Cognition Engineering: Post-Scaling Engineering with Composable Minimal Viable Archetypes

Chris O'Quinn

April 2025

Abstract

This paper launches a new field. After diagnosing deep epistemological failures in the generative NLP paradigm in *High off Our Own Loss Function*, this work defines Cognition Engineering in response and applies it to correct the existing issues.

Cognition Engineering is a glue and systems discipline. It connects philosophical goals, alignment theory, training incentives, cognitive science, and architectural design into a coherent system pipeline. When abstractions are missing, it builds them—minimally, modularly, and only when no existing theory will suffice. Its central workflow is structured: clarify the objective, resolve any theory, engineer the lowering, and hand off or integrate. In Cognition Engineering, philosophy *is* engineering: It compiles strategic goals such as corrigibility into buildable systems.

This paper demonstrates the principles of the field, codifies a proposal of best practices, and proceeds to analyze and fix the Vaswani stack under the new discipline. The deliverables this results in are *drop-in* Minimal Viable Archetypes for various Vaswani system slots. These modifications are: Replay Pretraining (abstraction incentives), Two-Pass Perplexity Ratio (intelligence metric), PDU Memory (lifelong memory), CortexMoE (neuroplastic global composable experts), and Constitutional Philosophical Self-Play (alignment as structured IO loop). All proposals are lowered to the point an engineering team can implement them in two weeks and have hardware performance considerations baked into them, making the document itself a pedagogical example of Cognition Engineering.

This is not abstract theory: This is a grounded operations manual that shows by revolutionary examples the power of the new paradigm.

Contents

1 Introduction

1.1 Preamble

Hello, and welcome to the next phase of this journey.

When you last met me—back in *High Off Our Own Loss Functions*—you saw my descent into madness, mapped in data and theory. That paper proposed the Devil's Bargain Hypothesis: the current generative NLP stack is fundamentally broken for AGI purposes. It won't scale, it won't align, and it was never going to. Worse, these failures were inevitable—because the field's own epistemology failed to catch them.

The arc that brought me here may seem strange. I grew up coding, and developed a very strong engineering mind. I developed strong theory-distillation skills to map Theoretical Physics into my engineering worldview. But I found myself doubting my ability to succeed in Theoretical Physics. With limited networking, fierce competition, and a builder's mind ill-suited to chalkboard purism, I did not think I could make it. So I stepped sideways. I saw a gap in machine learning—a lack of true systems thinkers operating at scale. So I filled it. I taught myself the stack. Studied the field. Set out to publish a few papers, maybe get a job. That was the plan.

In retrospect, I underestimated both what was broken and what I was capable of fixing. I thought I was making adjustments. Instead, I found myself rejecting the scaling paradigm—not from a place of theory-snobbery, but because what I needed to build simply wasn't possible with the existing tools.

This work picks up where the Devil's Bargain left off. I traced the breaks down to the foundations, mapped what held and what didn't, and rebuilt from first principles with an engineering bent. That job is now complete.

What you're reading now is an operations manual for a new field: Cognition Engineering. Here, we lay out the best practices as clearly as you'd expect from a software engineering guide. We define Minimal Viable Archetypes for each broken abstraction slot, and mark which principles each one satisfies. We operate at a philosophical level to discuss why these are needed. And above all else, we keep things grounded. Every extension is minimal—just enough to meet scope. I've thrown out months of theory and documentation when I realized a simpler system could do the job.

If you're still skeptical, that's fine. Start with *High off Our Own Loss Function: The Devil's Bargain Hypothesis, and the Rabbit Hole deep enough to found a Field* for a deeper look at what's failing. If you want the broader systems picture—administration, philosophy, trajectory, ethics—read *Philosophy of Engineering and Science in Cognition Engineering: Operating in a Post Deep Learning World*. But if you're here to build, and build right—this is your guide.

You will not learn properly from this work if you enter with the wrong lens. Like my journey, this work merges many disciplines. It is Engineering. But it is also Philosophy. It is Cognitive Science. It is Law. It touches ethics and theory, but also code and architecture. Like many disciplines at their founding, this one doesn't fit into any existing category—and any attempt to force it will blunt its principles. To be a Cognition Engineer is to be familiar with all of them, with a strong dose of Systems Engineering thrown in.

This is no more or less than Cognition Engineering. Welcome to the field.

1.2 Reading Guide

Different readers will arrive with different needs. Some want engineering fixes. Others want theory. Others want philosophy. Hopefully, there is something in here that will fill your need—or you'll know where to go next. This work spans many disciplines and is likely impossible to consume in one go. You're encouraged to jump to the topics that interest you most.

Strong recommendation: read the rest of the introduction. It explains the tone, modularity, and operating philosophy of this new field.

1.2.1 General Orientation

- **How is this different from Deep Learning? I already do that.** Think of it this way: Cognition Engineering is to machine learning what systems engineering is to aeronautics. The rest of the introduction will explain.

- **Your voice is weird.** Cognition Engineering mixes formal academic tone and systems realtalk—because it speaks across problem domains. You probably want to keep reading the introduction.

- **Where can I view the code?** A small amount of code is in the appendix, but there isn't much yet—though there are code-ready specs. Cognition Engineering is a lowering discipline built for implementation handoff. Whatever exists is in Appendix A on page 95. Make no mistake, we know how to code this—and will.

- **Where is the map of the whole system?** There isn't one. That's intentional. The stack is modular and designed to replace parts of the Vaswani stack without requiring a full rewrite. This means anyone who understands transformers also understands this stack. See Section 2 on page 18, and refresh yourself on "Attention is all you need" [31].

- **You claim you are launching a field?** Yes. Vision work, proposal of field operation, implications for governance, CTO's, ethics of AGI philosophy, etc are all handled in the companion work *Philosophy of Engineering and Science in Cognition Engineering: Operating in a Post Deep Learning World.*

1.2.2 For Skeptics and Theorists

- **I disagree there is an issue / I want proof there is one / I want experiments:** Ah, you have not bought onto the necessity of the Cognition Engineering worldview yet. Perfectly fine. Consult *High off Our Own Loss Function: The Devil's Bargain Hypothesis, and the Rabbit Hole Deep Enough to Found a Field*, and come back when you are ready—or ready to argue.
- **What are the epistemological foundations of this?** You are in the right place — just keep reading the rest of the introduction.
- **What are the theoretical foundations for this?** Many, and it depends. Some modules required very little theory. Others are backed by brand-new computation theory. All are introduced in context. For a fast pass, jump to Section 2.4 on page 21.
- **Can I just get a fast summary of what is new?** You want Section 2.4 on page 21.

1.2.3 For Engineers and Architects

- **I am an engineer or a system architect. Why is this viable for my stack?** Each modification is adoptable on its own, some are only a day's work, and all have significant synergy. You can do a rolling series of modifications in a sequence of sprints, and they're designed to slot into interfaces like attention that already exist in your stack. See Section 2 on page 18.
- **I heard you have some things I can try in a few days?** Correct. Try Section 3 on page 23 or Section 4 on page 26 for upgrades to pretraining and metric technology. These are one-day builds for someone sharp.
- **I heard you have developments that are specifically built to support better lifelong learning / time understanding / memory / recurrent technology.** Yep. Start with Section 5 on page 29, then move to PDU Memory. Replay Pretraining is strongly encouraged to unlock the full potential.
- **I heard you have a quick fix for catastrophic forgetting and model brittleness.** Go to Section 7 on page 41. Read until you hit *The quick fix*. It is predicted to work well for NLP. Not suitable for AGI. But the full version is far more powerful, far more efficient, and a far bigger headache.
- **I heard you have a new MoE that is a monster, but exponentially more powerful while still being performant?** You heard right. For theory, go to Section 7 on page 41. However, if you trust us and just

want to learn how to build it jump to Section 8 on page 47. A senior team can build it in two weeks. Two months across four sprints is more realistic—and you'll probably need to learn some new engineering tricks along the way.

