vLLM : Architectural Deep Dive

Understanding a High Throughput LLM Inference System

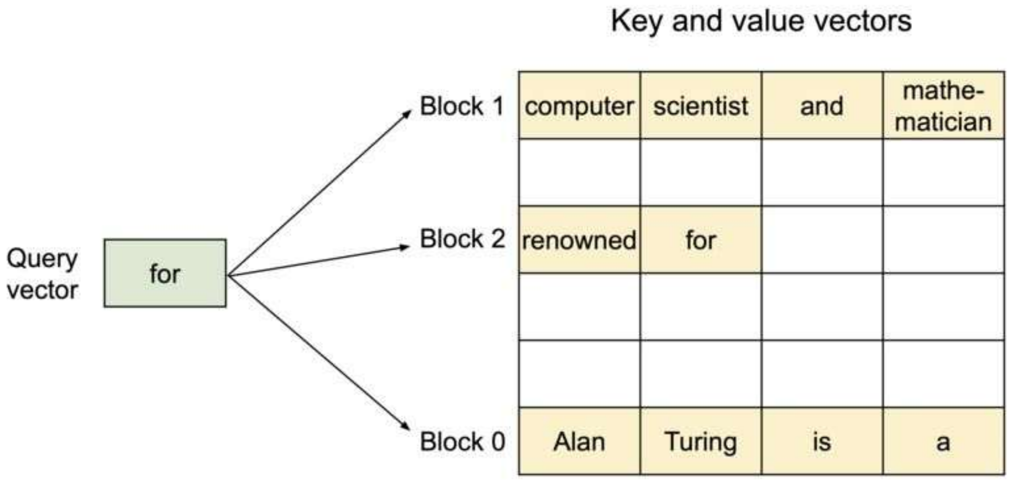
By: **Ayush Satyam**
Software Engineer at Red Hat
GitHub: **@ayushsatyam146**

Why vLLM is awesome!!

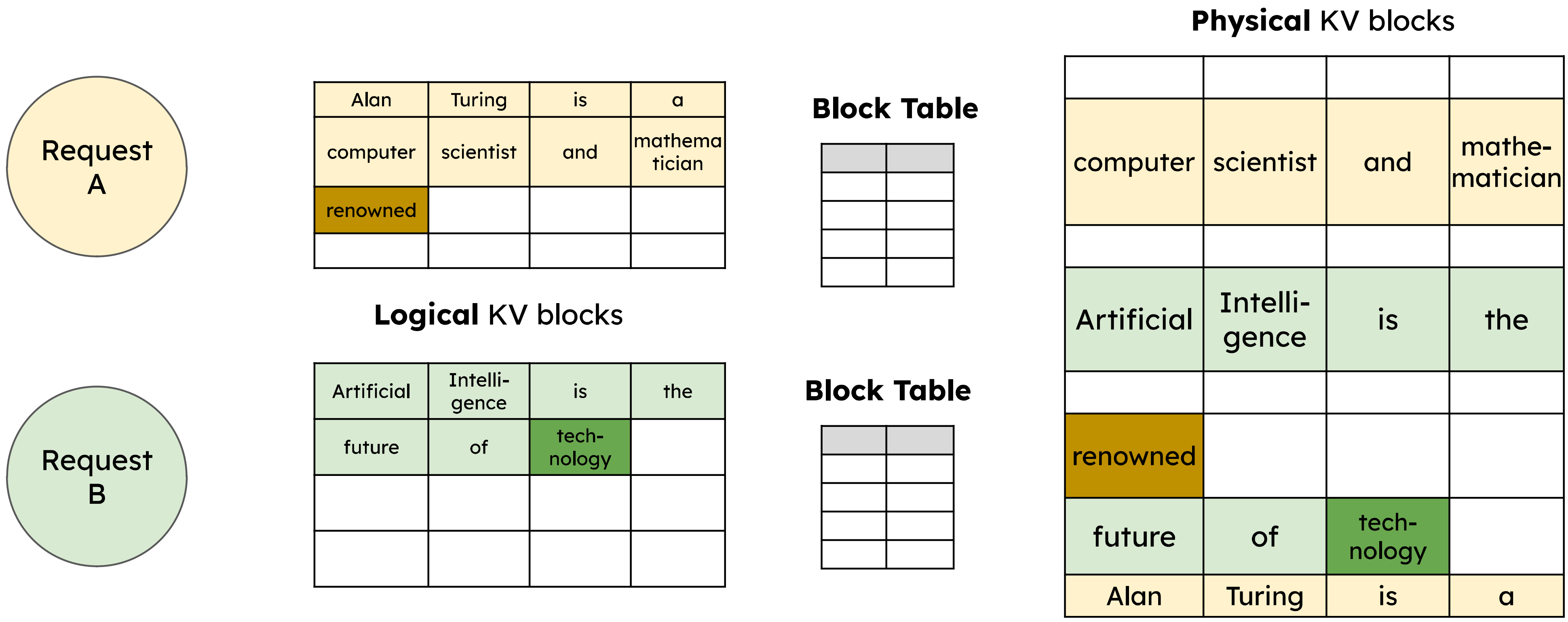
- Paged Attention
- Optimized K-V Caching
- Optimized CUDA kernels
- Speculative Decoding
- Chunked Prefill and many more

