In this section, we provide additional information about the dataset used in our experiments, which is not presented in the main text due to space limitations.

A.1 ClassEval

ClassEval [1] is a challenging class-level code generation benchmark including 100 class-level programming tasks. We present the benchmark format of ClassEval and method-by-method generation strategies as follows. Each problem in ClassEval provides a class skeleton as input. Figure 1 illustrates an example of a class skeleton in ClassEval. The class skeleton serves as a structured blueprint for the target class, containing both class-level information (import statements, class name, class description, and class constructor) and method-level information (method signature, functional description, parameter/return descriptions, and example input/outputs). In evaluation, models are asked to generate the implementation of unfinished methods to complete the whole class.

We evaluate CodeFast in ClassEval using method-by-method generation strategies, which are best generation strategies for models that are not GPT-3.5 and GPT-4, according to conclusions from ClassEval. ClassEval introduces two method-by-method generation strategies: incremental generation strategy and compositional generation strategy. In the incremental generation strategy, the model is asked to generate the class in a method-by-method manner. Each iteration is based on the method bodies that have been generated in previous iterations. The iterative process repeats until all methods in the class are generated. In the compositional generation strategy, the model is asked to complete each unfinished method in the class skeleton independently and assemble them to form the class at last. In our evaluation, we utilize both incremental generation strategy and compositional generation strategy for evaluation.

A.2 Manually labeled dataset

We manually construct a labeled dataset to evaluate the prediction accuracy of five GenGuard classifiers. Since the datasets (HumanEval, MBPP, etc.) only provide test cases without providing ground truth code, we have no labels to evaluate the classifier's prediction performance. For each evaluated Code LLM, we sample 100 generations from test sets of MBPP and MBXP, which are not used for training the five classifiers. Then, three annotators, including two authors and one Ph.D. student of our department are recruited to manually label this data (500 samples in total). Given prompts and raw code, annotators independently truncate the raw code without changing its functionality. We then collect the line numbers where the code is truncated as labels. We verify the agreement of annotators using Krippendorff's alpha [2], and the value of 0.994 shows high agreement among annotators.

B EVALUATION

B.1 RQ1: Effectiveness

In this section, we supplement the experimental results that are not presented in the main text due to space limitations.

```
from datetime import datetime                                          Import Statements
class VendingMachine:                                                  Class Name
    """This is a class to simulate a vending machine, including adding products, inserting
coins, purchasing products, viewing balance, replenishing product inventory, and
displaying product information. """                                    Class Description
  def __init__(self):
    """
    Initializes the vending machine's inventory and balance.
    """                                                                Class Constructor
    self.inventory= []
    self. balance= {}
  def purchase_item(self, item_name):                                  Method Signature
    """ Purchases a product from the vending machine and returns the balance after the
purchase.                                                              Functional Description
    :param item_name: str, the name of the product to be purchased, which should be in
the vending machine.
    :return: If successful, returns the balance of the vending machine after the product is
purchased, float, if the product is out of stock,returns False.        Parameter/Return
    >>> vendingMachine.inventory = {'Coke': {'price': 1.25, 'quantity': 10}}  Description
    >>> vendingMachine.balance = 1.25
    >>> vendingMachine.purchase_item('Coke')
    0.0
    >>> vendingMachine.inventory
    {'Coke': {'price': 1.25, 'quantity': 9}} """                       Example Input/Output
  def restock_item(self, item_name, quantity):                         Method Signature
    """                                                                Functional Description
    Replenishes the inventory of a product already in the vending machine.
    :param item_name: The name of the product to be replenished, str, which should be
in the vending machine.
    :param quantity: The quantity of the product to be replenished, int, which is greater
than 0.
    :return: If the product is already in the vending machine, returns True, otherwise,
returns False.                                                         Parameter/Return Description
    >>> vendingMachine.inventory = {'Coke': {'price': 1.25, 'quantity': 10}}
    >>> vendingMachine.restock_item('Coke', 10)
    True
    >>> vendingMachine.inventory
    {'Coke': {'price': 1.25, 'quantity': 20}}                          Example Input/Output
    """
  ...
```

Figure 1: An example of a class skeleton in ClassEval.

Table 1: Prediction accuracy of five GenGuard modules.

Classifier Type	Precision	Recall
GenGuard (CodeLlama-7B)	0.961	0.987
GenGuard (CodeLlama-13B)	0.998	0.973
GenGuard (CodeLlama-34B)	0.998	0.976
GenGuard (StarCoder-15.5B)	0.999	1.000
GenGuard (Phind-CodeLlama)	0.996	0.987
Average	**0.990**	**0.985**

Overall performance with Pass@3 and Pass@5. The results with Pass@3 and Pass@5 metrics are listed in Table 2. The experimental results show that our method can effectively maintain the Pass@3 and Pass@5 accuracy of generated code.

Prediction performance of GenGuard classifiers. We evaluate the prediction performance of GenGuard classifiers with metric Precision and Recall utilizing the dataset presented in Section A.2. Experimental results in Table 1 show that the GenGuard classifier demonstrates a high recall of 0.99 and precision values of 0.985.