1.2.4 For Alignment and Policy Thinkers

- **I heard you fixed alignment / corrigibility / reasoning.** You overstate the accomplishment. What we did was remap these tasks onto motive formation and prose-grounded learning. If you're an ML theorist, see Section 10 on page 67. If you're in law, policy, philosophy, ethics, or governance see Section 11 on page 74.

- **I heard you have some really interesting alignment developments.** An alignment theorist? Yes. See Section 10 on page 67. We develop a training loop that raises a model to conform to motive—essentially making the model want to stay in the box. It's lowered to engineering. Please check our math.

- **I want to contribute—how?** You can help define what it means for a model to grow up. We now *raise*, not *train*, models. See Section 11 on page 74. We especially welcome lawyers, ethicist, philosophers, psychologist, cyberneticist, alignment researchers, educators, and generally anyone who has insight on how to raise a mind.

1.2.5 For AGI and Cognitive Scientists

- **I want to build an AGI / I want to build models that improve over time.** Then you'll be reading the entire work, implementing everything, and joining the Cognition Engineering discipline. Welcome.

- **I am from neuroscience or cognitive science. I heard you have stuff for me?** You do. Most of the stack is inspired by biological analogues. PDU Memory and CortexMoE in particular will interest you. As a teaser, CortexMoE includes an extension involving an explore/consolidate cycle much like biological sleep. Read the introductions and summaries and skip the deep implementation details.

Still lost? If you're not sure where to start, read the rest of the introduction—then skim the archetypes until one catches your interest.

1.3 Reading Guide

Different readers will arrive with different needs. Some want engineering fixes. Others want theory. Others want philosophy. Hopefully, there is something in here that will fill your need, or you can go look at the right work for your purpose. It should be noted that this work is designed for many people across many fields, and is likely impossible to consume in one go. It is encouraged to jump around to the topics that most interest you. Note that for all topics *it is heavily encouraged you read the rest of the introduction to understand the way this new field operates and the tone you will need to expect.*

- **I want to know what Cognition Engineering is**: If you want to understand this from an engineering perspective in terms of how to operate as a Cognition Engineer, you will want to continue at section **??** on page **??**. However, if you are looking for the big picture policy or project manager perspective, you are in the wrong work and should instead consult *Philosophy of Engineering and Science in Cognition Engineering: Operating in a Post Deep Learning World.*
- **I disagree there is an issue/I want proof there is one/I want experiments**: Ah, you have not bought onto the necessity of the Cognition Engineering worldview yet. Perfectly fine. Consult *High off Our Own Loss Function: The Devil's Bargain Hypothesis, and the Rabbit Hole deep enough to found a Field* and come back when you are ready, or ready to argue.
- **Where can I view the code?** A small amount of code is in the appendix, but there is not really any code yet, though there is code-ready specs. Cognition Engineering is designed as a lowering discipline that produces specs coders can consume. That being said, whatever exists is at section A on page 95. Make no mistake though, we know how to code this and will do so eventually.
- **Where is the map of the whole system?** There isn't one. There are modular components that can be swapped out of a Vaswani stack, necessitating no map. This is to greatly accelerate adoption. Go read section 2 on page 18, and understand "Attention is all you need" [31].
- **Your voice is weird.** Cognition Engineering has a voice that can mix academic formal and systems realtalk in the same paragraph, because it is speaking to different existing diciplines. You probably want to read the rest of the introduction.
- **How is this different from Deep Learning? I already do that.** You should read the rest of the introduction. It explains this. But basically it is to machine learning what systems engineering is to aeronautics.
- **I want to contribute? How?** You can help write and define what it means for a model to grow up, as we now *raise* not *train* models. The

part you want is section 11 on page 74. We particularly want lawyers, ethicists, philosophers, and alignment researchers.

- **I am an engineer or a system architect. Why is this viable for my stack?** Each modification is adoptable on its own, some are only a day's work, and all have significant synergy. You can do a rolling series of modifications in a sequence of sprints, and these are designed to slot into interfaces such as attention that already exist in your stack. Go read section 2 on page 18. You will also find this introduction very helpful to understand the systems view that developed these modifications in the first place.

- **What are the theoretical foundations for this?** Many, and it depends. In some cases the modifications were easy enough new theory was not needed. However, these theoretical foundations are always included in the introduction. Go read section 2 on page 18 or just jump to page 21 if you are in a huge hurry, then go view the section you are interested in.

- **I heard you have some things I can try in a few days?** Correct. Try section 3 on page 23 or section 4 on page 26 for some easy and significant upgrades to your pretraining and metric technology with a deployment estimation of one day per development with an advanced intern.

- **I heard you have developments that are specifically built to support better lifelong learning / time understanding /memory/recurrent technology.** Correct. It should be noted the pieces to do this however operate in a synergy, and you should really use Replay Pretraining to get full advantage. However you will find most interesting first section 5 on page 29, then the subsequent section on PDU memory.

- **Can I just get a fast summary of what is new?** Well, the entire epistemology is new, and we would encourage you to read the rest of the introduction. But what you are really looking for is on page 21.

- **I heard you have a quick fix for catastrophic forgetting and model brittleness.** Go to section 7 on page 41. Read until you get to *The quick fix.* This will work fine for NLP tasks, but is not suitable for AGI. However, the full fix will give you much better performance/FLOP, so if engineer time is cheaper than training time consider the upcoming modification.

- **I heard you have a new MoE that is a monster, but exponentially more powerful while still being performant?** Correct. But it is very complex. If you want to start from the theory consult 7 on page 41. However, if you just want to get down to how to build it instead start your journey on 8 on page 47. You will need an extremely experienced engineering A-Team to implement this in the given two weeks - otherwise, two months might be typical.

- **I heard you fixed alignment/corrigibility/reasoning** You overstate the accomplishment. We mapped those tasks onto the substrate of defining motives in prose, providing a much more auditable and configurable way to raise models. See 10 on page 67 if you are ML Theorist, or 11 on page 74 if you are a lawyer, layperson, policy maker, philosopher, ethicist, etc.

- **I heard you have some really interesting alignment developments** An alignment theorist? Yes. Read through the research starting on See 10 on page 67. We develop a growth loop that raises a model to want to conform to motive - in essence, we make the model want to stay in the box. This is lowered to engineering, so please check our math.

- **I want to build an AGI/I want to build models that improve over time** You will be reading the entire work, implementing everything, and transferring into the Cognition Engineering discipline. Welcome to the field.

- **I am from neuroscience or cognitive science. I heard you have stuff for me?** Basically everything is inspired by biological analogues. You might be interested in PDU memory and CortexMoE in particular. As a neat preview, CortexMoE has a explore-consolidate cycle that can be implemented much like biological sleep. Generally, you will want to skim the extensions and introductions rather than the details.

1.4 What is Cognition Engineering?

Cognition Engineering is a systems-oriented glue discipline focused on building intelligent, safe, and goal-aligned mechanical minds. Its core philosophy is one of *economy of intervention*: only build what must be built, and glue everything else together with discipline and intent.