Paged Attention

An attention algorithm that allows for storing continuous keys and values in non-contiguous memory space.



Managing KV cache: Lessons from OS



Alan	Turing	is	a
computer	scientist	and	mathematician
renowned			

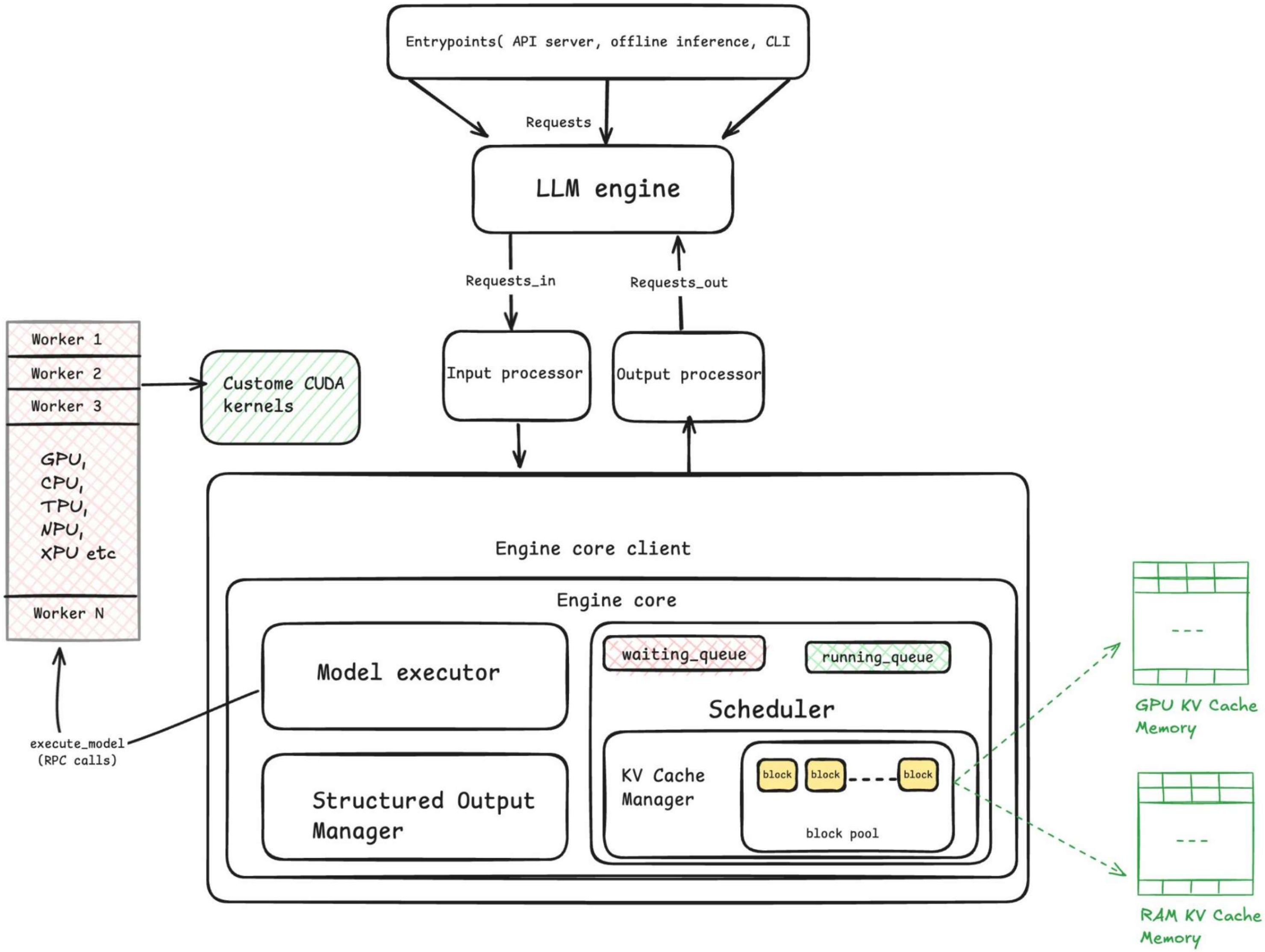
Artificial	Intelligence	is	the
future	of	technology	

computer	scientist	and	mathematician
Artificial	Intelligence	is	the
renowned			
future	of	technology	
Alan	Turing	is	a

