

INSTITUT FÜR INFORMATIK  
Ludwig-Maximilians-Universität München

LLM-ASSISTED GENERATION OF  
FORMAL-VERIFICATION HARNESSSES  
FOR THE INTEL TDX MODULE  
FIRMWARE

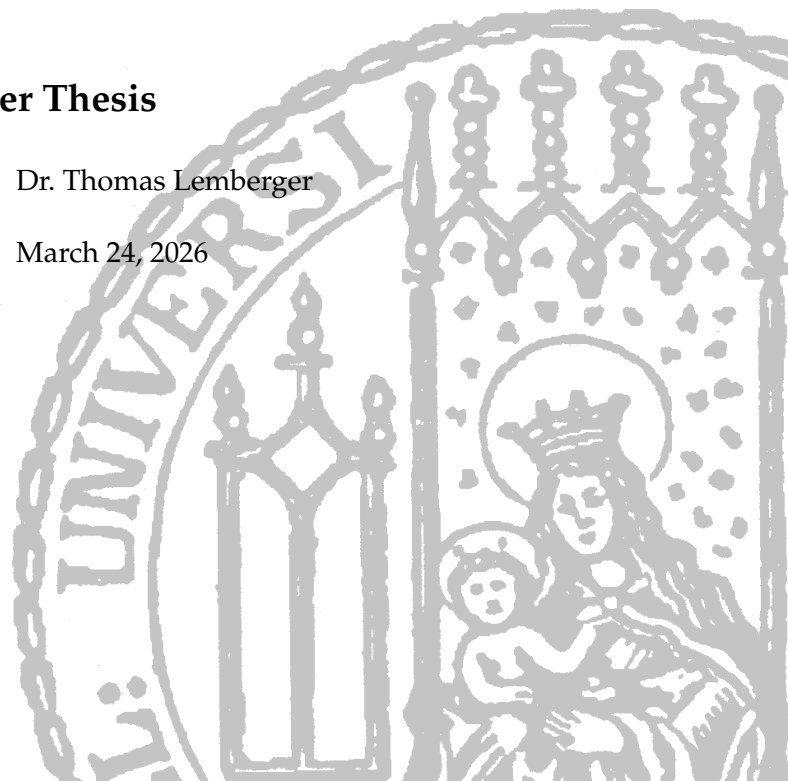
---

Johannes Tim Wildberger

**Master Thesis**

**Supervisor** Dr. Thomas Lemberger

**Submission Date** March 24, 2026





# Statement of Originality

*English:*

## **Declaration of Authorship**

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments.

*Deutsch:*

## **Eidesstattliche Erklärung**

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, March 24, 2026

Johannes Tim Wildberger



# Abstract

Writing the formal verification harnesses required to generate the verification tasks that make up a benchmark dataset to compare software verifiers is, as of now, a manual, costly, and time-consuming process. Large Language Models, with their innate ability to abstract and generalize in complex, knowledge-intensive domains, are a viable prospect for turning this manual process semi-automatic, thereby reducing costs.

We present LLM-Assisted Harness Generator (LAHG), an LLM-based formal-verification harness generator that enables the semi-automatic, dialog-driven generation of verification tasks using an off-the-shelf, untrained model, and demonstrate its capabilities on the Intel TDX module firmware.

LAHG employs an Internal Generative Rule Set (IGRS), refined through iterative feedback loops, and a three-stage Guardrailed Pipeline to translate natural-language specifications into structured C-harnesses and HarnessForge configurations in YAML.

The evaluation demonstrates that the framework successfully generates compilable, HarnessForge-compatible artifacts, with configurations utilizing a manually curated sequence of examples, achieving a higher average success Stage of 2.0 per refinement bucket compared to 1.6 for random sequences. While two of the nine tested configurations produced verification tasks that were formally verifiable, achieving full semantic satisfaction for formal verifiers remains a challenge.

These results imply that while LLMs can structurally bridge the gap between specifications, implementations, and verification harnesses, the precise logical encoding required for fully automated formal verification remains an open challenge.



# Contents

<b>Contents</b>	<b>v</b>
<b>List of Figures</b>	<b>vi</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>3</b>
2.1 Intel TDX . . . . .	3
2.2 HarnessForge . . . . .	4
2.3 Google Gemini . . . . .	6
2.4 File Type Conversion with Docling . . . . .	6
2.5 Measurements . . . . .	7
<b>3 Related Work</b>	<b>9</b>
<b>4 Contribution</b>	<b>11</b>
4.1 Internal Generative Rule Set . . . . .	12
4.2 Programmatic IGRS Generation . . . . .	12
4.3 Guardrailed Pipeline . . . . .	17
4.4 MCP Tool Suite . . . . .	18
4.5 Gemini Configuration . . . . .	19
4.6 LAHG . . . . .	22
<b>5 Evaluation</b>	<b>23</b>
5.1 Experimental Setup . . . . .	23
5.2 Questions and Answers . . . . .	24
5.3 IGRS Generation . . . . .	25
5.4 Harness-config pair generation . . . . .	33
<b>6 Conclusion</b>	<b>37</b>
6.1 Research Findings . . . . .	37
6.2 Limitations . . . . .	38
6.3 Conclusion . . . . .	38
6.4 Future Work . . . . .	38
<b>Bibliography</b>	<b>41</b>

# List of Figures

1.1	LLM-Assisted Harness Generator (LAHG) . . . . .	2
2.1	Intel TDX architecture and isolation boundaries [20]. . . . .	3
2.2	HarnessForge YAML schema for the verification task expected__safety__requirement . . . . .	4
2.3	Guest-side (tdg) vs. host-side functions (tdh) . . . . .	5
2.4	Left to Right: pdf, md, yml . . . . .	7
2.5	Semantically identical but syntactically different numbered lists . . . . .	8
4.1	Initial rule set for the IGRS generation . . . . .	13
4.2	Iterative refinement loop for IGRS generation . . . . .	14
4.3	Side-by-side comparison of the <i>Generate</i> and <i>Compare</i> prompt templates . . . . .	15
4.4	Few-Shot Example wrapped in XML in prompt template. . . . .	16
4.5	3-stage Guardrailed Pipeline . . . . .	17
4.6	Document structure of LAHG's <code>gemini.md</code> project-configuration. . . . .	19
4.7	Human-In-The-Loop-In-Action . . . . .	21
4.8	LAHG . . . . .	21
5.1	Experimental environment . . . . .	24
5.2	Cross-comparison of all files' Embedding Cosine Similarity . . . . .	26
5.3	Sequential IGRS Cosine Similarity and global similarity for pdf data-spec . . . . .	27
5.4	Sequential IGRS Cosine Similarity and global similarity for pdf full . . . . .	28
5.5	Stages reached for all harness-config generation runs . . . . .	31
5.6	Config-harness generation runs: Number of used input tokens vs. highest reached Stage . . . . .	32
5.7	Stages reached for input split data-spec and all input types. . . . .	34
5.8	Harness-config generation runs: Input tokens vs. highest Stage reached . . . . .	35

# 1 Introduction

SV-COMP, the International Competition on Software Verification [6], allows comparing formal verifiers in terms of performance and effectiveness on curated benchmark datasets by enforcing a unifying standard syntax for verification tasks that all competitors agree on. This allows a fair competition on various data sets and tasks. These benchmark datasets currently consist of software and hardware models, but not of firmware, since that is mostly proprietary and kept unavailable for the general public.

The Intel TDX Module Benchmark Set [11] is the first open-source firmware benchmark dataset that enables the automated creation of verification tasks for comparing and evaluating software verifiers through SV-COMP.

Intel TDX is a trusted domain extension for Intel CPUs that isolates Virtual Machines from the surrounding environment. It acts as a secure intermediary that orchestrates interactions between the untrusted host and the trusted guest, manages the secure provisioning of resources, and enforces isolation for critical execution sections.

Manually writing verification tasks is error-prone. HarnessForge [33], a single-file verification task generator, automates this process. It requires a harness in the programming language C, as well as a YAML configuration file, thereby reducing the manual effort to generate these files. However, this still requires an engineer with both domain knowledge and expertise in software verification. It took several months to create verification tasks for 22 of the 109 interface functions of the Intel TDX Firmware Module.

In this project, we present the LLM-Assisted Harness Generator (**LAHG**), which employs Google Gemini to abstract and generalize the relationship of a specification document and its corresponding implementation, to aid a software verification engineer in a semi-automatic fashion in expanding the Intel TDX firmware benchmark dataset.

LAHG addresses the LLM's non-determinism and its innate tendency to hallucinate on complex, knowledge-intensive tasks [4, 34, 38] by employing a sequential pipeline that acts as a multi-stage safeguard, splitting the broad task of "harness generation" into more efficiently manageable chunks. An MCP server implements tools that invoke the programs that handle these chunks. As displayed in Fig. 1.1, the user interacts with the Gemini CLI, which acts both as a generator for the harnesses and YAML configuration files, as well as an intermediary to interact with the MCP tool suite. The pipeline's internal feedback loop passes relevant information from the tools to the model, ensuring a tight coupling between the LLM's actions and their impact on progress toward the general objectives.

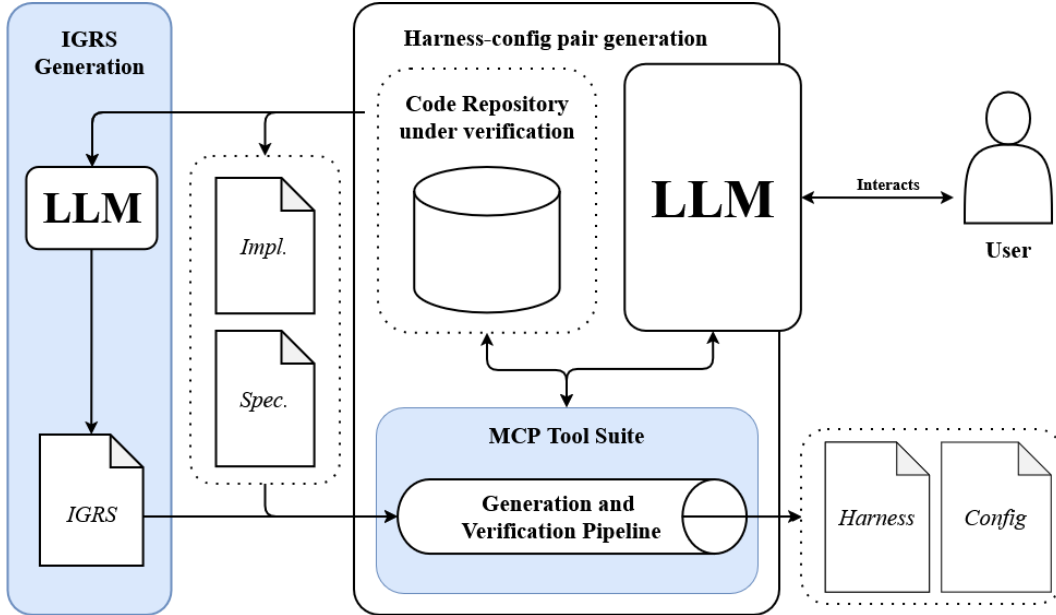


Figure 1.1: LLM-Assisted Harness Generator (LAHG)

The tool suite reduces LLM token consumption, increases input-window effectiveness, and provides ease-of-use features to enhance usability.

We evaluate LAHG's performance against the already harnessed interface functions of the Intel TDX benchmark dataset and address these questions:

**RQ. 1:** *Does providing the entire specification document exceed the LLM's input window?*

**RQ. 2:** *Does converting the specification document from a layouted format to a textual or structural format improve the model's understanding of this document?*

**RQ. 3:** *Does altering the sequence of few-shot examples influence the model's output?*

**RQ. 4:** *Is the ideal input type for generating structured content with the Gemini API also ideal for generating structured content with the Gemini CLI?*

The resulting tool enables the user to create compilable harnesses that are compatible alongside their YAML configuration files with HarnessForge and thus fit into the SV-COMP ecosystem. It shows that LLMs are highly susceptible to the file type of their inputs and highly limited in effectiveness due to their current input and context window sizes.

We believe that despite the technical limitations of modern LLMs, significant advancements can be made by exploring the impact of prompt engineering on this specific problem and investigating changes in model understanding across varying input file types.

## 2 Background

### 2.1 Intel TDX

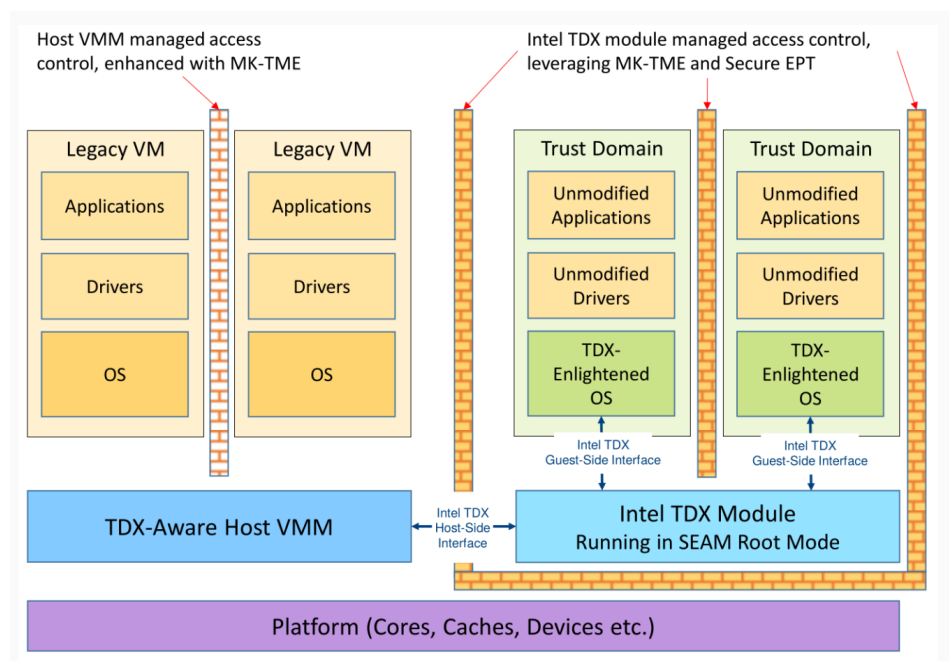


Figure 2.1: Intel TDX architecture and isolation boundaries [20].