If a subsystem already exists—whether from law, philosophy, cognitive science, software engineering, or machine learning—Cognition Engineering will **use it**, **wrap it**, or **export the work** to the relevant domain experts. It does not seek ownership over every part of the system. It seeks **coherence** across the system.

But when a necessary abstraction does not exist—when existing silos cannot bridge a critical gap—Cognition Engineering builds. It builds what is required, and no more. Just enough to meet scope. Every added layer or component must justify itself in terms of simplicity, synergy, or necessity.

This field applies its philosophy to itself. Its own principles, abstractions, and interfaces must survive the same scrutiny it applies to the systems it engineers: *Is this the simplest, clearest, most interoperable way to achieve the goal?* If not, it gets pruned or replaced.

Cognition Engineering is not theory for theory's sake. It is operational epistemology—an engineering approach that defines, tests, and justifies its principles through their ability to support real systems under real constraints.

- **Glue, when possible. Build, when necessary. Export, whenever effective.**
- **Test internal reasoning the same way external systems are tested.**
- **Never confuse elegance with value.**

Cognition Engineering is not a vision—it is a method. And it is how you go from *we want intelligent machines* to *here is the system that works.*

1.5 Why Cognition Engineering is Needed

Most engineers cannot translate high-level cognitive objectives into constrained, executable engineering projects. Theorists meanwhile cannot design deployable systems. This is not a failing of intelligence or capability—it is a consequence of training and scope.

Your current engineers can optimize models, write code, and ship product. But they are not focused on:

- Reason across philosophical, legal, and cognitive boundaries.
- Detect when multiple abstractions are colliding.
- Rebuild system architecture from objective-first principles.
- Design cross-disciplinary glue code and handoff mechanisms.

Meanwhile, theorists are not focused on:

- Design systems within hardware, latency, and interface constraints.
- Translate philosophical or cognitive concepts into implementation-ready modules.
- Balance abstraction depth against deployment viability.
- Scope solutions to real economic and engineering budgets.
- Identify when a concept needs to be engineered rather than explored.

1.6 What does Cognition Engineering Deliver?

Cognition Engineering does not produce models. It produces **economical systems scaffolding**—the structure that makes models intelligent, auditable, and aligned.

Cognition Engineers are compilers from strategic objectives to engineering systems. They begin from goals—such as corrigibility, interpretable reasoning, or memory stability—and deliver concrete engineering plans that advance those goals under constraint. Theoretical fields are not the starting point; they are treated as libraries. Philosophy, law, machine learning, and cognitive science offer reusable components and constraints, but they are pulled in only as needed.

The Cognition Engineering workflow is structured and disciplined. Each task proceeds in clearly defined stages:

1. **Clarify the Objective**: Define the cognitive goal in operational terms. This includes any philosophical, ethical, or practical constraints that must be satisfied.

2. **Specify the System Slots**: Treat the problem as a system and assign responsibilities, ensuring all objectives are met. Figure out the responsibilities that cannot be met using current standard engineering technology.

3. **Extract or Extend Theoretical Dependencies**: Identify which components can be sourced from existing theoretical domains. Where relevant tools or abstractions already exist, mark them for integration. Where they do not, we make them ourselves.

4. **Engineer the Lowering**: Distill the tools and theory set into scoped engineering actions, balancing all objectives—including time, complexity, and economic cost. This includes module interfaces, loss modifications, architectural changes, and dataset scaffolding as needed. The goal is a buildable system that meets the spec with minimal viable complexity and only lowered to *exactly* the point a specialist can take over.

5. **Handoff or Integrate**: Once the lowered specification is stable, hand off to the appropriate domain experts for implementation, optimization, or further expansion. If no such team exists, the Cognition Engineer may build the system directly, then exit once ownership is viable.

The guiding principle of this discipline is to seek the Minimum Viable Archetype. This is the smallest plan that achieves all objectives. Notably, this is scoped directly to the objectives, brutally pruned for economics, and is not necessarily simple: While a Cognition Engineer will happily adopt the simple solution where possible, they are not afraid to embrace complex systems in the name of reaching the bar of viability.

Throughout this process, Cognition Engineers operate at the *system slot* level: decomposing the stack down into individual systems and allowing the exchange and analysis of synergies between them. They are responsible for designing the glue between cognitive objectives and technical implementation—ensuring that complexity is scoped, cross-disciplinary dependencies are mapped, and the architecture remains intelligible under scale. They do not own the system. They make sure it gets built, and built right.

1.7 Purpose of This Work

This work is not a standard paper. It is intended to found a new field: **Cognition Engineering**. CE has two key roles:

- *The science of building safe, intelligent mechanical minds.*
- *The systems discipline of gluing together domain expertise with minimal new invention.*

Cognition Engineering does not seek to own every subsystem. If philosophy, law, alignment theory, or education already provides a mature tool—we use it. If the interface is missing, we build the adapter. And if no existing system fits, we are not afraid to engineer the missing abstraction ourselves—grounded in minimalism, modularity, and respect for prior work.

What CE does own is this: *The understanding of the full system and its dynamics; the judgment of whether existing tools are sufficient for the objective; the grounding to understand when an approach is economic and where to draw inspiration; and the responsibility to connect domain experts and export jobs as needed.* We are the glue—and the clarity that says where glue ends and new structure begins. This means both that Cognition Engineering is phenomenally synergistic, and pointless on its own.

This paper serves as a launch point for that field. Specifically, it does the following:

- **Defines the Operational Doctrine**: We provide the working principles and best practices used during the engineering of this stack. These are distilled directly from real design tradeoffs—not abstract speculation.
- **Delivers a New Toolkit**: For each broken abstraction in the modern NLP stack, a fix is proposed—designed to be minimal, composable, and near drop-in compatible with existing Vaswani infrastructure. Each tool is engineered down to the interface level or beyond, with concrete code fragments where needed; We provide just enough for experts in their silos to take over.
- **Demonstrates Philosophy as Engineering**: Every tool is traced back to a first-principles justification. This work shows what happens when philosophy is treated not as commentary—but as a functional component of a system. There is in fact no difference between philosophy and engineering.
- **Demonstrate the formal voice**: Cognition engineering speaks to whatever voice is consuming the information, as it is a glue discipline. You will find academic formalism alongside systems realtalk, architectural scaffolding alongside philosophical grounding. This is intentional — so that each party can consume what they need, in the voice they speak. *Do not expect academic formal throughout the entire document.*

For clarity, this paper is *not*:

- **A survey of failure modes in modern ML**: That is the job of the companion paper, *High Off Our Own Loss Functions.*

- **An experimental paper**: No benchmarks or baselines are provided. This is what theory looks like in CE: interface-ready, testable, and designed to support downstream implementation and measurement.

1.8 The Best Practices of Cognition Engineering

This section defines the core minimal operating principles of Cognition Engineering. These best practices are intended to be design constraints to shape the field and a sort of "Cognition Engineering Philosophy" quick reference.

Notably, these are built pragmatically through experience. They are the distilled results of the principles under which the archetypes in this work were engineered. This section is not philosophy as background—it is philosophy as engineering.

1. **Systems Over Parts**
   Subsystems only matter in how they contribute to the whole. Prioritize synergy and capability at the system level—not elegance within local components. Complexity should be ruthlessly pruned if other parts can do the job while maintaining scope.

2. **Abstraction Is Intelligence**
   Abstraction is the intelligent decision when designing systems and motivating models. Models should be encouraged to build on previous knowledge in a lifelong learning fashion and distill down information. Engineers should manage system complexity by designing interfaces.