10k feet view of the system

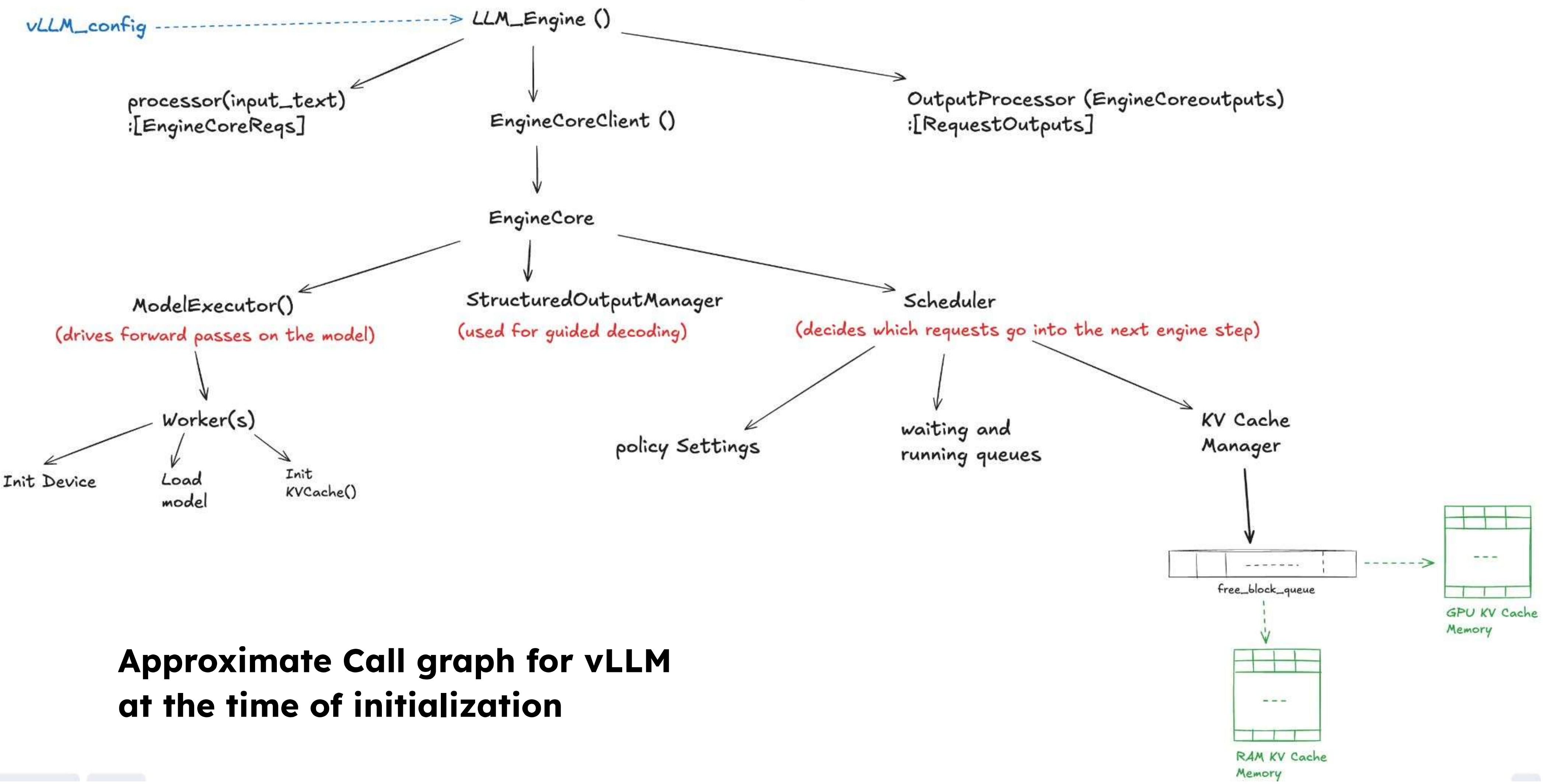
What are the core components of the system?

