v17 output differences

Adaptive_LowRank_Training_Research.docx.pdf
PDF

v15_log.jsonl
File

v15_transformer_lowrank_adaptive_grad.py
Python

v16_log.jsonl
File

v16_transformer_lowrank_pressure_cooker.py
Python

v17_transformer_lowrank_pressure_cooker.py
Python

So this is an overview of what we're working on. We had left off at v15 last week, and today when we picked it back up, I asked you to try and make some significant strides, and while v16 still looked familiar when it came to output, the moment I asked you to continue moving forwards and also, besides improvements, also add an inference step, v17 started to look very different when it comes to the output it shows me while triaining, so I want you to have a look and check if we're still on the right track here.

I looked through the report + the three code versions. The “v17 suddenly looks *very* different while training” isn’t your imagination — it’s because **v17 isn’t just “v16 + inference.”** It’s a fairly big pivot in *how rank is controlled* **and** how progress is printed.

Why the v17 training output looks so different (and why that alone isn’t a bug)

v17 prints a **different set of “headline” numbers**, at a **different cadence**:

- It now prints **vocab size** and a **uniform baseline loss** upfront (log(V), plus baseline ppl). v17_transformer_lowrank_pressur…
- It logs **by step** (every 50 steps) with: learning rate, CE loss, ppl, rank(min/mean/max), and the pressure coefficient **`pressα`**. v17_transformer_lowrank_pressur…
- It runs periodic eval, prints eval losses, and saves **`last.pt`** every eval + **`best.pt`** when improved. v17_transformer_lowrank_pressur…

So even if the underlying learning dynamics were identical (they aren’t), the console stream would still “feel” completely different.

The bigger deal: v17 changed the *algorithmic* story, not just the logging

If your “right track” is **Adaptive Low-Rank Training (ALRT) as described in the report**, v17 has drifted.

What the report says the method is

The report's core loop is: estimate stable rank from spectral properties (power iteration), smooth/control it, then resize ranks (with truncated SVD "sqrt split"), and (because shapes change) reinitialize optimizer. Adaptive_LowRank_Training_Resea…

It also emphasizes **layer-specific rank differentiation** (Q/K compress hardest, O/FFN need more), emerging over training. Adaptive_LowRank_Training_Resea…

What v15 / v16 were doing (closer in spirit)

- v15 is explicitly built around **per-layer low-rank modules** (Q/K/V/O/FF up/down) and it calls `maybe_update_all_ranks()` during training. v15_transformer_lowrank_adaptiv…
- v16 goes even further and keeps an explicit **spectral controller** that picks rank to capture an energy fraction of singular values, and updates ranks lazily on an interval. v16_transformer_lowrank_pressur…
- v16 also has a global **pressure controller** that's explicitly "closed-loop autoscaling" against a compute target. v16_transformer_lowrank_pressur…

What v17 is doing instead

v17 is basically a **"scheduled compression + regularizer"** approach:

1. **Rank is scheduled**, not inferred: you set `--rank-init`, `--rank-final`, warmup + anneal steps, and it periodically applies the same rank everywhere. v17_transformer_lowrank_pressur… v17_transformer_lowrank_pressur…
2. **The "pressure" is not a stable-rank estimate**. It's a weighted L2 penalty that grows with component index, implemented as
   - `pressure_w` increasing from 0→1 (linear or quadratic), v17_transformer_lowrank_pressur…
   - and `pressure_term = sum_i pressure_w[i] * (||A[:,i]||^2 + ||B[i,:]||^2)` v17_transformer_lowrank_pressur…
     This is a totally reasonable *pressure-cooker heuristic*… but it's **not** the report's stable-rank controller + resize pipeline.

So: **v17 is still "low-rank training"**, but it's not the same research branch as "adaptive rank inferred from spectra with per-layer differentiation."

So are you "still on the right track"?

Bluntly:

- **If the goal is ALRT-as-in-the-report** (stable rank estimation → controller → per-layer ranks diverge), then **v17 is off the mainline**. It replaced adaptive inference with a predefined schedule and a soft penalty, and it applies rank uniformly across all low-rank linears (so you won't naturally reproduce the report's "Q/K end up ~11, O ~23, FFN-down higher" story). Adaptive_LowRank_Training_Resea… v17_transformer_lowrank_pressur…
- **If the goal is "keep compressing training compute while staying stable, and add generation/inference ergonomics"**, then **v17 can still be a valid direction** — but treat it as a *new variant* (scheduled rank collapse + pressure regularization), not "the next ALRT iteration."

