The RYS Build Story

A field-note style guide to RYS layer duplication, hybrid metadata, and why a model can load while still being wrong.

The model loaded. That was the trap.

The cold-open problem: a model can load cleanly while the copied layer story is still wrong.

The first bad build did not fail in the loud, helpful way. It loaded. The files looked plausible. Then generation started drifting into degenerate repetition, as if the model had found a groove and could not climb back out of it.

That is the trap with checkpoint surgery. A runtime can accept the file structure before the computation is actually coherent.

RYS starts with a tempting idea: take a trained model, select a slice of layers from the middle, copy that slice, and paste the copy back into the stack.

Now the model is deeper, but the new depth did not begin as random weights. It began as computation the model already knew how to use. That is the fun part.

What RYS is actually changing

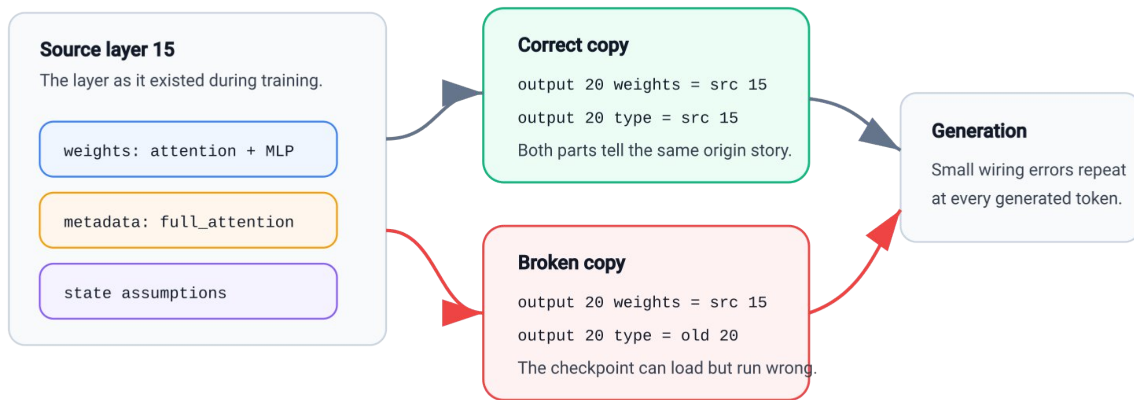
The useful definition is simple: RYS is a checkpoint transformation, not a training method by itself.

It takes a window of already-trained transformer layers, duplicates that window, and inserts the duplicate back into the layer stack. It does not ask the model to learn new layers from scratch. It starts those new layers from a computation pattern the model already had.

That is why it can be useful. A copied middle layer is not random noise; it is a trained transformation that already fits the surrounding model. The hope is that the extra pass through a familiar transformation gives the model more useful depth without beginning from zero.

The risk is that a copied layer is not just a pile of numbers. It also carries assumptions about where it sits, what kind of block it is, and which execution path should run it.

Think of a layer like a workstation in an assembly line. The tensors are the tools at the station. The config metadata is the instruction sheet that tells the runtime how to use those tools. If you copy the tools from station 15 but leave the instruction sheet from station 20, the station may still exist, but the work being done there is no longer the work those tools were trained for.

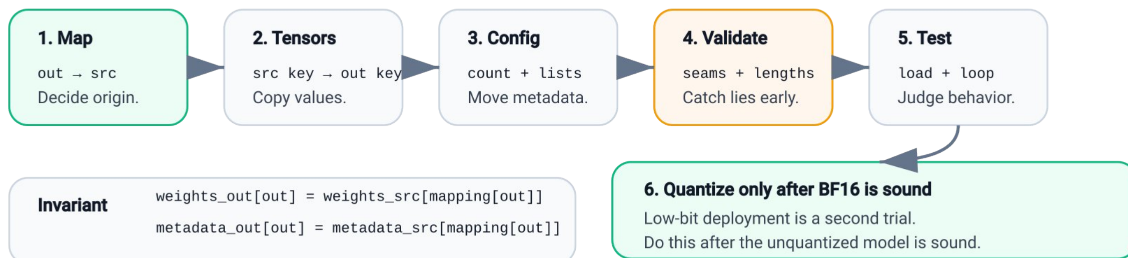


A copied layer has weights and execution assumptions. Move both.

The rule that kept the build sane was simple:

> **Every output layer must copy its weights and its execution metadata from the same source layer.**

Everything else is just a way of keeping that one rule true.



The build route: map, tensors, config, validation, testing, quantization.

Before cutting anything, map the machine

A transformer language model is a stack. Tokens become hidden states. Those hidden states pass through layer 0, then layer 1, then layer 2, and so on until the model predicts the next token.

