Conversation with Gemini - Sequence of memory Events during ComfyUI execution

win11. comfyui. explain to me the sequence of events ... disk ---> cpu --> reg memory ---> vid memory ... of the files involved in executing this json ... then again for the second iteration, what is differnet (things already in mem, etc)

-
- 2 successful queries

The execution of your ComfyUI workflow involves a sophisticated data flow between your computer's storage and memory components.

Here is an explanation of the sequence of events from disk to video memory (VRAM) for the first and second image generation iterations .

🚀 First Iteration (Cold Start)

This sequence is the **slowest** because all necessary files, especially the large model checkpoint, must be loaded from the **Disk** (SSD/HDD) into memory.

Step	ComfyUI Node	Data Flow & Memory Location
1. Initialization	(N/A)	The **CPU** executes the ComfyUI Python script (in **RAM**) and reads the workflow JSON to identify required files.
2. Model Loading	**CheckpointLoaderSimple** (Node 1)	The large checkpoint file (`flux.1_schnell...safetensors`) moves from **Disk RAM**. The heavy components (U-Net, VAE, CLIP) are then transferred from **RAM VRAM** (Video Memory) on the GPU.
3. Prompt Encoding	**CLIPTextEncode** (Nodes 2 & 3)	The text prompts ("a beautiful cabin...", etc.) are processed by the **CLIP** model (now in **VRAM**) to create **Conditioning** tensors, which are stored in **VRAM**.
4. Latent Setup	**EmptyLatentImage** (Node 7)	A 1024x1024 tensor of random noise, the initial **Latent Image**, is created and placed entirely in **VRAM**.
5. Sampling	**KSampler** (Node 4)	The **CPU** orchestrates the **GPU** to run the **U-Net model** (in VRAM) repeatedly. This heavy

Step	ComfyUI Node	Data Flow & Memory Location
		computation loop runs entirely within the **VRAM** and GPU cores, consuming the Latent Image and Conditioning.
6. Decoding	**VAEDecode** (Node 5)	The final **Latent Image** is processed by the **VAE Decoder** (in VRAM) to convert it into a standard pixel **Image** tensor, which temporarily stays in **VRAM**.
7. Saving	**SaveImage** (Node 6)	The final pixel **Image** is transferred **VRAM RAM**. The **CPU** compresses the image and writes the final file to the **Disk**.

⚡ Second Iteration (Warm Start)

The second iteration is dramatically faster because the largest file transfers from **Disk** are skipped. The key difference is that the large models remain in **VRAM**.

Component	First Iteration Action	Second Iteration Action	Difference
Checkpoint Model	Disk RAM VRAM	**Already in VRAM**	**Massive speed gain.** Skips the largest I/O (Input/Output) operation.
CLIP Encoder	Disk RAM VRAM	**Already in VRAM**	No disk access needed for encoding.
Prompts/Conditioning	RAM VRAM	The **new/same prompts** are encoded using the CLIP model already in **VRAM**. New Conditioning tensors replace old ones in VRAM.	Negligible speed difference.
Latent Image	Created in VRAM	A **new Latent Image** (with a new random seed, as per your JSON) is created in **VRAM**.	Process is the same, but the content is different.
KSampler & VAE	VRAM Computation	**VRAM Computation**	The computational process is identical, just using the new latent image.
Save Image	VRAM RAM Disk	VRAM RAM **Disk**	A new file is written to the disk.

The most significant difference is that the multi-gigabyte **Model Loading** step is **eliminated**, as the model is persistent in the **VRAM** as long as ComfyUI continues running.

