Request Flow Summary

End-to-End Trace Through the System

Generated: February 08, 2026

1. Complete Request Flow

This diagram shows the complete journey of a request through the system, from HTTP input to streaming output:

Step	Component	Description
1	HTTP Request	FastAPI receives incoming request
2	Frontend	Tokenizes text into input_ids
3	Backend	Receives UserMsg with input_ids
4	PrefillManager	Adds request to pending_list queue
5	Scheduler	Creates Batch with Req objects
6	Engine	Executes forward pass (prefill/decode)
7	Sampler	Generates next_tokens from logits
8	Detokenizer	Converts tokens back to text
9	Frontend	Creates UserReply with incremental_output
10	HTTP Response	Streams response back to client

2. Key Object Transformations

The request object transforms through multiple types as it flows through the system:

```
GenerateRequest (HTTP)
  ↓
TokenizeMsg (frontend)
  ↓
UserMsg (backend, has input_ids)
  ↓
PendingReq (waiting in queue)
  ↓
Req/ChunkedReq (scheduled for execution)
  ↓
Batch (grouped for GPU)
```

```
    ↓
ForwardOutput (next tokens)
    ↓
DetokenizeMsg (token to text)
    ↓
UserReply (final response)
```

Object Type	Key Fields	Purpose
GenerateRequest	prompt, max_tokens, temperature	HTTP API request
TokenizeMsg	uid, text, sampling_params	Tokenization request
UserMsg	uid, input_ids, sampling_params	Backend processing
PendingReq	uid, input_ids, chunked_req	Queued request
Req	input_ids, output_len, cache_handle	Scheduled request
Batch	reqs[], phase, input_ids	GPU batch processing
ForwardOutput	next_tokens_gpu, next_tokens_cpu	Generated tokens
DetokenizeMsg	uid, next_token, finished	Token to detokenize
UserReply	uid, incremental_output, finished	Streaming response

3. Request State Transitions

Each request goes through distinct states during its lifecycle:

```
WAITING: PendingReq in pending_list
    ↓
PREFILL: Req in Batch(phase="prefill")
    ↓
DECODE: Req in Batch(phase="decode")
    ↓
FINISHED: finished=True in DetokenizeMsg/UserReply
```

State	Location	Description
WAITING	PrefillManager.pending_list	Request queued, waiting for GPU resources
PREFILL	Batch with phase='prefill'	Computing KV cache for input tokens
DECODE	Batch with phase='decode'	Generating output tokens one by one
FINISHED	UserReply.finished=True	Generation complete, response sent

4. The 'yield' Pattern in Streaming

The system uses Python generators with 'yield' to enable token-by-token streaming. This is why you see tokens appear one-by-one instead of waiting for the complete response.

wait_for_ack function:

```
# In wait_for_ack:
for ack in pending:
    yield ack  # Returns UserReply one by one

# Each yield pauses execution and returns one item
# Next call resumes from where it left off
```

stream_generate function:

```
# In stream_generate:
async for ack in self.wait_for_ack(uid):
    yield f"data: {ack.incremental_output}\n"
    # Each token streamed immediately!
    if ack.finished:
        break
```

Aspect	Without yield	With yield
Return behavior	Returns all at once	Returns one item at a time
Memory usage	Holds entire result in memory	Processes items on-demand
User experience	Wait for complete response	See tokens as they generate

Network traffic	Single large payload	Multiple small chunks (SSE)

5. Key Request Object Fields

Important fields tracked throughout the request lifecycle:

Field	Type	Description
input_ids	torch.Tensor	Tokenized input (e.g., [101, 2023, 2003, ...])
output_ids	torch.Tensor	Generated tokens appended to input (grows during decode)
num_computed_tokens	int	Tracks KV cache length (how many tokens cached)
sampling_params	SamplingParams	Controls temperature, top_p, max_tokens, etc.
cached_len	int	Number of tokens with computed KV cache
output_len	int	Number of output tokens generated so far
table_idx	int	Index in attention table for this request
cache_handle	BaseCacheHandle	Handle to KV cache storage

6. Complete Code Flow Example

Here's how the key functions connect in the actual codebase:

```
# 1. HTTP endpoint receives request
@app.post("/generate")
async def generate(req: GenerateRequest):
    uid = state.add_request(req)
    return StreamingResponse(state.stream_generate(uid))

# 2. Frontend tokenizes and sends to backend
tokenize_msg = TokenizeMsg(uid=uid, text=req.prompt, ...)
backend_msg = UserMsg(uid=uid, input_ids=tokens, ...)

# 3. Backend adds to pending queue
def _process_one_msg(self, msg: UserMsg):
    self.prefill_manager.add_one_req(msg)

# 4. Scheduler creates batch
batch = self.prefill_manager.schedule_next_batch()

# 5. Engine forward pass
forward_output = self.engine.forward_batch(batch)

# 6. Detokenize and reply
detokenize_msg = DetokenizeMsg(uid=uid, next_token=token)
user_reply = UserReply(uid=uid, incremental_output=text)

# 7. Stream response
async for ack in self.wait_for_ack(uid):
    yield f"data: {ack.incremental_output}\n"
```

Summary

The request flow demonstrates a well-architected pipeline that efficiently handles LLM inference with streaming responses. Key design patterns include:

• **Asynchronous Processing:** Non-blocking I/O for concurrent request handling

• **Batch Processing:** Grouping requests for efficient GPU utilization

• **Streaming Output:** Using generators (yield) for real-time token delivery

• **State Management:** Clear state transitions (WAITING → PREFILL → DECODE → FINISHED)

• **Type Safety:** Strongly typed messages at each pipeline stage

• **Resource Management:** KV cache handles and table management for memory efficiency