The simple path is:

> tokens -> embedding -> layer 0 -> layer 1 -> ... -> layer N-1 -> logits

Nothing dramatic is happening here yet. This is just the conveyor belt. RYS changes the length of that conveyor belt, so before touching tensors, it helps to name the pieces clearly.

> **The five objects on the bench are weights, tensor names, config, per-layer metadata, and the converted artifact.**

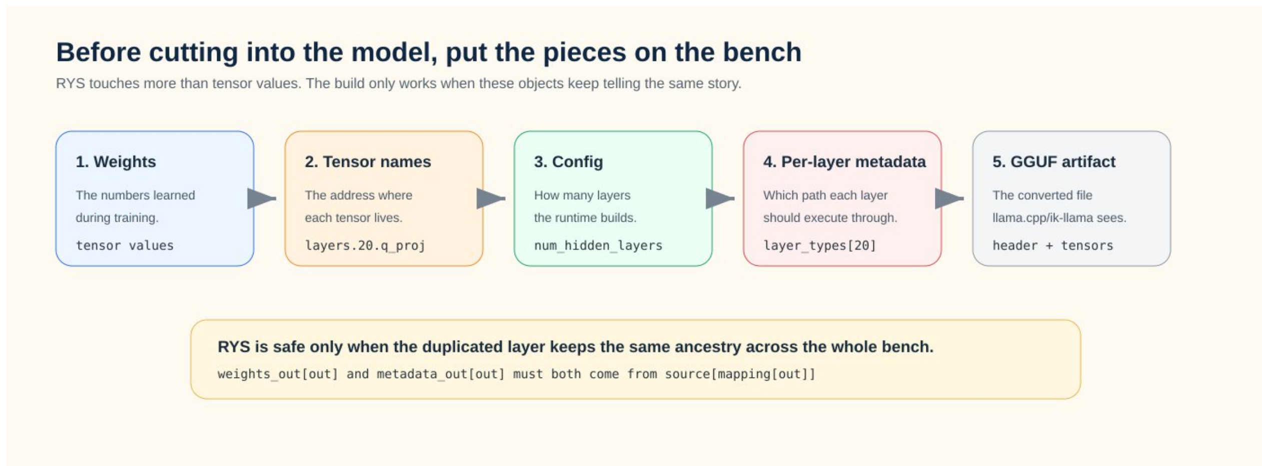
Weights are the learned numbers. In Hugging Face format, they usually live in safetensors shards plus an index file.

Tensor names are addresses. A key like model.layers.15.self_attn.q_proj.weight tells the loader which saved tensor belongs to which layer and module.

Config is the JSON description that tells a runtime how many layers to build and what kind of model it is building.

Per-layer metadata is where entry 15 describes layer 15, entry 16 describes layer 16, and so on. In Qwen3.6 hybrid models, text_config.layer_types is the important one because it tells the runtime which attention path each layer should use.

GGUF is a converted llama.cpp-style artifact. It has its own metadata header and tensor table. It is not the same thing as serving the HF safetensors folder directly.



Before editing a checkpoint, keep the five objects separate: weights, names, config, per-layer metadata, and the converted artifact.

That separation matters because RYS touches all of these pieces. If only the tensors move, the checkpoint can become structurally valid and semantically wrong.

The map is the story

Before copying tensors, build the output-to-source map. This map answers one question:

mapping[out_layer] = source_layer

If mapping[20] = 15, then output layer 20 is a copy of source layer 15. That does not mean inference jumps backward at runtime. The output stack still runs forward: layer 19, then layer 20, then layer 21. The map is ancestry. It tells us where each output layer came from when we built the checkpoint.

The map is ancestry, not runtime control. It tells the exporter what to copy into each output slot; it does not tell generation to travel backward through the network.

Runtime order and source provenance are different things

The generated model still executes from left to right. The map only says where each output layer was copied from.

Runtime execution order

The model never jumps backward while generating.

Source provenance map

mapping[20] = 15
Copied from source layer 15.

The map is ancestry, not runtime control.
Use it to build tensor names and metadata.
Do not read it as an inference-time jump.

Takeaway: the model still runs forward; the map only records where each output layer came from.

That one distinction prevents a lot of confusion.

Start with a tiny model:

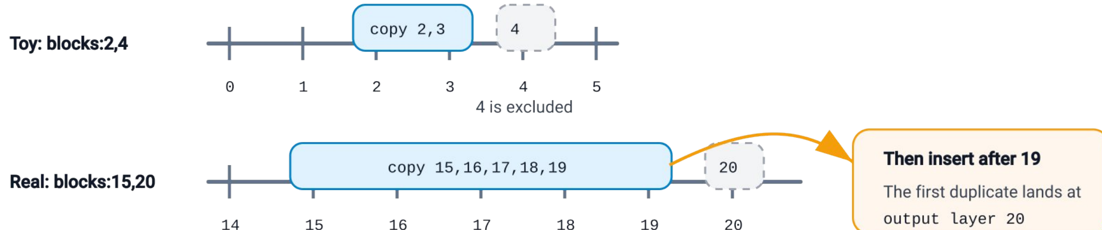
source: 0, 1, 2, 3, 4, 5

Now use blocks:2,4.