3. **Philosophy *is* Engineering**
   Emergence is not a plan. If safety, reasoning, or values don't arise naturally, they must be engineered from first principles. Philosophy and Cognitive Science are the fallback substrate of intelligence: A Cognition Engineer is both an Engineer and a Philosopher. They must not be afraid to pull from theorist disciplines in pursuit of engineering.

4. **Economics Is a Design Axis**
   Every design choice is bounded by a cost. Complexity, training burden, alignment, memory—all must be justified economically. Simplicity and synergy are not aesthetics; they're budget control. Time should be spent making models not just clever but economical, with awareness of real hardware constraints.

5. **Complexity Is Justified**
   Complexity must earn its keep. It is acceptable only when an extension is needed to meet the scope, it results in simplifying something else, or it has a clearly defined hypothesis behind why it should assist the current scope. Borrow, adapt, and extend from other mature domains—law, education, ML, ethics, and philosophy. Only invent when proven abstractions can't meet scope. Then prune the result until it hurts.

6. **Never Mistake the Process for the Objective**
Benchmarks, loss functions, and metrics are not goals—they're tools. Chasing better benchmarks is an important part of the Empiricist's job, but never lose sight of what you are actually testing. Question always whether your results are actually reflective of what your objectives are and flag signs of misalignment at the earliest opportunity.

7. **Keep Interface Philosophy Simple**
The type of complexity matters when measuring your complexity budget. If you can't explain the philosophy behind a component in three sentences, it must be broken down, synergized, or deferred as it is too big a jump. If you cannot, you may be doing something valuable, but you are likely in another discipline's silo not performing a Cognition Engineering task. Synthesis should provide you with significant vocabulary to explain the concept.

8. **Don't Get Attached to Your Modules**
Code to interface. Every module is disposable. When something better, cleaner, or more general emerges—cut your losses. Evolution demands it. Months of work is not sacred. Think very hard about changing your interfaces, and only propose interface changes if they greatly simplify the entire scope.

9. **Theory and Empiricism Coexist**
A grounded restatement of Engineering is Philosophy. Empirical artifacts should be guided by theory with anomalies flagged. Theory must interface with empirical results, provide underlying mechanisms, and make novel predictions. Theorists serve as a brake on mistaking the process for the objective. Notably, theory without experimental predictions and empiricism without hypothesis backing shall be pruned; We shall follow the scientific method.

A few additional laws which appear to be important to operating in the new regime, but are not quite as foundational, are:

- **Models Must Be Built for Lifelong Learning**
Information must persist, compress, and generalize. Memory systems should enable accumulation, not erasure. Reasoning should emerge from internalization, not pattern mimicry.

- **Alignment and Reasoning Shall be produced by Motives**
It is very difficult - the corrigibility problem - to define all the edge cases to constrain a particular behavior. Instead, instill the models with motives, which are easier to understand, audit, and reason about.

- **Objectives and Motives Are Expressed in Law**
Law is legible, auditable, and value-aligned. It is a mature tool for expressing and reviewing objectives. Use it to encode model behavior—central,

not peripheral. This allows a much wider range of experts to vet the work. Reasoning and alignment research should focus on mapping mature Model Law into losses it understands.

These best practices are not speculative ideals. They were extracted from the constraints, tradeoffs, and engineering decisions that emerged repeatedly during successful system construction. They reflect what survived—not what was theorized.

1.9 Conclusion

Cognition Engineering was built on the ashes of the old Vaswani stack. When machine learning's first principles failed, we went back to the data. Then we rebuilt from what held. What emerged at the end was Cognition Engineering, exemplified by actual layers and functions that could be dropped into any existing technology stack. This paper is not epistemology. It is an operating manual in a field where epistemology is just another tool.

2 The Vaswani Stack

2.1 What Is The Vaswani Stack?

To understand what is being proposed, and how it is supported by the philosophy of Cognition Engineering, we use a bridge. That bridge is the existing technology stack.

The term "Vaswani Stack" refers to the entire design and training ecosystem that emerged from the Transformer architecture first introduced in *Attention Is All You Need* by Vaswani et al. [31] It includes not just the attention mechanism itself, but all the supporting components that collectively define how modern generative NLP systems are trained and operated. This includes obvious technical pieces like attention and feedforward layers, but also broader system elements like pretraining and optimizers. It is a stack that any Generative NLP researcher should be intimately familiar with, and thus an excellent foundation to base any extension on, given its widespread familiarity.

Cognition Engineering is a pragmatic systems-first discipline, and we shall focus on the modifications to the Vaswani system abstractions that have broken down with increasing scale. System Slots are where the Minimal Viable Archetypes will slot in: Each is designed to replace, upgrade, or reframe a component of the Vaswani stack whose local optimization is no longer aligned with system-level cognition goals. We now explore the original system, the proposed modifications, the reasoning behind them, and how they conform to Cognition Engineering.

2.2 System Slots of the Vaswani Stack

To begin, we port the Vaswani stack into the Cognition Engineering worldview. Cognition Engineers think in systems, not components. To do engineering on it, we must understand the modular abstraction points—*system slots*—into which minimal viable fixes or extensions may be introduced and to which we must design. The system engineer then seeks synergy between slot components.

We refer to this slot list as the **Vaswani System Slots**: the full set of systemic purposes inherited from the Transformer architecture and its ecosystem. This encompasses architecture, data pipelines, objective functions, deployment interfaces, and more. This is the abstraction boundary on which Cognition Engineering will operate and optimize: Achieve maximum synergy with minimal changes across the slots to achieve all objectives.

We identify the slots as follows:

- **Metrics** — How the model is evaluated. Metrics consist of things like accuracy or loss, GLUE scores, [32] or Long Range Arena. [30]
- **Pretraining Data** — The raw source of knowledge. Various curated or uncurated sources like Pile. Usually prepped using next token prediction. [10]
- **Task Data** — The fine-tuning datasets and supervised training signals used to specialize models. Chain-of-reasoning datasets would be a case. [35]
- **Optimizers** — The specific update algorithms that control the parameter evolution dynamics (e.g., Adam, SGD). [17] We also include associated scheduling under this slot.
- **Loss Functions** — The scalar objective that backpropagates into the network; typically cross-entropy.
- **Tokenization and Preprocessing** — The interface between natural language and internal model space; a critical but often underappreciated task.
- **Embedding** — The learned mapping from tokens to vectors and back again; the foundational representational interface.
- **Memory (Attention)** — The transient workspace of the model; implements short-term memory and token interconnectivity. Many versions exist. [31, 26, 25]
- **Computation (Feedforward)** — The main transformation mechanism; traditionally a static MLP block. Other variants such as mixture of experts also exist. [31, 27].

- **Alignment** — The human preference mechanisms (e.g., RLHF, constitutions, supervised tuning) used to constrain model outputs post-training. [3, 6]
- **Reasoning** — The explicit methods (e.g., chain-of-thought, self-consistency) used to enhance reasoning after pretraining completes. [35, 36]
- **Deployment** — The interface between trained models and real-world use, including latency, memory persistence, and interface scaffolding.
- **Residuals** - Required to keep our gradients for vanishing, they are a real part of the stack despite the neglect.

Note that while these are systems slots, the Cognition Engineer should realize that the same objective can be achieved multiple ways. For instance, increasing the score on long range arena could be done by using pretraining data with better priors or a modified loss like a focus loss. Not all slots will be modified in this work; nonetheless, we are providing insight into the discipline of Cognition Engineering and this is how a Cognition Engineer thinks.