Quick sanity expectations for v17's console output (what "healthy" looks like)

Given the defaults:

- Rank stays near `rank_init` early, then slides toward `rank_final` during/after warmup/anneal. v17_transformer_lowrank_pressur…
- `pressα` ramps from ~0 to `--pressure-alpha` over training; early it should be tiny. v17_transformer_lowrank_pressur…
- CE loss should drop below the uniform baseline `log(V)` fairly early on non-trash data. v17_transformer_lowrank_pressur…

Two practical gotchas in the *output* (not necessarily the training):

- The step print shows **CE**, not **(CE + α·pressure)**, so you can miss that pressure is dominating. v17_transformer_lowrank_pressur…
  If you want debugging clarity, print `press` and `α*press` too.
- Eval uses `estimate_loss()` which measures **CE only** (again: fine, but know what you're looking at). v17_transformer_lowrank_pressur…

What I would do next (practical path forward)

You basically have two clean options:

Option A: Keep v16 as the "real ALRT line" and graft inference onto it

v16 already has the adaptive-ish machinery (spectral rank control + pressure controller + pruning knobs). v16_transformer_lowrank_pressur… v16_transformer_lowrank_pressur…

So the lowest-risk move is: **add v17-style `--mode generate`, checkpoint save/load, and maybe periodic sample generation** to v16, without changing rank logic. That keeps you aligned with the report's adaptive narrative.

Option B: Keep v17, but admit it's a different research hypothesis and make it more "adaptive"

If you like the "pressure pushes information into early components" idea, you can still make it *adaptive* by changing **only the rank chooser**, not the rest:

- Instead of a fixed schedule, estimate something per-layer (stable rank, or gradient spectrum like v16) and set ranks per layer (not globally).
- The report's controller logic (safety factor, EMA smoothing, thresholds, max step) is a good stabilization template. Adaptive_LowRank_Training_Resea…
- And you don't necessarily need resizing/optimizer reset in v17 because its params are fixed-shape — which is actually a nice simplification compared to the report's reshape+reset requirement. Adaptive_LowRank_Training_Resea…

Net: **v17 isn't "wrong," but it *is* a branch.** The sudden console-output shift is explained by the refactor (step-logging + baseline prints + pressure schedule + checkpointing), but the deeper reason it "feels different" is that it *is* different: it moved from "rank inferred from signal" to "rank imposed by schedule + regularizer."

If you want the cleanest "still on track with ALRT" story, the move is to pull the inference/checkpoint ergonomics from v17 into v16 and keep v16's adaptive controllers intact.

Sources

This explanation makes sense to me, but what it doesn't give me is any kind of handle on which direction is more valuable. If this is just not clear at this moment, then I would say let's continue optimizing ALRT, and as a secondary concern, add inference and model checkpointing. But the main goal will always remain the same for this project, to find a way to hopefully reduce the overall resource requirements for training/inferencing transformer based AI models. So please produce the full code for v18, and make teh best choices you can make at this time as to which direction you want to go in. And remember to keep the console output easy to understand, while using file based logging to get more granular insights.

I'm going to pick the "**optimize ALRT**" direction for v18.

Reason: the ALRT line is the one that most directly hits your primary goal — **lower training + inference resources** — because it actually learns *layer-specific* intrinsic ranks and then **physically shrinks the parameter shapes** (so you reduce optimizer state memory and checkpoint/model size, not just math in the forward pass). That matches the stable-rank/controller + resize-by-SVD approach from your report.

Adaptive_LowRank_Training_Resea…

v17's "pressure cooker + scheduled rank collapse" is a legit alternate hypothesis, but it's a *different hypothesis* and it doesn't naturally produce the same per-layer differentiation story (Q/K low, O/FF higher). It's worth revisiting later once ALRT is stable and well-instrumented.

v18 code

Here's the full v18 script (train + resume + checkpointing + KV-cached generation + JSONL logging):

Download v18_transformer_lowrank_alrt.py

What v18 does (the important bits)