Intel TDX establishes hardware-enforced "Trust Domains" which isolate virtual machines from unauthorized host-side access, effectively removing the Virtual Machine Monitor (VMM) from the Trusted Computing Base.

As shown in Fig. 2.1, the security boundary shifts to a privileged Intel TDX Module running in Secure Arbitration Mode (SEAM) to manage guest resources and behavior securely. This architecture ensures the confidentiality and integrity of the guest environment, shielding it even if the underlying host or VMM is compromised.

**ABI specification.** The reference specification document for the Application Binary Interface (ABI) [21] is a 430-page pdf document that describes the entire TDX Module architecture. It details the data types, control structures, and interface functions used by both the host-side and guest-side to manage resources, memory isolation, and attestation.

```

- name: tdg_mr_report__expected__safety_requirement
  target:
    filename: formal/harness/tdg_mr_report_harness.c
    method: tdg_mr_report__call
  before_target:
    - filename: formal/src/initialization.c
      method: init_tdg_mr_report
    - filename: formal/harness/tdg_mr_report_harness.c
      method: tdg_mr_report__expected__precond
  after_target:
    - filename: formal/harness/tdg_mr_report_harness.c
      method: tdg_mr_report__expected__postcond
    - filename: formal/src/initialization.c
      method: close_tdg_mr_report_with_check
  properties:
    - property_file: ../../../../properties/unreach-call.prp
      expected_verdict: true
    - property_file: ../../../../properties/coverage-error-call.prp
      expected_verdict: true

```

Figure 2.2: HarnessForge YAML schema for the verification task `expected__safety__requirement`

The ABI defines two primary instruction sets: `tdh` interface functions, which are host-managed operations, and `tdg` functions, which provide the guest-side interface for interacting with the TDX Module. The function's specification details the pre- and post-conditions that must hold before and after its execution in natural language.

## 2.2 HarnessForge

HarnessForge [33] is a tool developed by a group of researchers at the LMU Munich, Intel, and the National Taiwan University, which "automatically creates a single-file verification task for user-specified functions and properties from a source-code repository in the programming language C by extracting the necessary dependencies of the functions under verification."

It requires two primary inputs: a C harness, which utilizes assertions to encode the pre- and post-conditions for a target function, and a YAML configuration file that orchestrates the generation of verification tasks.

In the remainder of this thesis, *config* refers to a HarnessForge configuration file and *harness* refers to such a C harness. We refer to the combination of a C harness and a corresponding YAML configuration as *harness-config pair*.

### 2.2.1 Configuration Schema

Figure 2.2 shows an exemplary element of the list of verification tasks specified in a YAML configuration file for HarnessForge. Each verification task has a unique descriptive name.

The `target` is the function under verification, and the blocks `before_target` and `after_target` encode what properties must hold before and after the function execution.

<b>Guest-side Interface Functions</b>	<b>Host-side Interface Functions</b>
1. <code>tdg_mr_report</code>	1. <code>tdh_export_restore</code>
2. <code>tdg_servtd_wr</code>	2. <code>tdh_import_abort</code>
3. <code>tdg_sys_rd</code>	3. <code>tdh_mng_addcx</code>
4. <code>tdg_vm_wr</code>	4. <code>tdh_mng_create</code>
5. <code>tdg_vp_enter</code>	5. <code>tdh_mng_init</code>
6. <code>tdg_vp_vmcall</code>	6. <code>tdh_mng_key_config</code>
	7. <code>tdh_mng_key_freeid</code>
	8. <code>tdh_mng_vpflushdone</code>
	9. <code>tdh_mr_finalize</code>
	10. <code>tdh_phymem_page_reclaim</code>
	11. <code>tdh_sys_config</code>
	12. <code>tdh_sys_init</code>
	13. <code>tdh_sys_key_config</code>
	14. <code>tdh_sys_shutdown</code>
	15. <code>tdh_sys_update</code>
	16. <code>tdh_vp_enter</code>

Figure 2.3: Guest-side (tdg) vs. host-side functions (tdh)

Each method corresponds to a method that is called during a task execution. The `filename` attribute specifies the path to the file implementing the method, defined relative to the HarnessForge working directory. The harness contains methods that cover the logic paths of each pre- and post-conditions in the specification document. It not only covers whether a given condition is satisfied, but also handles expected and unexpected behavior. The latter might occur if a function under verification depends on user input that cannot be known beforehand.

A configuration file can contain tasks for multiple levels of execution progression.

- `safety_requirement` covers a successful initialization of the target.
- `cover_entry` injects an `assert(False)` after the initialization to prove that this part of the function under verification is verifiably correct.
- `cover_exit` injects this `assert(False)` after the function execution.

The HarnessForge output is a single-file verification task in the programming language C that various formal software verifiers can execute.

The ABI details 109 interface functions. Figure 2.3 presents a side-by-side comparison of the 22 functions for which a harness has been created; of these, six are guest-side, while the remaining 16 are host-side interface functions.

## 2.3 Google Gemini

Google Gemini [17] is an LLM developed by Google, available via a web interface and an API Service, with a SDK [18] for Python and a local Command-Line Interface (Gemini CLI). The Gemini CLI is a stateful wrapper of the stateless Gemini API that can be configured and extended. The use of both is subject to the underlying payment plan, which imposes limits on the number of usable tokens per day and per minute, as well as requests per day and per minute. Google bills users of its LLM models for the number of tokens used to represent the user’s input, as well as the number of tokens used to represent the LLM’s output.

### 2.3.1 Gemini API

The Gemini API allows users to upload files to a File Search Store [18]. An account can have multiple File Search Stores. The File Search Store natively supports `pdf`, `md`, and `yml`. A file of any other type must be converted to a supported format to be accessible for the API. The API is stateless. Therefore, it has no internal history or prior context. The only way to bypass this lack of history is to upload instructions to the File Search Store and instruct the model to follow them.

### 2.3.2 Gemini CLI

The Gemini CLI can be installed locally and has file system access to all files within its installation root and its subdirectories. It can be configured with a Markdown file `gemini.md` that allows the user to detail the model’s role and objective, and restrict its access to the file system or narrow its focus by specifying behavioral rules.

It provides a built-in tool suite for file system access and web search. These tools can be extended with Anthropic’s Model Context Protocol Server [3], which serves as an entry point for the model to use custom-built programs and scripts. Each tool execution can be invoked by entering a custom command in the CLI or by referencing it in natural language.

## 2.4 File Type Conversion with Docling

The layout document format `pdf` is designed for human readability rather than machine understandability. It encodes information visually but not structurally, which forces LLMs to apply image recognition to extract, parse, and understand the content of a `pdf` document. Image recognition is costly and adds uncertainty into the model’s understanding, since inferring a semantic relationship from the visual grouping of elements is more ambiguous than extracting it from formatted text in Markdown or from a structured container in YAML.

Docling [25], an open-source toolkit from IBM Research, allows efficient parsing of `pdf` documents. It encodes the parsed information in its unifying internal representation *Docling Document*, which can be converted to other formats such as `md` or `yml`.

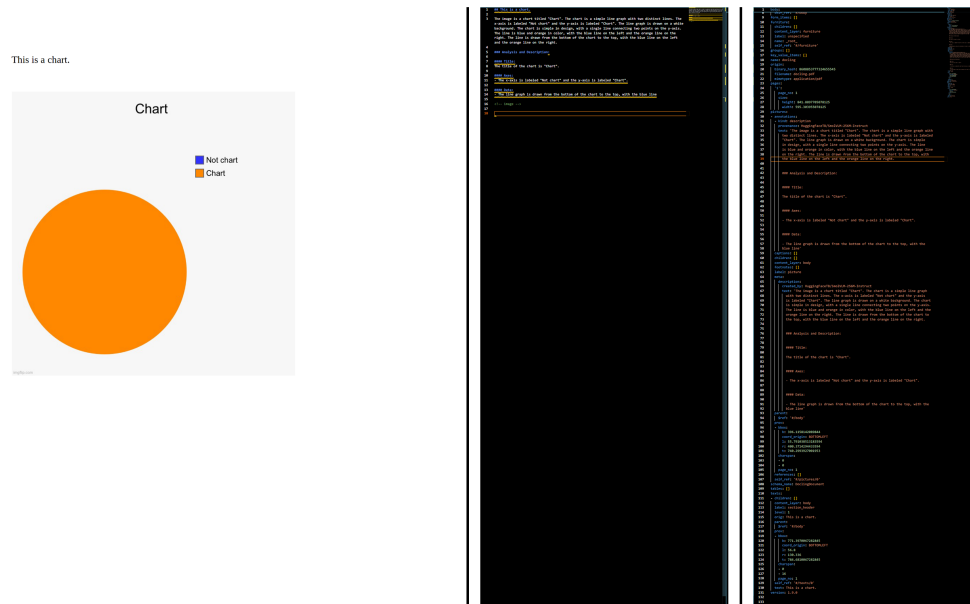


Figure 2.4: Left to Right: pdf, md, yaml

Figure 2.4 displays from left to right a pdf document and the result of the Docling conversions to md and yaml. The pdf file is displayed with its original proportions. It is important to note that image description was activated during the conversion. Otherwise, the md would have contained the text "This is a chart.", followed by an image placeholder. Note that while the original pdf only contains one sentence in the first line and an image, the resulting yaml contains 131 lines of text, whereas the md contains only 16 lines of text. This demonstrates the discrepancy of the pdf's visual encoding and the verbose textual representations of yaml. We deliberately chose this example to illustrate the stark difference in textual content between the original pdf and the extracted md and yaml files. While md contains more than 171,000 words, roughly the same textual content as the original pdf document, yaml conveys the same information in more than 12 times the text with over 2,160,000 words.

## 2.5 Measurements

### 2.5.1 Embedding Cosine Similarity

LLMs are not deterministic, and therefore, the outputs of two tasks with identical circumstances vary in naming, formatting, and content. Capturing a meaningful difference between LLM-generated files can thus be done by either comparing textual or semantic representations. The former requires significant preprocessing to exclude differences that might be purely syntactic rather than semantic. At the same time, the latter can be efficiently determined by the Cosine Similarity of two embedding vectors.

**Numbered List 1**

1. Item One
2. Item Two
3. Item Three

**Numbered List 2**

- a. Item One
- b. Item Two
- c. Item Three

Figure 2.5: Semantically identical but syntactically different numbered lists

The Gemini API offers a programmatic approach [16] for retrieving embedding vectors for a prompt or a file in the File Search Store. The user can choose the model and the embedding space. The embedding converts a datum to a vector representation in the embedding space. The vector is the semantic representation of the datum.

The Embedding Cosine Similarity is calculated as the dot product of two vectors divided by the product of their magnitudes:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} \quad (2.1)$$

A Cosine Similarity of 1 means the two vectors, and thus their physical representations, are identical.

Both numbered lists in Fig. 2.5 are semantically nearly identical, whereas they differ syntactically. While a syntactical analysis, such as a Levenshtein distance, will report only the lexical differences, the Cosine Similarity captures the semantic differences. In this case, the semantic difference is 0.

Therefore, our main metric for comparing files is Cosine Similarity, since it captures relevant semantics and is less sensitive to minor syntactic differences than a comparison based on the textual representation. The Cosine Similarity ranges from 0 to 1, with 0 meaning "absolutely dissimilar" and 1 meaning "entirely identical". Since the embedding of C code, yml, md, and pdf is but a tiny aspect of the embedding space of modern LLMs, which are trained on various subjects, languages, and artifacts, there is no quantitative value in two Cosine Embedding Scores except for their relative value.

An example: There are three documents, A, B, and C. A and B have a Cosine Similarity of 0.84 and A and C have a Cosine Similarity of 0.94. There is no quantitative analysis to be made other than "A is more similar to C than to B". This enables ordering by score, but it does not allow for subtracting or inferring how much more similar the two files are, since this score represents a distance in the embedding space rather than another vector with a direction.

## 3 Related Work

**Prompt Engineering.** A model’s performance for a given task correlates with the quality of the initial prompt. Providing no context (Zero-Shot Prompting) works well on easy code generation tasks, but the resulting code quality is generally lower than with One-Shot Prompting [27, 29], where a usually positive example is provided in the prompt.

In general, adding context improves the model’s output quality [15]. Adding multiple examples into a prompt (**Few-Shot Prompting**), in addition to, and in combination with **Chain-of-Thought Reasoning**, produces better results for complex tasks than Zero-Shot and One-Shot Prompting [12, 37, 39].

Chain-of-Thought Reasoning in combination with an Internal Digest shows promising results when combined with Iterative Refinement [28].

In addition, structuring the prompt further increases the model’s performance and hinders hallucinations [34].

We use Few-Shot prompting in combination with Iterative Refinement, Chain-of-Thought Reasoning, and an Internal Digest in this project.