- Tokenizer
- LLM Engine
- KV cache manager
- Worker
- Scheduler
- Model executor….
- … Let’s **SEE** in detail


Entrypoints(API server, offline inference, CLI
Requests
LLM engine
Requests_in
Requests_out
Input processor
Output processor
Worker 1
Worker 2
Worker 3
GPU,
CPU,
TPU,
NPU,
XPU etc
Worker N
Custome CUDA kernels
execute_model
(RPC calls)
Engine core client
Engine core
Model executor
Structured Output Manager
waiting_queue
running_queue
Scheduler
KV Cache Manager
block
block
block
block pool

GPU KV Cache Memory

RAM KV Cache Memory



vLLM_config

LLM_Engine ()

processor(input_text)
:[EngineCoreReqs]

EngineCoreClient ()

OutputProcessor (EngineCoreoutputs)
:[RequestOutputs]

EngineCore

ModelExecutor()
(drives forward passes on the model)

StructuredOutputManager
(used for guided decoding)

Scheduler
(decides which requests go into the next engine step)

Worker(s)

Init Device

Load
model

Init
KVCache()

policy Settings

waiting and
running queues

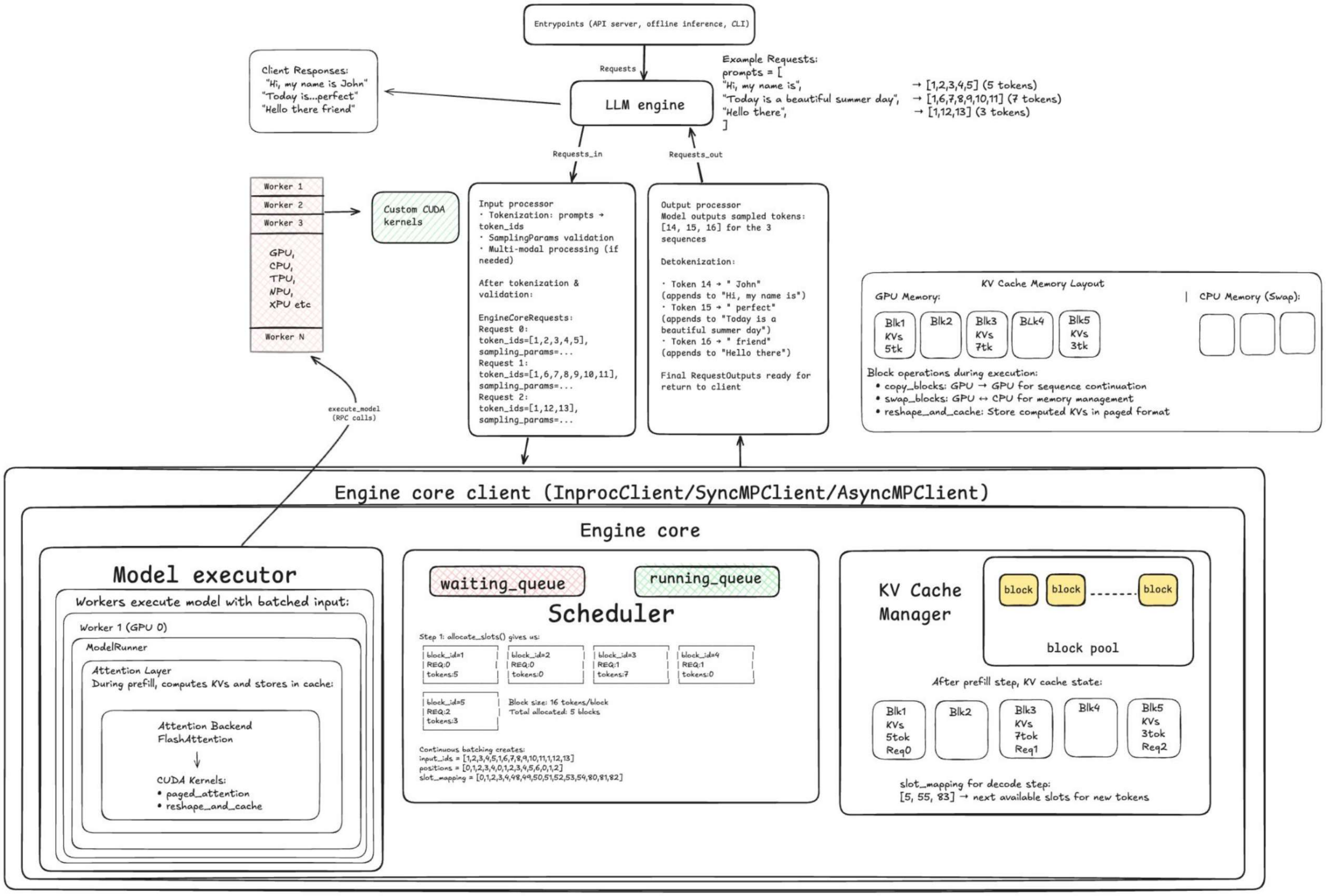
KV Cache
Manager

free_block_queue

GPU KV Cache
Memory

RAM KV Cache
Memory

**Approximate Call graph for vLLM
at the time of initialization**



Client Responses:
"Hi, my name is John"
"Today is...perfect"
"Hello there friend"

```
Example Requests:
prompts = [
"Hi, my name is",                    → [1,2,3,4,5] (5 tokens)
"Today is a beautiful summer day",   → [1,6,7,8,9,10,11] (7 tokens)
"Hello there",                       → [1,12,13] (3 tokens)
]
```

```
Input processor
· Tokenization: prompts → token_ids
· SamplingParams validation
· Multi-modal processing (if needed)

After tokenization & validation:

EngineCoreRequests:
