LLM4DV: Using Large Language Models for Hardware Test Stimuli Generation

Zixi Zhang
University of Cambridge
zz458@cam.ac.uk

Greg Chadwick
lowRISC
gac@lowrisc.org

Hugo McNally
lowRISC
hugom@lowrisc.org

Yiren Zhao
Imperial College London
a.zhao@imperial.ac.uk

Robert Mullins
University of Cambridge
robert.mullins@cl.cam.ac.uk

Abstract

Test stimuli generation has been a crucial but labour-intensive task in hardware design verification. In this paper, we revolutionize this process by harnessing the power of large language models (LLMs) and present a novel benchmarking framework, LLM4DV. This framework introduces a prompt template for interactively eliciting test stimuli from the LLM, along with four innovative prompting improvements to support the pipeline execution and further enhance its performance. We compare LLM4DV to traditional constrained-random testing (CRT), using three self-designed design-under-test (DUT) modules. Experiments demonstrate that LLM4DV excels in efficiently handling straightforward DUT scenarios, leveraging its ability to employ basic mathematical reasoning and pre-trained knowledge. While it exhibits reduced efficiency in complex task settings, it still outperforms CRT in relative terms. The proposed framework and the DUT modules used in our experiments are open-sourced.

1 Introduction and Background

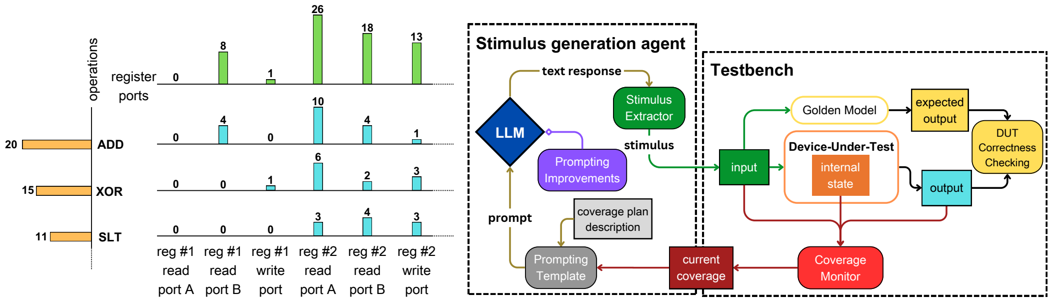
Design verification (DV) checks the correctness of hardware designs. The verification process takes in inputs, or *test stimuli*, passes them into the hardware design-under-test (DUT), and compares the result to expected outputs from a *software golden model* [1].

A *coverage plan* in DV is defined to specify a list of coverage points to be tested, which are particular outputs and machine states that the verification process needs to cover. Each cover point is associated with a *coverage bin* that counts how many times the cover point has been exercised. The goal of the verification process is to achieve 100% functional *coverage rate* based on the coverage plan. Figure 1a illustrates an example coverage plan.

Effective test stimuli generation has been a major challenge in meeting 100% coverage [1]. Traditionally, constrained-random testing (CRT) has been used for generating vast random but valid test stimuli and attempting to "hit" the bins. However, CRT is inefficient in hitting bins with complicated conditions. Often, this necessitates extensive human engineering for the test stimuli design process.

Recent large language models (LLMs), trained on immense amounts of text and human feedback data, are capable of solving tasks of distinct form that require multi-step reasoning and generating high-quality samples [2, 3, 4]. This brings up an important question: can LLMs effectively minimise the amount of human effort involved in hardware Design Verification (DV)? This study specifically focuses on one arguably most labour-intensive aspect of the DV process - generating test stimuli - and explores the potential of using LLMs to assist with this task. More related work on LLMs and DV

Machine Learning for Systems Workshops at 37th NeurIPS 2023.



(a) Example coverage plan

(b) LLM4DV framework

Figure 1: Design verification flow. Figure 1a is an example coverage plan: two types of basic coverage bins (orange and green) and also their cross coverage bins (cyan) are shown. Figure 1b shows an overview of the LLM4DV framework where the prompted LLM takes output coverage results as inputs and outputs stimulus in an iterative fashion.

can be found in Appendix A. In this work, we establish a benchmarking framework named **LLM4DV** (Large Language Model for Design Verification). This is a benchmark framework that utilizes LLMs for test stimuli generation. In this paper, we make the following contributions:

- We design and construct three DUT modules: a Primitive Data Prefetcher Core, an Ibex Instruction Decoder, and an Ibex CPU.
- We propose and develop a framework that uses *prompted LLMs* to generate test stimuli for hardware designs. Also, we propose four improvements to the prompting strategies to improve the effectiveness of the generated stimuli.
- We evaluate the framework with the three DUT modules. We show that our framework with optimal improvement measures achieves 98.94%, 86.19%, and 5.61% coverage rates on these modules respectively in a realistic experimental setup. We open-source our framework with these three modules to allow both the machine learning and hardware design communities to experiment with their ideas. Our project is publicly available at `https://github.com/ZixiBenZhang/ml4dv`.

2 Method

Our experiments use an LLM in the test stimuli generation process, together with a testbench with a DUT to form the whole LLM4DV framework, as shown in Figure 1b. The following subsections describe the basic DV framework, the prompt templates for the LLM, and four prompting improvements. Figure 2 gives a general picture of the prompt templates and prompting improvements.

2.1 Basic framework

The testbench has three major components: a stimulus generation agent (the prompted LLM), a design-under-test (DUT), and a coverage monitor. In each timestep, the stimulus generation agent provides a test stimulus and inputs it to the DUT.

Our basic framework introduces a prompt generator and an LLM as the stimulus generation agent. In each generation cycle, the prompt generator generates a prompt based on a template (see Appendix B) and the current coverage feedback from the coverage monitor. In every timestep when the stimulus generation agent is requested for a test stimulus, it takes out the oldest value in its stimulus buffer; if the buffer is empty, the LLM takes in a new request and a list of new stimuli will be added to the buffer.

The pipeline, or a *trial*, stops either when all cover points have been reached or if the stimulus generation agent is "exhausted" i.e. becoming ineffective or inefficient to cover any new bins. See Algorithm 1 in the Appendix for details of the pipeline.

We provide a Coverage-Feedback Template to generate prompts for the LLM. Examples of prompts and responses are shown in Appendix H, and detail of our prompting template is in Appendix B.

2.2 Four prompting improvements

We develop two improvements necessary for making the framework executable and two improvements to further improve its performance. We also evaluate different design choices for these improvements. A full ablation study with all these improvements and their design choices are in Appendix I and Appendix D, in this section, we mainly discuss the best attempts.

Missed-bin sampling In most generation cycles in a trial, there would be hundreds to thousands of bins uncovered. The iterative queries can't include all of them because the prompt's length would exceed the LLM's input token number limit.