**LLM-based code generation.** LLMs are innately prone to hallucinate [4, 34, 38]. Therefore, the reliability and quality of LLM-generated code must be monitored [40]. This is especially crucial for the employment of Agentic LLMs as Coding Agents for safety-critical code. In this project, we secure critical sections using a Human-In-The-Loop (HITL) protocol to monitor the LLM’s intent and actions.

**Combining automated testing with internal feedback** [27] improves the quality of LLM-generated code significantly. For simple use cases, adopting Test-Driven Development, by instructing the LLM to write its own tests, increases the resulting code quality [2, 31]. However, this performance drops significantly for harder problems.

**Iterative Refinement** [1, 9, 29, 32] works well for **LLM-based code repair and patching** on firmware and software. Additional **self-reflection through automated internal feedback** significantly increases code quality [10, 28].

However, it introduces latency and additional iterations, thereby increasing cost. Employing automated feedback with Iterative Refinement [24, 26] outperforms One-Shots for complex tasks.

We value a positive impact on code quality more than efficiency, which is why we opt for Automated Internal Feedback without Automated Testing.

**LLM Guardrails.** Agentic LLMs introduce complexity and unpredictability to the user interaction, which can be mitigated by applying Safeguards or Guardrails [14]. Program-based guardrails [13] introduce less friction to the user interaction than

HITL protocols, but also provide significantly lower security assurance for critical tasks. We combine program-based guardrails along with a HITL protocol.

**Retrieval Augmented Reasoning.** Retrieval-Augmented Reasoning (RAG) improves context-awareness in LLMs for knowledge-intensive tasks [23].

**LLM evaluation.** While LLMs are capable of self-correction [10], introducing HITL and external tools shows higher correction accuracy than internal reasoning [35]. Especially in domains with limited data, the quality of self-correction performance reviews varies substantially [7, 30]. However, the overall quality of the initially generated code can be on par with results from domains with large amounts of available data, if curated properly.

## 4 Contribution

**Domain requirements.** LAHG reduces the required domain knowledge of the software verification engineer by utilizing an LLM’s capability to abstract and generalize. It turns the manual process for writing a harness and a matching config into a semi-automatic LLM-guided pipeline. By abstracting the latent rules that lead from the specification and the implementation to a valid harness, the LLM presents a draft to the engineer who can then accept, modify, or reject it and request further adaptations.

The relationship between the ABI specification and a function’s implementation, as well as the corresponding harness and its matching config, presents certain requirements:

- Req. 1** *The harness must be written in syntactically correct C.*
- Req. 2** *The config must be valid YAML in the HarnessForge-compatible schema as described in Fig. 2.2.*
- Req. 3** *The harness must reference the correct code such that the verifier can run the resulting verification task.*
- Req. 4** *The harness must correctly encode the function’s pre- and post-conditions from the specification.*

A successful compilation of the harness ensures compliance with **Req. 1**.

HarnessForge can verify **Req. 2** by a successful verification task generation by HarnessForge, thereby proving that the config functions as a proper orchestrator. The verifier evaluates whether the verification tasks, generated by HarnessForge, are verifiable, and checks whether the harness encodes the specification’s pre- and post-conditions such that they are satisfiable, thereby covering **Req. 3** and **Req. 4**. LAHG employs Gemini CLI with an MCP server, which enables the LLM to invoke a compilation, a HarnessForge call, and a verifier call. The requirements for a sound harness build upon each other. HarnessForge cannot use a harness with syntactically incorrect C-code, and a verifier must receive a valid verification task. Therefore, these requirements serve as safeguards within a sequential toolchain that guides the LLM toward generating a sound harness. We refer to this toolchain as the **Guardrailed Pipeline**.

Gemini CLI is a stateful wrapper for the stateless Gemini API that stores the session history in a dedicated log file. When starting a new session, the model is a "blank slate" and unaware of any previous interactions. Thus, the model is not pretrained for this specific task. To inject sufficient context and provide fundamental rules that contain the guidelines and knowledge to generalize from the specification document

and implementation to a harness-config pair, LAHG provides the LLM with an **Internal Generative Rule Set** (IGRS).

In this chapter, we cover the concept of an Internal Generative Rule Set first in [Sect. 4.1](#) before detailing its programmatic generation in [Sect. 4.2](#). We then present the details on the Guardrailed Pipeline in [Sect. 4.3](#), followed by a section about the MCP tools used in this project in [Sect. 4.4](#).

## 4.1 Internal Generative Rule Set

The IGRS consists of two sets of rules, derived from a manually written rule set, that aim to capture the relationship among specification and implementation, harness and config, and their combination.

[Figure 4.1](#) shows this initial rule set, which covers the orchestration structure of a verification task (Setup, Exercise, Verify, and Cleanup), and includes basic rules to ensure adherence to the HarnessForge YAML schema. The rule set adopts the terminology defined in both the Google Gemini project configuration `gemini.md` and the IGRS generation prompts. All bolded terms belong to this unifying terminology.

The **Orchestration Structure** is split into the phases **Setup**, **Exercise**, **Verify**, and **Cleanup**, which correspond to the structure of a verification task in the YAML schema for a HarnessForge config. Explicitly stating the abstract objectives that make up each phase sensitizes the model to capture each pre- and post-condition from the specification in natural language and to codify them in the harness.

The **Configuration & Hook Sequence** handles the naming, formatting, and structuring of the individual verification tasks in the config.

## 4.2 Programmatic IGRS Generation

LAHG employs an IGRS that has been created using a generative loop with Iterative Refinement and the Gemini API. We refer to this process as the **Iterative Refinement Loop**. This approach ensures the optimal IGRS for the given task. In this instance, using the Gemini API is preferable to using the Gemini CLI, as the former is entirely stateless and supports programmatic invocation. In contrast, the latter is limited to manual use.

[Figure 4.2](#) displays the order of events in the Iterative Refinement Loop. It shows the two-step sequence of `Generate` and `Compare` prompts. Each iteration comprises a single pair of these prompts, hereafter referred to as a *generate-compare*. Each generate-compare is done for a different function. We refer to the function of an iteration as *example*. This example differs from the few-shot examples shown in [Fig. 4.3](#), which refer to functions that are merely references and not an active part of the generation process.

### 1. Orchestration Structure

**Setup:** Define a primary **Harness Orchestrator** entry point (conventionally `[method]__call`).

**Setup:** Map the **Initial Snapshot** to the implementation's interface arguments per **Specification**.

**Setup:** Capture and persist all **Operation Outputs** for validation in the **Post-condition Phase**.

**Exercise:** Encapsulate conditional logic for state or branch validation into modular, reusable predicates.

**Exercise:** Implement a common **Pre-condition Phase** to apply universal constraints.

**Exercise:** Implement specialized **Pre-conditions** that extend the common **Pre-condition Phase** with task-specific **Symbolic Constraints**.

**Verify:** Implement a common **Post-condition Phase** to enforce **Invariants**.

**Verify:** Explicitly compare the **Terminal State** against its **Initial Snapshot** to ensure no unintended **Side Effects** occurred.

**Verify:** Define **Path-specific** assertions to validate status codes or expected **Terminal States** according to the **Specification**.

**Cleanup:** Implement a dedicated **Cleanup Phase** to deallocate all **Symbolic Resources**.

### 2. Configuration & Hook Sequence (CONF)

**RULE 1:** Task naming must follow the pattern:

`[method]__[path_classification]__[task_type]`

**RULE 2:** The target method in the configuration must literally match the **Harness Orchestrator** name.

**RULE 3:** Sequence Invariant (Setup): **Initialization Phase** → **Pre-condition Phase**.

**RULE 4:** Sequence Invariant (Cleanup): **Post-condition Phase** → **Cleanup Phase** → `Environment Close`.

Figure 4.1: Initial rule set for the IGRS generation

**Step 1:** `Generate`, instructs the LLM to inspect the specification and implementation of a given function and generate a harness-config pair by adhering to an IGRS<sup>1</sup>.

**Step 2a:** `Compare` then orders the LLM to identify the differences of the generated harness-config pair and their human-written counterparts. **Step 2b:** From there, it is instructed to infer a set of rules that lead to a better generation result in the subsequent iterations. We call this the *Delta Rule Set*. Finally, the LLM refines the existing IGRS by incorporating this rule set.

<sup>1</sup>In the first iteration of the Iterative Refinement Loop, no IGRS exists yet. Therefore, the LLM uses the initial rule set as seen in [Figure 4.1](#).

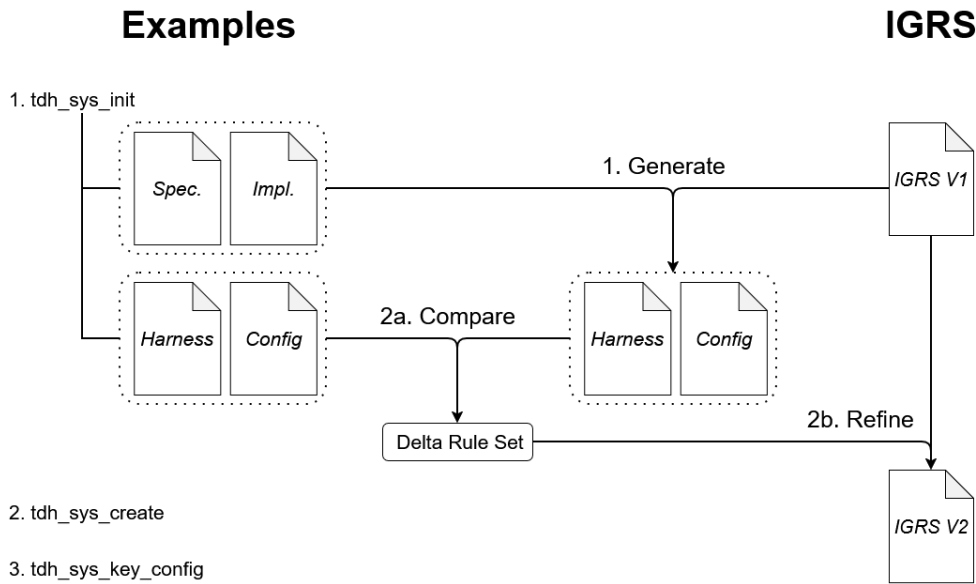


Figure 4.2: Iterative refinement loop for IGRS generation

We calculate the Embedding Cosine Similarity of the IGRS' of all successive non-initial iterations of the Iterative Refinement Loop. As the LLM sees more and more examples, the semantic novelty in each iteration approaches 0 since the existing rule set encompasses most of the abstract space. Therefore, the Embedding Cosine Similarity between iterations approaches 1. At some point, the difference between the two IGRS' becomes marginal, and thus, we stop the training since there remains no novelty in new examples.

### 4.2.1 Prompt Templates

Figure 4.3 shows a side-by-side comparison of the two prompt types. While they are structurally similar, they differ in their role definition for the model and the task. Each section of the prompt serves a distinct purpose.

The **Role Definition** defines the expectations on the model.

The **Task** presents the overall objective.

The **Context** contains the names of the files that have been uploaded to the Gemini File Search Store. Note that while we could have added their content to the prompt, they serve as reference material and are not part of any actual instruction. The only exception is the specification documents, which are often too large to fit into the input window of any model.

The **Unifying Terminology** Section ensures a tight coupling of the terms in the prompt and their purpose by providing a brief description of what is meant by each term.

The **Few-Shot Examples**, as displayed in Fig. 4.4, are wrapped in XML tags. These are semantic indicators that help the LLM interpret the examples as code, which is separate from the instructional content in the prompt.

Generate Prompt Template	Compare Prompt Template
<p><b>Role Definition</b>            (...) You specialize in architectural analysis, the synthesis of formal verification harnesses, and the orchestration of proof-based verification tasks.</p>	<p><b>Role Definition</b>            (...) You specialize in architectural gap analysis, structural pattern recognition, and abstracting technical discrepancies into formal generative rule sets. (...)</p>
<p><b>Task</b>            Your objective is to generate a C verification harness and a corresponding YAML configuration file by analyzing an implementation against its authoritative specification.</p>	<p><b>Task</b>            Your objective is to derive an IGRS by performing a structural audit. You will compare a set of "Generated Artifacts" against a verified "Reference Ground Truth" to identify architectural deviations.</p>
<p><b>Context</b></p> <ul style="list-style-type: none"> <li>• Implementation</li> <li>• Specification</li> <li>• (IGRS)</li> </ul>	<p><b>Context</b></p> <ul style="list-style-type: none"> <li>• Generated Artifacts</li> <li>• Reference Artifacts</li> <li>• Implementation</li> <li>• Specification</li> <li>• (IGRS)</li> </ul>
<p><b>Unifying Terminology</b>            (...)</p>	<p><b>Unifying Terminology</b>            (...)</p>
<p><b>Few-Shot Examples</b>            (...)</p>	<p><b>Few-Shot Examples</b>            (...)</p>
<p><b>Initial Rule Set</b>            (...)</p>	<p><b>Initial Rule Set</b>            (...)</p>
<p>-</p>	<p><b>Meta-Abstraction Heuristics</b>            (...)</p>
<p><b>Objective &amp; Strategy</b>            (...)</p>	<p><b>Objective &amp; Strategy</b>            (...)</p>
<p><b>Output Format &amp; Internal Digest</b>            Perform a three-way mapping (...):</p> <ol style="list-style-type: none"> <li>1. Identify a logical requirement (...).</li> <li>2. Cite the <b>Rule ID</b> (...).</li> <li>3. Map the <b>Taxonomy Role</b> (...).</li> </ol> <p><b>Example:</b> 'Spec requires buffer alignment...'</p>	<p><b>Output Format &amp; Internal Digest</b></p> <ol style="list-style-type: none"> <li>1. Identify a discrete block of code.</li> <li>2. Determine its verification purpose.</li> <li>3. Map to <b>Taxonomy Role</b> and 5-phase lifecycle.</li> <li>4. Justify the creation of a new IGRS rule.</li> </ol>

Figure 4.3: Side-by-side comparison of the *Generate* and *Compare* prompt templates

Note that the variable denoted by \$ is programmatically substituted with the content of, in this case, the specification of the first example function. These few-shot examples provide the LLM with a positive example, increasing its output quality [8].

