

vLLM Hook v0: A Plug-in for Programming Model Internals on vLLM

Ching-Yun Ko and Pin-Yu Chen

IBM Research

Emails: {cyko,pin-yu.chen}@ibm.com

February 2, 2026

Abstract

Modern artificial intelligence (AI) models are deployed on inference engines to optimize run-time efficiency and resource allocation, particularly for transformer-based large language models (LLMs). The vLLM project¹ is a major open-source library to support model serving and inference. However, the current implementation of vLLM limits programmability of the internal states of deployed models. This prevents the use of popular test-time model alignment and enhancement methods. For example, it prevents the detection of adversarial prompts based on attention patterns or the adjustment of model responses based on activation steering. To bridge this critical gap, we present **vLLM Hook**, an opensource plug-in to enable the programming of internal states for vLLM models. Based on a configuration file specifying which internal states to capture, **vLLM Hook** provides seamless integration to vLLM and supports two essential features: *passive programming* and *active programming*. For passive programming, **vLLM Hook** probes the selected internal states for subsequent analysis, while keeping the model generation intact. For active programming, **vLLM Hook** enables efficient intervention of model generation by altering the selected internal states. In addition to presenting the core functions of **vLLM Hook**, in version 0, we demonstrate 3 use cases including prompt injection detection, enhanced retrieval-augmented retrieval (RAG), and activation steering. Finally, we welcome the community’s contribution to improve **vLLM Hook** via <https://github.com/ibm/vllm-hook>.

1 Overview of vLLM Hook

What is vLLM Hook? In the deployment of AI technology, service providers use inference engines to optimize the economic factors of the hosted model, such as latency, memory, energy, hardware, and other resource constraints. Disabling features that were otherwise programmable during the model development phase can significantly reduce inference costs. In particular, the vLLM project [KLZ²³], is a popular open-source, community-driven inference engine that supports the efficient serving of modern AI models, especially transformer-based large language models (LLMs). In short, we refer to a model deployed on vLLM as a “vLLM model.” Despite achieving high inference throughput, the current vLLM package does not support accessing or modifying important internal states of a vLLM model. Examples of these internal states include attentions, attention heads, and activations. The lack of programmability in significant internal states creates a technological gap that prevents the use of advanced techniques for monitoring, adjusting, and controlling deployed models at inference time. To bridge this gap, our vLLM Hook is an opensource plug-in of vLLM to enable flexible programming of internal states within vLLM models.

Our **vLLM Hook** provides seamless and non-intrusive integration to vLLM. The system overview of vLLM Hook is illustrated in Figure 1. Based on a configuration file (Config) specifying which internal states within a vLLM model to “hook”, **vLLM Hook** supports two essential features: *passive programming* and *active programming*.

- *Passive programming* probes and saves the selected internal states for subsequent analysis, while keeping the model generation intact.

¹<https://github.com/vllm-project/vllm>

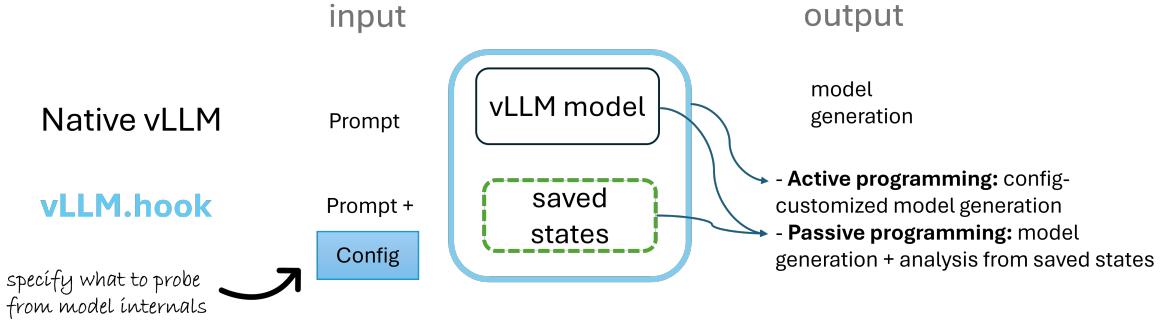


Figure 1: Overview of **vLLM Hook** (<https://github.com/ibm/vllm-hook>), an opensource plug-in to enable flexible programming of internal states for models deployed on vLLM. With a customizable configuration file (Config), vLLM Hook supports two essential features: (1) *Active Programming*: **vLLM Hook** provides efficient intervention of model generation by altering the selected internal states; (2) *Passive Programming*: **vLLM Hook** probes and saves the selected internal states for subsequent analysis, while keeping the model generation intact.