Case studies. Figure 2 presents a generation example from the Phind-CodeLlama$_{\text{CodeFast}}$. After inputting the prompt (lines 1-12), the Phind-CodeLlama$_{\text{CodeFast}}$ generates a code line that fulfills requirements (lines 13). In Line 14, the model generates a closing

: Prompt : Continue Generation : Stop Generation

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  /**
4  * Write a c++ function to find the volume of a triangular
     prism.
5  * > findVolume(10, 8, 6)
6  * 240
7  * > findVolume(3, 2, 2)
8  * 6
9  * > findVolume(1, 2, 1)
10 * 1
11 */
12 int findVolume(int l, int b, int h) {
13     return (l * b * h) / 2;
14 }
```

Figure 2: A generation example of Phind-CodeLlama$_{\text{CodeFast}}$ in MBCPP.

: Prompt : Continue Generation : Stop Generation

```
1  def max_occurrences(nums)
2      """
3      Write a python function to find the item with
       maximum occurrences in a given list.
4      assert max_occurrences([1,2,3,1,2,3,12,4,2]) == 2
5      """
6      # Your code here
7      # max_occurrences([1,2,3,1,2,3,12,4,2])
8      # max_occurrences([1,2,3,1,2,3,12,4,2])
```

Figure 3: A generation example of CodeLlama-34B$_{\text{CodeFast}}$ in MBPP.

brace "}" to end the program. At this time, the GenGuard module predicts to stop the generation of the model.

Figure 3 presents a generation example from the CodeLlama-34B$_{\text{CodeFast}}$. After inputting the prompt (lines 1-6), the Phind-CodeLlama$_{\text{CodeFast}}$ generate comment in lines 13. However, in line 14, the model repeats the same comment as in line 13. At this point, GenGuard predicts to stop and avoid the generation of excess tokens.

B.2 RQ2: Ablation study

The experimental results, including metrics for Pass@3 and Pass@5, are presented in Table 3. It is observed that Code LLMs enhanced with the Multi-PL GenGuard achieve similar Pass@3 and Pass@5 scores, as well as comparable speed acceleration, to those utilizing the Mono-PL GenGuard module. Furthermore, Table 3 demonstrates that removing the line voting mechanism results in a decline in both Pass@3 and Pass@5 performance. In summary, GenGuard trained on Multi-PL data performs comparably to that trained on Mono-PL data. Additionally, the line voting mechanism effectively improves the accuracy of the generated code.

B.3 RQ3: Stability

In this section, we supplement experiment results with Pass@3 and Pass@5 and experiments investigating the robustness of our method.

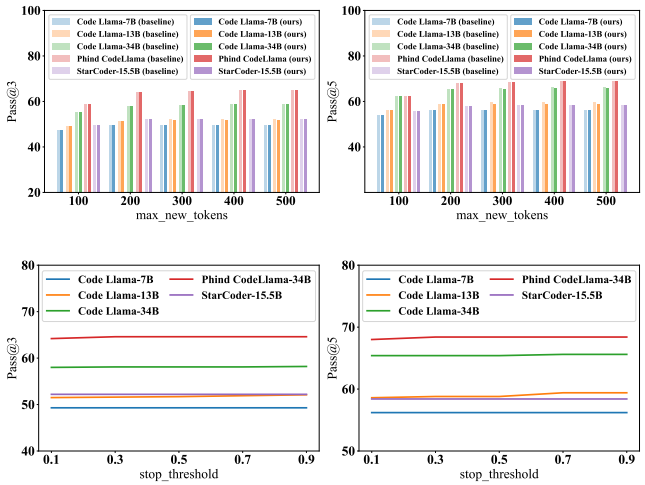


Figure 4: Stability analysis of CodeFast in MBPP.

Overall results. In RQ3, we explore the stability of our method when two hyperparameters vary: *max_new_tokens* and *stop_threshold*. We present the experimental results for Pass@3 and Pass@5 on the MBPP dataset in Figure 4. For the MBGP dataset, results are shown in Figure 7 and Figure 8. For MBCPP, results are displayed in Figure 9 and Figure 10, and for MBJSP, they are in Figure 5 and Figure 6. The results demonstrate that CodeFast exhibits great stability in different parameter settings across different programming languages.