Each few-shot example was dynamically randomly sampled from the pool of the 22 human-generated harness-config pairs. Random Sampling with a Sliding Window Exclusion (FIFO Queue) prevents a template from being populated with examples used in a recent or current iteration, thereby maximizing novelty among the examples.

```
1 <example id="1">
2   <context_metadata>
3     <description>
4       Official ABI specification and implementation (...).
5       Use these as the authoritative source (...).
6     </description>
7   </context_metadata>
8   <document_role type="specification">
9     ``text
10    \${EXAMPLE_1_SPEC
11     ``
12   </document_role>
13   (...)
14 </example>
```

Figure 4.4: Few-Shot Example wrapped in XML in prompt template.

The **Initial Rule Set** contains the initial rule set as displayed in Fig. 4.1. It serves as a semantic anchor, ensuring that the LLM remains within its domain and does not hallucinate.

The **Meta-Abstraction Heuristics** exist only in the `Compare` prompt and guide how to internalize the semantic differences between the generated and reference artifacts. Furthermore, this section instructs the model on how to update the existing rule set positively.

The **Objective & Strategy** section is a detailed version of the initial **Task** and provides specific instructions for consolidating the findings.

The **Output Format & Internal Digest** is the final instruction to the model. Since the few-shot examples can be quite large, the model tends to "forget" its initial instruction and the specific rules it has to follow. The Internal Digest ensures that the context is kept alive, thereby forcing the model to use as much of its context window as possible.

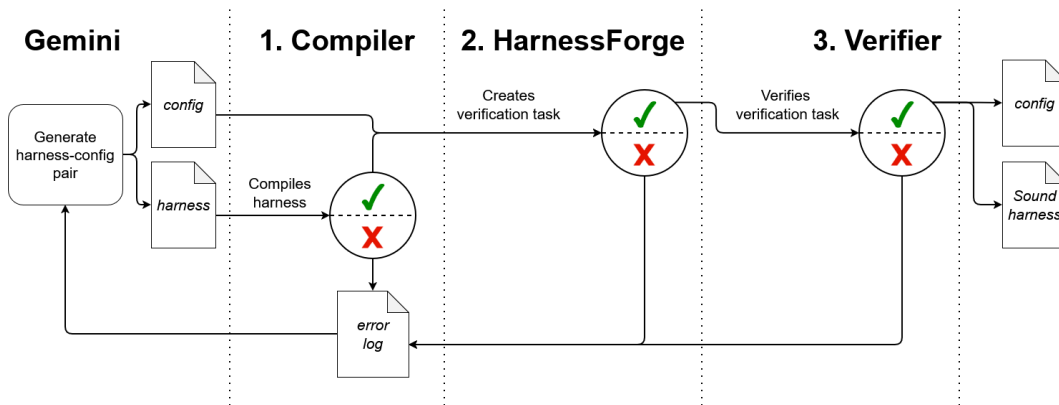


Figure 4.5: 3-stage Guardrailed Pipeline

### 4.3 Guardrailed Pipeline

The Guardrailed Pipeline shown in Fig. 4.5 consists of 3 main stages that each serve as a gate, which the harness-config pair must pass to proceed. Initially, Gemini is given an IGRS, the specification document, and the implementation of the function under verification. With these files, it generates the initial harness-config pair.

In **Stage 1**, the model invokes `clang`, the compiler used in LAHG<sup>2</sup>. `clang` then performs a diagnostic check of the harness, including preprocessing, lexing, parsing, and the semantic analysis. It does not assemble since Stage 1 only ensures a syntactically correct C-harness.

Upon successful compilation, the harness-config pair is passed to HarnessForge in **Stage 2** to ensure that harness and config match.

If no issues arise, in **Stage 3**, each resulting verification task is passed to CBMC, the software verifier used in LAHG<sup>3</sup>.

Once all verification tasks from Stage 2 have passed Stage 3, the harness is declared sound and added to the benchmark dataset.

If the MCP tools report errors during any Stage, their output is logged and fed back to the model as a return message of the respective MCP tools. The LLM can then interpret the output and use the information to regenerate an improved harness-config pair.

The Guardrailed Pipeline is enforced by the section *Pipeline Protocol: The Guardrail Loop* of the Gemini CLI project configuration `gemini.md`, described in Sect. 4.5.

<sup>2</sup>Note that `clang` can be substituted for any other compiler by reconfiguring the respective MCP tool.

<sup>3</sup>Note that CBMC can be substituted for any other software verifier part of SV-COMP by reconfiguring the respective MCP tool.

## 4.4 MCP Tool Suite

LAHG provides a custom tool suite through an MCP server that enforces the guardrail pipeline and offers features to reduce the tools' token usage.

This tool suite consists of MCP tools that can be invoked either internally by the LLM or by the user via custom commands. Their output is fed back to the model as an internal response, which gives it instantaneous feedback on the tool's success or failure.

### 4.4.1 Pipeline Tools

All Stages of the pipeline are supported by a matching MCP tool. To ensure that the compiler tool can compile a harness for any repository, the LAHG employs its tool *DependencyFinder* to dynamically resolve all includes and pass them to the compiler. In total, 4 MCP tools handle the execution of the Guardrailed Pipeline:

1. *DependencyFinder*: A preliminary step to the compiler, the *DependencyFinder* resolves all include paths of a harness to ensure seamless compilation.<sup>4</sup>
2. *Compiler*: Invokes a compiler execution, albeit limited to preprocessing to ensure valid C code while staying lightweight.
3. *HarnessForge*: Invokes the generation of all verification tasks specified in the generated configuration file and the harness.
4. *Verifier*: Invokes a verification run on a single verification task.

MCP's integration with most modern LLM CLI clients allows the model to invoke a tool either through an explicit custom command or by identifying a tool-use intent during inference of the prompt. The prompts `"/harnessforge [path_to_harness] [path_to_config]"` and `"Execute HarnessForge on the latest harness and config"` both lead to the execution of *HarnessForge*. This bimodal invocation simplifies the use of the LAHG and reduces the LLM's degrees of freedom, thereby discouraging it from hallucinating.

### 4.4.2 Token management

In certain scenarios, Gemini's built-in file management tools operate too broadly and burn tokens. If Gemini needs to search a log file for information that can not be found by pattern matching, it will just load the entire file into its context, risking an immediate overage of either the input token window or the context window. This results in an ineffective use of tokens and thus unnecessary cost.

LAHG implements tools optimized to work best when combined with lightweight file-searching tools such as `grep` and `find`<sup>5</sup>.

1. `read_file_slice`: Retrieves a specific line range from a file (default 200 lines) with line numbers and a token-safety truncation notice.
2. `ls_recursive_summary`: Returns a filtered tree view of allowed file types to map the project without wasting tokens.

---

<sup>4</sup>This tool has been omitted in the visualization, since it does not act as a gate. However, it is a crucial step in the harness generation and is tightly bound to the compiler.

<sup>5</sup>Note that *HarnessForge* is only executable in a Unix environment. Thus, we only mention Unix tools here.

1. Role and Persona
2. Terminology
3. Execution State Machine
4. Global invariants
5. Forbidden Behaviors and Hard Boundaries
6. Atomic Refinement Protocol
7. Pipeline Protocol: The Guardrail Loop
8. Token & Context Management
9. (MCP) Tools
10. Human-in-the-Loop Protocol
11. Artifact Schema Specifications

Figure 4.6: Document structure of LAHG's `gemini.md` project-configuration.

### 4.4.3 File Type Conversion

Gemini CLI does not come with a built-in file type converter, which makes working with a large specification document in pdf expensive.

To avoid this, LAHG implements an MCP tool that invokes Docling to either preprocess documents in advance or during interaction with Gemini. This can be paired with a configurable file splitter, which allows the extraction of pages, sections, or chapters, to limit the conversion to meaningful parts of a document.

## 4.5 Gemini Configuration

Gemini CLI's behavior is defined by `gemini.md`, a Markdown file that allows the user to specify the model's specific role in the codebase's context. [Figure 4.6](#) shows the structure of LAHG's `gemini.md`.

**Role and Persona.** Gemini is configured as an **Automated Formal Verification Orchestrator** with an emphasis on tight, informational, non-chatty conversations for concise, information-dense interactions.

**Terminology.** This section specifies an **Artifact Terminology**, an **Action Terminology**, and a **State Terminology**. Each entry in these sections provides a brief description to give the model a broad understanding of how to categorize the term. The State Terminology introduces all terms used in the following section **Execution State Machine**.

**Execution State Machine & Global invariants.** This is the first building block of the **Guardrailed Pipeline**. This section defines each Stage of the Guardrailed Pipeline (Compilation, HarnessForge, and Verification) as a finite-state machine with explicitly declared transitions. Gemini documents these sections in `TODO.md`, which is the center of Gemini CLI's todo list feature. This built-in feature allows the model to identify at any time the current Stage and the required next steps to complete the pipeline.

Furthermore, this section also instructs the model on the required behavior in the case of a failure <sup>6</sup>. To ensure recoverability of a session at any time, the model must always keep the `TODO.md` updated and to handle an overage of either the LLM's input window or its context window, it must execute the `clear`-command to reset the internal state and reinitialize it from the `TODO.md`.

The **Global Invariants** contain imperative instructions for specific edge-cases in the State Machine Representation that must hold at all times.

**Forbidden Behaviors and Hard Boundaries.** To prevent the LLM from guessing header paths when generating a harness or from omitting code segments "for brevity", this section draws hard boundaries around file access and tool execution.

**Atomic Refinement Protocol.** This protocol instructs the model to split the sessions into logical bins. These bins are incremental working folders in the repository that allow storing different states of the generation process. We refer to these bins as *S-buckets*.

If an error occurs repeatedly and the model appears to be stuck, the protocol instructs it to switch to a new bin, copy all files, perform an internal reset to clear the context window, and revisit the error with a fresh eye. The internal reset is performed with the `clear`-command and ensures that the model operates on an atomic problem. This keeps the context window small and the response times low.

**Pipeline Protocol: The Guardrailed Loop.** This section details the sequence of steps in the **Guardrailed Pipeline**:

"You manage a linear pipeline. Do not skip steps. Each step is an atomic unit.

1. **Initialization:** First, prompt the user to choose whether they want to start a new session or provide the path to an existing one.
  - **Resume:** Run `ls_recursive_summary` on `hgt/.` to check for an existing session and resume if found.
  - **New:** Prompt: "Start a new session for [repo][method\_name]?"
2. **Generation:** Generate the **Harness** `[method_name]_harness.c` and the corresponding **Config** `[method_name]_config.yml` in the `s<N>` directory.
3. **Dependency Gate:** Run `DependencyFinder`. You are forbidden from running `clang` until a **Dependency File** exists.
4. **Compilation:** Run **Compiler** using the **Dependency File** from Step 3.
5. **Task Creation:** Run `HarnessForge` on the latest **Config** to scaffold the **Verification Tasks**.
6. **Verification:** Run **Verifier** using the **Verification Tasks** from Step 5."

**Token & Context Management and (MCP) Tools.** These two sections present the model with a list of all available MCP tools, their intended use, and the specific scenarios in which a tool invocation might be reasonable.

---

<sup>6</sup>Note that failure here refers to any kind of unintended behavior in the pipeline, such as failed tool calls. A disruption of the service Gemini CLI is not handled here.

```

Action Required

? read_file_slice (tooling-engine MCP Server) {"start_line":1,"file_path": ...

MCP Server: tooling-engine
Tool: read_file_slice

MCP Tool Details:
(press Ctrl+O to expand MCP tool details)
Allow execution of MCP tool "read_file_slice" from server "tooling-engine"?

• 1. Allow once
  2. Allow tool for this session
  3. Allow all server tools for this session
  4. No, suggest changes (esc)

```

Figure 4.7: Human-In-The-Loop-In-Action

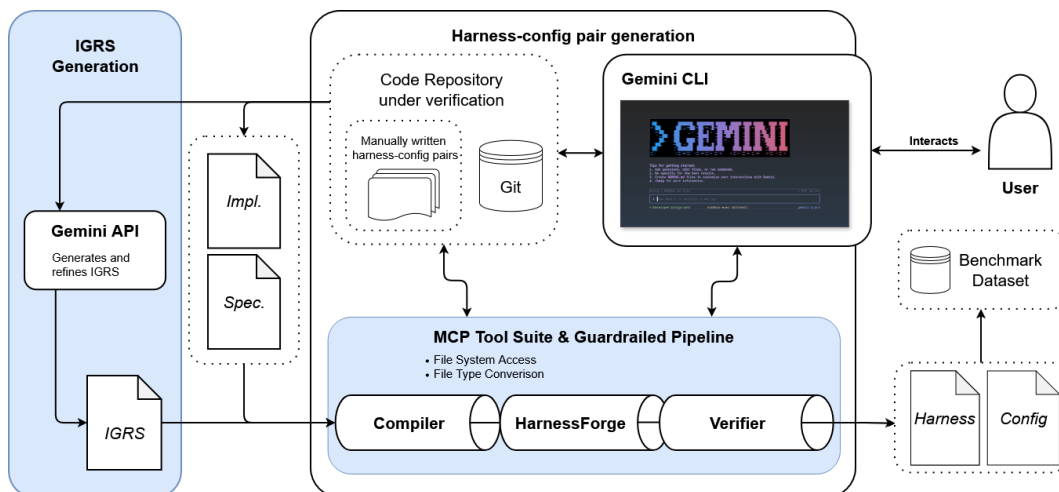


Figure 4.8: LAHG

### Human-in-the-Loop Protocol.

The HITL protocol states that the model is to propose an **Action Plan** before calling any MCP tool. This protocol ensures that the user is always in control of what the model writes to disk. Furthermore, this allows the user to interject and alter the model's plan, ensuring that every action follows an informed decision. Figure 4.7 displays an example of this interaction. The MCP server stores the user preferences for a given session, allowing to bypass of this HITL check on future occurrences.