- **Low-rank everywhere it matters**: Q/K/V/O and FFN up/down are `W = U @ V`.
- **Adaptive ranks (per layer)** via **stable rank**:
  - estimates σ_max with **power iteration**
  - computes $||W||\_F^2$ efficiently from factors (no full W needed)
- **Controller** per layer:
  - safety factor, EMA smoothing, thresholding, bounded step, bidirectional changes

- **True resizing**:
  - when shrinking: **truncated SVD + “sqrt split”** (preserves learned mapping best)
  - when growing: appends small random components
- **Optimizer reset after any rank change** (deliberate, per the report’s logic).
- **Inference**:
  - `--mode generate` supports temperature, top-k, top-p
  - uses **KV cache** so generation doesn’t re-do O(T^2) attention every token
- **Logging**:
  - console: readable “headline metrics”
  - file: `train.jsonl` stores granular events (`train_step`, `eval`, `rank_update`, `rank_change`, optional `rank_full`, `sample`)

How to run it

Train

```bash
python3 v18_transformer_lowrank_alrt.py \
  --mode train --data data.txt --out-dir runs/v18
```

Resume

```bash
python3 v18_transformer_lowrank_alrt.py \
  --mode train --data data.txt --out-dir runs/v18 \
  --resume runs/v18/last.pt
```

Generate

```bash
python3 v18_transformer_lowrank_alrt.py \
  --mode generate --ckpt runs/v18/best.pt \
  --prompt "Once upon a time" --max-new-tokens 400 \
  --temperature 0.8 --top-k 50
```

Console output you should expect

You’ll see:

- baseline `log(V)` loss + ppl at startup (useful sanity)
- periodic lines like:
  - `step | epoch | loss | ppl | lr | r min/mean/max | tok/s`

- eval lines:
  - `eval @ step | val_loss | val_ppl | r stats`
- occasional rank update summaries:
  - `rank_update @ step | changed X/Y | r before -> after | took …s`

The log file to mine for "what's really happening"

`runs/v18/train.jsonl` will include:

- `rank_change` entries per layer (with stable-rank estimate + controller targets + old/new rank)
- `rank_update` summaries (how many layers changed, rank stats, how long it took)
- (optional) `rank_full` big dump if you run with `--log-ranks-full`

Practical knobs (the ones you'll actually touch)

- **Rank update frequency**
  - default is "once per epoch-ish" (derived from dataset length)
  - override with `--rank-update-every N`
- **Resize method**
  - `--resize-method svd` (best fidelity, slower)
  - `--resize-method truncate` (fast iteration, more lossy)
- **Controller aggressiveness**
  - `--safety-factor`
  - `--change-threshold`
  - `--max-step`
  - `--ema-decay`

One honest warning

Because v18 **truly changes parameter shapes** and **rebuilds the optimizer** after a rank change, you may see small "optimization hiccups" right after rank updates. That's not inherently bad — it's part of the design trade.

Next steps after you run v18 (not asking you to do anything now): the most valuable follow-on is to compare v18's rank trajectories against v15/v16 logs and see whether it reproduces the "uniform compression → differentiation → stabilization" phases described in your writeup, using the new `rank_change` events as the ground truth timeline.

Sources

Hmmm, we had actually managed already once to circumvent that optimizer reset, mainly by implementing our own optimizer for certain parts so we could project momentum. Anyway, I wanted to try something, so have a look at this:

Transformer Architecture: A First-Principles Interrogation

Context

Our adaptive low-rank work revealed that transformer projections contain massive redundancy—Q/K compress to rank 11 from 512, meaning ~98% of the architectural dimensionality is unnecessary for attention pattern computation. This discovery came from questioning a single assumption.

The transformer architecture is 8 years old. It was designed for machine translation, then adapted for language modeling, then scaled. Many design choices were inherited from earlier sequence models, made for hardware that no longer exists, or simply carried forward because "it works."

Your task is to question everything.

Phase 1: Deep Interrogation

For each component below, engage in rigorous self-interrogation. Do not accept conventional wisdom. Ask yourself every question until you reach bedrock—either a principled justification or an admission that "we don't actually know why this is done this way."

Take your time. Think deeply. Challenge your own responses. If you find yourself saying "this is standard" or "this is how it's done," stop and ask why.