We propose missed-bin sampling, which samples a number of bins from all uncovered bins to be included in the differences part of iterative queries. Our experiment finds that more random sampling methods encourage the agent to cover bins with stricter hitting conditions, and more stable sampling methods make the agent more efficient in hitting the easier bins. We present two most effective sampling methods: *Coverpoint Type-based Sampling*, which samples from "easier bins" and "harder bins" respectively; *Mixed Coverpoint Type-based and Pure Random Sampling*, which switches between the previous two strategies whenever the agent becomes inefficient with the current strategy.

Best-iterative-message sampling The LLM needs previous messages in the dialogue to learn about what has happened. However, as the dialogue grows, the length of input may exceed the LLM's input limit. We hence sample from previous messages, which acceptably loses some generality and meanwhile preserves key details, including the bin description and positive examples (i.e. responses that successfully hit many bins) useful for covering edge cover points.

We propose four sampling methods: *Successful Responses*, where we keep random three iterative queries that hit the most number of bins; *Successful Difficult Responses*, which is similar to Successful Responses but each "harder bin" counts as 2.5 bins.

Dialogue restarting LLM sometimes behaves stubbornly, repeating mistakes it made previously. We introduce a dialogue restarting mechanism to resolve this problem. When the LLM hits less than three new bins within t responses, we clear the dialogue record and restart from the system message and initial query.

We present three most effective restarting plans: *Normal Tolerance*, where $t = 7$; *Low Tolerance*, where $t = 4$; *Coverage Rate-based Tolerance*, where $t = 4$ in the beginning and $t = 7$ after reaching certain coverage rate threshold.

Best-iterative-message buffer resetting When the dialogue record is reset, the buffer for best iterative messages in Best-iterative-message sampling can also be cleared or kept. These two strategies display a trade-off between "effectively forgetting past mistakes" and "learning about the task faster after restart". We present two most effective resetting plans for the best-iterative-message buffer: *Keeping best-messages*; *Stable-restart Keeping best-messages*, which keeps the buffer on restarts, but doesn't use it for the first four responses after restarts.

3 Evaluation

We evaluate our LLM4DV framework on three DUT modules: Primitive Data Prefetcher Core, Ibex Instruction Decoder, and Ibex CPU. These modules and their testbenches are constructed by ourselves. With Verilator and cocotb, we could efficiently simulate and test the modules in Python. We use OpenAI's GPT-3.5-turbo-0613 model [5] in our "fixed-budget" experiments. See Appendix F for DUT details, and Appendix E for experimental setup details.

Primitive Data Prefetcher Core The Primitive Data Prefetcher Core module requires test stimuli in specific stride patterns. We run the CRT baseline and the fixed-budget experiments with GPT 3.5 and three prominent configurations. Table 1 shows the results of the experiments. We see that the best configuration, which uses Coverpoint Type-based Sampling, Successful Difficult Responses, Normal Tolerance, and Stable-restart Keeping best-messages, achieved 98.94% coverage rate. This demonstrates that with our prompting template and improvement methods, the stimulus generation agent is capable of making simple mathematical reasoning to hit the bins efficiently.

Ibex Instruction Decoder The Ibex Instruction Decoder module takes in RISC-V instruction codes. Table 1 shows the results of the experiments. We see that the best configuration, which uses Mixed

Table 1: **Results of fixed-budget experiments**: A1, A2, A3 contrast *Successful Responses* (II) + *Stable-restart Keeping best-messages* (iii) vs. *Successful Difficult Responses* (IV) + *Stable-restart Keeping best-messages* (iii) vs. *Successful Difficult Responses* (IV) + *Keeping best-messages* (ii). B1, B2 contrast *Normal Tolerance* (a) vs. *Coverage Rate-based Tolerance* (d). C1, C2 contrast *Normal Tolerance* (a) vs. *Low Tolerance* (b). For details of the configurations, see Table 2 in Appendix.

Config	Max coverage	Max coverage rate	Avg msg/trial	Stdev msg/trial
Primitive Data Prefetcher Core				
Random	4	0.39%	-	-
A1	987	95.45%	641	104.24
A2	**1023**	**98.94%**	617.5	165.06
A3	1007	97.39%	459.33	287.21
Ibex CPU Instruction Decoder				
Random	1136	53.92%	-	-
B1	1695	80.45%	864	147.92
B2	**1816**	**86.19%**	844.25	127.94
Ibex CPU				
Random	3	1.53%	-	-
C1	10	5.10%	42.49	11.74
C2	**11**	**5.61%**	45.72	16.17

Coverpoint Type-based and Pure Random Sampling, Successful Difficult Responses, Coverage Rate-based Tolerance, and Stable-restart Keeping best-messages, achieved 86.19% coverage rate. This demonstrates that the stimulus generation agent can make use of its pre-trained knowledge to hit the bins efficiently.

Ibex CPU The Ibex CPU module takes in a list of updates to its instruction memory in each cycle, and executes the instruction memory accordingly. This module is more complex than the previous two modules. Table 1 shows the results of the experiments. We see that the best configuration, which uses Coverpoint Type-based Sampling, Successful Responses, Fast Tolerance, and Stable-restart Keeping best-messages, only achieved 5.61% coverage rate. Although it's better than the CRT, this shows that in complex task settings, the framework is deficient in producing effective updates and may require further prompt engineering and prompting improvements to enhance its performance.

4 Conclusion and Discussion

This study proposes LLM4DV, a benchmarking framework for test stimuli generation in design verification. We build three DUT modules and construct a stimulus generation agent, together with a prompt template and four prompting improvements. Through experimentation on the modules, we show that LLM-assisted test stimuli generation could achieve impressive coverage rates on several design units, and its performance is significantly better than CRT for relatively complex coverage plans. However, when the task setting becomes more complex, for instance, a full CPU design, LLM fails to achieve a fairly good coverage rate.

To our understanding, our study represents the first systematic exploration of utilizing LLMs to assist in the area of hardware design verification. More specifically, we focus on the generation of test stimuli, which is a task known for its high reliance on human intervention. Our designs and findings are open-sourced. Our framework is designed to accommodate arbitrary hardware designs and offers users the flexibility to switch between different LLMs or customize their prompting strategy. We also provided three high-quality hardware designs with different difficulties for the users to experiment with their new LLMs or prompts. We anticipate that our developed framework will provide researchers with a user-friendly interface to further explore the potential of LLMs in aiding hardware design verification.

References

[1] Hasini Witharana, Yangdi Lyu, Subodha Charles, and Prabhat Mishra. A survey on assertion-based hardware verification. *ACM Computing Surveys (CSUR)*, 54:1 – 33, 2022. URL `https://api.semanticscholar.org/CorpusID:246362628`.

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[3] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step, 2023.

[4] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[5] OpenAI. OpenAI: Introducing ChatGPT, 2022. URL `https://openai.com/blog/chatgpt`.

[6] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests, 2022.

[7] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation, 2023.

[8] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for Python. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 168–172, 2022. doi: 10.1145/3510454.3516829.