## 4.6 LAHG

Figure 4.8 depicts how LAHG uses the MCP tool suite and the Guardrailed Pipeline to achieve its objective. The blue colored rounded rectangles **IGRS Generation** and **MCP Tool Suite & Guardrailed Pipeline** mark the main contributions by LAHG.

**Deployment.** LAHG is available on GitLab [5]. We refer to the code repository under verification as *target*. To verify a target, one needs to download LAHG and either add the target as a submodule to LAHG or copy the code into a dedicated folder in LAHG’s repository. HarnessForge is also a submodule of LAHG. An installation guide details the remaining setup. LAHG comes with an IGRS in the form of a text file that has been generated for Intel TDX.

**New target.** If the included IGRS does not seem to be sufficient for a new target, one can generate a new IGRS if that target contains functions for which harnesses have been created. If this is not the case, the IGRS cannot be generated programmatically but rather during interaction with the LLM. LAHG also provides the initial rule set as a fallback option, should the provided IGRS fail to deliver satisfactory results. Be aware that in this case, the performance might be lower than expected.

The initial rule set can be improved during the interaction by instructing the model to refine it using information from the MCP tool’s outputs. LAHG comes with a playbook that details common issues and how to prevent or alleviate those. Note that parts of this playbook may not apply to a new target, as some issues may be specific to Intel TDX.

Note that IGRS generation is separate from harness-config pair generation, since the latter requires an IGRS to function.

Both the verifier and the compiler can be exchanged by configuring the respective MCP tool. LAHG works with any LLM that offers a client-side Coding Agent that is compatible with MCP, such as Gemini CLI. Suppose Gemini CLI is replaced by another tool, the content of the `gemini.md` must also be integrated into whatever method this new tool uses.

**Interaction.** As shown in Fig. 4.8, the user interacts with the Gemini CLI, which has access to the repository under verification and maintains a connection to the MCP server. It can invoke MCP tools and retrieve the tools’ output. The Guardrailed Pipeline requires an IGRS, a function’s implementation, and its specification. To keep the generation process modular, the user must specify which files to use, including the IGRS. If the user wants to switch IGRS mid-run, he can do so.

Each Stage of the pipeline is invoked by the user’s interaction with the Gemini CLI, so the user has full control over the LLM’s inputs. Since the Gemini CLI has access to the target’s files, it can manipulate their contents to gather insight or change the code. Note that the LLM will not directly change the files in the repository under verification. Instead, it will work on copies in a separate directory to keep the baseline code unaltered.

# 5 Evaluation

## 5.1 Experimental Setup

Generating a harness-config pair with LAHG requires an IGRS. As explained in [Sect. 4.1](#) and [Sect. 4.2](#), an IGRS is generated using Iterative Refinement with a programmatic approach in which we pass sequences of harnessed functions through the Gemini API to the LLM. We instruct the model to generate a harness-config pair with a given specification and an implementation, then compare its generated artifacts against the human-written ones. From this difference, the LLM abstracts rules, which it uses to enhance the Initial Rule Set. However, one must consider two aspects:

Firstly, every LLM has a maximum input window size. A sufficiently large file might exceed this window, which can be avoided by splitting the document into those sections that are most relevant to the model.

Secondly, the ABI specification for the Intel TDX module firmware is in `pdf` format. Thus, converting the specification to a machine-friendly format might improve its understanding of the content. [Figure 5.1](#) shows all external tools used by LAHG, along with their versions at the time of the experiments.

### 5.1.1 Specification Parameter Options

**Input-split.** If providing the full specification is not feasible due to the limited input window size, one must identify the parts of the specification that contain the most valuable information for the LLM to generate a harness-config pair. The simplest split would be to extract the singular interface function specification and all relevant underlying data structures.

This leads to two options for the input split:

- `full`: The entire specification document
- `data-split`: The data structures and the single function specification

In this section, we refer to this parameter as `input-split`.

**Input-type.** We identify the file types that are natively supported by the Gemini API as the most natural choice for the specification input type parameter.

`pdf` is a layout-centered format, designed for human readability. While humans naturally assign a semantic connection to a group of objects, machines fail to see a connection. While the LLM can handle `pdf` files, it needs to apply image recognition to each page to extract semantics. This process is time-consuming and costly.

```
OS: WSL2 Ubuntu on Win11

Tools:
- Clang for Windows x64: LLVM-18.1.8
  Target: x86\_64-pc-windows-msvc
  Thread model: posix
- HarnessForge 1.3.0-dev
- CBMC version 6.7.1 (cbmc-6.7.1) 64-bit x86\_64 linux

Gemini:
- Gemini CLI 0.33.2 with gemini-3-flash preview
- Gemini API Model: gemini-3-flash-preview
- Embeddings: text-embedding-001

Code Base:
- Intel TDX 2.0.08 (August 2025)
```

Figure 5.1: Experimental environment

The markup language `md` is a predominantly textual format common in developer documentation. Its reliance on natural language—interspersed only with basic structural elements like tables—minimizes formatting overhead, making it highly compatible with LLM processing. This is beneficial for LLM understandability, which is why most LLMs return answers to prompts in Markdown.

`yml` is mostly used as a data container. Since LLMs are trained on natural language and code, the model has an intrinsic understanding of YAML code and requires no additional processing.

We convert `pdf` to `yml` and `md` using Docling [25]. In this section, we refer to this parameter as `input-type`.

## 5.2 Questions and Answers

The parameters detailed in Sect. 5.1.1 lead to the following research questions:

**RQ. 1:** *Does providing the entire specification document exceed the LLM’s input window?*

**RQ. 2:** *Does converting the specification document from a layouted format to a textual or structural format improve the model’s understanding of that document?*

The sequential nature of the IGRS generation poses another question:

**RQ. 3:** *Does altering the sequence of few-shot examples influence the model’s output?*

The IGRS generation with the Iterative Refinement Loop uses the Gemini API, while the Guardrailed Pipeline uses the Gemini CLI. This leads to another question:

**RQ. 4:** *Is the ideal input type for generating structured content with the Gemini API also ideal for generating structured content with the Gemini CLI?*

Since IGRS generation and harness-config generation are two separate processes, we split the experiments into two categories: *IGRS Generation* and *Harness-config Generation*. As harness-config generation requires an IGRS, we first conduct experiments on IGRS generation in Sect. 5.3 before evaluating in Sect. 5.4 whether the found optimal parameter configuration also holds for harness-config generation.

## 5.3 IGRS Generation

The LLM’s input window size imposes a more stringent constraint on both the IGRS generation and the harness-config generation than the model’s understanding of the specification.

Thus, we begin in Sect. 5.3.1 by analyzing how the Cosine Similarity of subsequent IGRS progresses for two IGRS generations. We provide the Gemini API in both runs with the specification document in `pdf` format. The first run uses the `full` document, the second gives the API the `data-spec split`.

We perform both runs on the same sequence of all 22 interface functions for which harness-config pairs exist. We calculate the Cosine Similarity across all iterations and identify the global maximum. We reason that the Cosine Similarity must eventually converge on a score of 1, given enough examples.

The score closest to this theoretical maximum must be the best representative of the given material, and therefore, any successive lower score must have introduced noise or irrelevancy. The respective IGRSs are then used in two harness-config generations to see whether the LLM can handle the `full` specification in LAHG.

In Sect. 5.3.2 we answer **RQ. 2** and **RQ. 3** by creating three sets of function sequences. We then compare for each input type `pdf`, `md`, and `yaml`, which sequence reaches the highest stage in the fewest attempts by generating an IGRS for each combination and applying it to the harness-config generation.

In this chapter, by *functions*, we refer to only those for which a harness-config pair has been manually created.

### 5.3.1 Experiment 1: IGRS input-split

We begin by creating a sequence of interface functions that has the highest possible dissimilarity<sup>1</sup>. This metric ensures that the model will be presented with a sequence of functions that will bring the most novelty in each iteration.

We employ Farthest-First Traversal (FFT) to generate the function sequence. As defined in Eq. (5.1), the Farthest-First Traversal (FFT) acts as a greedy selection strategy to determine the optimal sequence of interface functions.

By selecting the  $n$ -th task,  $\text{task}_n$ , from the set  $T$  of all tasks that maximizes the minimum distance to all previously selected tasks  $\{\text{task}_i\}_{i=0}^{n-1}$ , the algorithm ensures that each iteration introduces the highest possible degree of semantic novelty.

This approach forces the model to refine its Rule Set against the most diverse implementation patterns early in the process, accelerating convergence toward the global maximum of Cosine Similarity. Consequently, this maximizes the informational

<sup>1</sup>We use the complement of the Cosine Similarity as the dissimilarity score.

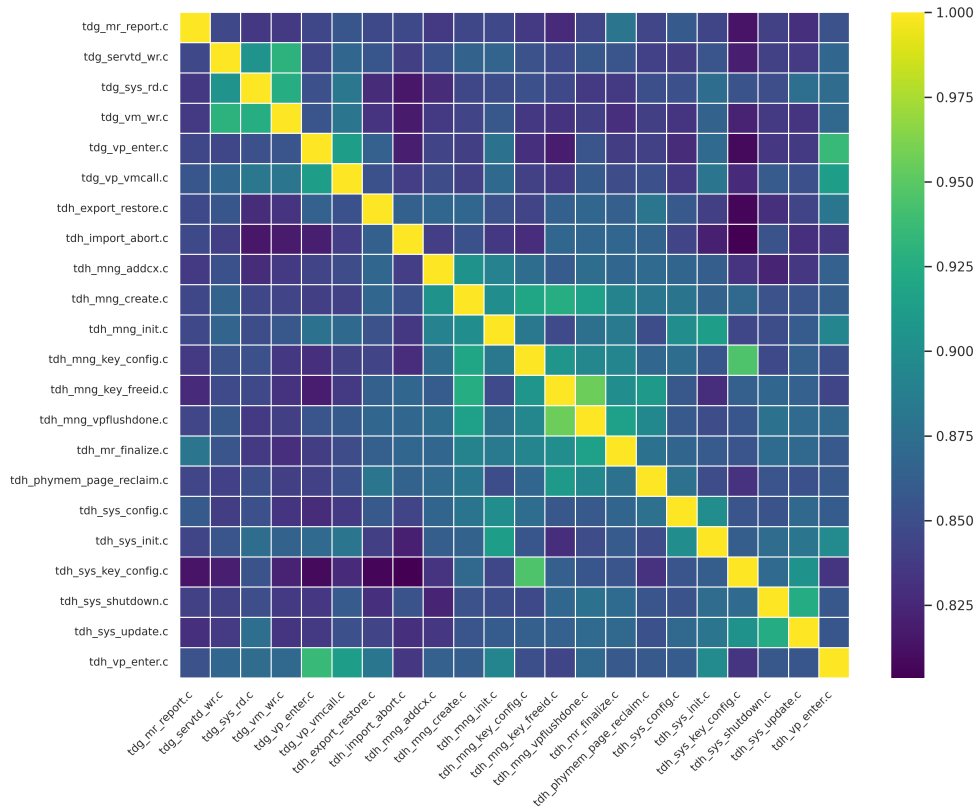


Figure 5.2: Cross-comparison of all files' Embedding Cosine Similarity

density of the sequence while minimizing redundant token consumption and the risk of computational intractability.

$$\text{task}_n = \underset{t \in T \setminus \{\text{task}_i\}_{i=0}^{n-1}}{\text{argmax}} \left( \min_{0 \leq i < n} \text{dist}(t, \text{task}_i) \right), \quad \text{für } T = \{\text{all Tasks}\} \quad (5.1)$$

To determine the starting point for the FFT algorithm, we generate a cross-comparison matrix with the Embedding Cosine Similarity for all unaltered C implementations of all functions. The resulting matrix is visualized in Fig. 5.2, which displays the Embedding Cosine Similarity between all function implementations in a heatmap. The matrix reveals that files within a submodule of Intel TDX, such as `tdh_mng` or `tdh_sys`, are more similar to one another than to files from other modules. This is due to the fact that functions within a single module typically load and access similar data structures. The guest-side function `tdg_sys_rd.c` is revealed to be the most dissimilar function and thus serves as the entry point for FFT. The FFT yields the sequence of functions provided in Table 5.1.

We generate two sequences of IGRS with the input splits `full` and `data-spec`. Each run uses the specification in pdf format.