This YouTube video offers a good visual walkthrough of the ComfyUI interface and workflow structure, which aligns with the steps described in the JSON file: [ComfyUI Basic Workflow for Beginners](#).

the vram is 12gig, the checkpoint is 16gig, the clip is another 242meg ... how does that change things ...

```
FETCH ComfyRegistry Data: 65/108

Cannot connect to comfyregistry.

FETCH DATA from: https://raw.githubusercontent.com/ltdrdata/ComfyUI-Manager/main/custom-node-list.json [DONE]

[ComfyUI-Manager] All startup tasks have been completed.

got prompt

model weight dtype torch.float8_e4m3fn, manual cast: torch.bfloat16

model_type FLOW

Using pytorch attention in VAE

Using pytorch attention in VAE

VAE load device: cuda:0, offload device: cpu, dtype: torch.bfloat16

Requested to load FluxClipModel_

loaded completely; 95367431640625005117571072.00 MB usable, 4777.54 MB loaded, full load: True

CLIP/text encoder model load device: cpu, offload device: cpu, current: cpu, dtype: torch.float16

Requested to load Flux

loaded partially; 9612.19 MB usable, 9611.75 MB loaded, 1728.56 MB offloaded, lowvram patches: 0

100%|██████████████████████████████████████████████████| 4/4 [00:20<00:00,  5.10s/it]

Requested to load AutoencodingEngine

loaded partially: 5932.54 MB loaded, lowvram patches: 0
loaded completely; 298.33 MB usable, 159.87 MB loaded, full load: True

Prompt executed in 00:15:40

got prompt

Prompt executed in 0.03 seconds

got prompt

loaded partially; 9569.07 MB usable, 9563.73 MB loaded, 1776.58 MB offloaded, lowvram
patches: 0

100%|████████████████████████████████████████| 4/4 [00:12<00:00,  3.08s/it]

Requested to load AutoencodingEngine

loaded partially: 5932.54 MB loaded, lowvram patches: 0

loaded completely; 296.33 MB usable, 159.87 MB loaded, full load: True

Prompt executed in 18.08 seconds
```

The limited VRAM (12 GB) compared to your model's size (16 GB checkpoint) significantly changes the data flow by forcing ComfyUI to use **Offloading** (moving data to slower RAM) and **Paging** (moving it back to VRAM) to execute the workflow.

This memory management technique, sometimes referred to as "lowvram" mode, is confirmed by the log entries **`loaded partially`** and **`offload device: cpu`**.

1. First Iteration: Cold Start (Full Load and Offload)

The first execution is the slowest because the system must initially load and distribute all the model's parts, encountering the memory bottleneck immediately.

Component	Log Evidence	Memory Flow (Disk RAM VRAM)	Impact on Speed
Checkpoint (16 GB)	`loaded partially; ... 9611.75 MB loaded, 1728.56 MB offloaded`	1. Full 16GB file moves from **Disk RAM**. 2. The critical **U-Net** portion is loaded to **VRAM** (9.6 GB). 3. The remaining model weights (1.7 GB) are **Offloaded** to **RAM** (`offload device: cpu`).	Very slow due to the large initial **Disk-to-RAM** transfer.
VAE	`loaded completely; ... full load: True`	Loaded directly and completely to **VRAM** (4.7 GB).	Fast access once loaded.

Component	Log Evidence	Memory Flow (Disk RAM VRAM)	Impact on Speed
CLIP Encoder	`CLIP/text encoder model load device: cpu, current: cpu`	Loaded entirely to **CPU RAM** (242 MB). It does not reside in VRAM.	Encoding is performed by the CPU, which is slower than a VRAM-native operation.
Sampling (KSampler)	`Prompt executed in 00:15:40` (Very Long)	During the sampling loop, the GPU frequently needs the model weights that were **Offloaded** to RAM. The system must rapidly **Page** these small pieces of data back and forth between **RAM and VRAM**, slowing down the entire process.	The long execution time (15 minutes 40 seconds) is primarily due to the repeated **Paging** between RAM and VRAM.

---

2. Second Iteration: Warm Start (Paging vs. Reloading)

The key difference is that the initial model transfer from **Disk RAM** is skipped, but the **Paging** between RAM and VRAM still occurs for the over-sized U-Net.