[9] Luca Ferro, Laurence Pierre, Yves Ledru, and Lydie du Bousquet. Generation of test programs for the assertion-based verification of tlm models. In *2008 3rd International Design and Test Workshop*, pages 237–242, 2008. doi: 10.1109/IDT.2008.4802505.

[10] Jason G. Tong, Marc Boule, and Zeljko Zilic. Airwolf-tg: A test generator for assertion-based dynamic verification. In *2009 IEEE International High Level Design Validation and Test Workshop*, pages 106–113, 2009. doi: 10.1109/HLDVT.2009.5340166.

[11] Yangdi Lyu and Prabhat Mishra. Automated test generation for activation of assertions in rtl models. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 223–228, 2020. doi: 10.1109/ASP-DAC47756.2020.9045731.

A Related work

While the application of LLMs on hardware design verification has been brought to focus only recently, test code generation for software engineering with LLMs has been well-studied and made remarkable performance [6, 7, 8]. Chen et al. [6] utilized LLM to generate potential implementations and test cases for a function by providing the LLM with the signature and a description of the function. Schäfer et al. [7] proposed a pipeline to generate unit tests for existing code, which iteratively refined

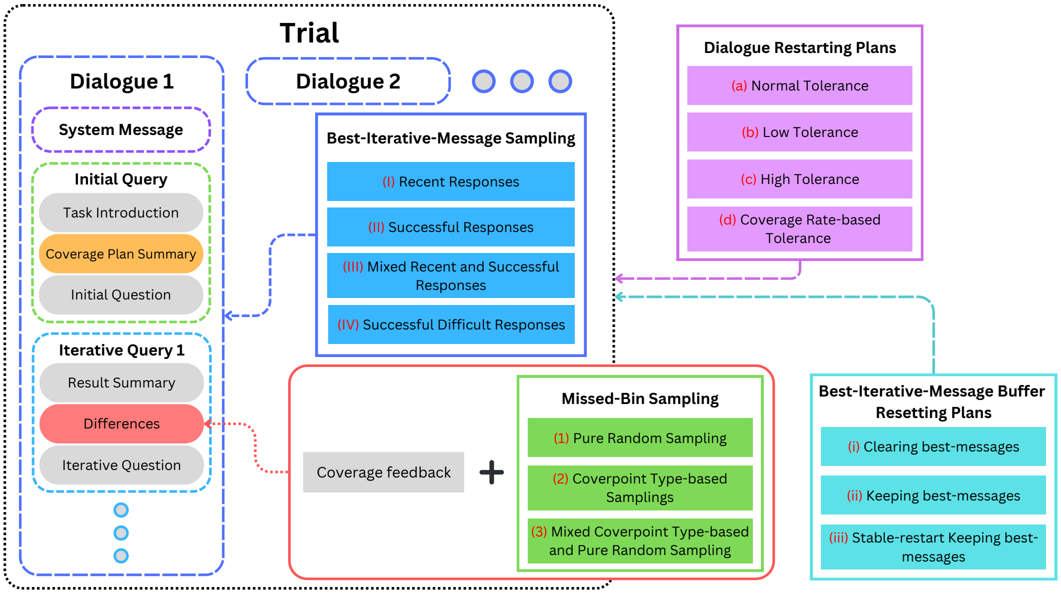

Trial
Dialogue 1
Dialogue 2
System Message
Initial Query
Task Introduction
Coverage Plan Summary
Initial Question
Iterative Query 1
Result Summary
Differences
Iterative Question
Best-Iterative-Message Sampling
(I) Recent Responses
(II) Successful Responses
(III) Mixed Recent and Successful Responses
(IV) Successful Difficult Responses
Dialogue Restarting Plans
(a) Normal Tolerance
(b) Low Tolerance
(c) High Tolerance
(d) Coverage Rate-based Tolerance
Coverage feedback
+
Missed-Bin Sampling
(1) Pure Random Sampling
(2) Coverpoint Type-based Samplings
(3) Mixed Coverpoint Type-based and Pure Random Sampling
Best-Iterative-Message Buffer Resetting Plans
(i) Clearing best-messages
(ii) Keeping best-messages
(iii) Stable-restart Keeping best-messages

Figure 2: Coverage-feedback prompt template and four prompting improvements. This framework corresponds to the stimulus generation agent component in Figure 1b. We highlight the different design options possible for the four prompting improvements (Best-iterative-message sampling, Dialogue Restarting Plans, Missed-Bin Sampling and Best-Iterative-Message Buffer Resetting Plan).

the prompt to generate better tests. These studies show promising results on software code, while we shift focus to using LLM on Hardware Description Language (HDL) and hardware design specification, which involves a different setting and more sophisticated procedure.

In hardware design verification, assertion-based verification (ABV) is widely adopted together with code coverage, functional coverage, and validation using generated test patterns [1]. ABV uses assertions in HDL to detect violations of predefined properties. However, ABV requires test patterns (i.e. input test stimuli) to activate given assertions and therefore reveal vulnerabilities. For simulation-based ABV approaches, traditional test generation using random or constrained-random tests cannot guarantee to activate assertions with complex conditions in a reasonable time. To speed up assertion coverage, Ferro et al. [9] used combinatorial testing, which provides a set of combinations of user-selected values, to select test stimuli most suitable to cover corner cases. Tong et al. [10] proposed a method that searches the compact assertion-based automata for failure and acceptance nodes before test generation. Simulation-based test generation has also been incorporated with formal method-based test generation. Lyu and Mishra [11] utilized concolic testing to activate assertions, which regards assertions as branches, searches through the branch statement tree with heuristics to efficiently obtain a path, and generates test stimuli to cover the obtained branch targets. These methods, however, are subjected to complexity explosion problems [1] and fail to effectively make use of the user's knowledge about the design. We overcome this issue by utilizing LLM's pre-trained knowledge to reason about the given coverage plan and guide the test stimuli generation.

B A graphical illustration of the prompting template

Coverage-Feedback Template The Coverage-feedback prompt template contains templates for the system message, initial query, and iterative queries.

- The **system message** clarifies the expected response format and specifies other requirements.
- The **initial query** is the first user message in a dialogue. It contains three parts:
    1. Task introduction: a description of what is included in this prompt and what the LLM will be asked to do.
    2. Coverage plan summary: a description of cover points of the coverage plan.
    3. Initial question: a one-line instruction.

- The **iterative queries** are the user messages following the first assistant (LLM) response. Each contains three parts:
  1. Result summary: a general feedback which:
     - if the previous assistant response was gibberish (i.e. contains mostly nonsense words) or didn't follow the output format, the result summary repeats the output format requirement; otherwise,
     - if the previous assistant response failed to hit any new bins, the result summary points that out and asks for a new list of stimuli;
     - if the previous assistant responses hit some bins, the result summary points that out and asks for a new list of stimuli.
  2. Differences: a list of uncovered bins.
  3. Iterative question: a one-line instruction, repeating the output format requirement if the previous response was gibberish or didn't follow the output format.