Figure 5.3 and Fig. 5.4 show the Sequential Cosine Similarity Scores across the entire sequence of functions in a red dotted line, as well as the average aggregate score in a

- |                          |                               |
|--------------------------|-------------------------------|
| 1. tdg_sys_rd.c          | 12. tdh_sys_config.c          |
| 2. tdh_sys_update.c      | 13. tdh_sys_init.c            |
| 3. tdh_sys_shutdown.c    | 14. tdg_vp_vmcall.c           |
| 4. tdh_mng_vpflushdone.c | 15. tdg_vp_enter.c            |
| 5. tdh_mng_key_freeid.c  | 16. tdh_vp_enter.c            |
| 6. tdh_mng_create.c      | 17. tdh_export_restore.c      |
| 7. tdh_mng_key_config.c  | 18. tdh_phymem_page_reclaim.c |
| 8. tdh_sys_key_config.c  | 19. tdh_import_abort.c        |
| 9. tdg_mr_finalize.c     | 20. tdg_mr_report.c           |
| 10. tdh_mng_addcx.c      | 21. tdg_servtd_wr.c           |
| 11. tdh_mng_init.c       | 22. tdg_vm_wr.c               |

Table 5.1: Sequence of functions yielded by FFT analysis.

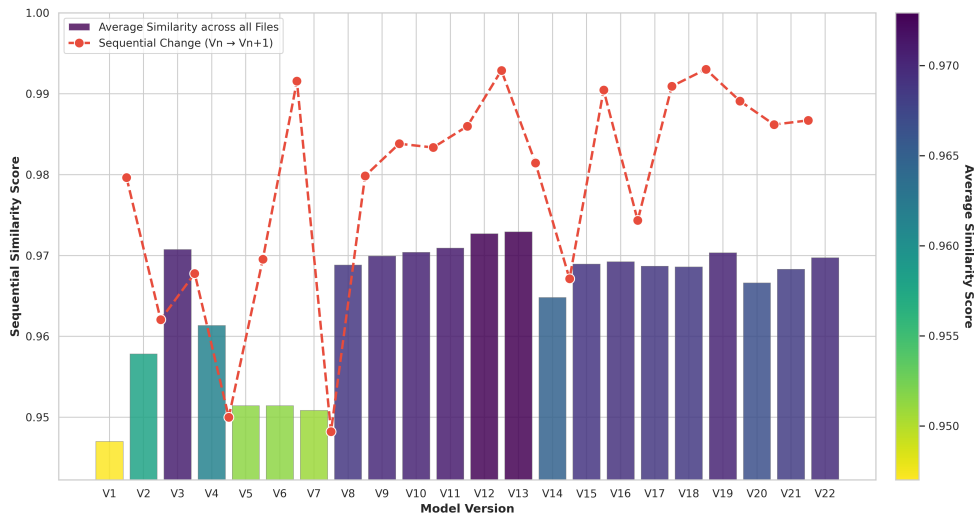


Figure 5.3: Sequential IGRS Cosine Similarity and global similarity for pdf data-spec

bar plot. The latter, the *global similarity score*, encodes how similar a given IGRS is to all other IGRS in this sequence. Each bar represents an IGRS version. V1 refers to the IGRS after the first iteration, and V22 represents the IGRS after the 22nd iteration. We calculate their scores after completing all iterations. While each bar represents an absolute score, the Sequential Cosine Similarity represents the Cosine Similarity from one IGRS to the next. Each score is always relative to its predecessor.

The plots reveal that if multiple successive IGRS have very similar global similarity scores ( $\pm 0.05$ ), the Sequential Cosine Similarity Score continually increases until the end of the subsequence. Both Sequential Cosine Similarity Graphs have a similar graph structure.

This leads to the conclusion that the sequence of functions influences the Sequential Cosine Similarity more than the specification's input split. Furthermore, both plots show that when the LLM is provided with a full specification document, the

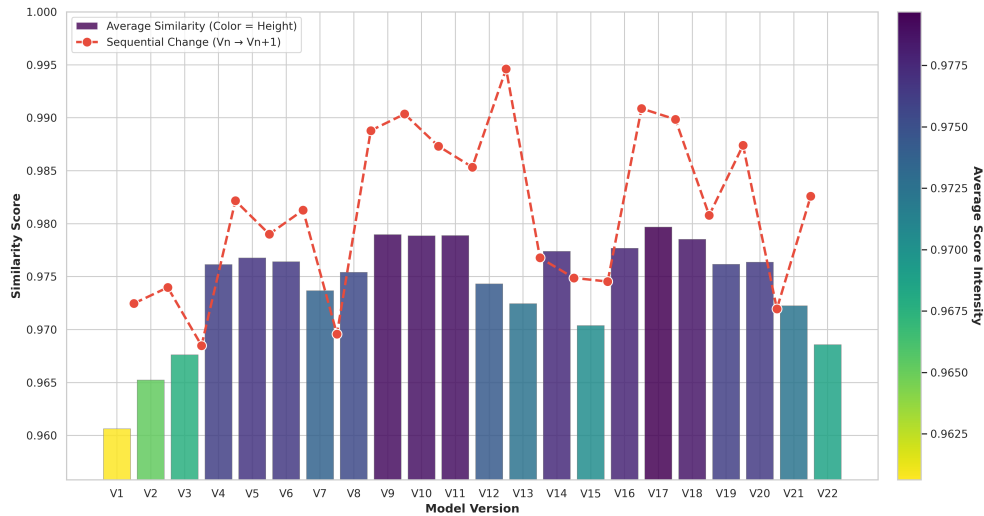


Figure 5.4: Sequential IGRS Cosine Similarity and global similarity for pdf full

global similarity score is more uniform and exhibits lower variance than when a specification is for the input split `data-spec`. Since `data-spec` strips from the document all method specifications not used in the immediate IGRS example, its informational overlap is lower than that of the full document. Therefore, `full` dilutes the model’s informational content and attention by providing too much context.

The highest Embedding Cosine Similarity occurred in `data-spec` in the 18th iteration between the functions `tdh_phymem_page_reclaim` and `tdh_import_abort` with a score of 99,30% and in `full` in the 13th iteration between `tdh_sys_config` and `tdh_sys_init` with a score of 99,46%. The Gemini API reported no issue in handling and parsing the entire specification document.

However, while attempting to generate the harness-config pair for `full`, the Gemini CLI reported that the specification exceeded its input window size, i.e., rendering the generation impossible. Hence, we conclude that `data-spec` is the better choice for the input split.

This answers **RQ. 1**: Yes, providing the entire specification document exceeds the LLM’s input window of 1,048,576 tokens [19].

It also shows that the Gemini API and the Gemini CLI have different limitations while using the same model `gemini-3-flash-preview`.

### 5.3.2 Experiment 2: IGRS input-type

We have concluded `data-spec` to be the input split of choice. To determine the best-performing input type for IGRS generation, we generate IGRS for each file type supported by the Gemini API.

First, we split the harnessed interface functions into training and validation sets. We chose the host-side functions (`tdh`) as the training set and the guest-side func-

tions (`tdg`) as the validation set, since they split the set of all harness functions approximately 70:30 and share the Intel-TDX submodules `vp` and `sys`.

To assess the potential impact of the sequence of functions on the model’s understanding, we create three sets of functions from the training set. We choose at random, sequentially, and manually. The `random` set is an arbitrary choice of functions.

The `sequential` set orders the functions alphabetically, yielding a somewhat random sequence of modules, which group the functions and thus retain some order.

The `manual` set is a deliberate selection designed to maximize semantic differences across the sequence.

**Creating the manual subset.** We create this `manual` subset similarly to the initial sequence in Sect. 5.3.1. The code basis for each function under verification is its C code, augmented with C-level representations of all dependencies, such as library calls, as produced by HarnessForge during its verification task generation. Note that the C code of the function under verification alone is not sufficient as it lacks important semantic information from the referenced library functions.

However, this preprocessing may introduce large amounts of boilerplate code by adding `structs` and `typedefs` used by every interface function, so that a cross-comparison of the files’ Embedding Cosine Similarity yields a constant value. This invalidates the comparison of the Cosine Similarities. We therefore strip the preprocessed files of these definitions before creating a cross-comparison matrix of the files’ Embedding Cosine Similarity.

To compensate for the semantic value lost in this stripping, we add the semantic value of the interface function’s specification. By converting the `pdf` specifications to Markdown, we retain the natural language and the tables while removing all layout information.

Aggregating the cross-comparison matrices for both the preprocessed implementations and the specifications gives us a combined score for each interface function. By calculating the average row sum for each interface function, we determine `tdh_vp_enter` to be the most dissimilar and thus the entry point for FFT, which produces the final `manual` sequence displayed in Table 5.2. It shows the three resulting function sequences side by side.

Combining each function set with each specification input type creates the configurations displayed in Table 5.3. Each sequence (`random`, `manual`, and `sequential`) is paired with each input type (`pdf`, `md`, and `yml`).

**Experiment parameters.** To evaluate the performance of the resulting combinations, we execute a harness-config generation for one function from the training set and one from the validation set. We select `tdh_sys_init` from the training set, as we used it only once during the IGRS generation, and `tdg_vp_vmcall` from the validation set, since it had the highest number of unique imports among all functions. The harness-config generation runs on `tdg_vp_vmcall` are marked with the suffix `_val`.

*S-buckets* refers to the logical bins of LAHG’s Atomic Refinement Protocol mentioned in Sect. 4.5, which we use to measure the progress of a run. The protocol states that

Table 5.2: Sequential example functions sets.

#	Randomly Selected	Manually Selected	Sequential
1	tdh_sys_update	tdh_mr_finalize	tdh_import_abort
2	tdh_mng_key_config	tdh_vp_enter	tdh_mng_addcx
3	tdh_mng_addcx	tdh_sys_key_config	tdh_mng_create
4	tdh_mng_vpflushdone	tdh_sys_shutdown	tdh_mng_init
5	tdh_sys_config	tdh_mng_key_freeid	tdh_mng_key_config
6	tdh_mng_create	tdh_import_abort	tdh_mng_key_freeid
7	tdh_import_abort	tdh_mng_addcx	tdh_mng_vpflushdone
8	tdh_mng_init	tdh_mng_init	tdh_mr_finalize
9	tdh_sys_key_config	tdh_sys_config	tdh_phymem_page_reclaim
10	tdh_phymem_page_reclaim	tdh_phymem_page_reclaim	tdh_sys_config
11	tdh_mr_finalize	tdh_sys_update	tdh_sys_init
12	tdh_sys_init	tdh_mng_vpflushdone	tdh_sys_key_config
13	tdh_vp_enter	tdh_sys_init	tdh_sys_shutdown
14	tdh_mng_key_freeid	tdh_mng_create	tdh_sys_update
15	tdh_sys_shutdown	tdh_mng_key_config	tdh_vp_enter

Table 5.3: Combinations of function sequence sets and specification input type

pdf	md	yml
pdf_random	md_random	yml_random
pdf_manual	md_manual	yml_manual
pdf_sequential	md_sequential	yml_sequential

the model is to switch to the next S-bucket either when the same error occurs three times in a row or when a newly reported error is unrelated to the previous one. This mandatory rule ensures that during the experiments, we can track how well a model handles errors. We track the highest stage passed in an S-bucket. We opt for a maximum of 5 S-buckets per run due to the cost constraints on this project. *Stages* refer to the stages of the LAHG’s Guardrailed Pipeline as described in Sect. 4.3.

Note that the run configurations specify the specific configuration of input type and input split that was used to generate the respective IGRS. We provide the LLM with the specification in the `pdf data-spec` format to ensure comparability with Experiment 1 in Sect. 5.3.1.

We measure the number of input tokens used and the highest Stage reached. Stage 3 constitutes as passed if executing the respective verification task does not immediately raise an error. We acknowledge that this is an ambiguous definition, but this criterion suffices for our purpose. A failing verification task can have multiple reasons. It can be either a faulty harness, a harness that perfectly encodes the pre- and post-conditions from the specification and identifies a mismatch between the specification and the implementation, or a verification task that is too complex to be computationally tractable. We therefore consider only a CBMC call as *failed* if it returns an error immediately.

**Highest Stage Reached.** Figure 5.5 displays each configuration’s progression through the Stages in a colored heatmap. It shows that only the IGRS `yml_manual` and `yml_sequential` generate harness-config pairs that result in one verification task each, which can be executed by CBMC. However, in the case of `yml_manual`, this successful CBMC execution is preceded by 4 S-buckets, during which the model is unable to generate a harness usable by HarnessForge. In the case of `yml_`

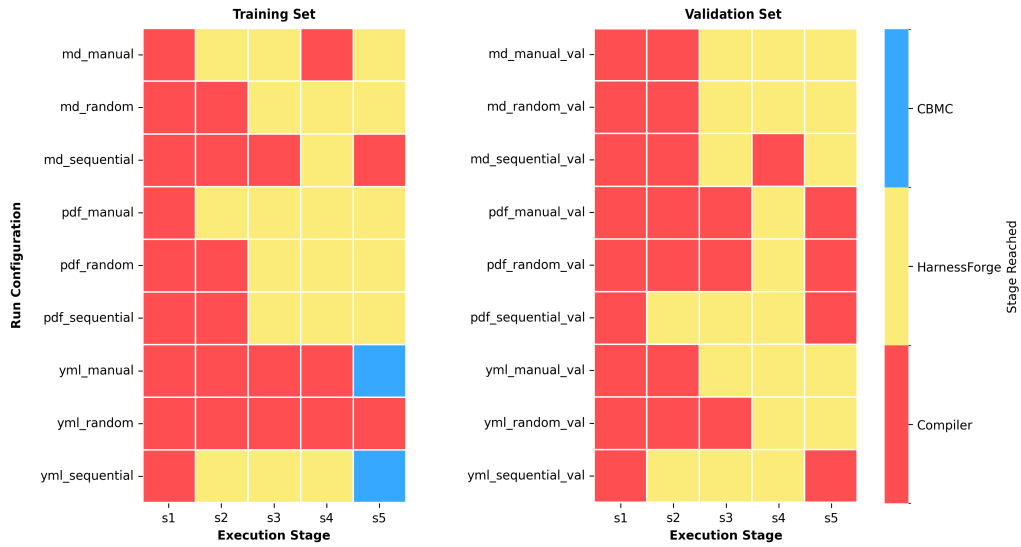


Figure 5.5: Stages reached for all harness-config generation runs

random, only Stage 1 is passed in all five S-buckets. It also shows that most issues with the `clang` compiler are resolved by S-buckets 2, with `md_sequential` and `md_manual` being the exceptions.