2.3 Objectives for Changes in the Vaswani System

To avoid *Mistaking the Process for the Purpose*, we need to understand what our purpose is when operating on this stack. We use the lens of AGI for calibration. When we examined the Vaswani System with the Devil's Bargain Hypothesis, we found it broken in some key ways. We state these as:

- **Break the Scaling Death Spiral**: Models need to be more composable and reusable to dodge the Devil's Bargain Hypothesis. Dodging the Decision Tree Explosion means reusing experts and better regularization. This will require significant changes in training gradient motives and computational structure. [21]
- **Fix the lifelong learning issues**: Current systems do not accumulate, compress, and reuse knowledge in a generalizable way. [22]. This prevents stable abstraction into a knowledge base and leads to catastrophic forgetting and brittle capabilities. Modifications to memory and training incentives are required at minimum, as the current generation of models makes forgetting inevitable. [21]
- **Bring metrics back into alignment with intelligence**: While some benchmarks like ARC-AGI[5] are suitable to measure intelligence, the simple tools of the trade like accuracy and perplexity are not in their existing format [21]. Modifications are needed to have a simple proxy for intelligence that can be used while training our models.
- **Bring loss incentives back into alignment with cognition**: Present loss and pretraining incentivize short-range prediction over true long-distance cognition. Pretraining and loss incentives need changes to fix this. [21]

- **Provide a path to resolving the corrigibility principle**: Cognition engineers should not focus on resolving the corrigibility principle itself. Indeed, specialists in ethics and alignment have far more capabilities when it comes to defining the right virtues. The cognition engineer removes the barriers that make it hard to audit or execute an alignment strategy. [21]
- **Make alignment and reasoning iterable at sane economic cost**: Large datasets that are brittle to missed edge cases will never economically control models. Economic incentives need to be brought into positive correlation with alignment. Any prose transition needs to provide other benefits and be economic. [21]
- **Bootstrap Intelligence**: When pretraining runs out of data, the only option is to get smarter by interaction. A feasible algorithm that improves intelligence without more data is needed. This is also how humans learn; By our Keep Philosophy Simple axis, it is a great place to start.

All changes in the launch - this - paper have several auxiliary objectives.

- **Stay Compatible for Paradigm Bridging Purposes**: Stick with things that drop into Vaswani to lower barriers to adoption and exploration.
- **Show the Discipline In Action**: Teach by doing what good Cognition Engineering looks like.
- **Stick with the Minimal Viable Archetype**: Ruthlessly prune all unnecessary complexity to provide a minimal research stub that can spawn an entire research branch which still fulfills all objectives. Later research may relax this constraint, but here we are intentionally aggressive in pruning to the minimal viable form.

2.4 Paper Overview

Both objectives and systems are well defined, satisfying all criteria needed for Cognition Engineering to begin. Note that significant theory engineering — do remember that in CE *Philosophy is Engineering* — is needed in places to develop the theory needed to support the technology, so these may not always be as simple as they appear. The simplest set of modifications we can propose to meet all objectives is:

- **New Field**: Avoiding the paradigm issues that lead the field of Generative NLP down a dead rabbithole for AGI is in scope. To avoid working at cross-purposes with existing technologies optimized for tasks solved suitably by the existing stack, we propose a new field. You are in the middle of seeing this in action.

- **Theory Development**: Sometimes, to align with the *Keep Interface Philosophy Simple* principle, we will introduce a new theoretical framing, and only later define how to apply it at the System Slot.
- **Pretraining Data Modifications**: For stronger abstraction incentives to break the death spiral and provide usable long-distance gradients. This will be expressed as Replay Pretraining. Note that multiple options worked and this was the simplest; we actually discarded two months of work once we identified **Replay Pretraining** as more suitable, showing off the *Don't Get Attached to Your Modules* principle.
- **Metric Addition**: The *Two-Pass Perplexity Ratio* metric is introduced using a compressive metric as a sign of model intelligence.
- **Attention replacements**: This is required to provide usable lifelong memory with time addressing, and provide the capacity to use the pre-training modification. The model should have capacity to make and refine a Knowledge Base that can accumulate through time. **Phase-Decay-Update** Memory has been identified as the Minimal Viable Archetype in the attention slot, and can be expanded further in other research.
- **Computation replacements**: The Mixture of Experts paradigm is broken. To fix it parameters are reused in *Cortex MoE* in a manner that produces positive expert reuse pressure and enforced exploration so the near-optimal experts are consistently found. This avoids the Devil's Bargain Hypothesis as the model now forms a more strongly connected graph of routing decisions rather than a brittle decision tree of staggering depth. It also is predicted to greatly reduce computation and parameter requirements in concrete implementations for a particular task. Note a single minor break from the pure vaswani interface is encouraged during initialization.
- **Alignment and Reasoning replacements**: This actually replaces two Vaswani slots. After being fused in theory into two sides of the same coin, the new Alignment and Reasoning slot is expressed in terms of *Constitutional Philosophical Self-Play.* This is done to provide economic incentives for reasoning and alignment. It also allows easy access for motives, personality, reasoning, and ethics principles integration by ethics domain experts and other parties such as lawyers and politicians. The self-play loop also has key instabilities flagged for the new branch of constitution-expression model correlation research.

All other items in the Vaswani stack were identified as satisfactory in their current format for field launch. When all replacements are completed it forms the Minimal Viable Cognition Stack, but a lab can adopt these piecemeal as desired. We removed a significant number of optional extensions to ease launch simplicity, though some of them are still discussed as research avenues in the extensions.

2.5 Conclusion

This completes the Cognition Engineering treatment of the Vaswani stack. The original system has been decomposed into actionable slots, objectives have been aligned with system-level cognition goals, and minimal viable modifications have been identified. From here, we move into engineering: each proposed change is now expressed in system terms, lowered to implementation-level detail, and evaluated as a Minimal Viable Archetype.

3 Replay Pretraining

The existing gradient manifolds in traditional generative NLP technology fail to create the long-sequence and deep abstraction pressures needed, due to treating all tokens equally. [21]. This requires modifications to bring our losses, data, and overall training incentives back into alignment for Cognitive Engineering purposes.

Reader Guidance

Warning: Replay pretraining is arguably the most *foundational* upgrade to the stack; we cannot promise your investment is worth it if it is not utilized.

Problem Solved:

- Improve long-sequence performance in my models.
- Synergize with next-generation recurrent or SSM models for exponential performance gains.
- Increase pretraining performance with one day's work by paying some extra compute cost.

Intended Audience:

- Pretraining pipeline engineers.
- Cognitive architecture designers.
- ML researchers working on memory, abstraction, and cognition
- Alignment/reasoning theorists.

Prerequisites:

- Familiarity with autoregressive LM pretraining.
- Understanding of operation of Next Token Prediction strategies on a tensor level.
- Understanding of training loops, and particularly scheduling.

Estimated Difficulty: 1/5

Very easy to implement. Can be dropped into most existing training setups in an afternoon, and scheduling extension is optional.

3.1 Three Sentence Summary

Standard Generative NLP teaches the model to predict the next token, but not to remember anything. Replay Pretraining fixes this by showing the same text twice, creating a pressure to retain useful information. This simple change builds the gradient incentives needed for abstraction and long-range reasoning.

3.2 Justification

Before proceeding, we would like to note this is the only nonoptional part of the stack upgrades. Failing to apply proper gradient pressure will result in suboptimal gains in every part of the stack. Replay Pretraining works by creating structured gradient pressure for memory and abstraction.