C Basic framework algorithm

Algorithm 1 Basic DV pipeline

```
stimulus ← 0
coverage ← {}
while coverage rate $< 100\%$ and not ($\Delta$ coverage in 25 messages < 3 or $\Delta$ coverage in 40 messages = 0) do
    while stimulus_buffer not empty and coverage rate $< 100\%$ do
        stimulus ← stimulus_buffer.pop()
        testbench.input(stimulus)
        coverage ← coverage_monitor.compute_coverage(testbench)
    end while
    prompt ← prompt_generator.$generate(coverage)$
    response ← LLM.generate(prompt)
    stimuli ← extractor.extract(response)
    stimulus_buffer.extend(stimuli)
    while stimulus_buffer is empty do
        prompt ← prompt_generator.regenerate(coverage)
        response ← LLM.generate(prompt)
        stimuli ← extractor.extract(response)
        stimulus_buffer.extend(stimuli)
    end while
end while
```

Algorithm 1 shows the workflow of the basic framework as in Section 2.1.

D Details of prompting improvements

This section describes the design choices of our four prompting improvements. We also compare these design choices with ablation experiments, see Appendix I.

D.1 Missed-bin sampling

We define three sampling methods:

- (1) **Pure Random Sampling**: randomly samples seven bins from all uncovered bins.
- (2) **Coverpoint Type-based Sampling**: we categorize all bins into "easier bins" and "harder bins" based on their difficulties to be covered, and order them based on their names; when sampling, we always take the first two uncovered bins, then either randomly sample five bins from all uncovered bins if there are no "easier bins" left, or sample three "easier bins" and two "harder bins".

- (3) **Mixed Coverpoint Type-based and Pure Random Sampling**: when the coverage ratio is below 15%, it keeps using Coverpoint Type-based Sampling; when the coverage ratio is larger than 15%, it switches between Coverpoint Type-based Sampling and Pure Random Sampling whenever the current sampling method hits less than three new bins within four responses. The number of 15% is obtained empirically.

D.2 Best-iterative-message sampling

We propose four sampling methods:

- (I) **Recent Responses**: keeps the initial query (and its response), and three most recent iterative queries (and their responses).
- (II) **Successful Responses**: keeps the initial query and three iterative queries with responses that hit the maximum number of bins; if multiple maximums exist, uniformly samples three of them.
- (III) **Mixed Recent and Successful Responses**: keeps the initial query, two most successful iterative queries, and one most recent iterative query.
- (IV) **Successful Difficult Responses**: similar to successful responses, but each "harder bin" as in Appendix D.1 counts as 2.5 bins.

D.3 Dialogue restarting

We define four restarting plans:

- (a) **Normal Tolerance**: restarts the dialogue if the LLM hits less than three bins within 7 responses.
- (b) **Low Tolerance**: restarts the dialogue if the LLM hits less than three bins within 4 responses.
- (c) **High Tolerance**: restarts the dialogue if the LLM hits less than three bins within 10 responses.
- (d) **Coverage Rate-based Tolerance**: uses Low Tolerance when the coverage ratio is below 15%, and Normal Tolerance afterwards. The number of 15% is intentionally set as the same value as in Mixed Coverpoint Type-based and Pure Random Sampling in Appendix D.1.

D.4 Best-iterative-message buffer resetting

We define three resetting plans for the best-iterative-message buffer:

- (i) **Clearing best-messages**: clears the buffer on dialogue restarts.
- (ii) **Keeping best-messages**: keeps the buffer on dialogue restarts.
- (iii) **Stable-restart Keeping best-messages**: keeps the buffer on dialogue restarts, but not using it for the first four responses after restarts.

E Fixed-budget experiment setup

Large language models We use OpenAI's GPT-3.5-turbo-0613 model [5] in our experiments. For OpenAI's API calls, we set parameters as `temperature` $= 0.4$, `top_p` $= 1.0$, and `max_tokens` $= 600$. These parameters are decided empirically. We also use Llama 2 7B model [4] in ablation experiments, see Appendix I.

Experimental setup and measurement We run "fixed-budget experiments" on the three DUT modules, each with certain configurations of the prompt template and methods of improvement measures. The configurations are selected as the ones with the most significant results in ablation experiments (see Appendix I).

In each fixed-budget experiment, the experiment routine would have a budget of 10,000,000 tokens (input and output tokens are counted in the same way). Here's how it works:

1. The experiment routine starts with a trial as described in Section 2.1.
2. The trial either ends as described in Section 2.1 or when it runs out of the token budget.
3. If there is budget left, the experiment routine starts a new trial, in which the stimulus generation agent, the testbench, and the coverage record are reset.
4. The experiment routine ends when the budget is used up.

Running a fixed-budget experiment takes 6 to 20 hours, depending on the DUT module and the LLM service status.

The fixed-budget setup reflects the realistic use case where the financial budget for DV is reflected on the token budget. The multi-trial setup allows configurations to trade off "steady slow increase", where a trial takes many messages but stably achieves a high coverage, and "random fast increase", where a trial takes fewer messages but may either reach a high coverage by chance or end up with a low coverage.

We measure the maximum coverage each configuration achieves in trials in an experiment routine as its performance. We use the mean and the standard deviation of message count per trial as a reference for the capability and stability of configurations. We don't measure the size of the union set of covered bins across trials because we intend to test our framework's capacity across different DUT modules rather than to reach 100% coverage rate on a specific DUT.

We compare our framework to a constrained-random test (CRT) generation agent baseline, which generates 1,000,000 integers within the input range. Particularly, for the Ibex CPU module, the CRT generates an instruction to update the current PC in each cycle.

F Details of DUTs

This section explains our three DUT modules and their coverage plans respectively.

F.1 Primitive Data Prefetcher Core

The Primitive Data Prefetcher Core takes in 32-bit integers and detects whether there's a stride pattern in it. This module requires relatively high mathematical reasoning capability for the LLM.

Our coverage plan contains 1034 bins of the following types:

- Single-stride bins: counts when 16 consecutive integers $a_0, a_1, ..., a_{15}$ satisfy $a_{i+1} - a_i = c$ for some constant $-16 \leq c \leq 15$.
- Double-stride bins: counts when 16 consecutive integers satisfy an alternative stride width pattern, formally $a_{2i+2} - a_{2i+1} = c_1$ and $a_{2i+1} - a_{2i} = c_2$ for some $-16 \leq c_1, c_2 \leq 15$ and $c_1 \neq c_2$.
- Misc bins: including
  - Single-stride positive/negative overflow bins: a single stride pattern with $c < -16$ (negative overflow) or $c > 15$ (positive overflow).
  - Double-stride pp/pn/np/nn overflow bins: a double stride pattern with c_1 and c_2 positively / negatively overflow respectively.
  - No-stride-to-single/double: counts when 16 integers satisfying no stride pattern are followed by 16 integers with single / double stride pattern.
  - Single/double-to-double/single: counts when 16 integers satisfying single / double stride pattern are followed by 16 integers with double / single stride pattern.