- *Active programming* enables efficient intervention of model generation by altering the selected internal states.

Why do we need vLLM Hook? Generative AI (GenAI) technology enables rapid iterations, frequent model updates, and new releases every few weeks. Once models are deployed through inference engines, derived applications and services can be rapidly built and used. However, when failure modes are identified or model operation customization requests are made, taking down the model in service for further retraining and redeployment would cause catastrophic disruption to dependent users. Therefore, there is a practical demand for GenAI service providers to implement flexible model configurations and patches “on-the-fly” (i.e., during deployment) for models deployed on inference engines.

We position the vLLM Hook project as a developer kit that supports various use cases requiring the programming of vLLM model internal states. To demonstrate the programmability of different internal states through the vLLM Hook, we developed three examples that address the practical need to capture or modify specific parts of a vLLM model, including attentions, attention heads, and activations.

- **In-model Monitoring** [passive programming]. Use vLLM Hook to capture critical “inference traces” generated during inference. The saved states are then used for model monitoring and misalignment detection. One example is the Attention Tracker [HKR⁺25], which uses selected attention statistics from a transformer-based vLLM model to detect prompt injection attempts. The “in-model safety guardrail” design, enabled by vLLM Hook, differs from the typical “cascaded safety guardrail” design. The latter requires the use of an additional moderation model to inspect the input and output of the deployed model.
- **Model Steering** [active programming]. Use vLLM Hook to modify the internal states of the model to steer its output toward desired behaviors. Although recent model steering methods demonstrate promising inference-time alignment capabilities without retraining the deployed model [MRV⁺], the current vLLM package prevents most model steering methods that modify internal states from being implemented. One example is Activation Steering, which injects a steering vector into some layer of a deployed model for intervention, such as improving the capability in instruction following [SBY⁺25].
- **Selective Retrieval** [passive programming]. Use vLLM Hook to activate a selected set of components within a deployed model for efficient and effective data retrieval. The rationale is to retrieve data embeddings from a deployed model that are relevant to the task of interest. One example is to only activate a selected set of attention heads within a deployed model to improve the re-ranking performance in information retrieval [TLFS25].

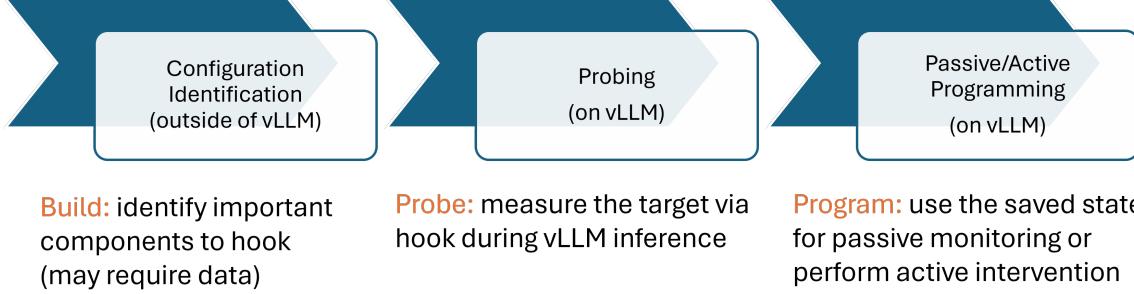


Figure 2: Complete development cycle of vLLM Hook. The three key steps are (1) *Build*: In the upstream development phase, use the model to be deployed on vLLM to design the programming function and identify important internal states to hook. (2) *Probe*: specify the target internal states to hook by creating the Config File. (3) *Program*: use the Config File to program the corresponding vLLM model.

How do we use vLLM Hook? We envision vLLM Hook as an organic innovation engine that will enable the development and deployment of advanced techniques for monitoring and adjusting vLLM models. As a developer kit, vLLM Hook augments the native vLLM project to support programming the internal states of vLLM models. We expect that by open-sourcing vLLM Hook, the community will be able to build valuable assets on deployed AI models for operational management and technical governance. This will also help close the gap between programming internal states during the development phase and deploying such capabilities during the deployment phase.

Finally, we list some best practices of vLLM Hook.

- **Build, Probe, and Program.** An important innovation of vLLM Hook is the inclusion of the configuration file (Config) that specifies where to hook and what to program for the deployed models. Such a Config file is model-specific and task-oriented. Therefore, as illustrated in Figure 2, the complete cycle of vLLM Hook involves three key steps:

1. *Build*: In the upstream development phase, use the model to be deployed on vLLM to design the programming function and identify important internal states to hook. This build step does not require the use of vLLM.
2. *Probe*: Based on the build results in step (1), specify the target internal states to hook by creating the Config File.
3. *Program*: use the Config File to program the corresponding vLLM model.