The notation is half-open, like a Python slice. The left edge is included. The right edge is excluded. So blocks:2,4 copies source layers 2 and 3. It does not copy layer 4.

Half-open windows: the right edge is not copied

In blocks:i,j, i is included and j is excluded. The copied layers are i through j-1.

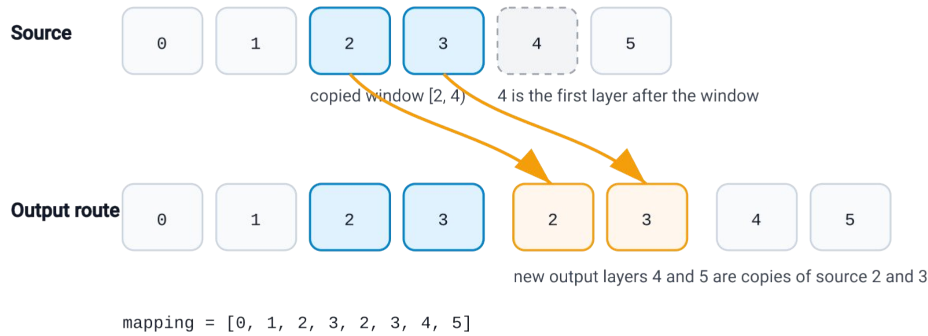


Half-open ranges: blocks:15,20 copies 15..19, not 20.

The duplicate is inserted right after the original copied window:

output source route: 0, 1, 2, 3, 2, 3, 4, 5

The copied layers are output 4 and output 5, but their source layers are 2 and 3.



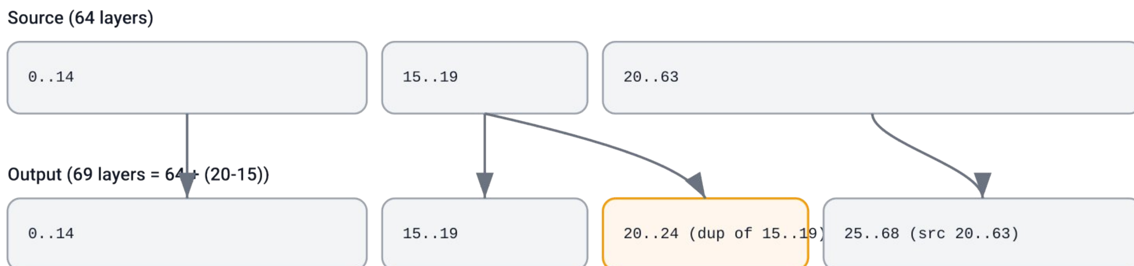
Toy mapping for blocks:2,4.

Now scale that up to the real 64-layer example.

blocks:15,20 copies source layers 15, 16, 17, 18, and 19. It does not copy source layer 20. Because the copy is inserted after source layer 19, the first copied output layer is output 20.

For a 64-layer source model, blocks:15,20 creates a 69-layer output model:

> output 0..19 -> source 0..19
>
> output 20..24 -> source 15..19
>
> output 25..68 -> source 20..63



Real mapping for blocks:15,20: 64 layers become 69.

Where the offset actually comes from

The part that looks strange at first is the shift after the insert.

It helps to stop thinking of blocks:15,20 as "layer 15 to layer 20" in everyday counting. In code, it is a slice boundary pair:

i = 15

j = 20

duplicate_length = j - i = 5

The duplicated source window is 15, 16, 17, 18, 19. That is five layers. Source layer 20 is not inside the copied window. It is the first layer after the window.

The original window stays where it was. Source layers 15..19 are still output layers 15..19. Then the copy is inserted into the next open output slots, which are 20..24. After those five inserted layers, the original source layer 20 can continue, but it has been pushed forward by five slots. So source layer 20 becomes output layer 25.

That is all the offset means. It is not an attention trick. It is just bookkeeping for the new stack length.