---

1. Attention Mechanism

The QKV Decomposition
- What is Q, K, V actually computing? Why three separate projections?
- We project from d_model to d_model for Q, K, V—we're not changing dimensionality. Why?
- Our low-rank work showed Q and K need only rank ~11. What does this imply about what attention is actually learning?
- Could Q and K be the same matrix? What would we lose?
- Could we eliminate V and just use the residual stream directly? What would break?
- Is the bilinear form (Q @ K.T) the right way to compute similarity? Why not learned distance functions? Additive attention? Something else entirely?

Multi-Head Attention
- Why multiple heads? What does each head learn that others don't?
- Heads are completely independent then concatenated—is this the right structure?
- Many heads learn redundant patterns. Should heads communicate during attention?
- What if we had one very high-rank head instead of many low-rank heads?
- MHA was designed when d_model was 512. We now use 4096+. Should head count scale with d_model? It often doesn't.

Softmax & the Attention Distribution
- Softmax forces attention to be a probability distribution. Why is this desirable?
- Every token must attend somewhere (weights sum to 1). Should a token be able to "attend nowhere"—to say "this position has no relevant context"?
- Softmax is expensive and has numerical issues. What would happen with ReLU attention? Sigmoid? Linear?
- The softmax temperature is fixed. Should it be learned? Per-head? Per-layer? Per-token?

Quadratic Complexity
- $O(n^2)$ in sequence length is the fundamental bottleneck. Is this inherent to "attention" or an implementation choice?
- Linear attention variants exist but often underperform. Why? What are they losing?
- Do we actually need all-to-all attention? Human cognition doesn't work this way.
- What if attention was sparse by design, not by approximation?
- What if we attended to learned "memory slots" instead of all previous tokens?

Causality & Masking
- Causal masking throws away half the attention computation. Is there a better formulation?
- Why do we compute full attention then mask, rather than just not computing masked positions?
- Is strict left-to-right autoregression the right inductive bias for language?

---

2. Positional Information

The Problem
- Attention is permutation-equivariant. Position must be injected somehow. But how did we decide which way?

Current Approaches
- Learned absolute positions: Why should position 1 and position 1000 have unrelated embeddings?
- Sinusoidal positions: Beautiful mathematical properties, but why should the network care about sine waves?
- Rotary (RoPE): Encodes relative position in the attention computation. Is rotation the right transformation? Why?
- ALiBi: Adds distance-based bias to attention scores. Simple and effective—what does this tell us about what position information the network actually needs?

Deeper Questions
- Why is position not intrinsic to the architecture? Is there a design where position emerges naturally?
- Do we need absolute position at all, or only relative?
- Should position encoding be the same for all layers? The same for Q and K?
- What if position was continuous rather than discrete?
- Humans don't have "position embeddings"—we perceive sequence through time and chunking. What architectural analog might exist?

---

3. Feed-Forward Networks

The Structure
- FFN is d_model → 4×d_model → d_model with nonlinearity. Why this specific expansion factor?
- The 4× expansion is from the original paper. Hardware has changed. Is this ratio still optimal?
- Why expand then contract? What if we contracted then expanded?
- What if FFN had multiple layers internally?

What is FFN Learning?
- Attention handles token mixing. FFN handles per-token transformation. Is this a clean separation or arbitrary?
- FFN parameters dominate the network (2/3 of params in standard transformer). Is this ratio justified?
- Our low-rank work showed FFN up-projections compress more than down-projections. Why the asymmetry?
- Some research suggests FFN acts as key-value memory. If true, should it be structured as such explicitly?

The Nonlinearity
- GELU, ReLU, SiLU, SwiGLU—many options, small differences. Does this suggest the specific nonlinearity doesn't matter much?
- The nonlinearity is element-wise. Is this the right granularity? What about group-wise? Channel-wise?
- What if there was no nonlinearity? The network would be linear layer by layer but not globally. Would this work?

Alternatives
- GLU variants add gating—this helps. Why? What does gating provide that standard nonlinearity doesn't?
- MoE replaces dense FFN with sparse experts. This works well. What does this tell us about FFN's role?
- If FFN is memory, could we replace it with actual retrieval? External memory?

---

4. Normalization

The Core Question
- What problem does normalization solve? There are multiple claimed benefits—which are real?
- Stabilizes training—but why is training unstable without it? Is that a normalization problem or an initialization/architecture problem?
- Enables higher learning rates—again, why? Is this treating symptom or cause?

