



# Automating Linux Kernel Documentation for Safety-Critical Compliance through Large-Language Models

Presented by Justin Stanley & Grant Stensland

LPC 2025



# Who We Are

## VES LLC | A Waséyabek Company

- Established in 2014
- 100% Tribally-owned holding company
- Team located across United States
  - Main headquarters is located in Maryland with an annex office in Michigan
  - Personnel located on all coasts from the DC Metro to Seattle
  - 80+ Full-time employees and growing rapidly

## Core Competencies

- Custom infrastructure development for embedded environments
- Embedded systems integration (OSs, BSPs, Application porting)
- Edge prototyping of emerging technologies
- Deployable software products with secure container orchestration
- Security and compliance via DevSecOps and Zero Trust
- ISO 9001-2015 Certified

# Acknowledgements

## Community Thanks

- Steven Rostedt for kernel documentation validity feedback
- The ELISA team for the insightful discussions and effort that helped spark this investigation



## Safe Systems with Linux Microconference

- Tomorrow starting at 10:00 AM in Hall B4
  - Aspects of Dependable Linux Systems
  - NVIDIA Approach for Achieving ASIL B Qualified Linux
  - Applying Program Verification to Linux Kernel Code
  - Defining and maintaining requirements in the Linux Kernel
  - KUnit Testing Insufficiencies
  - Exploring possibilities for integrating StrictDoc with ELISA's requirements template approach for the Linux kernel
  - BASIL: Open-Source Traceability for Safety-Critical Systems
  - Tooling and Sharing Traceability Discussion

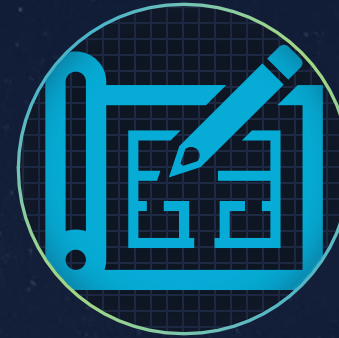
# Agenda



1) Motivation



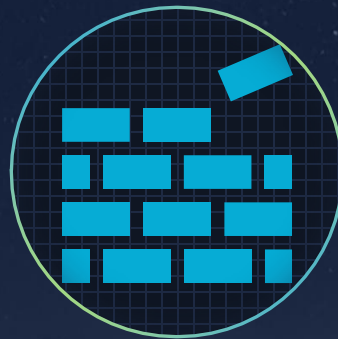
2) Case Study



3) Tech Overview



4) Results



5) Next Steps



6) Q&A

# Motivation

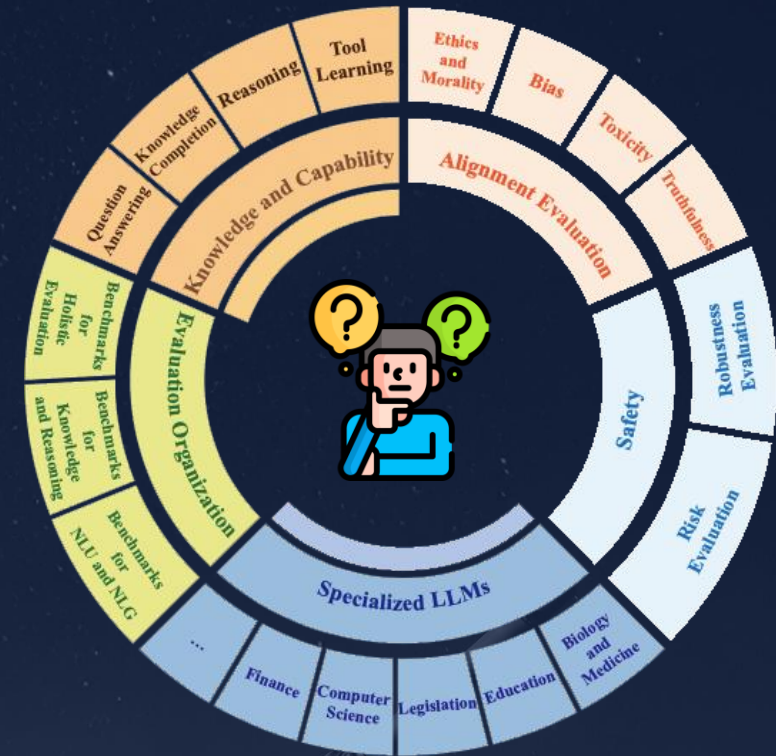
# Motivation: The Compliance Challenge

- Despite its technical maturity and reliability, Linux is unusable in safety-critical applications in the domains of aerospace, automation, and defense.
- The key barrier is regulatory compliance, not performance. Standards such as ASIL or DO-178C require rigorous standardized documentation.
  - This documentation emphasizes accuracy, reproducibility, and traceability.
  - Producing this documentation manually is slow and costly, often taking years of effort. Especially when not done up front.
- The Linux kernel, though technically robust and widely used, lacks certification-ready documentation, limiting its use in regulated domains.
- There is a growing gap between open-source innovation and the rigor demanded by certification authorities.



# Motivation: Leveraging LLMs

- Large Language Models (LLMs) can summarize complex functions, explain algorithms, and highlight assumptions.
- Models can potentially detect undocumented edge cases or conditions needing clarification
- LLMs can help "translate" expert-level kernel code into safety-auditor-friendly language
- LLMs cannot guarantee correctness, formal verification, or certification compliance.
- Outputs require validation by human experts. LLMs can assist, but do not replace safety engineers.
- The value lies in accelerating documentation, not in automating the entire certification process.



# Case Study

# Case Study: Applying LLMs to ftrace

## Case Study

- Explore whether locally deployed LLMs can automate documentation generation that aligns with safety-critical standards.
- Apply this approach to the Linux kernel's ftrace utility.
- Evaluation metrics:
  - Validity and standards alignment of generated documentation
  - Computational performance in relation to each LLM
  - Reproducibility of outputs

## Specifications

- Workstation specs:
  - Ubuntu 22.04
  - Intel i7 14700k CPU
  - Nvidia RTX 4000 Ada GPU
- Families of LLMs evaluated:
  - Meta Llama 3.2
  - StarCoder2
  - Mistral Devstral

# Case Study: Model Downselect



## Compatibility with Ollama

- All LLMs open source and compatible with Ollama front end
- All models relatively mature and stable at the time of the study
- Balanced spectrum of model size and capabilities

## Compute Efficiency Metrics

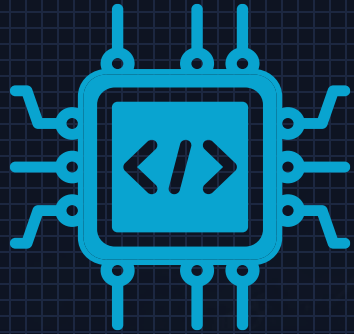
- Model size (GB)
- Parameter count (B)
- Speed (Tokens / Second)
- Maximum Context Window (K tokens)

## Prompt Used

"For Function's expectations: provide several lines of explanation for how every combination of arguments will affect the output start each explanation with a "-"  
MAKE SURE TO ALWAYS PRODUCE A C BLOCK COMMENT, only produce one block comment"