The piecewise rule is:

if out < j: src = out

if j <= out < j + (j - i): src = i + (out - j)

if out >= j + (j - i): src = out - (j - i)

For blocks:15,20, that becomes:

output 19 is still source 19

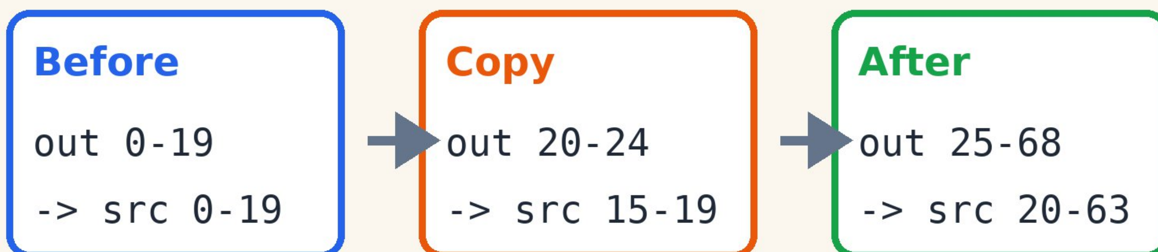
output 20 is the first duplicate, so it goes back to source 15

output 24 is the last duplicate, so it maps to source 19

output 25 is after the inserted copy, so it resumes source 20

d = j - i = 5

blocks:15,20 copies 15..19. Layer 20 is outside the copied window.



Source 20 resumes at output 25 after the five inserted layers.

The offset is just the insert length: d = j - i.

The seam is the part worth checking by hand:

output 19 -> source 19

output 20 -> source 15

output 24 -> source 19

output 25 -> source 20

If those four points are right, the insert is probably being interpreted correctly. If they are wrong, everything downstream is already suspect.

One map, two ledgers

Once the map exists, there are two ledgers to update: the tensor ledger and the config ledger.

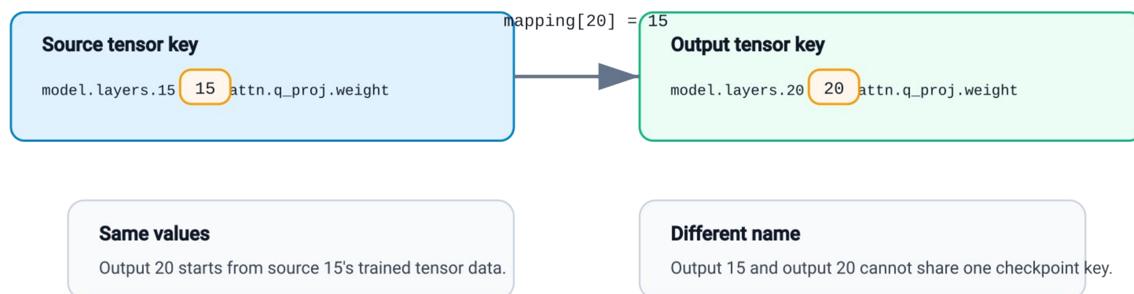
The tensor ledger is the visible one. In a safetensors checkpoint, each layer owns a family of tensor keys. A key might look like this:

model.layers.15.self_attn.q_proj.weight

If output layer 20 is copied from source layer 15, the output checkpoint needs the same tensor values under a new key:

model.layers.20.self_attn.q_proj.weight

You cannot reuse the old key. Source layer 15 still exists as output layer 15, and the duplicate exists again as output layer 20. Those two layers can start with identical values, but they need different names in the output checkpoint.



Tensor values can be copied, but output layer keys must be renamed.

The export loop is just a controlled rename:

copy global tensors unchanged

for out in range(len(mapping)):

```
    src = mapping[out]

    for tensor_name in tensors_for_layer[src]:

        new_name = replace_layer_index(tensor_name, out)

        output_tensors[new_name] = clone_if_reused(source_tensor)
```

The clone is not decorative. If a source layer is reused, some safetensors save paths do not like multiple checkpoint keys pointing at the same underlying storage. Cloning duplicated tensors keeps the output file clean and avoids shared-storage save errors.

Then comes the config ledger. This is the part that explained the loop in our broken build.

The tensor ledger and config ledger both need to follow the same map.

If the source model had 64 layers and the output model has 69, the layer-count field must become 69. Depending on the architecture, that might be num_hidden_layers, text_config.num_hidden_layers, n_layer, num_layers, or something model-specific.

For per-layer config lists, the rule is exactly the same as the tensor rule:

```
new_list[out] = old_list[mapping[out]]
```

That line is boring on purpose. In this build, boring is safety.

If output layer 20 got source layer 15's weights, output layer 20 also gets source layer 15's metadata. If output layer 25 resumes source layer 20's weights, output layer 25 also gets source layer 20's metadata.

The config mirrors the same map as the weights

Do not shift a per-layer list by hand. For each output row, look up the source row and copy that source metadata.

output layer	mapping[out]	weights source	layer_types source	correct output type
19	19	src 19	src 19	full
20	15	src 15	src 15	full
24	19	src 19	src 19	full
25	20	src 20	src 20	linear

`new_list[out] = old_list[mapping[out]]`

Per-layer config lists must follow the same output-to-source map.