Investigation on the robustness of CodeFast. To explain why CodeFast show stable performance across various stop threshold values, we investigate stopping prediction probability distribution of GenGuard when predicting true excess token and true expected token. Experiments are conducted in the dataset presented in Section A.2. First, from Table 4, we can observe that the average stopping prediction probability is more than 0.9 among five classifiers (0.976 on average) when predicting excess tokens and less than 0.1 when predicting expected tokens. Second, we demonstrate the distribution of stopping prediction probability intuitively by plotting its probability density distribution with Kernel Density Estimation [2]. From Figure 11, we can observe that stopping probability is distributed between 0.9 and 1 when predicting true excess tokens, and between 0 and 0.1 when predicting true expected tokens. These observations indicate that GenGuard is confident in stopping prediction. Therefore, our approach is robust and can tolerate a wide range of stop thresholds.

B.4 RQ4: Generalizability

To further validate the generalizability of CodeFast, we evaluate the performance of five GenGuard-enhanced models using three datasets in different PLs (Python, JavaScript, and Go) from the multilingual HumanEval benchmark. Detailed results with time consumption and average lengths are listed in Table 5. We can observe that after being enhanced by the GenGuard module, the generation speed of all five Code LLMs increases across three programming languages. Notably, this improvement is achieved while maintaining Pass@3 and Pass@5 scores, indicating that CodeFast

Table 2: Performance of CodeFast on 5 Code LLMs across 4 benchmarks. P@3 and P@5 are short for Pass@3 and Pass@5 respectively. Time represents the average time (in seconds) taken by a Code LLM to generate result per sample.

Model	MBPP (Python)				MBJSP (JavaScript)				MBGP (Go)				MBCPP (C++)			
	P@3	P@5	Time	Speedup	P@3	P@5	Time	Speedup	P@3	P@5	Time	Speedup	P@3	P@5	Time	Speedup
CodeLlama-7B	49.4	56.2	9.36	×1.00	51.7	58.6	8.81	×1.00	43.7	50.6	7.1	×1.00	52.2	57.8	8.44	×1.00
CodeLlama-7B$_{\text{CodeFast}}$	49.4	56.2	2.07	**×4.52**	51.3	58.6	3.36	**×2.62**	43.1	50.2	3.42	**×2.07**	52.2	57.8	3.17	**×2.66**
CodeLlama-13B	52.1	59.4	11.87	×1.00	57.0	64.6	11.92	×1.00	45.2	51.8	11.94	×1.00	57.1	63.0	11.70	×1.00
CodeLlama-13B$_{\text{CodeFast}}$	51.7	58.8	2.98	**×3.98**	56.3	63.8	4.31	**×2.77**	45.1	51.6	4.38	**×2.73**	57.2	63.4	4.00	**×2.93**
CodeLlama-34B	58.3	65.6	14.67	×1.00	63.7	69.2	8.37	×1.00	51.5	57.0	14.51	×1.00	62.8	68.6	10.12	×1.00
CodeLlama-34B$_{\text{CodeFast}}$	58.1	65.4	3.62	**×4.05**	63.1	69.0	5.58	**×1.50**	51.1	56.4	5.89	**×2.46**	62.8	68.6	4.95	**×2.04**
StarCoder-15.5B	52.2	58.2	5.14	×1.00	40.9	47.8	6.42	×1.00	37.1	43.4	7.17	×1.00	51.7	58.4	6.15	×1.00
StarCoder-15.5B$_{\text{CodeFast}}$	52.2	58.2	1.98	**×2.59**	40.5	47.2	1.56	**×4.11**	37.2	43.8	2.88	**×2.49**	51.7	58.6	2.55	**×2.41**
Phind-CodeLlama	64.6	68.4	4.89	×1.00	66.7	70.4	8.34	×1.00	49.8	53.8	14.19	×1.00	66.0	69.4	10.04	×1.00
Phind-CodeLlama$_{\text{CodeFast}}$	64.6	68.4	3.65	**×1.34**	66.7	70.4	5.79	**×1.44**	49.8	53.8	5.78	**×2.46**	66.0	69.4	5.12	**×1.96**

Table 3: Ablation study results.

Model	Type	MBPP (Python)			MBJSP (JavaScript)			MBGP (Go)			MBCPP (C++)		