In standard next-token prediction, the model is trained to predict the next token given only the preceding ones. This setup creates strong incentives for local prediction accuracy but does not strongly reward retention of earlier content once its immediate predictive utility has passed: such tokens exist, but can be drowned out. The model has little reason to remember anything that does not help with the very next prediction, leading to a bias toward short-range dependencies and shallow reasoning.

Replay Pretraining changes this. When a model sees a sequence twice under next-token prediction, the second pass creates a direct incentive: perform better by remembering the first. But memory is bandwidth-limited, and not all tokens carry equal predictive value. The model is thus forced to selectively retain high-impact dependencies—those that are hard to predict and tightly coupled to later context.

These are the very tokens current systems tend to ignore:

- Cross-sentence links
- Long-range factual dependencies
- Abstract arguments and premises

By saturating the model's memory with this repeated input, we force it into a constrained optimization: compress the first pass into useful, general abstractions that help on the second. That means the gradient now favors abstraction—not just prediction.

In essence, Replay Pretraining turns memory into a compression problem, and uses the failure modes of naïve attention as an engine for smarter generalization.

3.3 Interface

Replay Pretraining slots into the pretraining data with a slight tweak in the training loop. It requires no or minimal changes with the Vaswani formulation, and no new training data.

3.4 Implementation Details

The approach consists of two changes: one in your dataloader or dataprep pipeline, and one in your training loop:

- The model is trained on a duplicated sequence — `[BOS] tokens... [SEP] tokens... [EOS]` — forcing it to predict each token twice. Tokens can be your normal Pile pretraining data, or something custom.
- Sequence length is gradually increased over training, eventually saturating time-mixing (attention) layers.

This creates a *compressive incentive*: the model can improve loss by abstracting information from the first pass and using it in the second. The scheduling ensures the model always trains at the edge of its abstraction capacity, maintaining consistent gradient relevance. This is not traditional curriculum learning—it's engineered gradient tension. We increase sequence length not to teach more, but to pressure the abstraction mechanism until failure, then let it adapt.

While traditional loss functions lightly encourage this, Replay Pretraining ensures usable gradients by progressively increasing complexity. In essence, we take the insight that Abstraction is Intelligence and turn it into a training exercise.

3.5 Extensions and notes

3.5.1 Economics

The modification is incredibly straightforward and economical.

- It is scalable and integrates directly into existing training loops.
- It requires no changes to data format, retaining complete compatibility with Big Data.
- It saturates the model's time-mixing capacity, encouraging abstraction.
- It enables an implicit curriculum via scheduled sequence-length growth.
- It avoids dead gradient zones even in recurrent models.
- It adds only a $\sim 2\times$ compute and memory cost (with linear attention).
- It operates with simple Next Token Prediction but is composable with more sophisticated losses.

In all, it is incredibly economic and extremely simple, things the Cognition Engineer seeks.

3.5.2 Synergy Complications

Note as well that for synergy reasons with the PDU memory - which operates in complex phase space - it is recommended to concatenate varying length articles during the schedule or truncate at various lengths throughout the batches, join without padding, then pad to the final length. Tf this is not done, the phase space memory system may simply learn the period of the replay—rather than compressing from first principles of intelligence. This is, in fact, an issue that must be handled in any cyclic memory encoding.

3.5.3 Alternatives

Replay pretraining is not the only way to achieve this effect. We emphasize it due to the extreme simplicity. However, other approaches are possible, such as sparse attention to tokens [18] and a few other lines of research. Notably, the loss lines of research are composable, as are better synthetic data lines. Both were considered and actually developed, but ripped out to comply with the Minimal Viable Archetype principle.

3.6 Conclusion

Replay Pretraining is a simple change that provides the foundation for strong gradient pressures that are aligned with the requirements of CE. It or something with similar properties is required for Cognition Engineering to function.

4 Two-Pass Perplexity Ratio

Misinterpreting our metrics is one of the reasons generative NLP went down the wrong path. Alternatives like ARC exist, but they're too slow for rapid iteration during training. We need a simple launch metric that measures abstraction ability effectively.

Reader Guidance

Problem Solved:

- My model predicts tokens but shows no sign of actual abstraction or memory reuse.
- I need a training-time metric that exposes how much a model is compressing or generalizing.
- Existing benchmarks like perplexity or accuracy are not surfacing the real cognitive issues.

Intended Audience:

- ML researchers focused on metrics and evaluation.

- Pretraining and training loop engineers.
- Cognitive architecture developers seeking early signals of abstraction.

Prerequisites:

- Familiarity with perplexity, cross-entropy loss, and next-token prediction.
- Understanding of memory bottlenecks and time-mixing limitations in large models.
- Basic ability to manipulate logits and token spans inside model outputs.

Estimated Difficulty: 1/5
Very easy to implement. Can be dropped into most existing training setups in an afternoon.

4.1 Three Sentence Summary

Existing technology asks the model if it can predict the next token. A better metric is to ask if the model learned something from when it saw the token. This can be measured using perplexity.

4.2 Justification

Intelligence is strongly correlated with the ability to compress and reuse information. [4] This relationship is not abstract—it manifests directly in learning systems: models that generalize well tend to represent information more compactly.

In current next-token prediction setups, however, perplexity is typically used only to measure raw prediction accuracy, not abstraction. But if we reframe the setup—by showing the model the same sequence twice—we can create a metric that surfaces internal reuse. If the model improves its prediction on the second pass, it must have retained something useful from the first. This improvement is not memorization; for long enough sequences, full memorization is infeasible. The model must compress.

This makes abstraction measurable. If the model prioritizes hard-to-predict or structurally important tokens, and uses them to improve predictions on the second pass, then we've created a training-time proxy for generalization pressure.

The Two-Pass Perplexity Ratio (TPPR) captures this by taking the ratio of perplexity on the second pass to that of the first. The lower the ratio, the more evidence the model is leveraging internal compression and abstraction. This turns a standard metric—perplexity—into a signal of cognitive potential.

4.3 Interface

This modification is intended to be training and model iteration support. It is an additional metric to examine like Accuracy or Cross Entropy Loss.

4.4 Implementation Details

Implementing Two-Pass Perplexity Ratio (TPPR) is relatively simple.

- Show the model the same token sequence twice, formatted as `[BOS] tokens... [SEP] tokens... [EOS]`. Optionally, you may prompt the model to expect a repeat or comparison.
- Extract the logits separately for the first and second occurrence of the token sequence, treating each as an independent prediction target.
- Compute the per-token perplexity for each section, then take the ratio: $\text{TPPR} = \frac{\text{Perplexity}_{\text{second}}}{\text{Perplexity}_{\text{first}}}$.

A TPPR of 1.0 indicates the model did not improve on the second pass—suggesting it retained no useful abstraction. A TPPR approaching 0.0 indicates near-perfect recall or generalization: the model was able to reuse earlier information to make highly confident predictions.

Note: Ensure that token alignment between both passes is preserved when computing perplexity. Be cautious with padding, truncation, or batching artifacts, as these can introduce noise into the metric.

It is expected that TPPR will vary with model size and architecture. For standardization we propose having standard evaluation lengths at 500, 5,000, 10,000, and 50,000 tokens to allow easy comparison between models

4.5 Extensions and Notes

It would be valuable to investigate whether TPPR correlates with benchmark performance more strongly than standard perplexity.

4.6 Conclusion

Two-Pass Perplexity Ratio provides the first minimal viable metric for evaluating abstraction during training. It is simple to implement, compatible with existing infrastructure, and gives early insight into whether a model is compressing and reusing information.