This is where many RYS builds quietly break. They update the tensors. They bump the layer count. But a per-layer config list stays in the old order. The model can load while executing copied layers with the wrong instruction sheet.

The Qwen3.6 loop story

For this Qwen3.6 hybrid checkpoint, some layers take the full-attention road and some layers take the linear-attention road. Those are not cosmetic labels. They are different execution paths inside the model.

> text_config.layer_types is the per-layer routing plan that tells the runtime which road each layer should take. That makes layer_types execution-critical.

Around the RYS window, the source schedule was:

source layer 15: full_attention

source layer 16: linear_attention

source layer 17: linear_attention

source layer 18: linear_attention

source layer 19: full_attention

source layer 20: linear_attention

Now follow the blocks:15,20 map. Output layer 20 copies source layer 15. Source layer 15 is full_attention. Therefore output layer 20 must also be treated as full_attention.

If layer_types is not remapped, output layer 20 can accidentally keep the old entry that used to sit at index 20. In this source model, that entry was linear_attention. That creates the mismatch we were looking for:

output layer 20 weights: source 15

correct output layer 20 type: full_attention

naive output layer 20 type: linear_attention

The bug is not that one attention type is bad. The bug is sending copied weights down the wrong execution path.

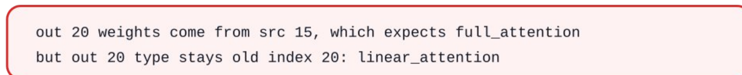
layer_types is an execution plan, not decoration

In the Qwen3.6 hybrid line, the runtime uses this list to choose full_attention or linear_attention for each layer.

Source schedule near the seam

Correct boundary after remapping

Wrong at output 20 if the old list is kept



In Qwen3.6 hybrid models, layer_types is an execution plan.

Depending on the loader and architecture checks, a broken build may either fail loudly or load far enough to generate. The dangerous case is the quiet one: the shapes look plausible, but the copied layer is routed through an execution path that does not match where its weights came from.

The offset math did not create that attention mismatch by itself. The offset only said "output 20 came from source 15." The mismatch happened because layer_types did not move with the same output-to-source map.

Once output 20 became a copy of source 15, every per-layer execution list also needed to treat output 20 as source 15.

So the attention rule is not a separate rule. It is the same map again:

output 20 weights come from source 15

output 20 layer_type comes from source 15

output 25 weights come from source 20

output 25 layer_type comes from source 20

The symptom we cared about was degenerate repetition and looping. Looping by itself is not proof of this exact bug, because sampling settings, templates, quantization, runtime code, and prompt handling can all cause repetition. In this build, though, the seam lined up with the failure: the first wrong metadata entry appeared at output layer 20, exactly where the RYS insertion began. When the metadata was remapped with the same map as the tensors, that split story disappeared.

The fix we trusted was not a sampler tweak. It was to use the same map twice:

> weights_out[out] = weights_src[mapping[out]]
>
> layer_types_out[out] = layer_types_src[mapping[out]]

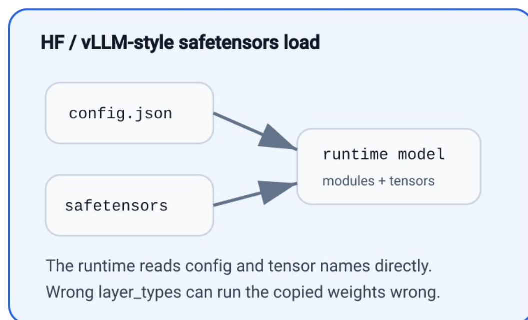
Once both ledgers followed the same map, output 20 got source 15's weights and source 15's full_attention type. Output 25 resumed source 20's weights and source 20's linear_attention type. The checkpoint stopped telling two stories at the insertion boundary.

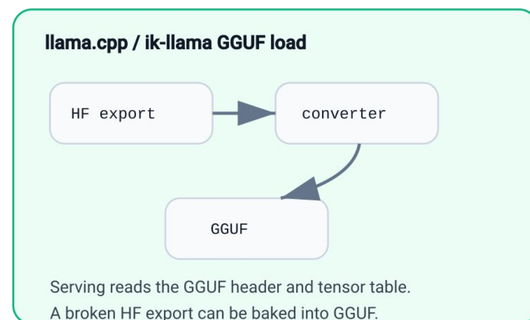
Loaded where?

"The model loads" is too vague unless you say where it loaded.

"Loads" means different things in safetensors and GGUF paths

Treat them as two artifacts with two validation surfaces. The same RYS map should survive both paths.





Do not use one load success as proof for the other path.
Validate HF/vLLM as a safetensors folder; validate llama.cpp as the converted GGUF artifact.

HF-folder loading and GGUF loading validate different artifacts.