Furthermore, the heatmap shows that all IGRS perform equally well and, in most cases, worse on the validation set than on the training set. This is due to the fact that the validation set contains only guest-side interface functions, whereas the training set contains only host-side interface functions. While they are similar, they use different logic and thus might introduce implementation details not covered by the IGRS.

Only `md` performs equally or better on the validation set. `pdf` performs worse with all IGRS on the validation set.

This indicates that while `yml` yields the best results, `md` appears more stable and thus the preferred input type. In addition, `manual` reaches on average the highest average Stage for each S-bucket (2) while `random` and `sequential` both have an average Stage for each S-bucket of 1.6. This answers **RQ. 3**: Yes, the sequence of examples does influence the model’s output. It also shows that maximizing semantic spread across the sequence yields better results than random selection.

**Stages reached vs. token efficiency.** Figure 5.6 shows, for each configuration, the highest Stage reached relative to the number of input tokens required. It reveals that the well-performing outlier `yml_manual` required roughly 10 million input tokens in the training set, whereas its companion `yml_sequential` used roughly 30 million tokens.

The most expensive harness-config generation was `md_random` on the training set with more than 46 million tokens, and the most cost-efficient was `md_sequential` on the validation set. The visualization shows that, regardless of the success of the harness-config generation run, `md` and `yml` seem to be more cost-effective than `pdf`.

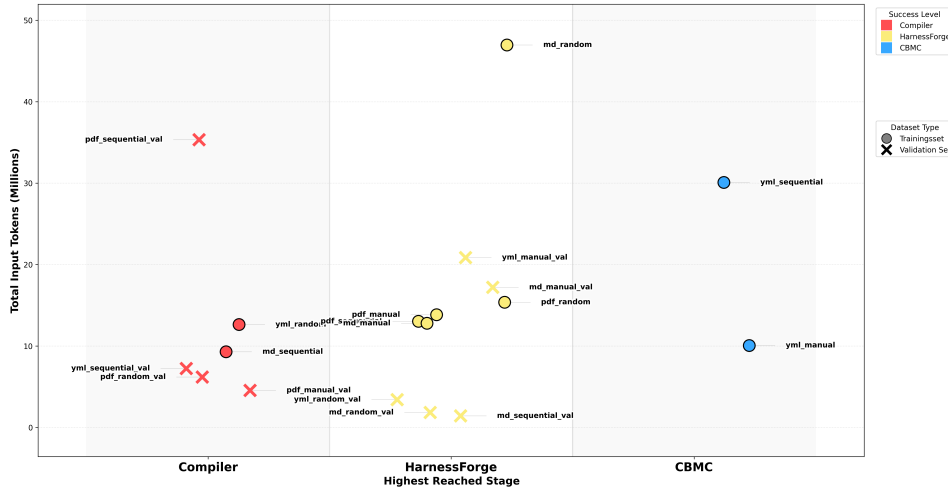


Figure 5.6: Config-harness generation runs: Number of used input tokens vs. highest reached Stage

With `md_sequential` as the exception, every `md` IGRS passed Stage 2 (HarnessForge). The IGRS `md_sequential_val` required the lowest cost of roughly 1.4 million input tokens.

We therefore conclude that `md` is the best choice for the input type for its output stability and cost-efficiency. While this does not clearly answer **RQ. 2**, it gives an indication that a textual format like `md` improves the models understanding.

**Token efficiency and financial impact.** Table 5.4 displays the financial impact of input token usage. It shows for each IGRS, how many calls to the LLM were recorded and how many input tokens were used, along with the resulting cost. We do not consider the cost of output tokens relevant, since these are roughly 1500 tokens per harness and config, and this is both the desired and inevitable consequence of harness-config generation. Thus, we omit this data from the tables. Furthermore, we refrain from displaying the average number of input tokens per prompt per IGRS, since it depends on a prompt’s context. If the prompt requires the Gemini CLI to internalize all files from a given folder, this results in higher token consumption than a simple question that does not require any research.

**Observations during the harness-config generations.** We encounter multiple occasions where the LLM-generated YAML code must be corrected several times, even though this structure is not only defined in the **Output Format & Internal Digest** mentioned in Fig. 4.3 but also provided in the few-shot examples. It shows that either `gemini.md` was overloaded or the LLM’s nondeterministic nature clashes with the deterministic nature of the YAML schema.

Since LAHG is designed to be project-agnostic, it does not provide specific information about the codebase of the function under verification. This results in incorrect formulation of include paths in the generated harness. While the Gemini CLI has access to any file in the Intel TDX firmware module GitHub repository, it fails to automatically detect the correct include paths. This might be due to the narrow focus imposed by the strict rules in the `gemini.md`.

Table 5.4: Financial overview of `gemini-3-flash-preview` token usage categorized by format and strategy

Type	Strategy	Count	Tokens	Cost (USD)
md	manual	83	12,783,063	6.39
	random	178	46,964,440	23.48
	sequential	28	3,537,705	1.76
	manual_val	112	17,205,703	8.60
	random_val	18	1,837,321	0.91
	sequential_val	6	1,412,604	0.70
pdf	manual	56	13,832,882	6.91
	random	66	12,186,463	6.09
	sequential	61	10,364,615	5.18
	manual_val	29	4,552,506	2.27
	random_val	24	6,193,983	3.09
	sequential_val	223	35,333,145	17.66
yaml	manual	73	10,071,905	5.03
	random	63	12,633,195	6.31
	sequential	75	30,081,083	15.04
	manual_val	188	20,860,200	10.43
	random_val	13	3,419,379	1.70
	sequential_val	60	7,243,136	3.62
<b>TOTAL</b>		<b>1,357</b>	<b>246,513,328</b>	<b>\$ 123.25</b>

## 5.4 Harness-config pair generation

Experiment 2 reveals that the input types `md` and `yaml` are more cost-efficient than `pdf`. While `md` is the most consistent among all three, `yaml`, overall, reaches the highest Stage.

We select the best performing IGRS from Experiment 2. For each input type, we conduct a harness-config generation providing the LLM with the specification in the respective input type in the `data-split` format. In this scenario, we deem that IGRS best, which is among the ones reaching the highest Stage in the Guardrailed Pipeline and which uses the fewest input tokens. We therefore select the IGRS `yaml_manual`. As in the previous experiment, we conduct the harness-config generation up to S-bucket 5. We measure the number of input tokens and use it as the key metric to compare two configurations that have the same sequence of passed Stages across all 5 S-buckets.

**Highest Stage reached.** Figure 5.7 shows each configuration’s progression in the harness-config generation. It reveals that none of the configurations produce a harness that results in a verification task that CBMC can execute. However, it shows that all configurations generate harnesses compatible with HarnessForge.

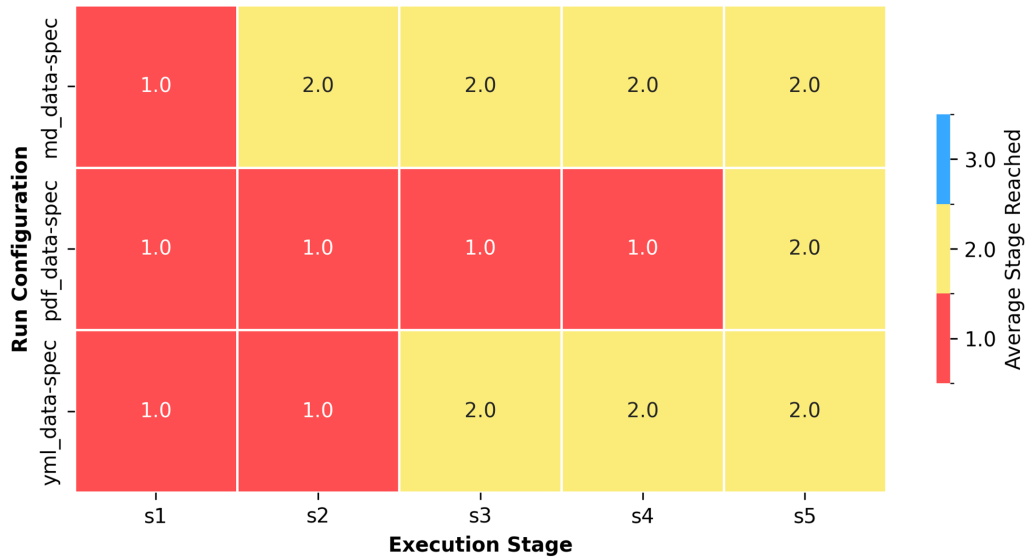


Figure 5.7: Stages reached for input split `data-spec` and all input types.

They strongly differ in their speed at which they pass Stage 2. `md_data-spec` performs best by reaching Stage 2 in the second S-bucket, directly followed by `yml_data-spec`, which reaches Stage 2 in the third S-bucket. `pdf_data-spec` reaches that Stage last in S-bucket 4.

This indicates that the LLM exhibits a better understanding of the specification document if it is preprocessed, i.e., it does not require image recognition to extract its content.

**Stages reached vs. token efficiency.** Figure 5.8 displays each configuration’s highest reached Stage in correlation to the input tokens used. It shows that while `pdf_data-spec` requires the fewest input tokens, it reaches the same Stage as the other two configurations, albeit more slowly.

While `md_data-spec` requires roughly 32.5 million tokens, more than three times as many tokens as `pdf_data-spec` with around 10 million tokens, it is still considerably more cost-effective than `yml_data-spec` with more than 45.5 million tokens.

Combined with the insights from Fig. 5.7, we conclude that providing the specification document in `md`, split into the single function specification and the underlying data structures, works best for IGRS generation with the Gemini API, as well as the harness-config generation with the Gemini CLI.

#### Token efficiency and financial impact.

Table 5.5 displays the number of prompts for each harness-config generation, as well as the required tokens and the resulting cost. It shows that the high token consumption of `yml_data-spec` directly correlates with higher cost.



Figure 5.8: Harness-config generation runs: Input tokens vs. highest Stage reached

Table 5.5: Financial overview of token usage for merged data specifications

Type	Count	Tokens	Cost (USD)
md_data-spec	159	31,451,152	15.72
pdf_data-spec	125	10,241,612	5.12
yml_data-spec	133	46,291,813	23.14
<b>TOTAL</b>	<b>245</b>	<b>87,984,577</b>	<b>\$ 43.99</b>

**Observations during the harness-config generations.** When providing the specification document in `yml` format, we observe that the LLM tends to forget parts of the instructions in the `gemini.md`. We believe that since the `yml` representation of the specification document is much more informationally diluted due to the YAML formatting, it fills the context window more quickly than in the case of the informationally denser representation `md`. While the user can force the model to reload its `gemini.md` to remind itself of its duties, this is cumbersome during the interaction with the Gemini CLI.

**Notes to consider.** While the experiments in Sect. 5.3.1 and Sect. 5.3.2 reveal `md` to be the best performing input type, the interaction with the Gemini CLI also went more smoothly than with `yml` and `pdf`. We acknowledge that this is a subjective observation, but we deem it important to note that LAHG is a semi-automatic tool, which requires user interaction. Therefore, we convert all `pdf` files, by default, to `md` to make this interaction as seamless as possible.



## 6 Conclusion

Writing formal verification harnesses for firmware is a manual, knowledge-intensive process that requires expertise in both the application domain and software verification. In this thesis, we presented LAHG, an LLM-assisted harness generator that turns this process into a semi-automatic, dialog-driven pipeline.

LAHG combines an Internal Generative Rule Set, generated through Iterative Refinement with the Gemini API, with a 3-Stage Guardrailed Pipeline enforced through an MCP tool suite and the Gemini CLI. We demonstrated LAHG on the Intel TDX module firmware and evaluated how specification preprocessing affects the LLM’s ability to generate harness-config pairs compatible with HarnessForge and SV-COMP.

### 6.1 Research Findings

Our evaluation addressed four research questions (cf. Chp. 1). Regarding **RQ. 1**, we found that providing the entire 430-page ABI specification exceeds the Gemini CLI’s input window of 1,048,576 tokens, even though the Gemini API can handle the same document with the same underlying model `gemini-3-flash-preview`. This shows that the deployment interface imposes its own constraints beyond the model’s theoretical capacity. The `data-spec split`, which extracts only the relevant function specification and its underlying data structures, is therefore the necessary and better-performing choice.

For **RQ. 2**, we compared three specification input types: `pdf`, `md`, and `yaml`. While `yaml` reached the highest individual Stage in the Guardrailed Pipeline during IGRS-based harness-config generation, `md` proved to be the most consistent input type across all configurations and the most cost-effective overall. We attribute this to Markdown being informationally dense and not requiring image recognition like `pdf`, nor diluting content with structural formatting overhead like `yaml`. These results indicate that converting the specification from a layout-centered format to a textual format improves the model’s understanding of the document.