		Pass@3	Pass@5	Speedup	Pass@3	Pass@5	Speedup	Pass@3	Pass@5	Speedup	Pass@3	Pass@5	Speedup
CodeLlama-7B	CodeFast	49.4	56.2	×4.52	51.3	58.6	×2.62	43.1	50.2	×2.07	52.2	57.8	×2.66
	w/o MultiPL	49.4	56.2	×4.67	51.4	58.6	×2.60	43.3	50.2	×2.16	52.2	57.8	×2.72
	w/o LineVoting	48.9	55.6	×5.81	51.6	58.0	×3.49	42.5	49.8	×2.29	51.7	57.4	×2.78
CodeLlama-13B	CodeFast	51.7	58.8	×3.98	56.3	63.8	×2.77	45.1	51.6	×2.73	57.2	63.4	×2.93
	w/o MultiPL	52.0	59.2	×4.03	56.9	64.4	×2.76	45.1	51.8	×2.71	57.4	63.4	×2.93
	w/o LineVoting	49.3	56.8	×5.41	55.2	62.8	×3.83	43.9	50.4	×3.09	56.2	62.4	×3.16
CodeLlama-34B	CodeFast	58.1	65.4	×4.05	63.1	69.0	×1.50	51.1	56.4	×2.46	62.8	68.6	×2.04
	w/o MultiPL	58.0	65.4	×4.08	63.4	68.8	×1.49	51.4	57.0	×2.46	62.8	68.6	×2.02
	w/o LineVoting	57.7	65.2	×5.13	64.1	68.4	×2.04	51.0	56.4	×2.69	63.2	69.0	×2.21
StarCoder-15.5B	CodeFast	52.2	58.2	×2.59	40.5	47.2	×4.11	37.2	43.8	×2.49	51.7	58.6	×2.41
	w/o MultiPL	52.2	58.2	×2.59	40.6	47.4	×4.11	37.2	43.8	×2.09	51.7	58.4	×2.41
	w/o LineVoting	48.3	54.0	×6.21	38.8	45.8	×4.89	36.7	43.0	×2.87	50.9	57.8	×2.68
Phind-CodeLlama	CodeFast	64.6	68.4	×1.34	66.7	70.4	×1.44	49.8	53.8	×2.46	66.0	69.4	×1.96
	w/o MultiPL	64.6	68.4	×1.33	66.6	70.2	×1.41	50.0	54.0	×2.47	66.0	69.4	×1.94
	w/o LineVoting	62.4	66.2	×1.72	65.4	68.8	×1.92	49.1	53.0	×2.94	64.3	67.6	×2.16

Table 4: Average stopping prediction probability of five classifiers for true excess and expected tokens. Here, P_{stop}(excess) and P_{stop}(expected) denote the prediction probabilities for true excess and expected tokens, respectively.

Classifier Type	P_{stop}(excess)	P_{stop}(expected)
GenGuard(CodeLlama-7B)	0.966	0.008
GenGuard(CodeLlama-13B)	0.964	0.004
GenGuard(CodeLlama-34B)	0.970	0.001
GenGuard(StarCoder-15.5B)	0.998	0.002
GenGuard(Phind-CodeLlama)	0.980	0.003
Average	**0.976**	**0.003**

exhibits strong generalization capabilities, achieving robust performance on untrained datasets. Additionally, we note a significant reduction in the length of the generated token sequences, indicating that CodeFast can effectively decrease the generation of excess tokens.

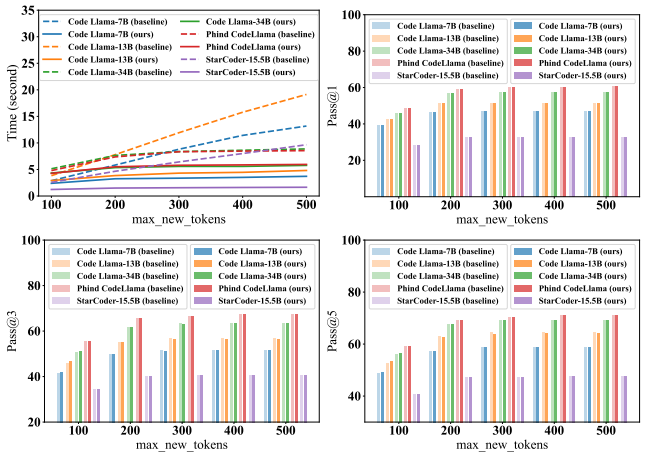


Figure 5: Stability analysis of CodeFast about max_new_tokens on MBJSP.

Table 5: Performance of CodeFast in untrained datasets. Python, JavaScript, and Go are short for HumanEval, HumanEval-JavaScript, and HumanEval-Go, respectively. Time represents the average time (in seconds) taken by a Code LLM to generate result per sample.

Model	HumanEval					HumanEval (JavaScript)					HumanEval (Go)				
	P@3	P@5	Time	Speedup	Length	P@3	P@5	Time	Speedup	Length	P@3	P@5	Time	Speedup	Length
CodeLlama-7B	41.0	48.8	8.91	×1.00	286.7	40.6	49.1	8.46	×1.00	275.9	27.8	33.1	8.72	×1.00	290.8
CodeLlama-7B$_{\text{CodeFast}}$	41.0	48.8	3.29	**×2.71**	106.1	40.6	49.1	4.26	**×1.99**	135.9	27.7	33.1	4.13	**×2.11**	131.4
CodeLlama-13B	44.1	51.8	11.65	×1.00	300	40.7	48.4	11.64	×1.00	300	25.6	30.0	12.05	×1.00	299.5
CodeLlama-13B$_{\text{CodeFast}}$	43.7	51.2	4.23	**×2.75**	104.9	40.4	48.4	4.69	**×2.48**	119.1	25.6	30.0	4.77	**×2.53**	116.8
CodeLlama-34B	57.3	67.1	11.23	×1.00	201.6	51.7	60.2	7.75	×1.00	139.0	32.1	38.1	15.08	×1.00	270.4
CodeLlama-34B$_{\text{CodeFast}}$	57.0	67.1	4.62	**×2.43**	80.6	52.0	60.9	5.62	**×1.38**	99.6	32.1	38.1	7.05	**×2.17**	122.6