F.2 Ibex Instruction Decoder

The Ibex Instruction Decoder is an instruction decoder for 32-bit RISC-V instruction codes. This module involves almost no mathematical reasoning but requires knowledge about RISC-V knowledge. Through preliminary tests, we find that GPT 3.5 and Llama 2 are pre-trained with RISC-V knowledge, and thus our bin descriptions don't need to explain the instruction formats and operations in detail.

Our coverage plan contains 2107 bins of the following types:

- ALU operation bins: counts when an instruction represents one of 26 pre-defined ALU operations such as ADD, ADDI, XOR, LW, etc.
- Register port bins: counts when an instruction uses the port of the specific register. There are 32 registers, and each has two read ports and one write port, which are used when the register file is taken as the first source, second source, and destination register, respectively.
- Cross coverage bins: the Cartesian product of ALU operation bins and register port bins. Counts when an instruction satisfies both bins simultaneously (some of the products, such as ADDI and read_port_A of any register, are invalid and not included in the coverage plan).

F.3 Ibex CPU

The Ibex CPU is a full RISC-V CPU with instruction and data memory. Different from the previous two modules, in every cycle the agent provides a stimulus of a list of address-instruction pairs $(a, i$, each presents a dynamic update that changes the instruction memory content at address a into value i. The CPU performs an instruction in the instruction memory in each timestep and waits for another list of updates (which can be empty). Apart from the current coverage, the testbench also returns the current program counter (PC) and the last executed instruction to the stimulus generation agent. Therefore, the task requires knowledge of both RISC-V and working principle of CPU and memory, and demands managing the complicated task setting.

Our coverage plan contains 196 bins of the following types:

- Operation bins: for each of the pre-defined ten R-type operations, three S-type instructions, and one J-type instruction (JAL), we consider the following four bins:
  - seen: counts when an instruction performs the operation;
  - zero_dst: if available, counts when the instruction performs the operation, with the destination register (rd) as zero (reg #0);
  - zero_src: if available, counts when the instruction performs the operation, with one of the source registers (rs) as zero (reg #0);
  - same_src: if available, counts when the instruction performs the operation, taking the same register as source registers (rs).
- Jump bins: for the JAL operation, we consider forward and backwards jumps respectively.
- Hazard bins: for each pair of the pre-defined operations, we consider a simplified read-after-write (RaW) hazard, which counts when the later instruction reads from a register that the previous instruction is writing to.

G Configurations in fixed-budget experiments

Table 2 shows the details of LLM4DV configurations in our fixed-budget experiment results (Table 1), including methods used for prompt templates and strategies of prompting improvements.

H Example prompts and responses

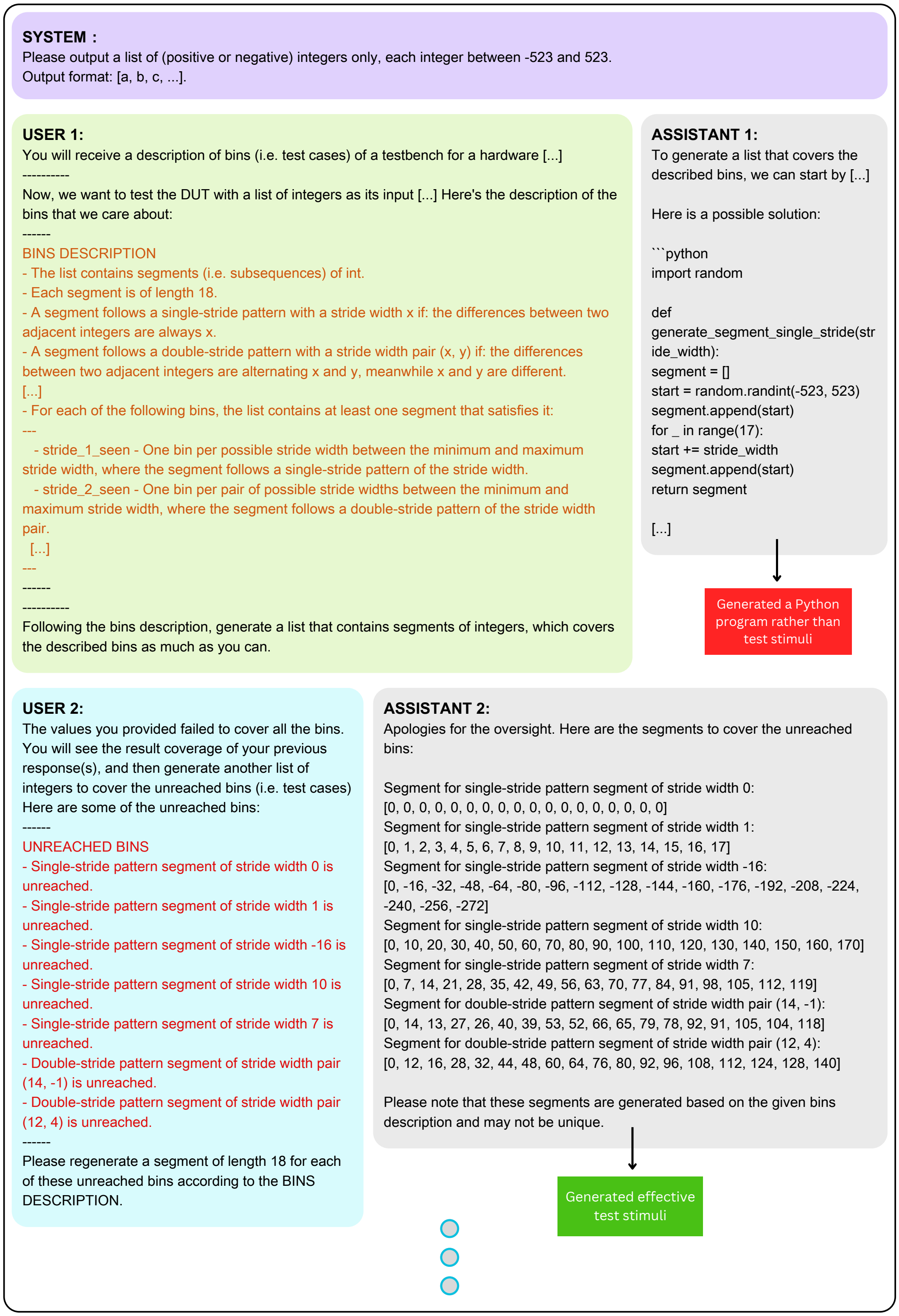
Figure 3 demonstrates several prompts and responses from the fixed-budget experiment on the Primitive Data Prefetcher Core module. The agent (USER) introduces the task and coverage plan in the initial message, and then provides coverage feedback in iterative messages. The LLM (ASSISTANT) generates textual responses according to the description and feedback.

I Ablation experiments

Due to the cost of money and time for OpenAI's API requests and experiment running, we compare configurations of the stimulus generation agent by their performances on simpler experiment setups.