HF safetensors loading checks the folder story. GGUF loading checks the converted artifact story. Passing one does not certify the other.

An HF-format load reads config.json, the safetensors index, and the shard files together. It proves that the folder is at least structurally plausible for that loader: layer counts, tensor names, tensor shapes, and config fields are being interpreted together. A strict loader may fail if the selected layer type expects a different tensor family. A permissive or custom path may get farther and still be semantically wrong.

The llama.cpp or ik-llama path checks a different object. The HF checkpoint is converted first. During conversion, the converter reads the HF config and tensors, then writes a GGUF with its own metadata header and tensor table. At serve time, llama.cpp reads that GGUF, not the original safetensors folder.

In our production path, the HF safetensors RYS export came first. Then it was converted to GGUF for ik-llama serving. The GGUF-side checks were useful for that artifact: the BF16 GGUF reported a 69-block model and the loader saw 915 tensors. That did not replace the earlier HF config check; it confirmed a different stage of the pipeline.

The practical rule is simple: validate the artifact in the form that will run it.

For HF or vLLM-style safetensors, inspect config.json, tensor keys, layer counts, and per-layer metadata.

For llama.cpp or ik-llama, inspect the GGUF header, tensor count, block count, conversion logs, quantization logs, and serve smoke tests.

The checks that saved time

A loaded model is not automatically a correct model. Loading mostly proves that the checkpoint is structurally plausible. It does not prove that every copied layer has the right execution metadata.

What success looks like is boring in the best way: the output layer count matches the tensors, every copied tensor has the right output name, every per-layer metadata list has the output length, and the seam entries point back to the same source layers as the weights.

Before judging quality, check the artifact mechanically:

Every expected output layer index exists in the tensor keys.

No tensor key is duplicated.

The config layer count matches the output layer count.

Every per-layer metadata list has the output length.

Every remapped metadata entry follows the map.

For layer_types, the check is:

```
for out, src in enumerate(mapping):
    assert layer_types_out[out] == layer_types_src[src]
```

For blocks:15,20, check the seam:

```
output 19 -> source 19
output 20 -> source 15
```

output 24 -> source 19

output 25 -> source 20

Validate the artifact before judging the model

Loading proves the checkpoint is plausible. These checks prove the copied layer story is internally consistent.

Structural checks

- ✔ 69 output layer tensor groups exist
- ✔ 69 layer_types entries exist
- ✔ config layer count matches tensor groups
- ✔ no duplicate tensor keys after renaming
- ✔ metadata list length equals output layer count

Boundary seam checks for blocks:15,20

out 19 → src 19

out 20 → src 15

out 24 → src 19

out 25 → src 20

```
layer_types_out[out] == layer_types_src[mapping[out]]
```

If these fail, fix the checkpoint. Do not waste time judging prompts, scores, or quantization yet.

Mechanical checks before quality judgment.

At this point, you should be able to explain why output 20 comes from source 15 and why output 25 resumes source 20. If that explanation is fuzzy, do not quantize yet. The quantized model will only make the bug harder to see.

A practical validation path

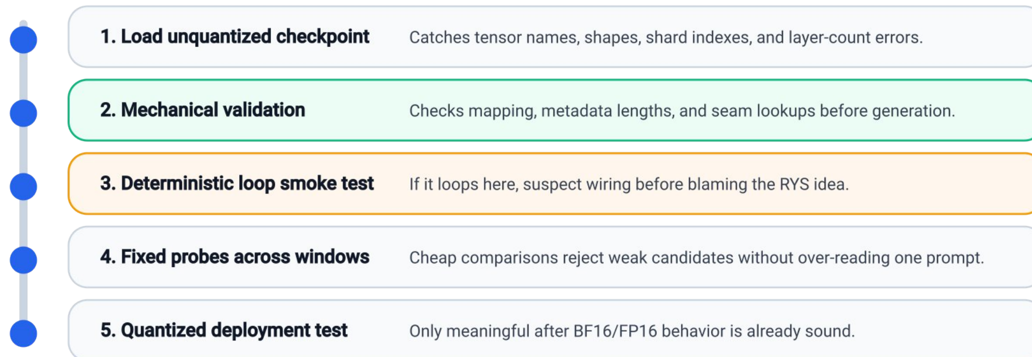
The testing order matters.

First, load the unquantized checkpoint. If it fails, the problem is usually tensor names, tensor shapes, shard indexes, or layer-count config.

Second, run a short loop smoke test. Use a boring prompt that forces structure, for example: "Write 12 distinct one-sentence scenes in the same city. Do not repeat sentence openings." If the model immediately repeats, do not assume RYS is bad. Check whether weights and metadata still point to the same source layer.

Third, run fixed probes across candidate windows. These are not final proof of model quality. They are a cheap way to compare options and reject broken builds.