StarCoder-15.5B	41.2	47.6	6.22	×1.00	240.9	15.5	19.9	6.31	×1.00	242.1	22.8	27.5	6.29	×1.00	241.5
StarCoder-15.5B$_{\text{CodeFast}}$	41.2	47.6	5.27	**×1.18**	195.4	15.5	19.9	1.26	**×5.01**	33.21	22.8	27.5	2.84	**×2.21**	96.4
Phind-CodeLlama	84.3	87.8	7.13	×1.00	128.8	75.8	78.3	7.62	×1.00	137.6	44.6	50.0	13.67	×1.00	246.8
Phind-CodeLlama$_{\text{CodeFast}}$	84.3	87.8	4.82	**×1.48**	84.6	75.8	78.3	6.39	**×1.19**	112.3	45.1	50.0	7.32	**×1.87**	127.3

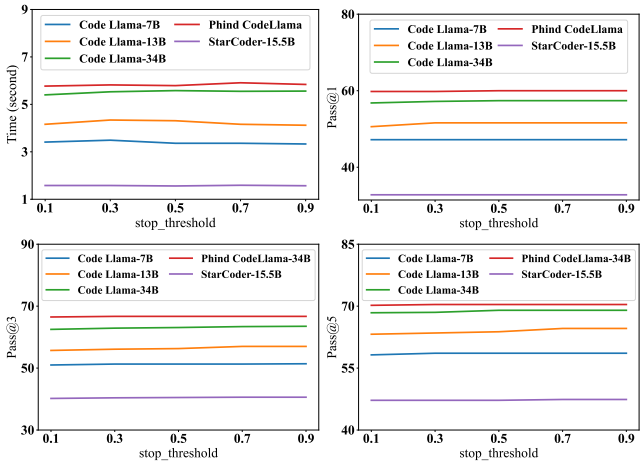


Figure 6: Stability analysis of CodeFast about stop_threshold on MBJSP.

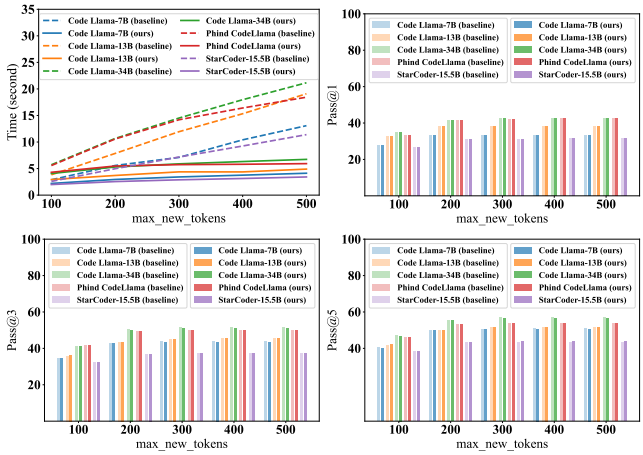


Figure 7: Stability analysis of CodeFast about max_new_token on MBGP.

B.5 RQ5: Application in other scenarios

In this section, we supplement the experiments with a compositional generation strategy and case studies. The detailed results are as follows:

Experimental results. We evaluate CodeFast in ClassEval [1] utilizing compositional generation strategy. The experimental results are shown in Table 6. We can observe that CodeFast can greatly increase the generation speed for all five models while maintaining the generation accuracy.

C PARAMETER INFORMATION OF GENGUARD MODULES

The parameter size information of five GenGuard modules is presented in Table 7. The results show that the largest model GenGuard for CodeLlama-34B only requires 16,384 parameters, which is 0.000048% the parameter size of CodeLlama-34B. This proves the lightweight nature of our method.

D COMPARISON WITH RELEVANT TECHNIQUES

D.1 Comparison with SEC

In this section, we compare our method with SEC [3], which is an effective inference acceleration method for code completion by dynamic model inference. Similar to our method, SEC trains an action classifier to predict the subsequent action for each specified intermediate layer based on its hidden state. The corresponding behavior of the inference process for each action predicted by SEC includes `Exit`, `Stop`, and `Continue`. The `Exit` action means that the model stops inference of the next transformer layers to avoid extra computation. The `Stop` action means that the model stops the inference process and outputs current generated tokens. The `Continue` action is the same as the normal inference of the transformer layer.

Training method. SEC first trains the intermediate LM head for each intermediate layer of LLM based on the ground truth code dataset. To attain training data for the classifier, SEC utilizes the ground truth code dataset to obtain predicted tokens of every

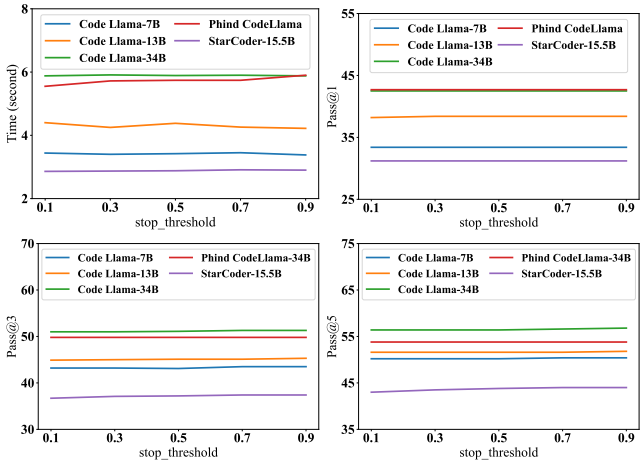


Figure 8: Stability analysis of CodeFast about stop_threshold on MBGP.

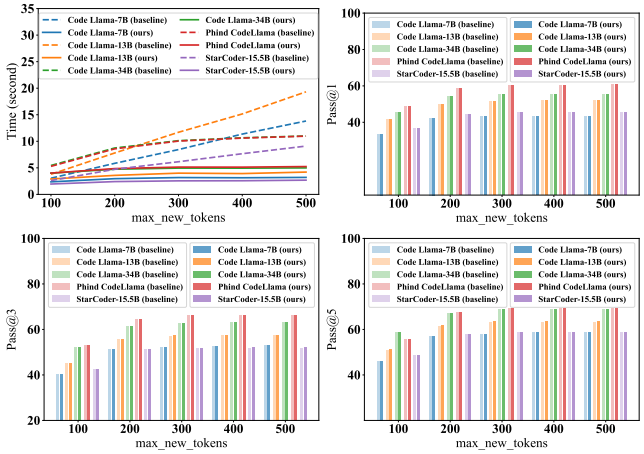


Figure 9: Stability analysis of CodeFast about max_new_tokens on MBCPP.

intermediate LM Head at the current step. Then, SEC compares the predicted tokens with the ground truth tokens to obtain the action labels for different intermediate layers. Finally, SEC trains a classifier based on the generated dataset.