Component	Log Evidence	Memory State & Flow	Why it is Faster/Slower
Checkpoint	`got prompt... Prompt executed in 0.03 seconds` (Fast)	The main checkpoint file is **already in RAM** and distributed between RAM/VRAM from the first run. The initial **Disk I/O is skipped**.	The **0.03 second** timing indicates that the massive Disk RAM load is avoided.
U-Net	`loaded partially; ... 1776.58 MB offloaded`	The system *re-activates* the model components from RAM/VRAM. The model remains in its **partially loaded/offloaded** state. The 1.7 GB of weights remain **offloaded to RAM** but are ready to be paged in.	The **Sampling** process (the actual generation) is still slow (18.08 seconds) because the necessary **RAM-to-VRAM Paging** must still occur every time the sampler needs the offloaded parts of the model.
VAE & CLIP	`loaded completely; ... full load: True`	These components are **already loaded** to their respective memory (VRAM for VAE, RAM for CLIP) and are reused.	The memory is "hot," meaning the data is already in fast access memory, speeding up their reuse.

In summary, the first run is slow because of the **Disk Load** and the **VRAM Paging**. The second run skips the **Disk Load**, but the main model must still be broken up, meaning the expensive **RAM-to-VRAM Paging** operation remains, resulting in a generation time of 18 seconds, which is still much longer than what would be possible with sufficient VRAM.

it is presently using a sata hdd, i am installing an nvme soon, how does that change things

The upgrade from a **SATA HDD** (Hard Disk Drive) to an **NVMe SSD** will provide a **massive, immediately noticeable speed increase** during the initial setup of your workflow, but it will **not** fundamentally change the slower second-iteration generation time.

Here is a breakdown of how the data flow changes:

1. Initial Model Loading (The Cold Start Boost)

This is where you will see the largest benefit.

Component	HDD (Current)	NVMe SSD (New)	Impact
Speed	50–150 MB/s (Sequential Read)	1,500–7,000+ MB/s (Sequential Read)	**10x to 50x faster.**
Disk RAM	The 16GB model file takes *minutes* (5 to 15 minutes, as suggested for large Flux models on HDDs) to transfer from **Disk RAM**. This is the main reason your first run takes 15 minutes.	The same 16GB file will load from **Disk RAM** in **seconds** (e.g., 10 to 60 seconds).	Your entire first execution time will be dramatically reduced by removing this bottleneck.

The move to NVMe effectively solves the slowest part of your **First Iteration** sequence: the waiting time for the massive checkpoint file to move from slow physical storage into system memory (RAM).

---

2. Generation & Paging (The Bottleneck Remains)

Once the model is loaded, the process of generating the image (the KSampler loop) relies on the GPU, VRAM, and System RAM. This sequence is limited by the **PCIe bus**, not the speed of the disk where the model file is stored.

Memory Interaction	Limiting Factor	NVMe Impact
VRAM RAM Paging	**PCIe Bus Bandwidth.** Since your 16GB model is 4GB larger than your 12GB VRAM, the system must constantly shuffle (page) data back and forth between VRAM and RAM.	**None.** The speed of this transfer is limited by the **PCIe connection** between the GPU and the CPU/RAM, not the drive's speed. Your generation time (e.g., 18 seconds) will likely **stay the same**.

Memory Interaction	Limiting Factor	NVMe Impact
System RAM Disk (Page File)	**Storage Speed**	**High Impact.** If your system runs low on **System RAM** while loading or processing, it uses the hard drive as a **Page File** (virtual memory). If this Page File is currently on the slow **SATA HDD**, the system is severely throttled. Moving the Page File to the **NVMe SSD** will make this emergency transfer much faster, preventing a potential system crash or an even more significant slowdown if RAM is exhausted.

In summary:

- **You will save:** Huge amounts of time whenever you switch models, restart ComfyUI, or load the workflow for the first time.
- **You will not save:** Time on the core image generation loop (the KSampler steps), as that time is spent moving data between VRAM and RAM, which is limited by the connection speed on your motherboard.Gemini can make mistakes, including about people, so double-check it. Your privacy & Gemini Opens in a new window