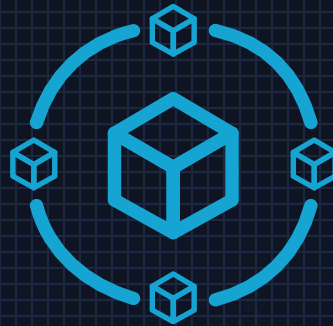
# Technical Overview

# Requirements



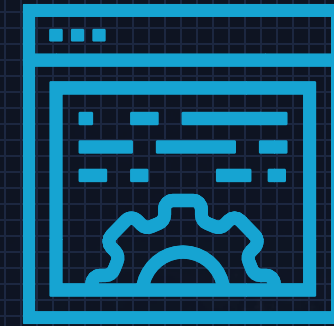
## Conciseness

- Easily reviewable for practical integration
- Concise for effective requirement tracing
- Minimize ambiguity to support audits



## Reproducibility

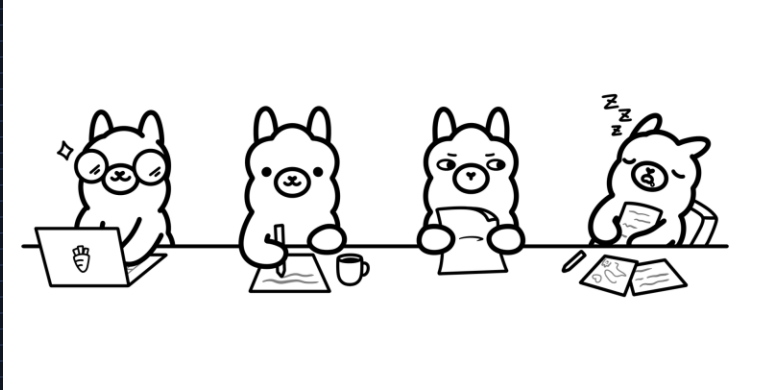
- Reduced noise in outputs
- More reliable tests and prompt tuning
- Stable results across minor prompt changes and configurations



## Model Robustness

- Easily deploy LLMs to target hardware
- Remote inference for deployment flexibility
- Modular deployment to support model substitution for evaluation

# Existing Software



## Ollama

- Open-source LLM backend
- Remote inference via REST API
- Public model registry, easily swappable
- Supports multiple models for comparison and benchmarking

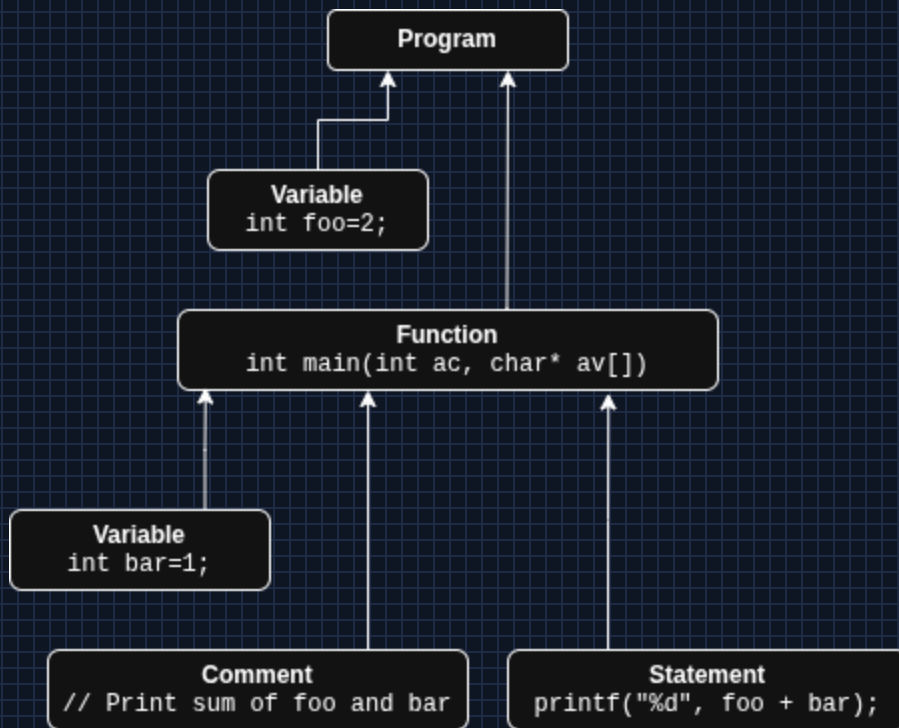


## Tree-sitter

- Open-source incremental language parser supporting multiple languages
- Provides reliable syntax trees for analyzing and reconstructing code
- Enables consistent validation of LLM-generated output

```
int foo=2;

int main(int ac, char* av[]) {
    int bar=1;
    // Print sum of foo and bar
    printf("%d", foo + bar);
}
```