Request 0:
token_ids=[1,2,3,4,5],
sampling_params=...
Request 1:
token_ids=[1,6,7,8,9,10,11],
sampling_params=...
Request 2:
token_ids=[1,12,13],
sampling_params=...
```

```
Output processor
Model outputs sampled tokens:
[14, 15, 16] for the 3 sequences

Detokenization:

· Token 14 → " John"
(appends to "Hi, my name is")
· Token 15 → " perfect"
(appends to "Today is a beautiful summer day")
· Token 16 → " friend"
(appends to "Hello there")

Final RequestOutputs ready for return to client
```

KV Cache Memory Layout

GPU Memory: | CPU Memory (Swap):

Blk1 KVs 5tk	Blk2	Blk3 KVs 7tk	BLk4	Blk5 KVs 3tk

Block operations during execution:
- copy_blocks: GPU → GPU for sequence continuation
- swap_blocks: GPU ↔ CPU for memory management
- reshape_and_cache: Store computed KVs in paged format

Workers execute model with batched input:

Worker 1 (GPU 0)

ModelRunner

Attention Layer
During prefill, computes KVs and stores in cache:

Attention Backend
FlashAttention
↓
CUDA Kernels:
- paged_attention
- reshape_and_cache

Step 1: allocate_slots() gives us:

block_id=1 REQ:0 tokens:5	block_id=2 REQ:0 tokens:0	block_id=3 REQ:1 tokens:7	block_id=4 REQ:1 tokens:0
block_id=5 REQ:2 tokens:3			

Block size: 16 tokens/block
Total allocated: 5 blocks

```
Continuous batching creates:
input_ids = [1,2,3,4,5,1,6,7,8,9,10,11,1,12,13]
positions = [0,1,2,3,4,0,1,2,3,4,5,6,0,1,2]
slot_mapping = [0,1,2,3,4,48,49,50,51,52,53,54,80,81,82]
```

After prefill step, KV cache state:

Blk1 KVs 5tok Req0	Blk2	Blk3 KVs 7tok Req1	Blk4	Blk5 KVs 3tok Req2

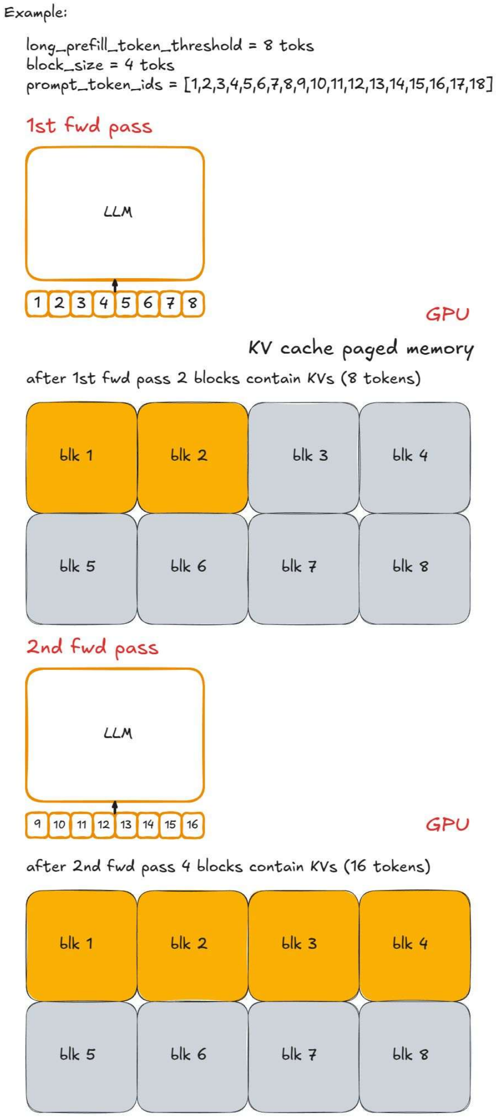
slot_mapping for decode step:
[5, 55, 83] → next available slots for new tokens