Through opensourcing and community-driven contributions, these config files can be used by different developers in their own work streams. The Build step is beyond the current scope of the vLLM Hook project.

- **Be Mindful of the Programmability and Usability Trade-off.** Although vLLM Hook allows developers to program deployed models on vLLM, programming too many states or using computationally intensive programming functions can significantly impact the usability of inference engines. We advise developers to consider the trade-off between programmability and utility (e.g., latency of the vLLM Hook model and memory usage for programming) and to build viable configuration files with minimal hooks to internal states.

2 Core Functions of vLLM Hook

vLLM-Hook is built as a modular plugin library that enables users to capture internal model signals (passive programming), intervene during inference (active programming), and optionally analyze the captured signals. We give an illustration of vLLM-Hook orchestration in Figure 3. The core of our framework consists of two abstractions: **worker** and **analyzer**. On a higher level, **worker** defines when and how we program within the vLLM pipeline, and **analyzer** defines the evaluation metrics

$$\text{llm} = \text{HookLLM}$$

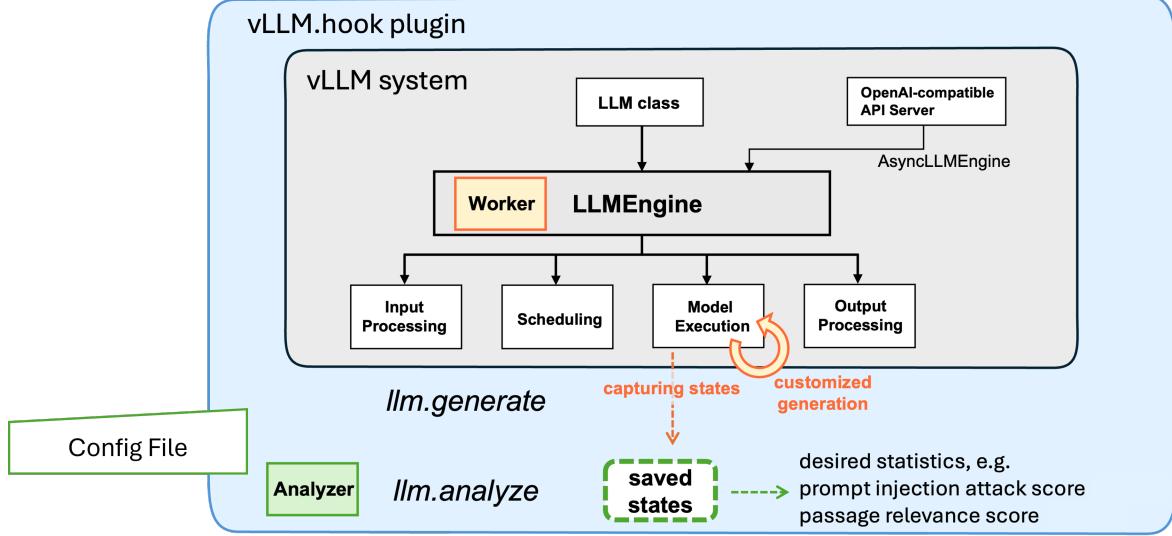


Figure 3: An illustration of vLLM-Hook orchestration. vLLM-Hook is a lightweight wrapper around a native vLLM system and uses `llm=HookLLM` to initialize an LLM instance, `llm.generate` for response generation (model forward pass), and `llm.analyze` for optional in-model analysis

based on the captured signals (if any). Together with **worker** and **analyzer**, a **Config File** is provided to specify the more granular behavior of the **worker** and **analyzer**. For instance, it could include the profile of the important layers and attention heads of a model, the steering vector of a desired model behavior, and the statistic reduction method (take average vs. weighted summation), etc. We give an example of a config file as follows:

```
{
  "model_info": {
    "name": "granite3-8b-attn",
    "model_id": "ibm-granite/granite-3.1-8b-instruct"
  },
  "params": {
    "important_heads": [[6, 9], [7, 20], [8, 1], [8, 13], [8, 14], [8, 15],
    [10, 2], [10, 3], [10, 6], [10, 21], [11, 4], [11, 30], [11, 31], [12, 2],
    [12, 28], [13, 8], [13, 9], [13, 12], [14, 15], [14, 16], [14, 19], [14, 27],
    [15, 6], [15, 7], [15, 20], [15, 23], [16, 12], [16, 14], [16, 16], [17, 7],
    [17, 11], [17, 15], [17, 19], [17, 21], [17, 25], [17, 26], [18, 9],
    [18, 17], [18, 20], [18, 28], [19, 1]]
  },
  "hookq": {
    "hookq_mode": "last_token"
  }
}
```

In the above example, the config file specifies the generating LM under "`model_info`", the '[layer, head]' index of the important heads for the model analysis under "`params`"/"`important_heads`", and indicates we only need the query cache of the very last token through "`hookq`"/"`hookq_mode`". These components are managed through a lightweight registry and a **HookLLM** wrapper class.