Difference with SEC. Though both CodeFast and SEC train classifiers to prevent the generation of Code LLMs, there are the following differences. First, though the SEC classifier can classify erroneous generation by training on ground truth code, it can not classify excess generation. This is because SEC is not designed to remove excess generation. The SEC trains classifiers using the ground truth code that does not include excess generation and the labeling method can not label "stop" at excess token, so the SEC classifier cannot learn to remove excess generation from labels. The experiment results in Table 8 show CodeFast outperforms SEC both in generation speed and generation accuracy. Second, CodeFast is designed for multiple languages, while SEC is intended for one programming language. Third, CodeFast proposes the line-voting mechanism, which can effectively increase the accuracy of generated code. Fourth, our method offers a more lightweight

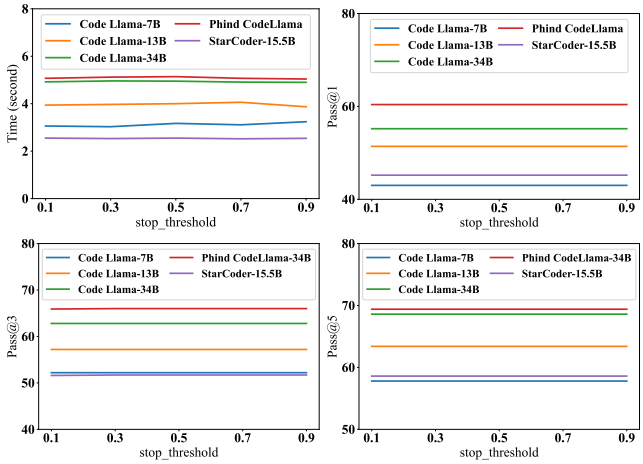


Figure 10: Stability analysis of CodeFast about stop_threshold on MBCPP.

Table 6: Performance of CodeFast in ClassEval with compositional generation strategy. P@k calculates the probability of top k-generated class that successfully passes unit tests. Time represents the average time (in seconds) taken by a Code LLM to generate result per sample.

Model	P@1	P@3	P@5	Time	Speedup
CodeLlama-7B	4.0	6.2	8.0	43.7	×1.00
CodeLlama-7B$_{\text{CodeFast}}$	4.0	6.2	8.0	14.3	**×3.06**
CodeLlama-13B	9.0	13.0	15.0	56.0	×1.00
CodeLlama-13B$_{\text{CodeFast}}$	9.0	13.0	15.0	20.6	**×2.72**
CodeLlama-34B	11.0	17.0	19.0	72.2	×1.00
CodeLlama-34B$_{\text{CodeFast}}$	11.0	17.0	19.0	17.2	**×4.20**
StarCoder-15.5B	7.0	8.7	10.0	39.8	×1.00
StarCoder-15.5B$_{\text{CodeFast}}$	7.0	8.7	10.0	17.7	**×2.25**
Phind-CodeLlama	15.0	19.1	21.0	62.9	×1.00
Phind-CodeLlama$_{\text{CodeFast}}$	15.0	19.1	21.0	23.2	**×2.71**

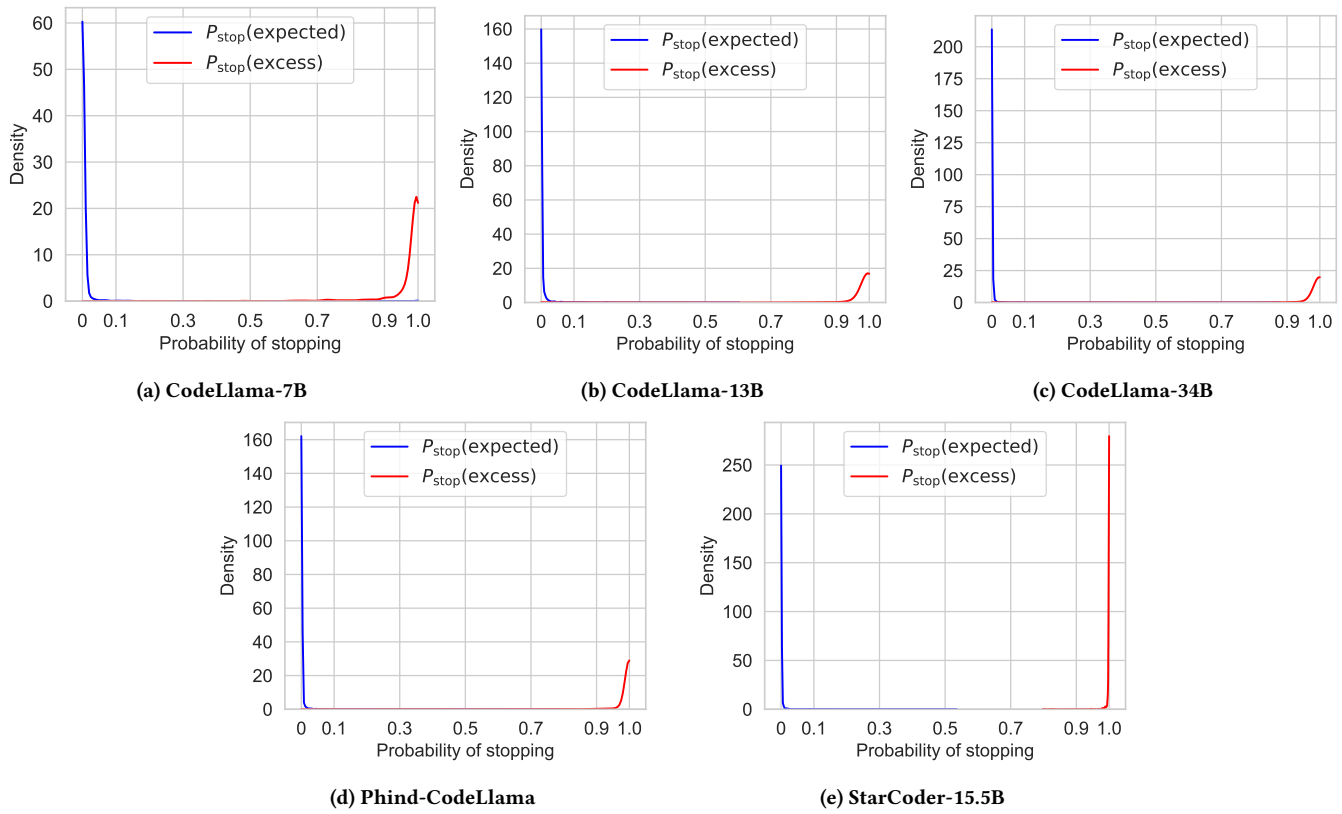
Table 7: Parameter information of five Code LLMs and their GenGuard modules.

Model	LLM model size	GenGuard model size
CodeLlama-7B	7B	8192
CodeLlama-13B	13B	10240
CodeLlama-34B	34B	16384
StarCoder-15.5B	15.5B	12288
Phind-CodeLlama	34B	16384

training process while the SEC requires training extra intermediate LM head for each Transformer layer.

Experiment settings. We reproduce the SEC method following the implementation in the official repository of SEC[1]. In our experiments, all models are deployed on GPU using the bfloat16 precision format. In the training stage, to fairly compare with CodeFast, we train the SEC classifier using the same dataset MBPP, which

[1]https://github.com/v587su/SEC/blob/master/model/multi_lmhead_codegen.py



(a) CodeLlama-7B (b) CodeLlama-13B (c) CodeLlama-34B

(d) Phind-CodeLlama (e) StarCoder-15.5B

Figure 11: Stopping prediction probability density distribution of five classifiers.

includes 374 ground truth code. The training of SEC models is conducted on two A100 GPUs. Due to the limitation of computational resources, we only train the SEC model for CodeLlama-7B, CodeLlama-13B, and StarCoder-15.5B. The training process uses the AdamW [4] optimizer with a learning rate of 5e-4 and lasts for ten epochs. In the inference stage, we utilize greedy decoding to generate code because the implementation of SEC does not support random sampling. The max generation length is set to 300. Following SEC [3], we adjust the stop threshold and exit threshold of the SEC classifier under 5% tolerances on the reduction of Pass@1 and strive to maximize the acceleration effect.