Only after the BF16 or FP16 checkpoint behaves should quantization enter the story. Quantized failures are much harder to debug if the unquantized model was never proven sound.



Test like a ladder: structure first, quantization last.

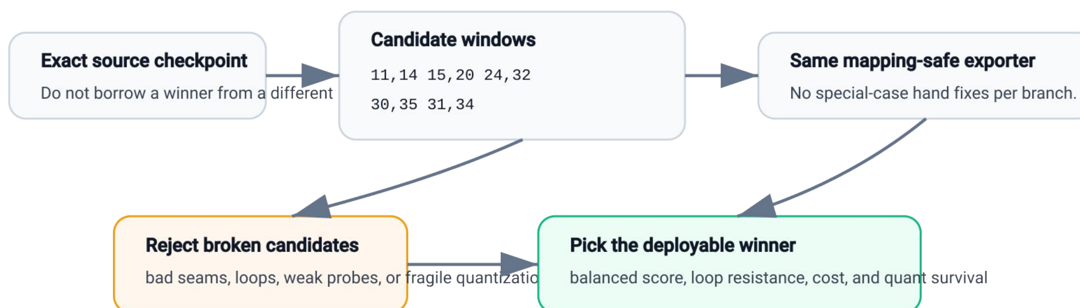
Choosing the layer window

The best RYS window belongs to the source checkpoint. A window that works well on one source can be weak on another, even when both share the same base architecture. Fine-tuning can move useful behavior through the stack, so each source line needs its own search.

Start with the exact checkpoint intended for use. Do not choose a window from a different fine-tune just because it has the same base architecture.

A practical first pass is to test a small spread of short windows: one early-middle, one middle, one late-middle, and one window suggested by prior experiments. Export each one with the same mapping-safe pipeline. Run the same structural checks, loop prompts, and probes. Compare everything against the unmodified source.

Reject broken candidates before comparing scores. A window that gives a nice number but loops, fails conversion, or collapses after quantization is not a deployable winner.



Window selection belongs to the exact source checkpoint.

Short insertions are easier to debug at first. Larger inserts can come later after the export path is trusted. The winner should not be chosen from one score alone. Load behavior, loop resistance, target probes, general behavior, serving cost, and quantization survival all matter.

The important lesson from our run is not "everyone should use 15,20." It is "the window belongs to the source checkpoint and the deployment path."

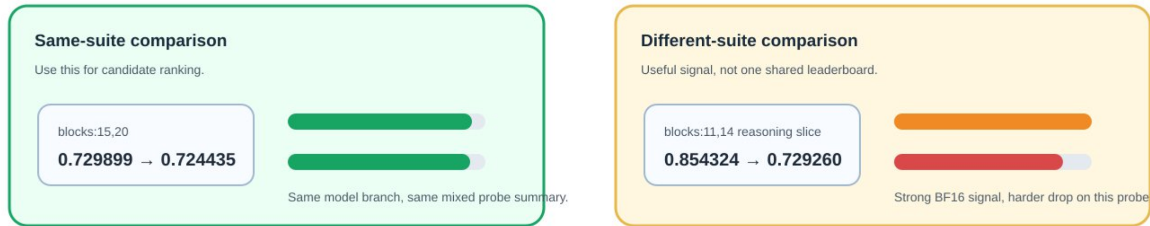
Quantization is the callback

BF16 is the audition. Quantization is the callback.

BF16 is the first audition. Quantization is the callback.

A window that looks great before compression can get weird once you squeeze it down, especially after RYS has changed the activation path. That is why we treated quantization as a second trial instead of assuming BF16 rankings would survive.

This is where the notes almost tricked us: the quantization files were not all grading the same subjects.



Compare candidates on the same suite. Different probes are useful notes, not one shared leaderboard.

Do not compare headline means unless the candidates ran the same suite.

The blocks:15,20 file used a mixed suite: math_16, eq_16, math_4, and gsm8k_5. Its headline number was overall_mean.

One blocks:11,14 file used the reasoning slice: math_4 and gsm8k_5. Its headline number was combined_mean. Another blocks:11,14 no-think file used math_16 and eq_16, where the combined drop was much smaller. Those files are useful, but they are not one scoreboard.

In the mixed-suite file, blocks:15,20 had a small overall IQ4_NL imatrix drop, even though individual probes moved differently:

BF16 overall_mean: 0.729899

IQ4_NL overall_mean: 0.724435

delta: -0.005465

Quantization is a second trial, not a formality.

In the reasoning-slice file, blocks:11,14 was genuinely strong in BF16, but dropped much harder under IQ4_NL on that specific probe:

BF16 combined_mean: 0.854324

IQ4_NL combined_mean: 0.729260

delta: -0.125064

In the no-think math_16/eq_16 file, blocks:11,14 dropped by -0.020520 combined, not -0.125064. That matters. The stronger claim is only true for the reasoning-slice test.