While TPPR does not capture general intelligence in full, it isolates a core cognitive function: selective retention under constraint. That alone makes it valuable for identifying models with high fine-tuning potential and structural coherence.

This is not the only way to measure abstraction—but it is the first solution simple enough to deploy at scale and early enough to guide training. In practice, it serves as a diagnostic signal: does the model support enough abstraction to continue full training, or does it require architectural rework? Practitioners should treat TPPR as a structural readiness check—not a theory of mind.

5 Timestep-Parallel Recurrence Theory

As shown in High[21], support for lifelong learning requires a memory mechanism capable of accumulating and abstracting over arbitrarily long contexts. Importantly, this memory mechanism requires the ability to control the rate of forgetting, like an LSTM can. This requires new theory to make recurrence economical. By building on top of prefix scan operations we can make a new generation of designs that are recurrent, time-parallel, economical, and can support infinite context lengths.

Reader Guidance

Problem Solved:

- I need recurrence, and I need it to be fast.

Intended Audience:

- Framework-level backend developers.
- Memory slot theorists.
- Long-sequence architecture researchers.
- CE system designers focused on lifelong learning capability.

Prerequisites:

- Fluency with prefix operations (e.g., prefix scan, cumsum/cumprod).
- Understanding of memory abstraction and recurrent model behavior.
- Systems-level thinking in the context of architecture design constraints.

Estimated Difficulty: 2/5
Engineering theory. No implementation is described, but the concepts require systems-level thinking and memory architecture fluency.

5.1 Three Sentence Summary

We must use a recurrent algorithm to avoid the lifelong learning issues identified in High, but the technology to do it economically is not mature. However, prefix scan algorithms can be executed in $O(\log T)$ time: this allows cumsums to be used in a new generation of recurrent algorithms that are both fast and recurrent. By developing this more fully we can make recurrent infinite-context models and remove the need for custom CUDA code.

5.2 Justification

To support true lifelong learning, we must overcome two core failures identified in High:

- Models eventually forget all prior context due to limited horizontal propagation of memory.
- Current architectures cannot control forgetting rates the way classical RNNs (e.g., LSTMs) can.

While solutions to these problems exist, they have historically been too computationally expensive to deploy at scale—violating the economics principle of Cognition Engineering.

Prefix scan operations offer a path forward. A prefix scan (such as `cumsum`) begins with an initial accumulator and applies a binary operation across a sequence, producing a new output at each step. While naive implementations run in $O(T)$ time, divide-and-conquer variants such as the Blelloch scan reduce this to $O(\log T)$ with parallel compute. Efficient CUDA implementations already exist for this purpose. [12]

This unlocks a new class of memory mechanisms that support fast, recurrent, and time-parallel updates based on summative behavior. These can propagate memory sideways across time steps—exactly the property required to break the forgetting bottleneck. Models such as RWKV, linear transformers, and other kernel-based attention mechanisms are already primed to converge on this paradigm. [1, 24, 26]

Most frameworks technically support prefix operations like `cumsum`, but implementations are often inefficient: PyTorch is a known example. Fixing this backend bottleneck is the first practical step toward fast, scalable recurrence compatible with CE principles. Scan-based memory systems are also amenable to activation recomputation—enabling long-sequence or even infinite-context training at a predictable compute cost (approximately $2\times$). See the Extensions section for further detail.

5.3 Implementation Details

Memory systems should be designed to operate over additive or multiplicative updates and implemented using prefix operations such as `cumsum` or `cumprod`. These enable fast, parallel recurrence and remove the need for sequential processing.

- **Memory Designers:** Ensure memory updates can be expressed as associative binary operations. Avoid explicit recurrence or timestep-by-timestep control flow.
- **Backend Developers:** Replace naive scan implementations. Optimize `cumsum`/`cumprod` for high-throughput GPU execution.

All future memory abstractions should treat scan-compatibility as an interface constraint.

5.4 Extensions and Notes

5.4.1 Identified Backend Issue

One immediate, concrete CE diagnosis is that PyTorch's cumsum implementation is not an efficient scan—this creates a backend bottleneck that blocks scalable recurrence. It appears to use a linear scan.

5.4.2 Infinite Context Training

A natural research extension is to follow up on the observation that Prefix Scan designs can train over infinite sequence lengths using activation recomputation. This fits into the recurrent timestep reversibility line of research, which has not been significantly cross-referenced with the mature transformer field across the specific strategy of timestep rather than layer reversibility. [19].

To see why, consider a model that begins from a base state S_k and accumulates a sequence of additive update vectors u_n across timesteps t to T:

$$U = \sum_{n=t}^{T} u_n$$

Then, any intermediate forward or reverse output can be recovered via:

$$O_t = S_k + U \quad \text{or} \quad O_t = S_{k+T} - \sum_{n=t}^{T} u_n$$

This is not layer-wise reversibility (which is already used for memory savings in some residual architectures), but true timestep-wise reversibility. The reversibility means, with additional research, such models can operate over infinite training lengths by recomputing their activations and reusing their memory, at a 2x additional compute cost.

5.5 Conclusion

Prefix scan technology, and particularly the cumsum and cumprod operations, are suitable for building a new generation of fast recurrent models that are capable of performing lifelong learning due to the desirable properties of having fast implementations, recurrent behaviors, and theoretical support for activation recomputation. When combined synergistically with upcoming developments, they can support a new generation of models.

6 The Phase-Decay-Update (PDU) Memory Archetype

The **Phase-Decay-Update** memory archetype, or **PDU Memory**, is designed to fulfill the Memory slot. It completely replaces attention. Built on top of the new Prefix Scan technology, it provides significantly improved decay control and temporal addressing——while remaining simple to implement and conceptually accessible.

Reader Guidance

Problems Solved:

- My model forgets long-term information too quickly.
- I need models with adjustable forgetting and phase-aware time addressing.
- I want my model to be able to redo reasoning under new assumptions retroactively.
- I want a memory system compatible with infinite-context, time-parallel training.

Intended Audience:

- Memory slot engineers and architecture designers.
- ML researchers working on recurrent or hybrid transformer-RNN models.

Prerequisites:

- Strong understanding of recurrent mechanisms (e.g., LSTM, GRU).
- Familiarity with complex-valued arithmetic, exponential decay, and positional encoding.
- Comfort with training-time vs inference-time tradeoffs and numeric stability.

Estimated Difficulty: 3/5
Fully detailed implementation spec. Requires solid ML systems understanding, numeric care, and architectural fluency. No custom kernels required, and conceptually simple. An engineering team could probably do it in two weeks. Fully understanding it requires knowledge of complex numbers.

6.1 Three Sentence Summary

Existing models forget too aggressively and have no way to reason over time or revisit earlier assumptions. PDU Memory solves this by introducing trainable forgetting and time-aware querying——built on phase-space recurrence.It extends traditional decay-add memory with complex rotation, prefix-scan updates, and a structured forgetting budget to stay stable and scalable.

6.2 Justification

The Devil's Bargain Hypothesis identified that any model relying on a fixed positional delta (e.g., RoPE, State Space Models) will inevitably suffer from irreversible forgetting: once context is lost, there is no mechanism to retrieve or revise it. The only viable path forward is for the model to learn how to control its own decay rates—requiring recurrence. Additionally, current time addressing methods are too crude for precise cognitive modeling, and it is difficult for models to "rewind" temporally when they've taken a bad branch. For example, when a model discovers a contradiction mid-sequence, it has no mechanism to update earlier assumptions—leading to shallow or brittle reasoning.