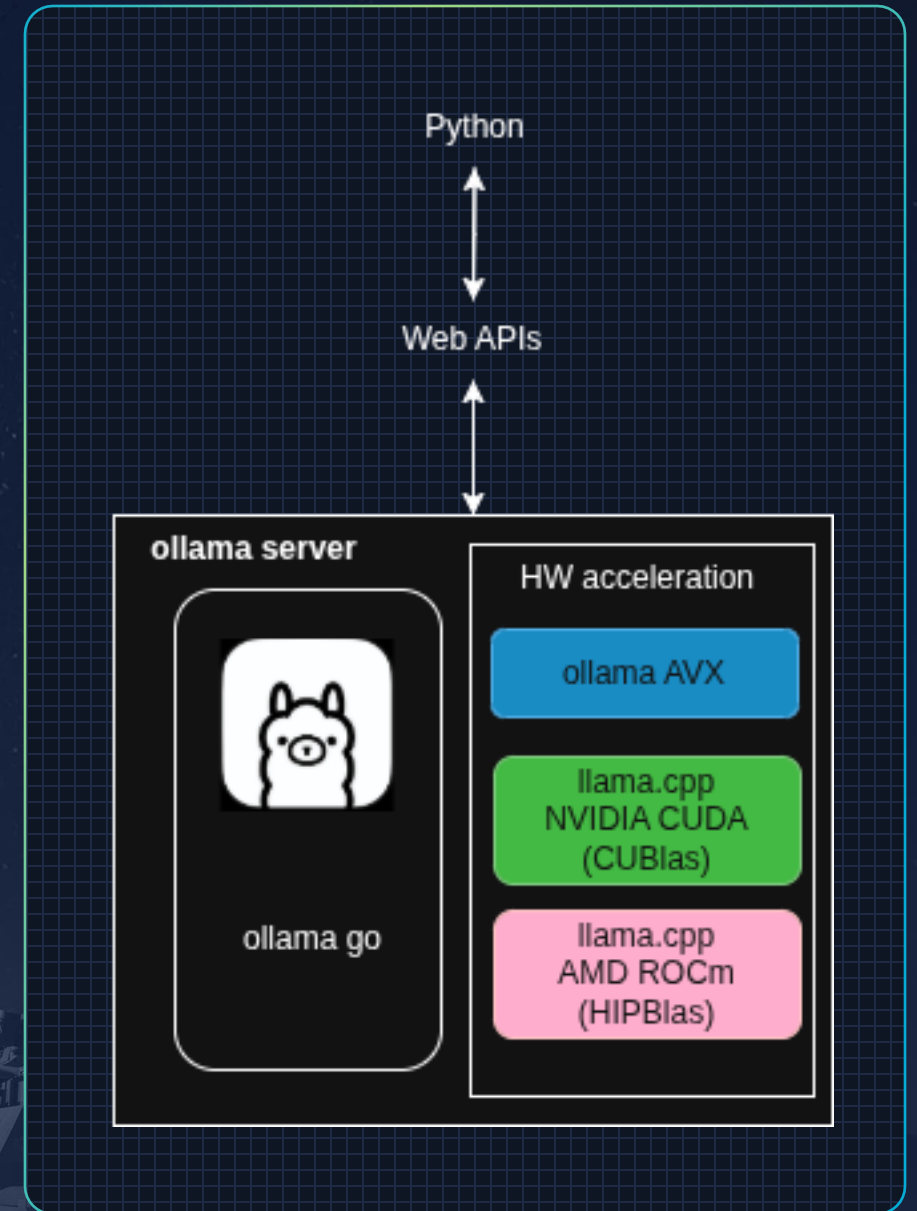


# Tree-sitter and ASTs

- Tree-sitter provides an open-source, language agnostic parser
- We focus on parsing C code to target the Linux kernel
  - Parsing other languages requires minimal changes, just loading new parser modules
- Parser converts source code text into **AST** structures (**Abstract Syntax Tree**)
- ASTs describe the program's semantic meaning
  - Used similarly by compilers for code generation
- We use them to extract functions for documentation, and later for building and verifying newly generated source files

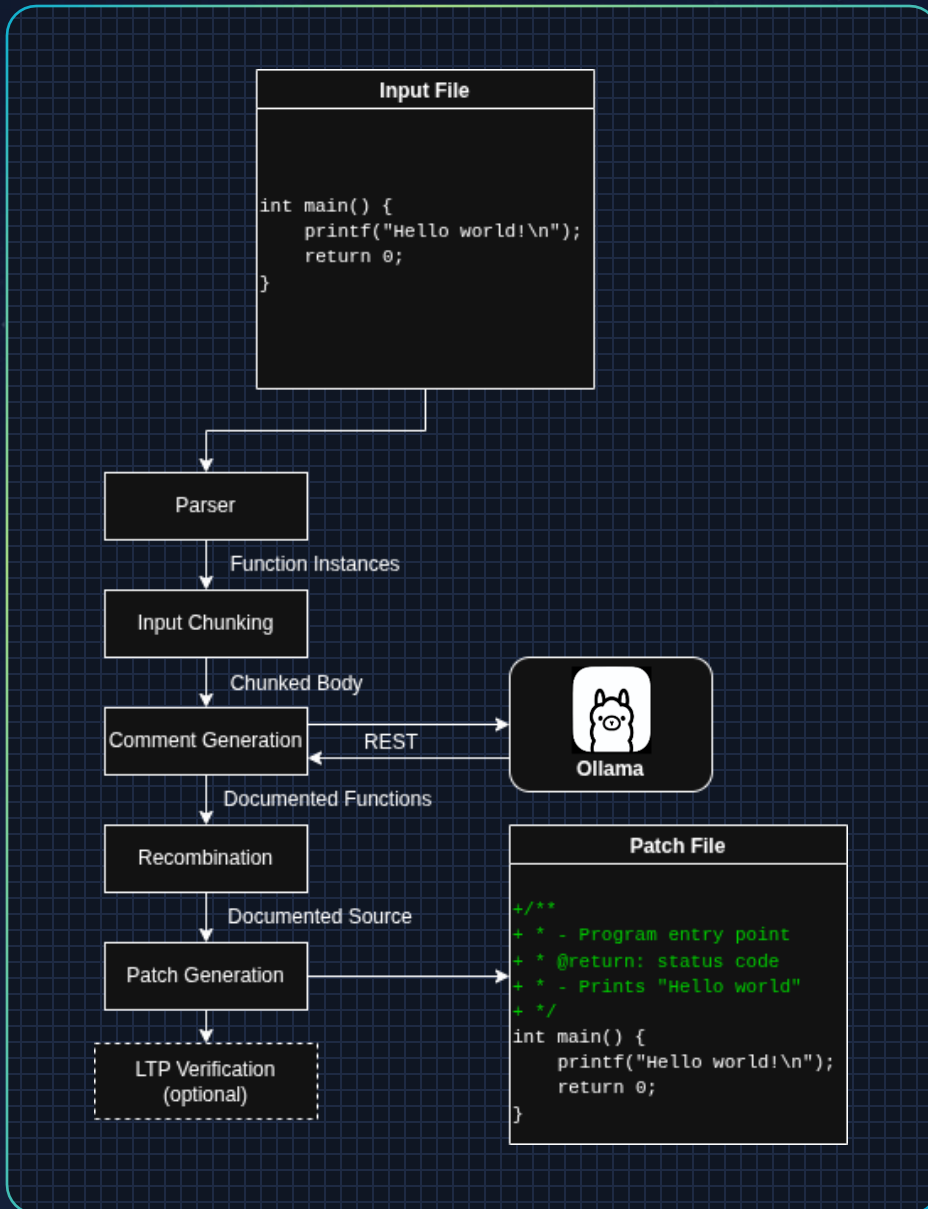
# Ollama

- Robust LLM backend, supporting wide range of target hardware for both local and on-prem deployment
- GPU-agnostic hardware acceleration using CUDA, ROCm, AVX, and other backends
- Remote inference via REST API for simple and easy integration
- Straightforward scaling with built-in support for load balancing
- Mature client libraries (Python, JavaScript, Go, etc.) enabling seamless integration
- Public model registry with easily interchangeable open-weight models