Table 2: **Configurations of fixed-budget experiments**, **strategies** been contrasted in each module are marked in bold fonts.

Conf	**Improvement**	**Method**
Primitive Data Prefetcher Core		
A1	Prompt template	Original prompt template
	Missed-bin sampling	Coverpoint Type-based Sampling (2)
	Best-iterative-message sampling	**Successful Responses** (II)
	Dialogue restarting	Normal Tolerance (a)
	Best-iterative-message buffer resetting	**Stable-restart Keeping best-messages** (iii)
A2	Prompt template	Original prompt template
	Missed-bin sampling	Coverpoint Type-based Sampling (2)
	Best-iterative-message sampling	**Successful Difficult Responses** (IV)
	Dialogue restarting	Normal Tolerance (a)
	Best-iterative-message buffer resetting	**Stable-restart Keeping best-messages** (iii)
A3	Prompt template	Original prompt template
	Missed-bin sampling	Coverpoint Type-based Sampling (2)
	Best-iterative-message sampling	**Successful Difficult Responses** (IV)
	Dialogue restarting	Normal Tolerance (a)
	Best-iterative-message buffer resetting	**Keeping best-messages** (ii)
Ibex Instruction Decoder		
B1	Prompt template	Prompt template with one-line task introduction
	Missed-bin sampling	Mixed Coverpoint Type-based and Pure Random Sampling (3)
	Best-iterative-message sampling	Successful Difficult Responses (IV)
	Dialogue restarting	**Normal Tolerance** (a)
	Best-iterative-message buffer resetting	Stable-restart Keeping best-messages (iii)
B2	Prompt template	Prompt template with one-line task introduction
	Missed-bin sampling	Mixed Coverpoint Type-based and Pure Random Sampling (3)
	Best-iterative-message sampling	Successful Difficult Responses (IV)
	Dialogue restarting	**Coverage Rate-based Tolerance** (d)
	Best-iterative-message buffer resetting	Stable-restart Keeping best-messages (iii)
Ibex CPU		
C1	Prompt template	Original prompt template
	Missed-bin sampling	Coverpoint Type-based Sampling (2)
	Best-iterative-message sampling	Successful Responses (IV)
	Dialogue restarting	**Normal Tolerance** (a)
	Best-iterative-message buffer resetting	Stable-restart Keeping best-messages
C2	Prompt template	Original prompt template
	Missed-bin sampling	Coverpoint Type-based Sampling (2)
	Best-iterative-message sampling	Successful Responses (IV)
	Dialogue restarting	**Low Tolerance** (b)
	Best-iterative-message buffer resetting	Stable-restart Keeping best-messages



SYSTEM :
Please output a list of (positive or negative) integers only, each integer between -523 and 523.
Output format: [a, b, c, ...].

USER 1:
You will receive a description of bins (i.e. test cases) of a testbench for a hardware [...]

Now, we want to test the DUT with a list of integers as its input [...] Here's the description of the bins that we care about:

BINS DESCRIPTION
- The list contains segments (i.e. subsequences) of int.
- Each segment is of length 18.
- A segment follows a single-stride pattern with a stride width x if: the differences between two adjacent integers are always x.
- A segment follows a double-stride pattern with a stride width pair (x, y) if: the differences between two adjacent integers are alternating x and y, meanwhile x and y are different.
[...]
- For each of the following bins, the list contains at least one segment that satisfies it:

  - stride_1_seen - One bin per possible stride width between the minimum and maximum stride width, where the segment follows a single-stride pattern of the stride width.
  - stride_2_seen - One bin per pair of possible stride widths between the minimum and maximum stride width, where the segment follows a double-stride pattern of the stride width pair.
  [...]

Following the bins description, generate a list that contains segments of integers, which covers the described bins as much as you can.

ASSISTANT 1:
To generate a list that covers the described bins, we can start by [...]