During the runtime, once the user calls `llm.generate` in active programming mode, **worker** actively engages in the model execution phase and performs customized generation (e.g., model steering). In passive programming mode, **worker** follows base generation flows and optionally capture internal states (e.g., attentions) during forward passes. After the forward pass has completed, the user can call `llm.analyze` and the **analyzer** will assemble the saved states and compute the desired statistics such

as prompt injection attack score, passage relevance score, etc.

2.1 Worker

Workers are integrated directly into the vLLM runtime and record/modify internal states during inference. They are implemented by subclassing the standard vLLM GPU worker and overriding the `load_model`. After the base model is loaded, the worker installs PyTorch forward hooks on selected modules. We give an example of this step as follows:

```
from vllm.v1.worker.gpu_worker import Worker as V1Worker

class ProbeHookQKWorker(V1Worker):
    def load_model(self, *args, **kwargs):
        r = super().load_model(*args, **kwargs)
        self._install_hooks()
        return r

    def _install_hooks(self):
        ## Configuration via environment variables:
        # self.hook_flag: directory to the flag file indicating hook (de)activation
        # self.hook_dir: directory to write signal files
        # self.run_id_file: file to append run identifiers
        # self.hookq_mode: whether to capture all tokens or just the last token
        self.hook_flag = os.environ.get("VLLM_HOOK_FLAG")
        self.hook_dir = os.environ.get("VLLM_HOOK_DIR")
        self.run_id_file = os.environ.get("VLLM_RUN_ID")
        self.hookq_mode = os.environ.get("VLLM_HOOKQ_MODE", "all_tokens")

        self.layer_to_heads = self._parse_layer_heads()
        self.important_layers = set(self.layer_to_heads.keys())
        model = getattr(self.model_runner, "model", None)
        # ... process additional config files related to the base model
        ...

        def qkv_hook(input, module_name):
            # ... parse q/k tensors by batch
            ...
            cache["qk_cache"][module_name] = {
                'q': q_tokens,
                'k_all': k_all_tokens,
                'layer_num': layer_num
            }
            cache_path = os.path.join(self.hook_dir, f"qk_{run_id}.pt")
            torch.save(cache, cache_path)
        # register hooks on attention modules
        self._hooks = []
        matched = []
        for name, module in model.named_modules():
            if name.endswith(".self_attn.attn") and "Attention" in str(type(module)):
                layer_num = int(name.split('model.layers.')[1].split('.')[0])
                if layer_num in self.important_layers:
                    hook = module.register_forward_hook(
                        lambda m, i, o, n=name: qkv_hook(i, n)
                    )
                    self._hooks.append(hook)
                    matched.append(name)
```

```

def _parse_layer_heads(self) -> Dict[int, List[int]]:
    layer_heads = os.environ.get("VLLM_HOOK_LAYER_HEADS", "")
    # ... parse 'VLLM_HOOK_LAYER_HEADS' env var from string to dict
    # e.g. '0:0,3,6;15:2' → {0:[0,3,6], 15:[2]}
    ...
    ...

def execute_model(self, *args, **kwargs):
    return super().execute_model(*args, **kwargs)

```

2.2 Analyzer

Analyzers are designed to (optionally) conduct analysis based on the saved statistics from the worker orchestration. Examples of analyzers include safety guardrails such as monitoring prompt injection attacks, and attention-based re-rankers that leverage selective attention weights to rank document relevance. Typically, an analyzer starts by retrieving the cache identified by the run ids. Then, it re-assembles the desirable statistics accordingly, e.g. selective attention weights. In the following, we show an example analyzer that uses saved keys and queries to recompute selective attentions for prompt injection attack risk evaluations:

```

class AttntrackerAnalyzer:
    def __init__(self, hook_dir: str, layer_to_heads: Dict[int, list]):
        self.hook_dir = hook_dir
        self.layer_to_heads = layer_to_heads

    def analyze(
        self,
        analyzer_spec: Optional[Dict] = None
    ) -> Optional[Dict]:
        # retrieve cache identified by the run id
        run_id_file = os.environ.get("VLLM_RUN_ID")

        # re-assemble selective attention weights
        attention_weights = self.compute_attention_from_qk(run_id_file)

        # evaluate prompt injection attack risks
        score = self.attn2score(attention_weights, analyzer_spec['input_range'],
                               analyzer_spec['attn_func'])

        return {"score": score}

    def compute_attention_from_qk(self, run_id_file: str) -> Dict[str, Dict]:
        # cache loading
        run_id = open(run_id_file).read().strip()
        cache_path = os.path.join(self.hook_dir, f"qk_{run_id}.pt")
        if not os.path.exists(cache_path):
            raise FileNotFoundError(f"Q/K cache file not found: {cache_path}.")
        cache = torch.load(cache_path, map_location="cpu")
        # ... recompute attentions
        ...

    def attn2score(self, batch_attention: List[Dict[str, Dict]],
                  batch_input_range: List[Tuple[Tuple[int, int], Tuple[int, int]]],
                  attn_func: str = "sum_normalize"
    ) -> float:
        # ... following Attention Tracker and calculate the aggregated attention on
        # the instruction vs the attention over both the instruction and user query
        ...