Let’s see some Advanced features

- Chunked Prefill
- Prefix Caching
- Guided decoding (FSM)
- Disaggregated PD
- and more…

Chunked Prefill

Chunked Prefill is a technique for handling long prompts by splitting their prefill step into smaller chunks



Example:

```
long_prefill_token_threshold = 8 toks
block_size = 4 toks
prompt_token_ids = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18]
```

1st fwd pass

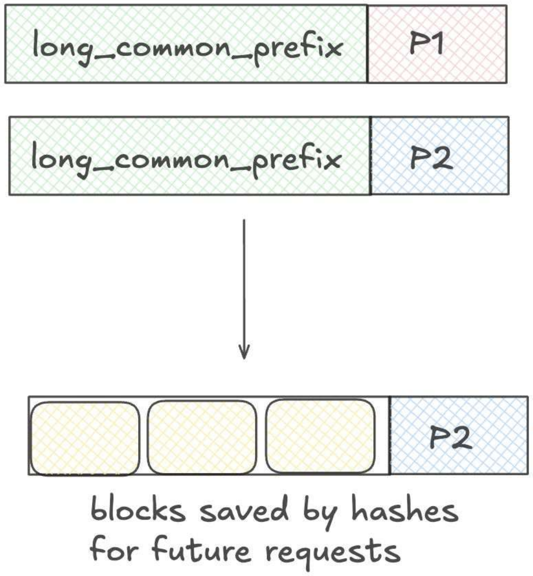
after 1st fwd pass 2 blocks contain KVs (8 tokens)

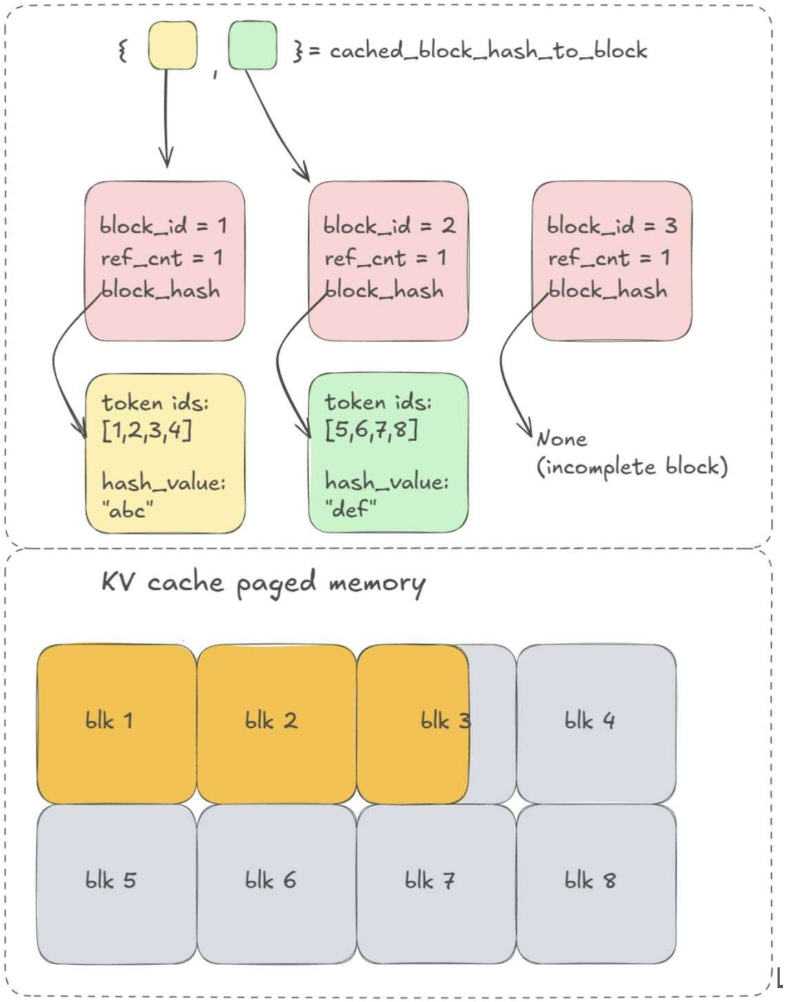
2nd fwd pass

after 2nd fwd pass 4 blocks contain KVs (16 tokens)