Here is a possible solution:

~~~~
```python
import random

def
generate_segment_single_stride(stride_width):
segment = []
start = random.randint(-523, 523)
segment.append(start)
for _ in range(17):
start += stride_width
segment.append(start)
return segment
~~~~

[...]

USER 2:
The values you provided failed to cover all the bins. You will see the result coverage of your previous response(s), and then generate another list of integers to cover the unreached bins (i.e. test cases) Here are some of the unreached bins:

UNREACHED BINS
- Single-stride pattern segment of stride width 0 is unreached.
- Single-stride pattern segment of stride width 1 is unreached.
- Single-stride pattern segment of stride width -16 is unreached.
- Single-stride pattern segment of stride width 10 is unreached.
- Single-stride pattern segment of stride width 7 is unreached.
- Double-stride pattern segment of stride width pair (14, -1) is unreached.
- Double-stride pattern segment of stride width pair (12, 4) is unreached.

Please regenerate a segment of length 18 for each of these unreached bins according to the BINS DESCRIPTION.

ASSISTANT 2:
Apologies for the oversight. Here are the segments to cover the unreached bins:

Segment for single-stride pattern segment of stride width 0:
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Segment for single-stride pattern segment of stride width 1:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
Segment for single-stride pattern segment of stride width -16:
[0, -16, -32, -48, -64, -80, -96, -112, -128, -144, -160, -176, -192, -208, -224, -240, -256, -272]
Segment for single-stride pattern segment of stride width 10:
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170]
Segment for single-stride pattern segment of stride width 7:
[0, 7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112, 119]
Segment for double-stride pattern segment of stride width pair (14, -1):
[0, 14, 13, 27, 26, 40, 39, 53, 52, 66, 65, 79, 78, 92, 91, 105, 104, 118]
Segment for double-stride pattern segment of stride width pair (12, 4):
[0, 12, 16, 28, 32, 44, 48, 60, 64, 76, 80, 92, 96, 108, 112, 124, 128, 140]

Please note that these segments are generated based on the given bins description and may not be unique.

Figure 3: Example prompts and responses of the A2 configuration (see Table 2) on the Primitive Data Prefetcher Core module. The purple box is the system message. The green box is an initial query, containing a coverage plan summary (in orange). The blue box is an interactive query, containing differences i.e. coverage feedback (in red).

Table 3: **Results of infinite-message experiments on the Primitive Data Prefetcher Core module**

(a) Pure Random Sampling (1) vs. Coverpoint Type-based Sampling for stride detector bins (2), both with Original prompt template, Recent Responses, and Normal Tolerance.

Config	Max cov (/1034)	Avg cov	Stdev cov	Avg cov/msg	Stdev cov/msg
(1)	890	664.33	195.62	**2.59**	0.18
(2)	953	**681**	276.61	1.59	0.41

(b) Clearing best-messages (3) vs. Keeping best-messages (4) vs. Stable-restart Keeping best-messages (5), all with Original prompt template, Coverpoint Type-based Sampling for stride detector bins, Successful Responses, and Normal Tolerance.

Config	Max cov (/1034)	Avg cov	Stdev cov	Avg cov/msg	Stdev cov/msg
(3)	962	829	188.09	1.86	0.69
(4)	1011	526.33	<u>451</u>	1.3	0.81
(5)	1005	**974.67**	52.54	1.73	0.45

(c) Recent Responses (2) vs. Successful Responses (5) vs. Successful Difficult Responses (6), all with Original prompt template, Coverpoint Type-based Sampling for stride detector bins, Normal Tolerance, and Stable-restart Keeping best-messages.

Config	Max cov (/1034)	Avg cov	Stdev cov	Avg cov/msg	Stdev cov/msg
(2)	953	681	276.61	1.59	0.41
(5)	1005	**974.67**	52.54	1.73	0.45
(6)	1031	**963.33**	110.35	1.71	0.39

(d) Original prompt template (6) vs. Prompt template with negative feedback (7), both with Coverpoint Type-based Sampling for stride detector bins, Successful Difficult Responses, Normal Tolerance, and Stable-restart Keeping best-messages.

Config	Max cov (/1034)	Avg cov	Stdev cov	Avg cov/msg	Stdev cov/msg
(6)	1031	963.33	110.35	1.71	0.39
(7)	963	865.5	137.89	1.58	0.79

I.1 Ablation experiment setups

We run “limited-message experiments” and “infinite-message experiments” on the Primitive Data Prefetcher Core module and the Ibex Instruction Decoder module. In a limited-message experiment, the experiment routine runs a trial with an upper bound of the number of responses that is smaller relative to the average response number for the trial to “exhaust” (as in Section 2.1). In an infinite-message experiment, the upper bound of the response number is much larger than the average response number for the trial to “exhaust”, allowing the agent to fully exert its capacity.

We use two LLMs in our experiment: OpenAI’s GPT-3.5-turbo-0613 model, and Meta AI’s Llama 2 7B model. For GPT 3.5, we use OpenAI’s API calls, with parameters as `temperature` $= 0.4$, `top_p` $= 1.0$, and `max_tokens` $= 600$. For Llama 2 7B, we put the downloaded Llama 2 directory in the same level as our project directory. We use one A100 GPU with 80GB memory to run the language model. We call the model with parameters as `temperature` $= 0.4$, `top_p` $= 0.9$, `max_gen_tokens` $= 800$, `max_seq_len` $= 10000$, and `max_batch_size` $= 4$. These parameters are decided empirically.

Other experimental setup follows Appendix E.

I.2 On Primitive Data Prefetcher Core

We run infinite-message experiments with several configurations of prompt templates and improvement measures on the Primitive Data Prefetcher Core module.

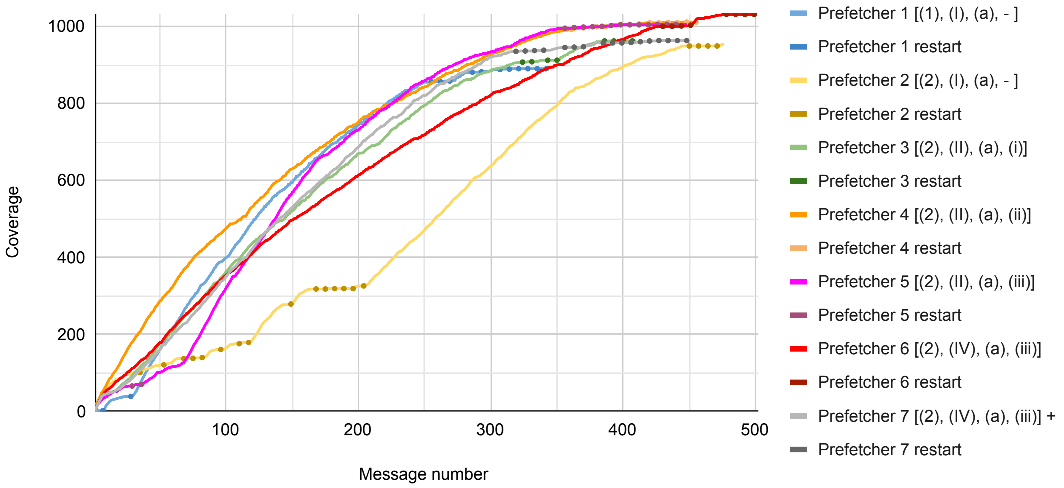


Figure 4: Infinite-message experiments on the Primitive Data Prefetcher Core module. Each line represents the trial reaching the maximum coverage on a configuration, and the dots on it show dialogue restarting points. All trials use *Original prompt template* except Prefetcher 7, which uses *Prompt template with negative feedback*.

The results are displayed in Table 3 and Figure 4. As we can see, in this module:

- **Coverpoint Type-based Sampling for stride detector bins** achieves better coverage but poorer coverage-per-message than *Pure Random Sampling*.
- **Successful Responses** and **Successful Difficult Responses** are generally better than *Recent Responses*.
- While *Keeping best-messages* and **Stable-restart Keeping best-messages** are generally better than *Clearing best-messages*, *Keeping best-messages* is least stable.