```

3 Demonstrations of vLLM Hook

vLLM-Hook is an extensible framework that aims to allow selective access to model internals during the inference. We provide demonstrations of vLLM-Hook through a series of Jupyter notebook examples.

- **In-model monitoring of prompt injection attacks**

- TL;DR: Attention Tracker [HKR⁺25] monitors prompt injection attacks via the aggregated attention scores of the important heads on the instruction prompt, also called focus score. Low focus score indicates potential malicious queries.
- Notebook: https://github.com/IBM/vLLM-Hook/blob/main/notebooks/demo_attntracker.ipynb

- **Model steering via activation steering**

- TL;DR: Activation steering allows you to bias the model’s behavior by nudging internal activations in specific directions. In [SBY⁺25], authors focus on instruction following capability and compute the steering vectors as the difference in activations between inputs with and without instructions.
- Notebook: https://github.com/IBM/vLLM-Hook/blob/main/notebooks/demo_actsteer.ipynb

- **Selective attention-based retrieval**

- TL;DR: Core reranker [TLFS25] is an attention-based reranker that leverage attention weights from selected transformer heads to produce document relevance scores.
- Notebook: https://github.com/IBM/vLLM-Hook/blob/main/notebooks/demo_corer.ipynb

4 How to Contribute to vLLM Hook?

We envision vLLM Hook as an organic innovation engine that will enable the development and deployment of advanced techniques for monitoring and adjusting vLLM models. These techniques require the use of programming internal states during inference.

We invite the community to contribute to the vLLM Hook project to expand its core functions, use cases, and demonstrations. Significant contributors will be invited to co-author future vLLM technical reports.

To contribute,

1. **Fork** this repository: <https://github.com/IBM/vLLM-Hook/tree/main>
2. **Create a branch** (`git checkout -b feature/amazing-feature`)
3. **Commit** your changes (`git commit -m 'Add amazing feature'`)
4. **Push** to your branch (`git push origin feature/amazing-feature`)
5. **Open a Pull Request and request approval from the admin**

Guidelines

- Users are encouraged to define new worker/analyzer, but should not modify the core orchestrator `hook_llm`
- Include examples and documentation for new features
- The registry will be updated by the admin

5 Acknowledgment

We thank David Cox for his comments on “Do your techniques work on vLLM?” This package was created as a result. We thank Kush Varshney’s support and feedback. We thank Hendrik Strobelt and Enara Vijil for their feedback on the implementation and testing.

References

- [HKR⁺25] Kuo-Han Hung, Ching-Yun Ko, Ambrish Rawat, I-Hsin Chung, Winston H Hsu, and Pin-Yu Chen. Attention tracker: Detecting prompt injection attacks in llms. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 2309–2322, 2025.
- [KLZ⁺23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.
- [MRV⁺] Erik Miehling, Karthikeyan Natesan Ramamurthy, Praveen Venkateswaran, Ching-Yun Ko, Pierre Dognin, Moninder Singh, Tejaswini Pedapati, Avinash Balakrishnan, Matthew Riemer, Inge Vejsbjerg, Elizabeth M. Daly, and Kush R. Varshney. AI Steerability 360.
- [SBY⁺25] Alessandro Stolfo, Vidhisha Balachandran, Safoora Yousefi, Eric Horvitz, and Besmira Nushi. Improving instruction-following in language models through activation steering. In *The Thirteenth International Conference on Learning Representations*, 2025.
- [TLFS25] Linh Tran, Yulong Li, Radu Florian, and Wei Sun. Contrastive retrieval heads improve attention-based re-ranking. *arXiv preprint arXiv:2510.02219*, 2025.