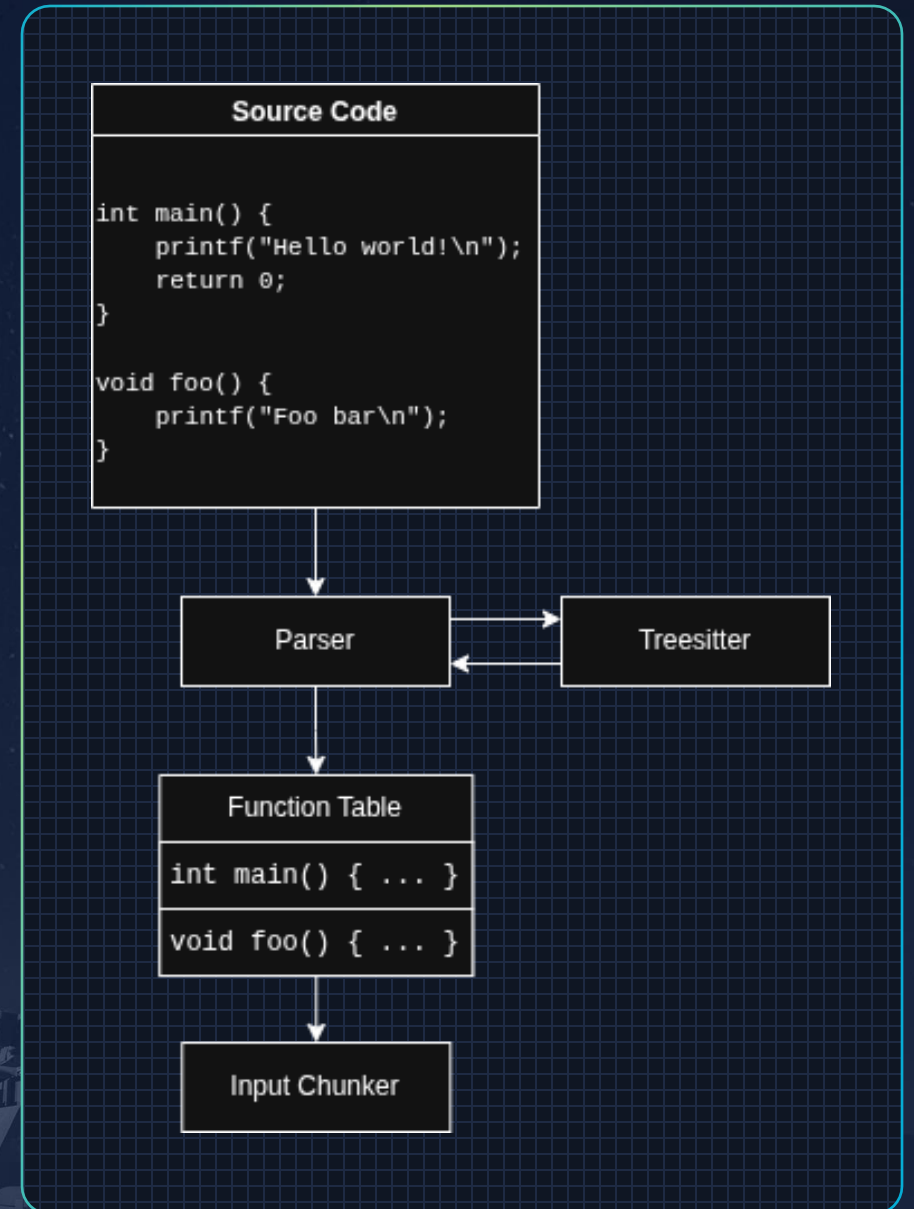
# Pipeline Overview

- Highly parallelizable (per input file)
- Swappable, load-balanced LLM backend
  - Implicit verification before patch generation
  - Explicit verification by running tests on newly built binaries
    - Linux Test Project
- Git-ready patch files produced for each input source file



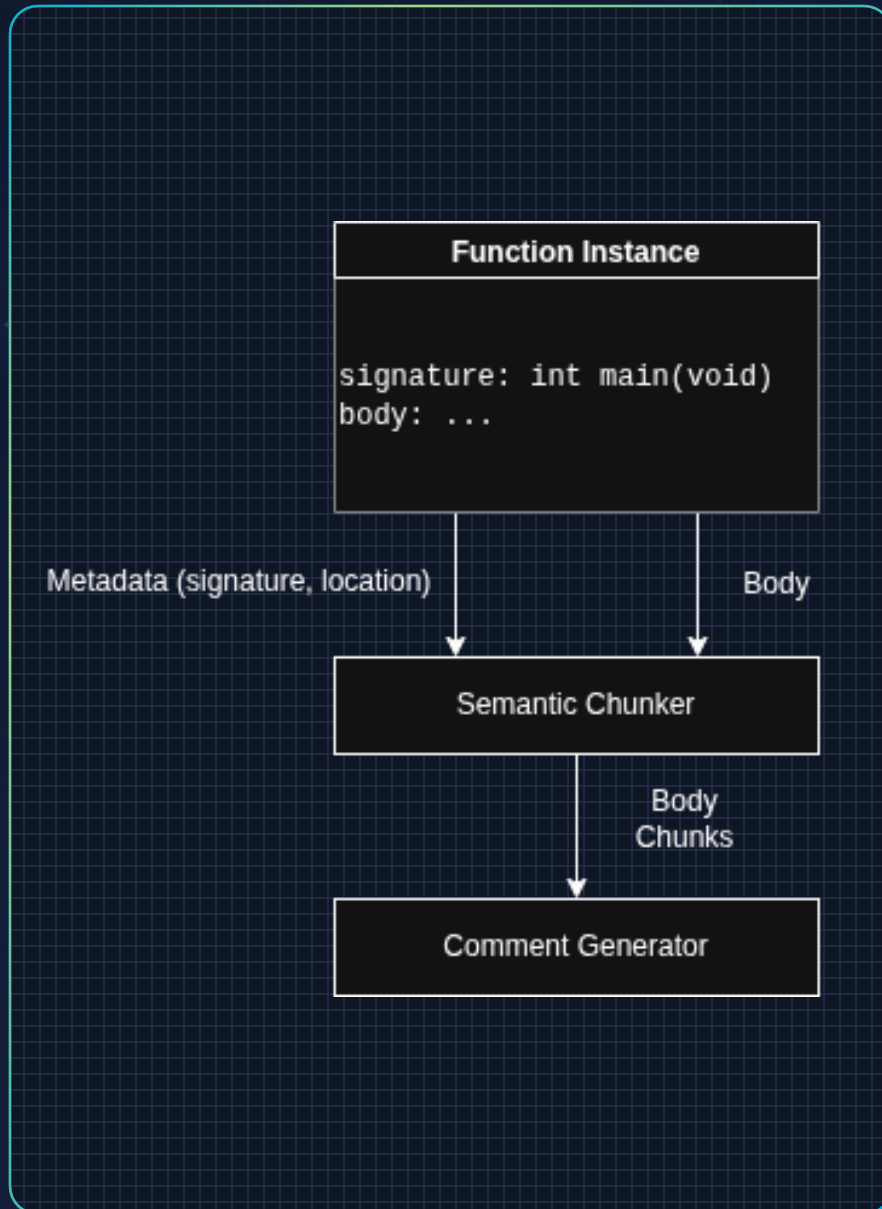
# Function Parsing

- Initial stage of the input pipeline, responsible for extracting structured information from source code
- Source code is passed to **Tree-sitter** which collects body and metadata for each function in top-level scope
- Extracted data includes:
  - Function body (source code)
  - Function signature (params, return value)
  - Function location (file, line number)
- Parsed function data is stored and passed to the next pipeline stage, **Input Chunking**