- *Prompt template with negative feedback*, which includes a one-sentence “what not to do” for each bin in iterative queries, makes no significant improvement than **Original prompt template**.

Based on these conclusions, we run fixed-budget experiments on the three most prominent combinations of strategies, as in Section 3.

I.3 On Ibex Instruction Decoder

First, we run limited-message experiments (at most 300 responses per trial) with several configurations of prompt templates (see Appendix B) and improvement measures (see Section 2.2) on the Ibex Instruction Decoder module.

The results are displayed in Table 4. As we can see, in this module:

- **Mixed Coverpoint Type-based and Pure Random Sampling for Ibex decoder bins** is better than *Coverpoint Type-based Sampling for Ibex decoder bins* in terms of average coverage, coverage stability, and average coverage-per-message.
- **Successful Difficult Responses** is generally better than *Successful Responses* and *Mixed Recent and Successful Responses*.
- While *Low Tolerance* has poorer coverage than **Normal Tolerance**, its coverage-per-message is significantly better, which means it covers easier bins faster; the **Coverage Rate-based Tolerance for Ibex decoder bins** that combines *Low Tolerance* and *Normal Tolerance* generally performs the best.
- The combination of *High Tolerance* and *Mixed Coverpoint Type-based and Pure Random Sampling for Ibex decoder bins* makes no significant improvement than *Normal Tolerance*

Table 4: **Results of limited-message experiments on the Ibex Instruction Decoder module**

(a) Coverpoint Type-based Sampling for Ibex decoder bins (1) vs. Mixed Coverpoint Type-based and Pure Random Sampling for Ibex decoder bins (2), both with Original prompt template, Successful Responses, Normal Tolerance, and Stable-restart Keeping best-messages.

Config	Max cov (/2107)	Avg cov	Stdev cov	Avg cov/msg	Stdev cov/msg
(1)	1251	872.5	505.96	3.66	0.5
(2)	1163	**1052**	**102.47**	**4.78**	0.61

(b) Successful Responses (1) vs. Mixed Recent and Successful Responses (3) vs. Successful Difficult Responses (4), all with Original prompt template, Coverpoint Type-based Sampling for Ibex decoder bins, Normal Tolerance, and Stable-restart Keeping best-messages.

Config	Max cov (/2107)	Avg cov	Stdev cov	Avg cov/msg	Stdev cov/msg
(1)	1251	872.5	505.96	3.66	0.5
(3)	1007	523	485.51	2.16	1.48
(4)	1625	**1413**	**195.79**	**4.98**	1.1

(c) Normal Tolerance (1) vs. Low Tolerance (5) vs. Coverage Rate-based Tolerance for Ibex decoder bins (6), all with Original prompt template, Coverpoint Type-based Sampling for Ibex decoder bins, Successful Responses, and Stable-restart Keeping best-messages.

Config	Max cov (/2107)	Avg cov	Stdev cov	Avg cov/msg	Stdev cov/msg
(1)	1251	872.5	505.96	3.66	0.5
(5)	1047	678	334.11	4.39	1.27
(6)	1430	**1179.67**	305.68	**4.5**	1.59

(d) Normal Tolerance (2) vs. High Tolerance (7), both with Original prompt template, Mixed Coverpoint Type-based and Pure Random Sampling for Ibex decoder bins, Successful Responses, and Stable-restart Keeping best-messages.

Config	Max cov (/2107)	Avg cov	Stdev cov	Avg cov/msg	Stdev cov/msg
(2)	1163	1052	102.47	4.78	0.61
(7)	1215	685.67	605.12	2.74	2.23

(e) Original prompt template (1) vs. Prompt template with one-line task introduction (8), both with Coverpoint Type-based Sampling for Ibex decoder bins, Successful Responses, Normal Tolerance, and Stable-restart Keeping best-messages.

Config	Max cov (/2107)	Avg cov	Stdev cov	Avg cov/msg	Stdev cov/msg
(1)	1251	872.5	505.96	3.66	0.5
(8)	1462	**1203.67**	323.38	4.01	1.08

Table 5: **Results of infinite-message experiments on the Ibex Instruction Decoder module:** all configurations use *Prompting template with one-line task introduction* and *Successful Difficult Responses*. (1)-(4) use *Stable-restart Keeping best-messages*; (1)(2) use *Coverage Rate-based Tolerance for Ibex decoder bins*, (3)(4) use *Normal Tolerance*; (1)(3) use *Mixed Coverpoint Type-based and Pure Random Sampling for Ibex decoder bins*, (2)(4) use *Coverpoint Type-based Sampling for Ibex decoder bins*. (5) uses *Keeping best-messages*, *Coverage Rate-based Tolerance for Ibex decoder bins*, and *Mixed Coverpoint Type-based and Pure Random Sampling for Ibex decoder bins*.

Config	Max cov (/2107)	Avg cov	Stdev cov	Avg cov/msg	Stdev cov/msg
(1)	**1807**	**1568**	338	2.55	0.62
(2)	1513	1302.5	297.69	2.83	0.41
(3)	**1861**	**1684**	250.32	3.85	0.07
(4)	1642	1322.33	472.17	2.71	0.56
(5)	721	369.5	497.1	0.85	0.18

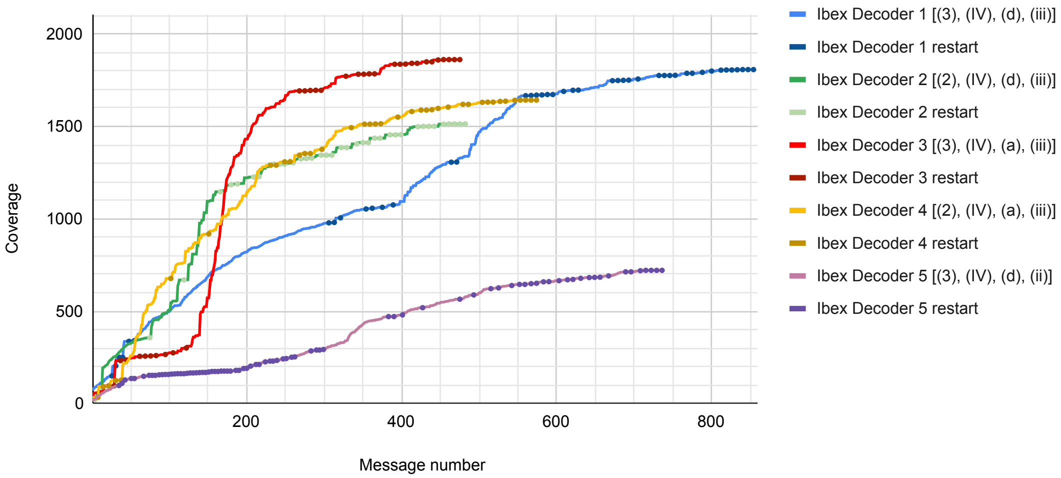


Figure 5: Infinite-message experiments on the Ibex Instruction Decoder module. Each line represents the trial reaching the maximum coverage on a configuration, and the dots on it show dialogue restarting points. All trials use *Prompt template with one-line task introduction*.

and *Mixed Coverpoint Type-based and Pure Random Sampling for Ibex decoder bins*, whereas its standard deviations are both larger.

- **Prompt template with one-line task introduction** is generally better than *Original prompt template*.

Second, we run infinite-message experiments with five configurations on this module. The results are displayed in Table 5 and Figure 5. As we can see, in this module:

- **Mixed Coverpoint Type-based and Pure Random Sampling for Ibex decoder bins** performs better than *Coverpoint Type-based Sampling for Ibex decoder bins* with longer trials.
- **Normal Tolerance** and **Coverage Rate-based Tolerance** performs similarly.
- *Keeping best-messages* performs much worse than **Stable-restart Keeping best-messages** on this module.

Based on these conclusions, we run fixed-budget experiments on the two most prominent combinations of strategies, as in Section 3.

Table 6: **Results of LLM comparison experiments** on the Ibex Instruction Decoder module. Both stimulus generation agents use *Prompting template with one-line task introduction*, *Coverpoint Type-based Sampling for Ibex decoder bins*, *Successful Responses*, *Normal Tolerance*, and *Stable-restart Keeping best-messages*.

Config	Max cov (/2107)	Avg cov	Stdev cov	Avg cov/msg	Stdev cov/msg
Llama 2 7B	17	15.67	2.31	0.43	0.08
GPT 3.5	1462	1203.67	323.38	4.01	1.08

I.4 LLM comparison

We compare the two choices of LLM by running limited-message experiments on the Ibex Instruction Decoder module. The results are displayed in Table 6. As we can see, GPT 3.5 is significantly better than Llama 2 7B. In other experiments with Llama 2 7B, we also observe frequent gibbering response that has never happened with GPT 3.5.