Prefix Caching

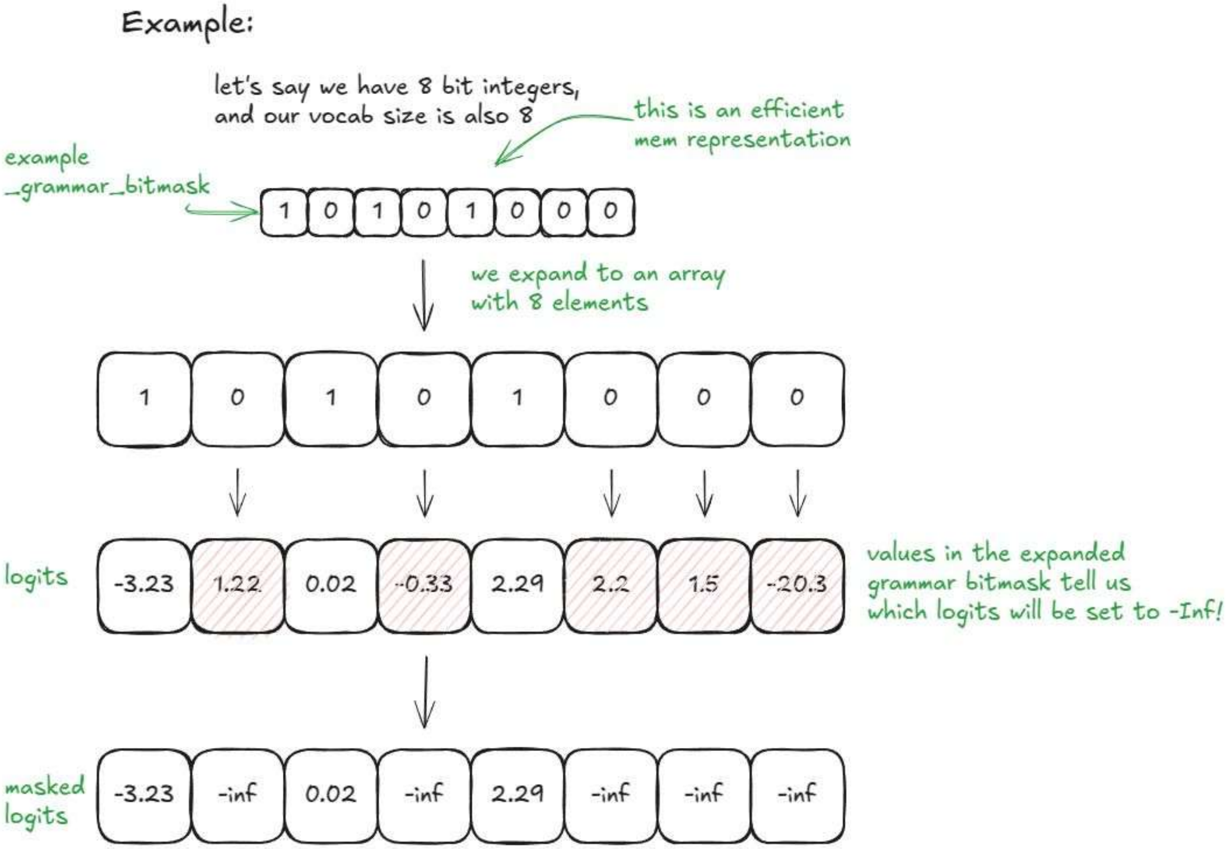
Prefix Caching avoids recomputing tokens that multiple prompts share at the beginning - hence prefix.





Guided Decoding

Guided Decoding is a technique where, at each decoding step, the logits are constrained by a grammar-based finite state machine.