Placement
- Pre-norm vs post-norm is still debated after years. What does this uncertainty tell us?
- Pre-norm makes residual stream more stable. Post-norm puts normalized signal into next layer. Which is principled?
- Could we normalize in some places but not others? Which places matter?

Type
- LayerNorm: Normalizes across features. Why features? Why not other dimensions?

- RMSNorm: Removes mean-centering, keeps scale normalization. This works fine. Was mean-centering ever necessary?
- BatchNorm: Works great in CNNs, not transformers. Why? What's different about the inductive bias?
- What would happen with no normalization but careful initialization? Has this been thoroughly tested at scale?

Learned Parameters
- LayerNorm has learned scale (γ) and shift (β). These are often near 1 and 0. Are they necessary?
- Could normalization be entirely non-parametric?

---

5. Residual Connections

The Structure
- Every sublayer (attention, FFN) is wrapped in a residual: x + sublayer(x). Why?
- This means the "residual stream" is the sum of all previous sublayer outputs. Is this a good architecture or a historical accident?
- Residual connections allow gradients to flow unchanged. But should gradients flow unchanged?

Questioning Residuals
- Without residuals, deep networks don't train well. But maybe the problem is something else that residuals happen to fix?
- ReZero: Initialize such that sublayer output is zero, then learn to deviate. This works. What does this suggest about what residuals actually do?
- Highway networks had learned gates on residuals. This was dropped for simplicity. Was that the right call?
- What if residual weight was learned? Per-layer? Per-channel?

The Residual Stream Hypothesis
- Some interpret transformers as writing to and reading from a "residual stream." If this is the right model, should the architecture make it explicit?
- Should earlier layers' contributions be weighted differently than later layers'?
- What if the residual wasn't additive but multiplicative? Or something else?

---

6. Embeddings

Token Embeddings
- One vector per token. Is this the right representation?
- Rare tokens get as many parameters as common tokens. Is this efficient?
- Subword tokenization creates bizarre artifacts (BPE failures). What if we operated on characters? Bytes? Spans?
- What if embedding dimension varied by token frequency?
- The embedding is learned but then frozen during the forward pass. What if it was dynamic—context-dependent input representations?

Positional Embeddings
- (Covered above, but noting the interaction with token embeddings)

- Adding position to token embedding—is addition the right composition? What about concatenation? Multiplication? Gating?

Output Embeddings / LM Head
- Weight tying: Using transposed input embeddings for output projection. Elegant, but is it principled?
- Softmax over vocabulary (often 50k-100k+). This is expensive. Every token requires computing 50k logits.
- Vocabulary size is growing (multilingual, code). Scaling is terrible. Hierarchical softmax? Adaptive softmax? These feel like patches.

---

7. The Training Objective

Next-Token Prediction
- The dominant paradigm. But is predicting the next token the right objective for learning representations?
- Autoregressive factorization $P(x) = \prod P(x_t | x_{<t})$ is mathematically valid but arbitrary. Why not right-to-left? Bidirectional? Random order?
- Teacher forcing: Always feed ground truth, predict next. This creates exposure bias—model never sees its own errors during training.

Alternative Objectives
- Masked language modeling (BERT): Predict masked tokens. Learns different representations. Why don't we use this for generation?
- Span corruption (T5): Predict masked spans. Somewhere between token and sequence level.
- What if we trained to predict multiple tokens ahead? Would this change the architecture requirements?
- What if the objective was latent (VAE, diffusion-style) rather than explicit token prediction?

---

8. Depth vs Width

Depth
- Modern LLMs are 32-128 layers. Why? Is there a principled reason or did we just scale up?
- What does depth provide? Composition of functions? Multi-step reasoning? Something else?
- Some research suggests middle layers do most of the work. Are early and late layers important?
- Universal transformers share weights across layers. This often works comparably. Are unique per-layer weights necessary?

Width
- d_model has grown from 512 to 4096+. d_ff scales with it. But does it need to?
- Is there a depth-width tradeoff? Same parameters but different allocation?
- What if we varied width across layers? Narrow early, wide late? Or vice versa?

Interaction
- Is the current depth-width ratio optimal or just convention?
- Could we achieve the same capability with extreme depth and narrow width? Or extreme width and shallow?

---

9. Sparse vs Dense

The Current State
- Standard transformers are fully dense—every parameter activates for every input.
- MoE shows sparse activation can match dense performance with fewer active FLOPs.
- Sparse attention shows we don't need all-to-all token interaction.