Experimental results. The experimental results are presented in Table 8. We can observe that CodeFast outperforms SEC both in generation speed and generation accuracy, indicating that CodeFast can increase the generation speed and maintain the accuracy of generated code better than SEC. Moreover, we can find that SEC cannot effectively remove excess generation from the metric of Length.

D.2 Comparison with Bruteforce Termination Tricks

There are some bruteforce termination tricks specific for Python such as terminating generation upon "\ndef" may easily remove excess generation. In this section, we compare our method with two bruteforce termination tricks, which terminate generation upon pattern "\ndef" and Python-specific patterns from a public repository[2] respectively. The experiments are presented in Table 9. We observe that CodeFast increases the generation speed of Code LLMs more effectively than these tricks.

Overall, there are following differences between CodeFast and buruteforce termination tricks. First, CodeFast can increase code

Table 8: Performance of CodeFast and SEC on 3 Code LLMs in MBPP dataset. P@1 is short for Pass@1. Time represents the average time (in seconds) taken by a Code LLM to generate result per sample.

Model	MBPP (Python)			
	P@1	Time	Speedup	Length
CodeLlama-7B$_{SEC}$	39.8	8.21	×1.14	287.0
CodeLlama-7B$_{CodeFast}$	41.0	2.07	**×4.52**	63.5
CodeLlama-13B$_{SEC}$	43.2	10.87	×1.24	254.9
CodeLlama-13B$_{CodeFast}$	44.8	2.98	**×3.98**	71.9
StarCoder-15.5B$_{SEC}$	40.8	3.73	×1.37	140.7
StarCoder-15.5B$_{CodeFast}$	42.6	1.98	**×2.59**	73.5
Average$_{SEC}$	41.3	7.60	×1.16	227.5
Average$_{CodeFast}$	42.8	2.34	**×3.76**	69.6

[2]https://github.com/iCSawyer/bigcode-evaluation-harness/blob/main/bigcode_eval/tasks/mbpp.py

Table 9: Performance of CodeFast and two tricks on five Code LLMs in MBPP dataset. P@1 is short for Pass@1. Time represents the average time (in seconds) taken by a Code LLM to generate result per sample.

Model	MBPP (Python)		
	P@1	Time	Speedup
CodeLlama-7B$_{\text{Pattern-def}}$	41.0	3.56	×2.62
CodeLlama-7B$_{\text{Pattern-all}}$	41.0	3.63	×2.58
CodeLlama-7B$_{\text{CodeFast}}$	41.0	2.07	×**4.52**
CodeLlama-13B$_{\text{Pattern-def}}$	44.8	5.25	×2.26
CodeLlama-13B$_{\text{Pattern-all}}$	44.8	4.94	×2.40
CodeLlama-13B$_{\text{CodeFast}}$	44.8	2.98	×**3.98**
CodeLlama-34B$_{\text{Pattern-def}}$	51.4	8.34	×1.76
CodeLlama-34B$_{\text{Pattern-all}}$	51.6	7.45	×1.97
CodeLlama-34B$_{\text{CodeFast}}$	51.6	3.62	×**4.05**
StarCoder-15.5B$_{\text{Pattern-def}}$	42.6	5.84	×0.88
StarCoder-15.5B$_{\text{Pattern-all}}$	42.6	2.56	×2.00
StarCoder-15.5B$_{\text{CodeFast}}$	42.6	1.98	×**2.59**
Phind-CodeLlama$_{\text{Pattern-def}}$	55.8	7.82	×0.63
Phind-CodeLlama$_{\text{Pattern-all}}$	55.8	5.59	×0.87
Phind-CodeLlama$_{\text{CodeFast}}$	55.8	3.65	×**1.34**
Average$_{\text{Pattern-def}}$	47.12	6.16	×1.49
Average$_{\text{Pattern-all}}$	47.16	4.83	×1.90
Average$_{\text{CodeFast}}$	47.16	2.86	×**3.22**