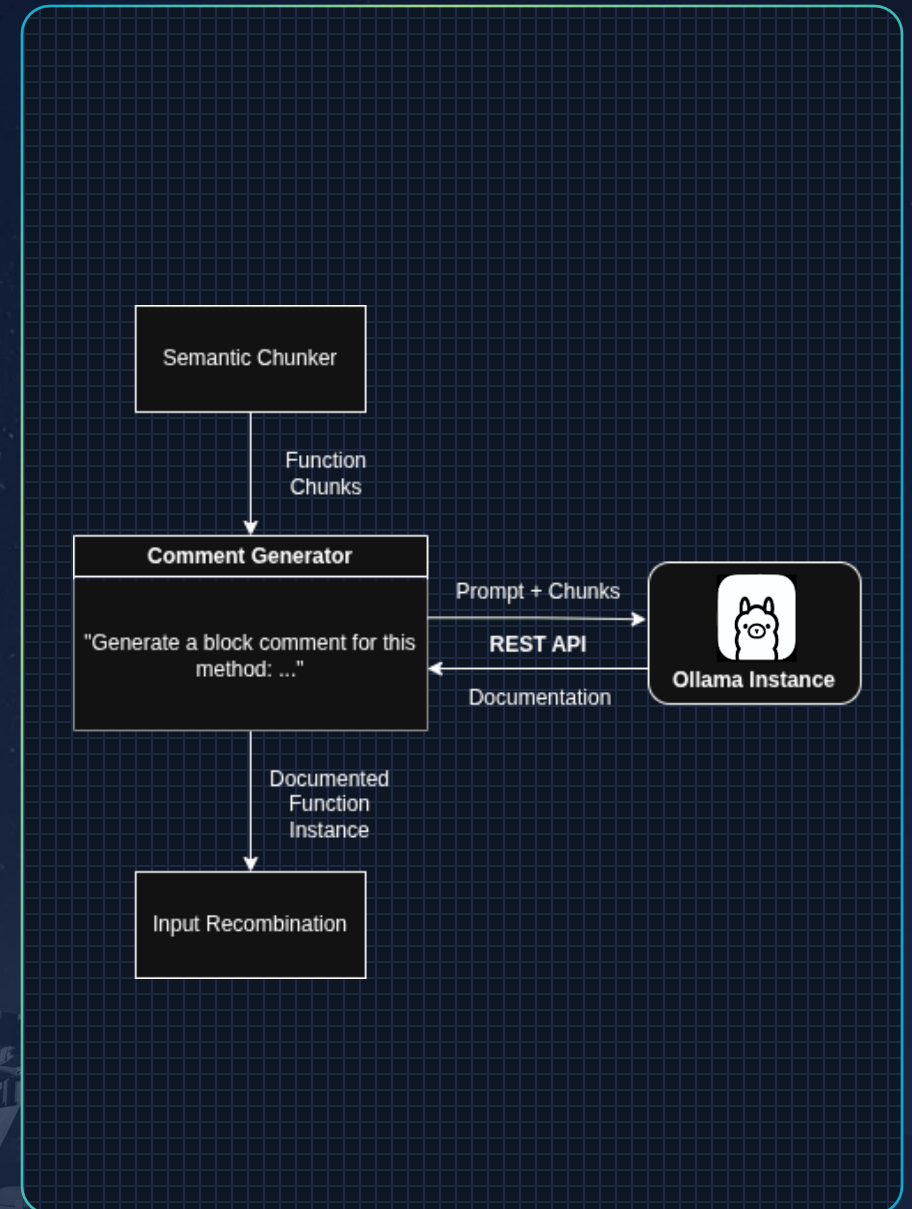
# Input Chunking

- After collecting functions from the original source, we prepare to submit them to the LLM
- However, LLMs have input size limitations
- If functions are too large, we split the body into **chunks**
- How to pick chunk boundaries?
  - Naïve chunking: align chunks exactly with LLM token limit
    - Source code tokens might land on chunk boundary and "confuse" LLM
  - Smart chunking: use **Tree-sitter** to pick a chunk boundary between source code tokens
    - Code is broken into ingestible, independent chunks



# Comment Generation

- Chunks of function bodies are sent to the LLM after input chunking stage
  - Uses a fixed prompt with documentation instructions combined with source code
  - Request are made to the Ollama instance via the REST API for inference
  - Documentation received is saved along with function metadata (file, location, body)
- Newly documented functions are then forwarded to the next stage: [Input Recombination](#)



# Prompt Selection

- We use a fixed prompt to instruct LLM to generate documentation, but how do we pick a strong prompt?
  - Follow established formatting from existing ELISA efforts
    - ▶ Emphasis on producing **traceable requirements**
  - Example patches for ftrace source code
    - ▶ SPDX tags require unique IDs, referencing – LLMs shouldn't generate these
    - ▶ Future work: extending pipeline to directly add SPDX tags
  - Iteratively refine prompt, identifying one that produces consistently formatted output

```
kernel/trace/trace_events.c
...
763 763     } while_for_each_event_file();
764 764 }
765 765
766 + /*
767 +  * SPDX-Req-ID: [TODO: automatically generate it]
768 +  * SPDX-Req-Ref: [TODO add the SPDX-Req-ID of __ftrace_set_clr_event_nolock].
769 +  * SPDX-Text:
770 +  * __ftrace_event_enable_disable - enable or disable a trace event.
771 +  * @file: trace event file associated with the event.
772 +  * @enable: 0 or 1 respectively to disable/enable the event (any other value is
773 +  * invalid).
774 +  * @soft_disable: 1 or 0 respectively to mark if the enable parameter IS or
775 +  * IS NOT a soft enable/disable.
776 +  *
777 +  * Function Expectations:
778 +  * - If soft_disable is 1 a reference counter associated with the trace
779 +  * event shall be increased or decreased according to the enable parameter
780 +  * being 1 (enable) or 0 (disable) respectively.
781 +  * If the reference counter is > 0 before the increase or after the decrease,
782 +  * no other actions shall be taken.
783 +  *
784 +  * - if soft_disable is 1 and the trace event reference counter is 0 before
785 +  * the increase or after the decrease, an enable value set to 0 or 1 shall set
786 +  * or clear the soft mode flag respectively; this is characterized by disabling
787 +  * or enabling the use of trace_buffered_event respectively.
788 +  *
789 +  *
790 +  */
```