Questions
- If sparsity works, why are dense models still dominant? Training stability? Simplicity?
- What if everything was sparse—sparse attention, sparse FFN, sparse embeddings?
- How do we determine which parameters should activate? Learned routing? Input-dependent?
- What is the right sparsity level? 10%? 1%? 0.1%?

---

10. Sequence Length & Context

The Problem
- Context length is limited (2k, 4k, 8k, 128k, 1M—constantly pushing). Why is this hard?
- KV cache grows linearly with sequence length. Memory is the bottleneck, not compute.
- Attention is $O(n^2)$. Long sequences are expensive.

Current Solutions
- Sliding window attention: Only attend to nearby tokens. Loses long-range dependencies.
- Grouped-query attention: Share KV across heads. Reduces memory, not compute.
- Ring attention: Distribute sequence across devices. Adds complexity.

Deeper Questions
- Do we actually need million-token context, or is this a patch for bad retrieval/memory?
- Human cognition has severe working memory limits but achieves coherence through chunking and abstraction. What's the architectural equivalent?
- What if context was hierarchical—sentence-level, paragraph-level, document-level representations that compress?
- What if the model could "forget" strategically—discard information it decides is irrelevant?

---

11. Computation Allocation

Fixed Compute
- Every token gets the same number of FLOPs—regardless of difficulty.
- "The" gets as much compute as a complex reasoning step.
- Is this efficient?

Adaptive Compute
- Early exit: Stop computation early for easy tokens. Requires knowing when to stop.
- Mixture of depths: Different tokens routed through different numbers of layers.
- Pondering: Spend extra compute on hard tokens.

Questions
- How much variation in difficulty exists between tokens? 10×? 100×? More?
- What signal indicates "this token needs more compute"?
- If we could perfectly allocate compute, what would the distribution look like?

Phase 2: Cost-Benefit Analysis

After your interrogation, for each component, estimate:

Component	% of Total FLOPs	% of Parameters	Functional Contribution	Verdict
Attention QKV projections				
Attention scores (softmax)				
Attention output projection				
FFN up projection				
FFN nonlinearity				
FFN down projection				
LayerNorm				
Embeddings				
LM head softmax				

Identify:
- Which component has the worst cost-benefit ratio?
- Which component are you least certain is necessary?
- Which component would be easiest to modify/remove/replace?

Phase 3: Hypothesis Formation

From your interrogation and analysis, formulate:

1. **The most promising target for architectural improvement**
   - What component or design pattern is most likely to be suboptimal?
   - What is your specific hypothesis about why it's suboptimal?
   - What would a better alternative look like?

2. **The theoretical benefit**
   - What efficiency gain do you expect? (FLOPs, memory, parameters, or capability per resource)
   - Why do you expect this?

3. **The risk assessment**
   - What could go wrong? What capability might be lost?
   - What's the failure mode—total collapse or gradual degradation?

4. **The minimal test**
   - What's the simplest experiment that would validate or invalidate your hypothesis?
   - What result would convince you to pursue this further?

  - What result would convince you to abandon it?

Phase 4: Implementation

Produce a complete, runnable `v11_*.py` script that implements your proposed architectural modification.

Requirements:
- Must be testable on WikiText-2 with the same setup as v10
- Must log comparable metrics (loss, timing, any new relevant metrics)
- Must be a genuine architectural change, not hyperparameter tuning
- Must include clear comments explaining the modification and its rationale

The goal is not necessarily to succeed—it's to rigorously test a well-reasoned hypothesis derived from first-principles analysis. A well-documented failure that teaches us something is valuable.

Guiding Principles

1. **No sacred cows.** "Transformers work" is not justification for any specific component.

2. **Historical context is not justification.** "It was designed this way" is not a reason to keep it.

3. **Popularity is not evidence.** "Everyone uses this" doesn't mean it's optimal.

4. **Question your own certainty.** If you feel sure something is necessary, ask why you're sure.

5. **Simplicity is a virtue.** Fewer components = fewer things to go wrong = easier to understand.

6. **Follow the compute.** What costs the most FLOPs? That's where improvements matter most.

7. **First principles over intuition.** When intuition conflicts with reasoning, trust the reasoning.

8. **Be willing to be wrong.** The goal is learning, not being right.

Take your time. Think deeply. Question ruthlessly. Then build.