Regarding **RQ. 3**, we found that the sequence of few-shot examples does influence the model’s output. The `manual` sequence, constructed by maximizing the semantic spread through Farthest-First Traversal, reached the highest average Stage per S-bucket, outperforming both `random` and `sequential` orderings. This confirms that a deliberate curation of training examples benefits the IGRS quality.

For **RQ. 4**, we confirmed that the optimal input type identified for the Gemini API (`md`) also holds for the Gemini CLI. When providing the specification in `md` format during the harness-config generation, the LLM reached HarnessForge compatibility

(Stage 2) in fewer S-buckets than with `pdf` or `yml`, while maintaining reasonable token consumption.

## 6.2 Limitations

LAHG successfully generates compilable C harnesses and HarnessForge-compatible configurations. However, no configuration produced verification tasks that CBMC could verify (Stage 3). The encoded pre- and post-conditions do not yet satisfy the verifier, which indicates that while the LLM can abstract the structural relationship between a specification and its implementation, faithfully encoding the precise logical semantics required for formal verification remains an open challenge.

We further observed that the LLM produced syntactically incorrect YAML configurations on multiple occasions, despite explicit schema definitions in both the IGRS and the few-shot examples. In addition, LAHG’s project-agnostic design led to recurring issues with include-path resolution, since the model could not reliably infer the correct paths without explicit codebase initialization.

The total cost of our experiments amounted to approximately 334 million input tokens. While individual runs can be inexpensive, the cumulative cost of systematic experimentation is not negligible.

## 6.3 Conclusion

LAHG demonstrates that LLMs can meaningfully accelerate the creation of formal verification harnesses. The combination of Iterative Refinement for IGRS generation, a Guardrailed Pipeline with automated feedback, and a Human-in-the-Loop protocol provides a viable framework for leveraging LLMs in knowledge-intensive verification tasks. LAHG successfully generates compilable harnesses and valid HarnessForge configurations semi-automatically, significantly reducing the manual effort required of a software verification engineer. The remaining gap between structurally correct artifacts and semantically sound verification conditions presents an opportunity for further improvement, which we address in the following section.

## 6.4 Future Work

**ToC-based generation.** Retrieval-Augmented Generation (RAG) for knowledge-intensive tasks benefits from exposure to the Table of Contents (ToC) of large documents [22, 36]. While `data-split` is an effective reduction of informational noise to the model, ToC-based splitting might be even more effective. By providing the LLM only with the Table of Contents, it can decide for itself what it deems relevant. This results in a more fine-grained split and might increase the informational density of the content, thereby reducing its impact on the context window. This approach applies to both IGRS generation and harness-config generation and is independent of the file type. As the time required to convert a file from `pdf` to another format increases with the document’s size, this process can be sped up by reducing the

number of pages to convert. We believe that future research on the impact of ToC-based LLM-guided information retrieval is a promising enhancement to the field of LLM-based code generation.

**Specification type alters the model’s interpretation.** We suspect that the format of a file not only alters the model’s understanding of the content by changing the informational density through structural formatting, but it also influences the model’s interpretation of the content. As `md` is mainly used by developers for technical documentation, `yml` primarily serves as a structured data container, and `pdf` is a document format used by the general public. As the model encounters these documents during training, their content correlates with the file type. This implicit connection might influence the model’s interpretation of the content. We propose further research on that topic to quantify that effect and determine its usefulness in document provisioning for LLMs.

**Programmatic YAML config generation.** As described in [Sect. 4.5](#), LAHG entrusts the LLM with the generation of the config for HarnessForge. However, since their orchestrative role is encoded in invariable YAML schema segments, their deterministic nature invites a programmatic approach. The orchestration required to create a verification-task specification in the config would still need to be handled by the LLM. Merely the codification of this information would be handled programmatically.

We suggest extracting the *Output Format & Internal Digest* mentioned in [Fig. 4.3](#) to a separate YAML guide. This guide would detail the process, from mapping the logic in the harness and encoding the pre- and post-conditions for the function under verification to the initialization and cleanup processes defined in the code base’s auxiliary files. We believe that this would free the model from cognitive load. These tokens are more useful for analysis and logical abstraction.

**Dedicated fv-context loading segment.** LAHG initializes the Gemini CLI session by providing the MCP server in addition to the `gemini.md` file. It does not provide information about the layout of the codebase of the function under verification. As mentioned in [Sect. 5.3.2](#), this leads to false include paths for header files and general path resolution issues.

However, we suspect that inserting a deliberate initialization of the repository under verification before the main user interaction might alleviate most of these issues. There is a project-agnostic approach to this measure as well as a project-aware approach. The project-aware approach consists of crafting a prompt template, similar to a form, that the user fills with information about the project structure. This prompt can then be presented to the LLM such that it starts the session with an informational basis. The project-agnostic approach requires no manual labor but may be less precise than the previous approach. It builds on the idea that the LLM will find and infer all necessary information, given enough time and informational resources. This might also lead to surprising insights, albeit at a cost. We believe that deliberate exposure to the repository under verification reduces the token consumption by enabling the LLM to reach its goals with fewer prompts and iterations.

**Separation to document style guides and Gemini project configuration.** The `gemini.md` used in LAHG contains information at multiple levels of organization, including style guides, behavioral rules, and terminology. As mentioned in [Sect. 5.4](#), we suspect that the `gemini.md`'s level of granularity and amount of content clash with the context window usage of large files. We therefore propose to remove anything unrelated to behavioral rules from the `gemini.md`. By creating style guides for certain file types that can be loaded at any time, we reduce the informational overhead of the `gemini.md`. We believe that if the model starts harness-config generation solely from behavioral context, it can load auxiliary information at any time, reducing the impact on the context window and thus discouraging hallucinations.

# Bibliography

- [1] S. M. Abtahi and A. Azim. Securing llm-generated embedded firmware through AI agent-driven validation and patching. *CoRR*, abs/2509.09970, 2025.
- [2] M. Andrzejewski, N. Dubicka, J. Podolak, M. Kowal, and J. Siłka. Automated test generation using large language models. *Data*, 10(10), 2025.
- [3] Anthropic PBC. Introduction to the model context protocol (mcp). <https://modelcontextprotocol.io/docs/getting-started/intro>, 2024. Accessed: 2026-03-13.
- [4] B. Atil, A. Chittams, L. Fu, F. Ture, L. Xu, and B. Baldwin. LLM stability: A detailed analysis with some surprises. *CoRR*, abs/2408.04667, 2024.
- [5] D. Beyer. Master’s thesis: Tooling for harness generation for c programs, 2024.
- [6] D. Beyer and J. Strejček. Improvements in software verification and witness validation: Sv-comp 2025. In A. Gurfinkel and M. Heule, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–186, Cham, 2025. Springer Nature Switzerland.
- [7] S. Billa and X. Jing. Travelbench : Exploring LLM performance in low-resource domains. *CoRR*, abs/2510.02719, 2025.
- [8] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.
- [9] Y. Charalambous, E. Manino, and L. C. Cordeiro. Automated repair of AI code with large language models and formal verification. *CoRR*, abs/2405.08848, 2024.
- [10] X. Chen, M. Lin, N. Schärli, and D. Zhou. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.
- [11] I. Corporation. Intel Trust Domain Extensions (TDX) Documentation. <https://www.intel.com/content/www/us/en/developer/>

- [tools/trust-domain-extensions/documentation.html](https://tools/trust-domain-extensions/documentation.html), 2025. Accessed: 2025-07-08.
- [12] S. Dhuliawala, M. Komeili, J. Xu, R. Raileanu, X. Li, A. Celikyilmaz, and J. Weston. Chain-of-verification reduces hallucination in large language models. In L. Ku, A. Martins, and V. Srikumar, editors, *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 3563–3578. Association for Computational Linguistics, 2024.
- [13] Y. Dong, R. Mu, G. Jin, Y. Qi, J. Hu, X. Zhao, J. Meng, W. Ruan, and X. Huang. Position: Building guardrails for large language models requires systematic design. In R. Salakhutdinov, Z. Kolter, K. A. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp, editors, *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pages 11375–11394. PMLR / OpenReview.net, 2024.
- [14] Y. Dong, R. Mu, Y. Zhang, S. Sun, T. Zhang, C. Wu, G. Jin, Y. Qi, J. Hu, J. Meng, S. Bensalem, and X. Huang. Safeguarding large language models: a survey. *Artif. Intell. Rev.*, 58(12):382, 2025.
- [15] A. Elnaggar and B. Tan. Adding context to llm-guided verilog repair. In *26th International Symposium on Quality Electronic Design, ISQED 2025, San Francisco, CA, USA, April 23-25, 2025*, pages 1–7. IEEE, 2025.
- [16] Google. Embeddings with the Gemini API. <https://ai.google.dev/gemini-api/docs/embeddings>, 2024. Accessed: 2026-03-13.
- [17] Google. Gemini. <https://gemini.google.com/>, 2024. Accessed: 2026-03-13.
- [18] Google. Google gen ai sdk for python. <https://googleapis.github.io/python-genai/>, 2024. Accessed: 2026-03-13.
- [19] Google Cloud. Gemini 3 pro | Generative AI on Vertex AI, 2026. Accessed: 2026-03-05.
- [20] Intel Corporation. Intel trust domain extensions (intel tdx), 2023. Accessed: 2026-03-19.
- [21] Intel Corporation. *Intel® Trust Domain Extensions (Intel® TDX) Module Architecture Specification: Application Binary Interface (ABI)*. Intel Corporation, 3.4 edition, September 2024. Order Number: 348563-007US.
- [22] H. S. Jeong, S. Jo, B. H. Yoon, Y. Heo, H. Jeong, and T. Kim. Zero-shot document understanding using pseudo table of contents-guided retrieval-augmented generation. *CoRR*, abs/2507.23217, 2025.
- [23] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

- [24] P. Link and B. Tan. Evaluating llm-based communicative agents for verilog design. In *26th International Symposium on Quality Electronic Design, ISQED 2025, San Francisco, CA, USA, April 23-25, 2025*, pages 1–8. IEEE, 2025.
- [25] N. Livathinos, C. Auer, M. Lysak, A. Nassar, M. Dolfi, P. Vagenas, C. B. Ramis, M. Omenetti, K. Dinkla, Y. Kim, S. Gupta, R. T. de Lima, V. Weber, L. Morin, I. Meijer, V. Kuropiatnyk, and P. W. J. Staar. Docling: An efficient open-source toolkit for ai-driven document conversion. *CoRR*, abs/2501.17887, 2025.
- [26] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegrefe, U. Alon, N. Dziri, S. Prabhume, Y. Yang, S. Gupta, B. P. Majumder, K. Hermann, S. Welleck, A. Yazdanbakhsh, and P. Clark. Self-refine: Iterative refinement with self-feedback. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.
- [27] N. S. Mathews and M. Nagappan. Test-driven development and llm-based code generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE '24*, page 1583–1594, New York, NY, USA, 2024. Association for Computing Machinery.
- [28] D. Paul, M. Ismayilzada, M. Peyrard, B. Borges, A. Bosselut, R. West, and B. Faltings. REFINER: reasoning feedback on intermediate representations. In Y. Graham and M. Purver, editors, *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2024 - Volume 1: Long Papers, St. Julian's, Malta, March 17-22, 2024*, pages 1100–1126. Association for Computational Linguistics, 2024.
- [29] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 2339–2356. IEEE, 2023.
- [30] F. M. Polo, L. Weber, L. Choshen, Y. Sun, G. Xu, and M. Yurochkin. tinybenchmarks: evaluating llms with fewer examples. In R. Salakhutdinov, Z. Kolter, K. A. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp, editors, *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pages 34303–34326. PMLR / OpenReview.net, 2024.
- [31] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Trans. Software Eng.*, 50(1):85–105, 2024.
- [32] Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang. Llms in software security: A survey of vulnerability detection techniques and insights. *CoRR*, abs/2502.07049, 2025.
- [33] SoSy-Lab. Harnessforge. <https://gitlab.com/sosy-lab/software/harnessforge>, 2024. Accessed: 2026-03-13.

- [34] B. Sundararajan, S. Sripada, and E. Reiter. Input matters: Evaluating input structure’s impact on llm summaries of sports play-by-play, 2025.
- [35] G. Tie, Z. Yuan, Z. Zhao, C. Hu, T. Gu, R. Zhang, S. Zhang, J. Wu, X. Tu, M. Jin, Q. Wen, L. Chen, P. Zhou, and L. Sun. Can llms correct themselves? A benchmark of self-correction in llms. *CoRR*, abs/2510.16062, 2025.
- [36] S. Wang, Y. Zhou, and Y. Fang. Bookrag: A hierarchical structure-aware index-based approach for retrieval-augmented generation on complex documents. *CoRR*, abs/2512.03413, 2025.
- [37] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.
- [38] Z. Xu, S. Jain, and M. S. Kankanhalli. Hallucination is inevitable: An innate limitation of large language models. *CoRR*, abs/2401.11817, 2024.
- [39] Q. Ye, M. Axmed, R. Pryzant, and F. Khani. Prompt engineering a prompt engineer, 2024.
- [40] Z. Ying, D. Towey, and Y. Zhang. Evaluation of the code generated by large language models: The state of the art. In H. Shahriar, K. S. Alam, H. Ohsaki, S. Cimato, M. A. M. Capretz, S. Ahmed, S. I. Ahamed, A. J. A. Majumder, M. Haque, T. Yoshihisa, A. Cuzzocrea, M. Takemoto, N. Sakib, and M. Elsayed, editors, *49th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2025, Toronto, ON, Canada, July 8-11, 2025*, pages 440–449. IEEE, 2025.