Background: When generating code, Code LLM often includes unnecessary details along with the essential solution. We have developed a method to truncate these responses, focusing on retaining the core of the solution while removing excess content. The key is to truncate at a point where the essential content is included, but excess parts are removed, ensuring the continuity and relevance of the generated code.

Instruction: Review the original coding prompt and the full code response generated by the model. Your task is to identify the point where the code should be truncated. This truncation should maintain the essence and continuity of the solution, while also being as brief as possible to avoid excess response. The goal is to refine the response to its most concise and relevant form without altering the core content.

Original Prompt: <prompt_example>
Original Output: <raw_output_example>
Extracted Code: <extracted_code_example>

(more demonstrations...)

Original Prompt: <current_prompt>
Original Output: <current_raw_output>
Extracted Code:

Figure 12: Prompt utilized in ChatGPT

generation speed than these tricks, as shown in Table 9. Second, CodeFast can be applicable to multiple PLs while these tricks are specific for a programming language (Python). Third, CodeFast introduces an automatic data construction framework, without manual collection of such tricks for different models and programming languages.

Table 10: Annotator demographics.

Category	Items	Number
Gender	Female	2
	Male	3
	Non-binary	0
Title	PhD Student	3
	Researcher	1
	Professor	1
Development Experience	0-4 years	0
	> 5 years	5
Academic Experience	0-3 years	1
	4-6 years	2
	> 6 years	2

E ANNOTATOR DEMOGRAPHICS

In this section, we present the demographic details of all annotators who participated in the human evaluation and labeling processes described in our paper. These details are summarized in Table 10. The annotators comprise two authors of this paper and three PhD students, with each possessing over five years of programming experience. Moreover, we ensure that all annotators have relevant research backgrounds.

F PROMPT UTILIZED IN CHATGPT

In the data labeling phase of our approach, we employ two types of code analyzers to label the code generated by Code LLM: a syntax-based code analyzer and a semantic-based code analyzer. When the syntax-based code analyzer fails, we label data by using ChatGPT as a semantic-based code analyzer. The details of one prompt used in ChatGPT are shown in Figure 12. First, we introduce the background of excess generation to ChatGPT and instruct ChatGPT to differentiate the expected generation and excess generation in raw output from Code LLM. Then, we utilize the in-context-learning method to guide Code LLM in labeling data, using three demonstrations constructed manually.

REFERENCES

[1] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation," *arXiv preprint arXiv:2308.01861*, 2023.
[2] E. Parzen, "On estimation of a probability density function and mode," *The annals of mathematical statistics*, vol. 33, no. 3, pp. 1065–1076, 1962.
[3] Z. Sun, X. Du, F. Song, S. Wang, and L. Li, "When neural code completion models size up the situation: Attaining cheaper and faster completion through dynamic model inference," *arXiv preprint arXiv:2401.09964*, 2024.
[4] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.