GitHub: elisa-tech/linux

```
Generate a C-style block comment for the following function.

Limit overall width to 80 columns.

Start the comment with "* <method> - <description of the method>".
Follow with "@<param>: - <description of the param>" for each parameter.
Follow with "* Function Expectations:".
Follow with "- <behavior expectation>" for each major function behavior.
Follow with a description of the return value.

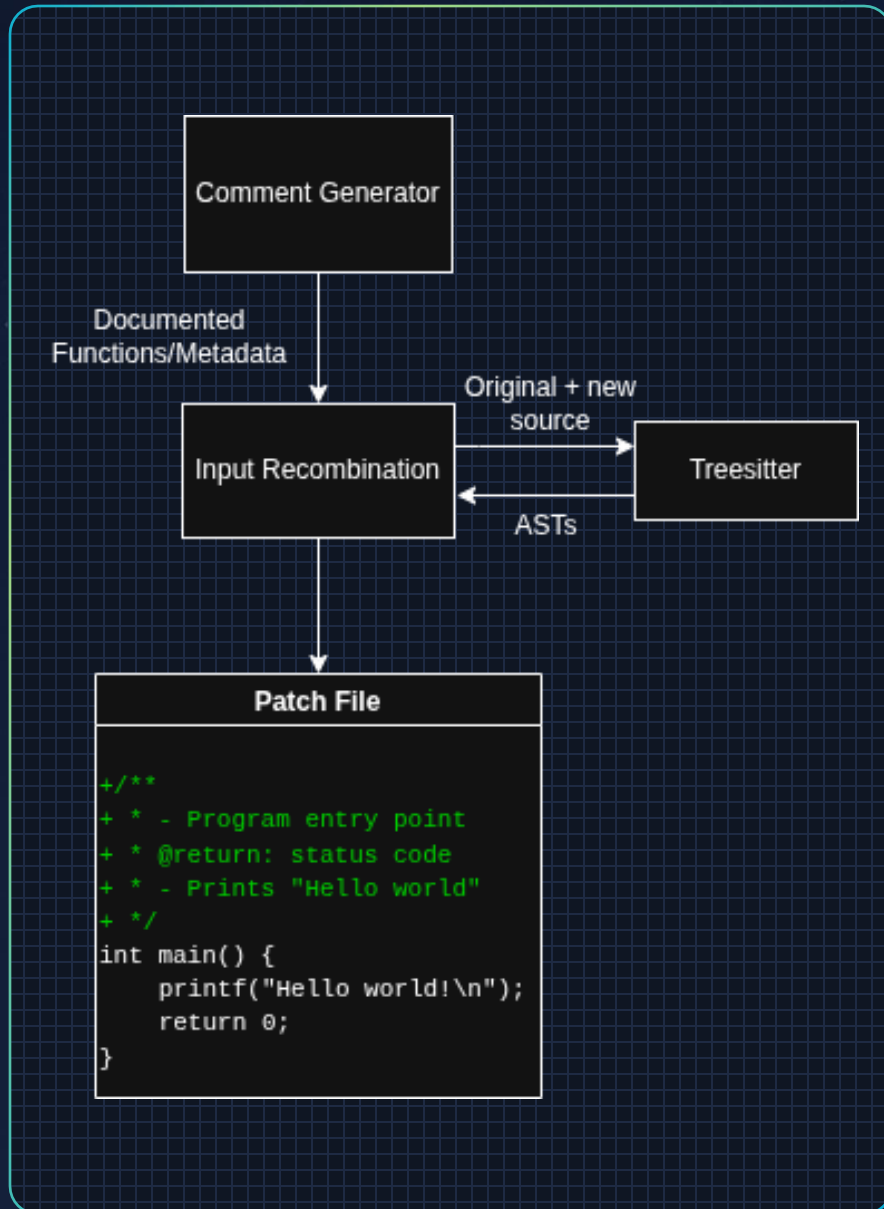
--

int main(int argc, char* argv[]) { ... }
```