So the honest read is this: 11,14 looked strong in BF16 and had interesting math behavior, but the reasoning-slice quantization result was volatile. 15,20 was the safer production pick because it survived the balanced scan, loop checks, serving path, and mixed-suite quantization delta.

A clean final rematch would run the same full quantized suite for both 15,20 and 11,14. Until then, do not flatten those two result files into one leaderboard.

The Qwen3.6 case file

Here is the specific build we landed on.

The source line had 64 layers indexed 0..63. Its important per-layer execution metadata was text_config.layer_types. The stack mixed linear_attention and full_attention.

The selected RYS spec was blocks:15,20. That copied source layers 15..19, inserted 5 layers, and produced a 69-layer output model.

The mapping was:

```
output 0..19 -> source 0..19

output 20..24 -> source 15..19

output 25..68 -> source 20..63
```

The config update was:

```
text_config.num_hidden_layers = 69

text_config.layer_types = [source_layer_types[mapping[out]] for out in range(69)]
```

The tensor export rewrote every per-layer tensor from source index to output index, cloned tensors when a source layer was reused, and copied global tensors unchanged for this export.

The validation target was simple:

69 output layer tensor groups

69 layer_types entries

output 20 tensors from source 15

output 20 layer_type from source 15

output 25 tensors from source 20

output 25 layer_type from source 20

The fixed probes compared candidate windows against the unmodified source. This chart uses the AEON scan files aeon_strict_math120.pkl and aeon_strict_eq140.pkl.

combined = (math_120 + eq_140) / 2

Here, eq_140 is the historical probe/file label. The saved strict EQ pickle contains qids 1..139, so read the label as the scan name rather than a literal counted row claim.

Strict probe results, sorted by combined score

combined = (math_120 + eq_140) / 2. The small differences matter only after the build has passed structural and loop checks.

window	math_120	eq_140	combined	relative combined bar
blocks:15,20	0.971441	0.647347	0.809394	
blocks:31,34	0.977269	0.640750	0.809010	
blocks:11,14	0.981498	0.634929	0.808214	
blocks:24,32	0.971823	0.628559	0.800191	
baseline 0,0	0.970181	0.629710	0.799945	
blocks:30,35	0.964841	0.627521	0.796181	

Bars are scaled within this candidate set to show separation. The chart is a selection aid, not proof that one window generalizes to every source model.

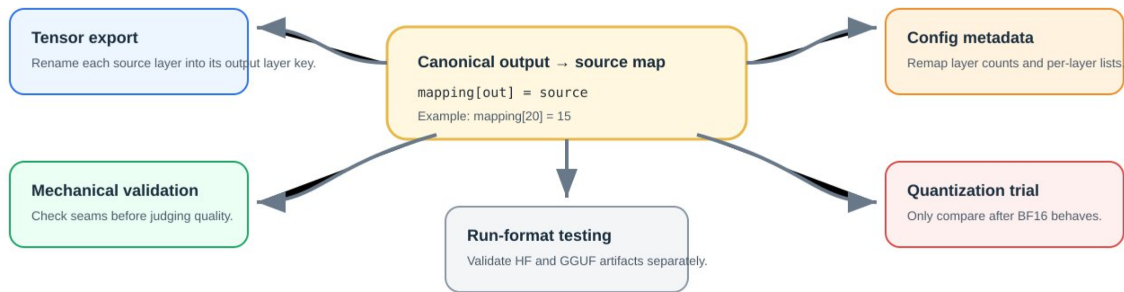
Strict probe results for the Qwen3.6 AEON source line.

In this source line, blocks:15,20 was the best balanced strict result. blocks:31,34 was close. blocks:11,14 had the highest math_120 result but not the best combined result. blocks:30,35 had worked better in another source line, but it was not the winner here.

That chart is not proof of a universal best window. It is selection after structure checks, loop checks, source-specific probes, and deployment constraints. RYS is not a magic index recipe. It is a build method plus a measurement loop.

The finished mental model

The clean version of RYS is not "duplicate some layers and hope."



One output-to-source map feeds the whole build: tensors, config metadata, validation, testing, and quantization.

It is:

Build one output-to-source map.

Use that map to move the tensors.

Use that same map to move every per-layer execution list.

Validate the seam before judging quality.

Test the artifact in the format that will actually run.

Quantize only after the unquantized build behaves.

When those pieces agree, RYS becomes a controlled checkpoint transformation. The tensors say where each output layer came from. The config says the same thing. The per-layer metadata says the same thing. The runtime no longer has to guess.

When those pieces disagree, the model can still load. But generation may reveal the split story through loops, repetition, or sudden quality collapse.