Example:

let's say we have 8 bit integers, and our vocab size is also 8

this is an efficient mem representation

example _grammar_bitmask

1 0 1 0 1 0 0 0

we expand to an array with 8 elements

1 0 1 0 1 0 0 0

logits

-3.23 1.22 0.02 -0.33 2.29 2.2 1.5 -20.3

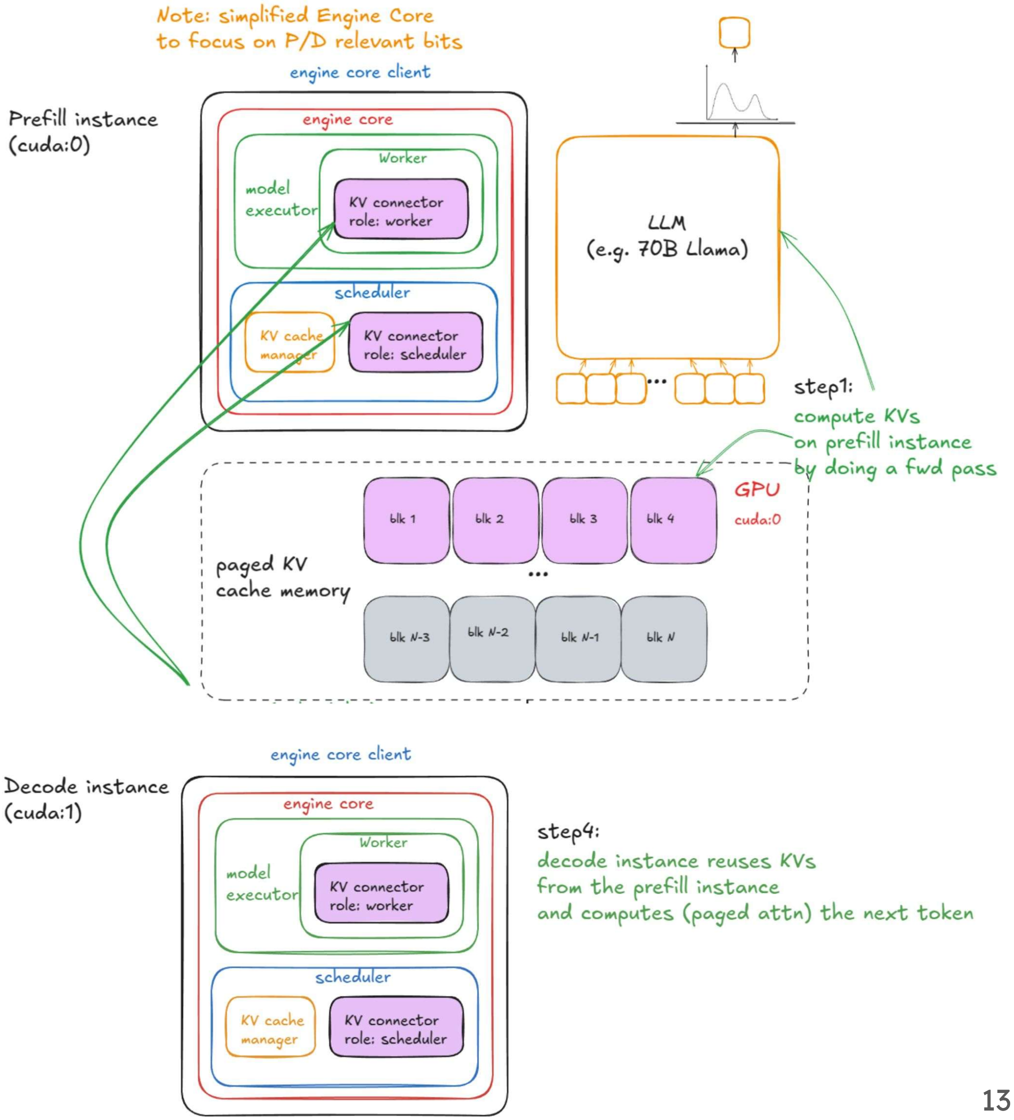
values in the expanded grammar bitmask tell us which logits will be set to -Inf!

masked logits

-3.23 -inf 0.02 -inf 2.29 -inf -inf -inf

Disaggregated PD

Prefill and decode have very different performance profiles (compute-bound vs. memory-bandwidth-bound), so separating their execution is a sensible design. It gives tighter control over latency — both TFTT (time-to-first-token) and ITL (inter-token latency



Note: simplified Engine Core to focus on P/D relevant bits

step1: compute KVs on prefill instance by doing a fwd pass

step4: decode instance reuses KVs from the prefill instance and computes (paged attn) the next token

References:

- Anatomy of vLLM Blog
- vLLM codebase (Must read!)
- Talk by WooSuk Kwon & Zhuohan Li
- Modal notebook here
- vLLM paper here

THANK YOU

Reach out to me on:

@ayushsatyam146

@ayushsatyam146