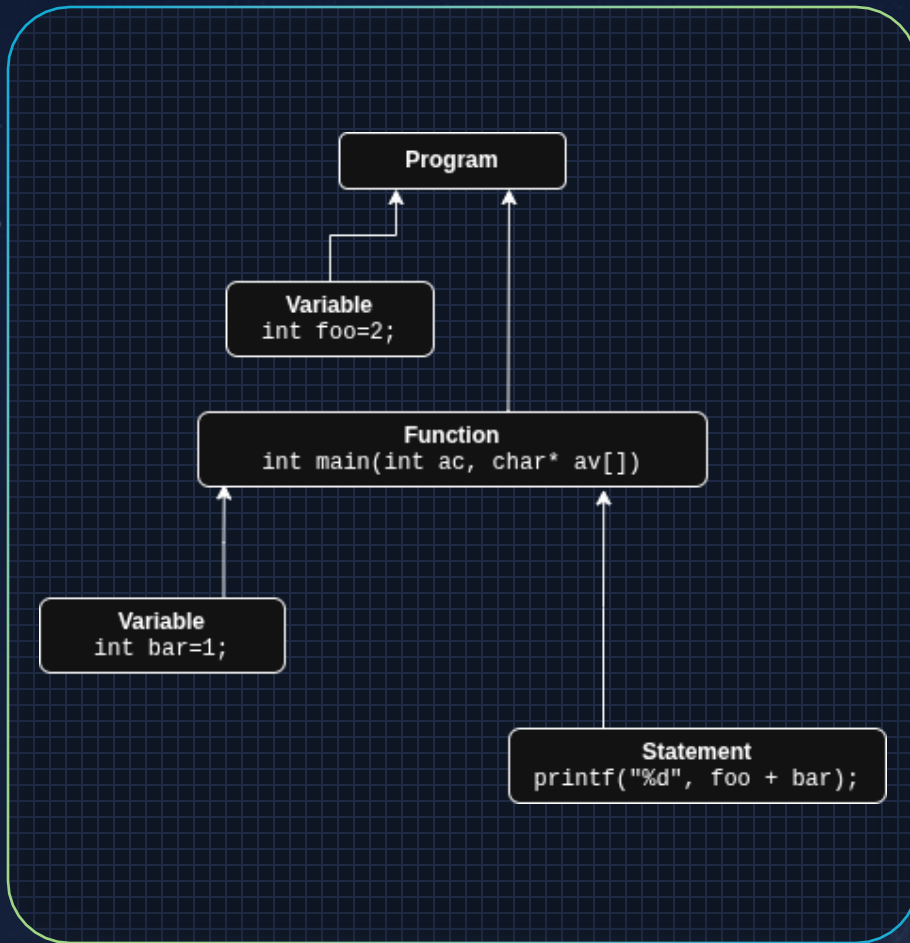
System prompt

# Input Recombination

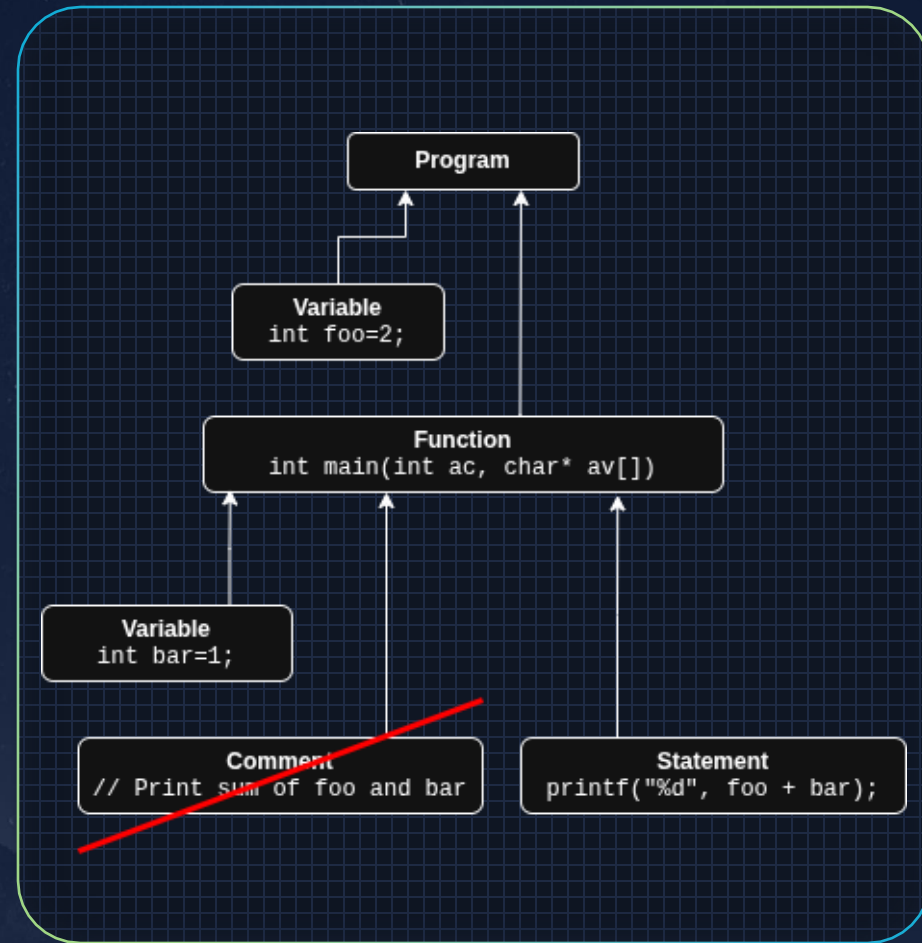
- The comment generator produces a set of documented functions, but we still need to combine them into a complete, documented source file
- Need to keep the rest of the code unchanged! Behavior should never differ
- We build the output and verify it using [Tree-sitter](#)
  - Function metadata includes [location](#), which we use to correctly insert documentation into the new source file
  - After building a complete output file, we parse it into a new AST
  - Now, we have an AST for the original source alongside an AST for the new source
  - We compare the ASTs, verifying comment nodes are the only difference
- Output file is guaranteed to not change behavior



# AST Verification



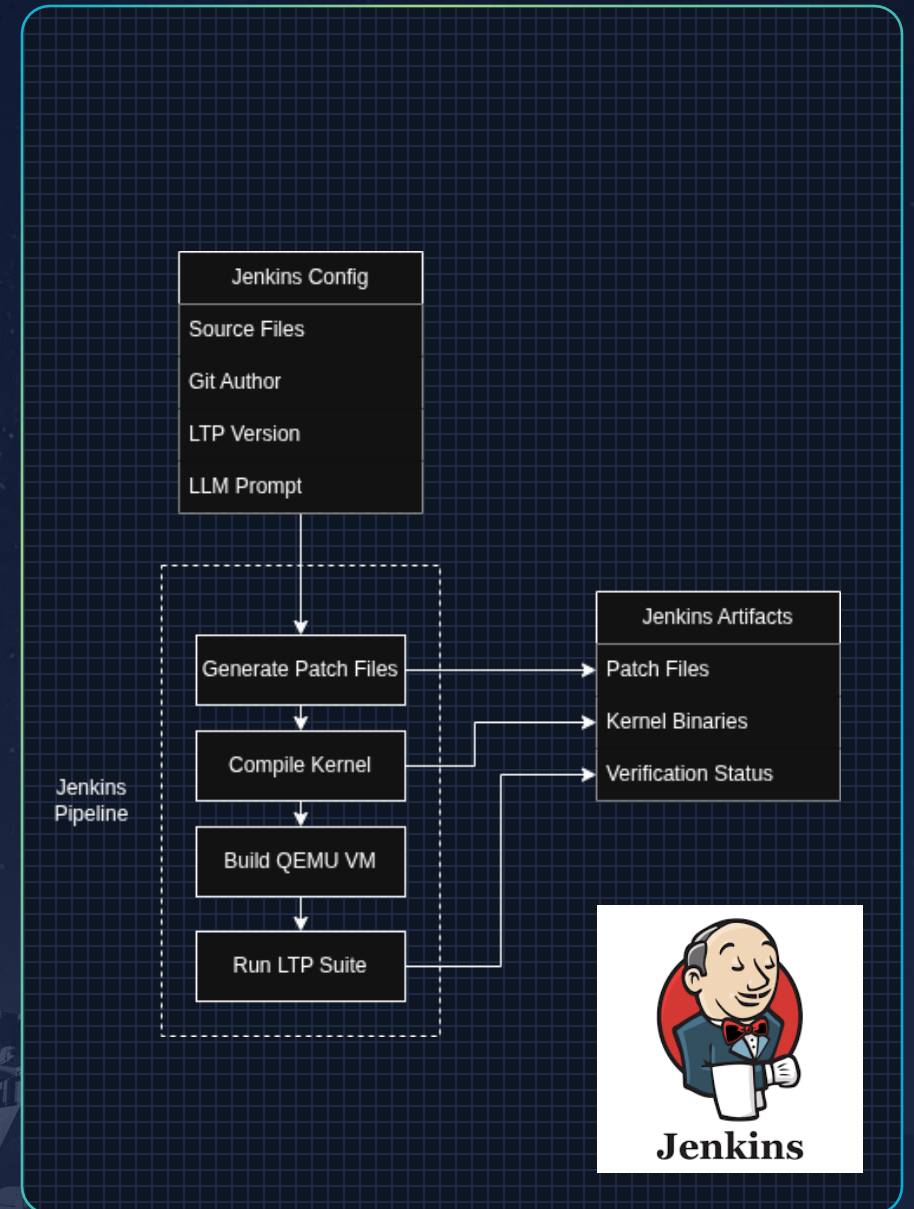
Original undocumented source



New documented source

# Jenkins & LTP Verification

- We deployed the pipeline into Jenkins CI to support development and improve validation
  - Rapid iteration and testing
  - More verification by running tests on newly built kernels
- LTP (Linux Test Project) used to run tests on modified subsystems
- Pipeline automation
  - User provides source files to document, git information
  - Jenkins pipeline produces patch files, binaries and verification results



# Results

# Test Run

```
/* What to set function_trace_op to */
static struct ftrace_ops *set_function_trace_op;

bool ftrace_pids_enabled(struct ftrace_ops *ops)
{
    struct trace_array *tr;

    if (!(ops->flags & FTRACE_OPS_FL_PID) || !ops->private)
        return false;

    tr = ops->private;

    return tr->function_pids != NULL || tr->function_no_pids != NULL;
}

static void ftrace_update_trampoline(struct ftrace_ops *ops);

/*
 * ftrace_disabled is set when an anomaly is discovered.
 * ftrace_disabled is much stronger than ftrace_enabled.
 */
static int ftrace_disabled __read_mostly;

DEFINE_MUTEX(ftrace_lock);
```

Original undocumented source

```
/**
 * ftrace_pids_enabled - Check if PID filtering is enabled for a given trace operation
 * @ops: Pointer to the ftrace_ops structure containing tracing options and flags.
 *
 * Function Expectations:
 * - If the FTRACE_OPS_FL_PID flag is not set in ops->flags or ops->private is NULL,
 *   the function will return false, indicating that PID filtering is not enabled.
 * - If the FTRACE_OPS_FL_PID flag is set and ops->private is not NULL, the function
 *   will check if either tr->function_pids or tr->function_no_pids is non-NULL. If
 *   either of these pointers is non-NULL, the function will return true, indicating
 *   that PID filtering is enabled.
 *
 * Return Value:
 * - Returns true if PID filtering is enabled for the given trace operation,
 *   false otherwise.
 */
bool ftrace_pids_enabled(struct ftrace_ops *ops)
{
    struct trace_array *tr;

    if (!(ops->flags & FTRACE_OPS_FL_PID) || !ops->private)
        return false;

    tr = ops->private;

    return tr->function_pids != NULL || tr->function_no_pids != NULL;
}
```

Documented function

# Test Run (cont.)

```
/**
 * ftrace_pids_enabled - Check if PID filtering is enabled for a given trace operation
 * @ops: Pointer to the ftrace_ops structure containing tracing options and flags.
 *
 * Function Expectations:
 * - If the FTRACE_OPS_FL_PID flag is not set in ops->flags or ops->private is NULL,
 *   the function will return false, indicating that PID filtering is not enabled.
 * - If the FTRACE_OPS_FL_PID flag is set and ops->private is not NULL, the function
 *   will check if either tr->function_pids or tr->function_no_pids is non-NULL. If
 *   either of these pointers is non-NULL, the function will return true, indicating
 *   that PID filtering is enabled.
 *
 * Return Value:
 * - Returns true if PID filtering is enabled for the given trace operation,
 *   false otherwise.
 */
bool ftrace_pids_enabled(struct ftrace_ops *ops)
{
    struct trace_array *tr;

    if (!(ops->flags & FTRACE_OPS_FL_PID) || !ops->private)
        return false;

    tr = ops->private;

    return tr->function_pids != NULL || tr->function_no_pids != NULL;
}
```

Documented function

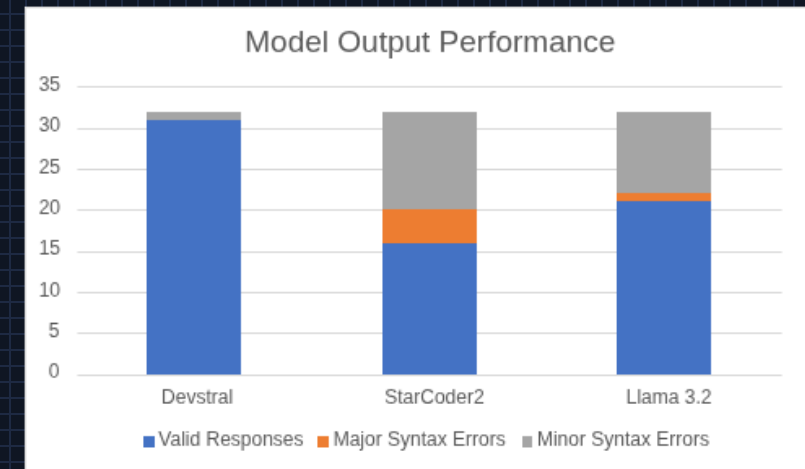
```
diff --git a/source/ftrace.c b/result/ftrace.c
index 728ecda..2edced2 100644
--- a/source/ftrace.c
+++ b/result/ftrace.c
@@ -100,6 +100,22 @@ struct ftrace_ops *function_trace_op __read_mostly = &ftrace_list_end;
 /* What to set function_trace_op to */
 static struct ftrace_ops *set_function_trace_op;

+/**
+ * ftrace_pids_enabled - Check if PID filtering is enabled for a given trace operation
+ * @ops: Pointer to the ftrace_ops structure containing tracing options and flags.
+ *
+ * Function Expectations:
+ * - If the FTRACE_OPS_FL_PID flag is not set in ops->flags or ops->private is NULL,
+ *   the function will return false, indicating that PID filtering is not enabled.
+ * - If the FTRACE_OPS_FL_PID flag is set and ops->private is not NULL, the function
+ *   will check if either tr->function_pids or tr->function_no_pids is non-NULL. If
+ *   either of these pointers is non-NULL, the function will return true, indicating
+ *   that PID filtering is enabled.
+ *
+ * Return Value:
+ * - Returns true if PID filtering is enabled for the given trace operation,
+ *   false otherwise.
+ */
bool ftrace_pids_enabled(struct ftrace_ops *ops)
{
    struct trace_array *tr;
@@ -139,6 +155,37 @@ const struct ftrace_ops ftrace_list_ops = {
    .flags = FTRACE_OPS_FL_STUB,
};
```

Resulting patch

# Model Comparison Results

- Open-source models match closed-source performance
  - Llama 3 and StarCoder2 performed comparably to the GPT models in the high-level documentation tasks
  - Completeness scores demonstrated minimal performance gaps across model families
- No strong link between metrics and documentation quality
  - No correlation between parameters, size, or speed and documentation accuracy
  - Qualitative metrics remained the most reliable indicators
- Three main models were all viable candidates
  - Llama, StarCode, and Devstral were all roughly consistent
  - All demonstrated maturity and capabilities for document-related LLM tasks
- Devstral showed highest validity in testing
  - Produced most syntactically correct outputs
  - Outperformed Llama 3.2 and StarCoder2 despite higher resource usage



Model Performance	StarCoder2	Llama 3.2	Devstral
GPU Usage (%)	96	95	98.5
GPU VRAM (GB)	11.3	5	16.5
Time to Completion (m)	6.5	2	8.33
Valid Responses (x/32)	16	22	31

# Next Steps

# Current Limitations

- Model limitations

- Inconsistent mapping between natural-language and kernel-level code
- Difficulty maintaining context across large, multi-file subsystems

- Technical gaps

- Requirements to code traceability not fully automated end-to-end
- No reliable SPDX relationship extraction or metadata generation

- Operational constraints

- Heavy SME involvement still required for validation
- Training require kernel-specific datasets that are still being gathered and worked on

# Next Steps

- Further prompt refinement
  - Improve prompt specifically for code generation, summarization, and compliance tasks
  - Address known failure patterns (hallucinations, ambiguity, and edge-case handling)
- Investigate multiple training pathways
  - Compare LLM fine-tuning, real time look up, and hybrid approaches
  - Benchmark further to identify best performing LLMs
- Ongoing collaboration with maintainers
  - Validate generated output formatting, structure, and correctness
  - Incorporate upstream feedback into model to reduce reviewer workload and rework cycles
- SPDX-driven requirement tracing
  - Generate SPDX metadata tied to code regions
  - Establish traceability from requirements to code blocks all the way to SPDX artifacts

# GitHub Repos



## lamacoop-docgen

- Python scripts for automating code documentation using LLMs
  - Open to contributors of all experience levels
  - <https://github.com/vesllc/lamacoop-docgen>




## lamacoop-jenkins

- Jenkins pipeline script used to automate build, test, and deployment processes
  - Open to contributors of all experience levels
  - <https://github.com/vesllc/lamacoop-jenkins>

# Questions?

# Thank You.

-  ves.solutions
-  info@ves.solutions
-  6180 Guardian Gtwy Ste. 102 Aberdeen Proving Ground, MD 21005