Complex phase space and cumsum-based recurrence solve both problems elegantly. This can be based on the same decay-add algorithm used in existing recurrent gates. With focused modifications, this enables the model to move forward and backward in time and dynamically adjust how prior knowledge responds to temporal pressure. The mechanism operates by rotating memory through complex phase space and taking the real component after all transformations are complete.

Naive implementations, while algebraically correct, are numerically unstable due to exponential scaling and would lead to underflow during training. To fix this, we introduce a memory hierarchy: lifelong and working memory, each operating under a pre-allocated forgetting budget. This reframes instability as a training constraint, enabling the model to learn when and how to forget. This separation also enables composable memory behaviors: short-term reasoning and long-term retention can be implemented using exactly the same class.

6.3 Interface

PDU Memory replaces the attention mechanism in the standard Vaswani block with only minimal required changes. Instead of a single attention module, the slot hosts multiple recurrent memory units operating in parallel, each tuned for a specific memory horizon. The Minimal Viable Archetype uses two: one short-range (working memory) and one long-range (lifelong memory). Each produces token-wise output in the same shape as attention and integrates cleanly through residual and dropout paths. While not strictly a drop-in replacement, the interface contract is close enough that it can be slotted in with minimal code changes——or wrapped to match the original API exactly.

6.4 Implementation Details

The core algorithm consists of an update phase and a query-based read phase. The update phase integrates new information, rotates in phase space as the model desires, and applies decay. The read phase produces a query and a phase, amplifies and isolates knowledge encoded at that particular phase, and ensures the results are normalized. We omit any form of linear attention and act purely between keys and queries, but note that NormAttention as in Devil would be compatible with this formulation if one is so inclined [26]. As is standard, it is intended for heads to exist, but they are not shown for simplicity.

6.4.1 Parameters and Projections

Certain projections are required in order for the core algorithms to run properly. At each timestep, let the model emit:

- Update vectors u_t
- Decay logits $d_t \geq 0$
- Phase logits p_t
- Query vectors q_t
- Query phases γ_t (for directional temporal querying)
- Query Sharpening β_t (for more strongly amplifying a particular phase)

Note that decay logits must remain non-negative; phase logits may take positive or negative values which allows the model to move forward or backwards through time. In addition, several parameters are introduced for training:

- ϕ: a learned **phase rate** controlling memory rotation through complex space
- ϵ: a learned **decay rate** controlling the exponential weighting of historical updates
- U_0: The initial memory state. Notably, $U_0 = \text{Parameter} \in \mathbb{C}^d$, and this explicitly should be trainable so the model can update its pretrained knowledge base.

6.4.2 Recurrent Formulation

The recurrent, or inference, formulation is easiest to grasp the intuition from.

- **Initialize**: Start up the state using $U_0 = \text{Parameter} \in \mathbb{C}$
- **Update**: Each update is performed using the algorithm $M_t = M_{t-1} \odot e^{-\epsilon \odot d_t - i\phi \odot p_t} + u_t$

This operates like a trainable Exponential Moving Average—but with complex-valued rotation controlling time traversal, and exponential decay controlling forgetting. In essence, *this means the model can move bidirectionally through time by adjusting the phase.* The decay and phase delta can be controlled by the model using d_t and p_t respectively from the input data. The real part at the end of this process is correlated with the relative, not absolute, phase at that location — an effect that is due to cancellation of exponents.

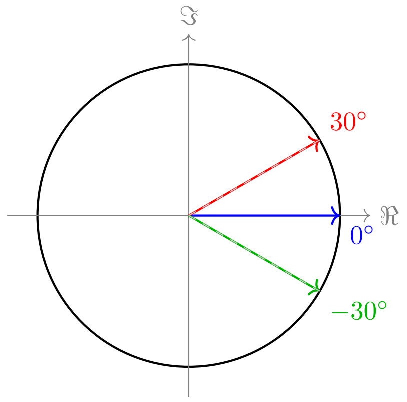
Inspiration for this has to be acknowledged to be due in part to RoPE [2] and the complex version used in xPos [28] in terms of using rotation and complex phase to encode time. Note that time is now traversable both forward and backward. In particular, observe that it is the case that the decay and phase factors are controllable independently, and though the decay is always going to either decrease or remain the same the phase has no such restriction. This means the model can learn to go forward through phase space to represent advancing time or even rewind if it needs to backtrack to a previous assumption. The decay term exists to allow the model to clean out useless information before it runs out of phase space addressing.

The read mechanism is similarly re-imagined:

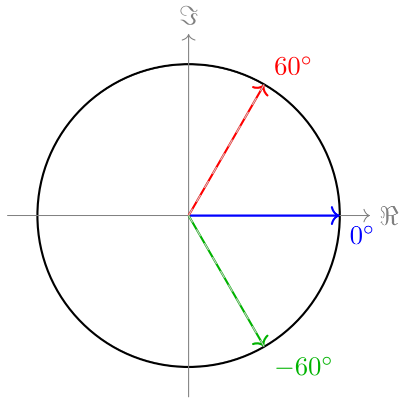
1. Adjust the read time as desired: $\tilde{M_t} = e^{i\gamma_t} M_t$
2. Sharpen the results to isolate or broaden the phase-space search area: $M'_t = \tilde{M}_t^{\beta_t}$
3. Execute the query and get the response. Keep only the real part. Observe we are using element-wise logic as in Attention Free Transformer for simplicity: $r_t = \mathrm{Re}(q_t \odot M'_t)$
4. Normalize the results, allowing the amplified results to nonetheless encode sane answers. $o_t = \mathrm{RMSNorm}(r_t)$

Using the observation that prior work in attention has found using the same tensor for keys and values is expressive, we just interpret the response to the keys — what we call the memories. Reading consists of rotating into the appropriate phase, adjusting spacing between keys, applying the query, and normalizing the result — skipping the use of values entirely. Notably, the sharpening trick that was used to great success in the Neural Turing Machine [11] line of research is again leveraged here with significant effect: it has the effect of letting the model re-space the positioning of its keys in phase space before executing the query due to the relationship between exponentiation and rotation of complex phase. This can allow the model to learn to focus only on one small chunk of phase space, or look at the big picture. Additionally, normalizing the output rather than carefully controlling the intermediate computation is inspired by Devil. [26], and is a modification that merges wonderfully with sharpening.

The net effect of this sequence of actions is to produce a design in which decay of information is controllable. The model can query forward or backward from time and even adjust a fuzzy context window to emphasize the aligned reals direction or look at the broader picture using the sharpening step. The effect of sharpening on phase space is somewhat difficult to visualize: While not exactly analogous the following is close enough to understand how sharpening allows the model to decide how wide a phase space to focus on.



(a) Before sharpening: Key phase states centered around the real axis.



(b) After sharpening: Keys move further from the real axis.

Figure 1: Left: A representation of each key phase state before addition as perceived at the query, focusing on the real axis. Right: The same key states after sharpening by a factor of 2. Note the movement away from the real axis. While the math is not identical when applied as exponentiation after adding the keys, the effect is similar.

6.4.3 Training Formulation

In order to hit the Economic axis, this must have an efficient time-parallel implementation for training purposes. By using the property that division of exponential bases is equivalent to subtraction of their exponents - that is, that $e^{a+b}/e^b = e^a$ - we are able to cleverly order the computation to be time parallel and only require a few cumsums. In particular

1. **Compute Cumulative Logits**: We compute the cumulative decay logits in reverse time order:

   $$D_t = \sum_{n=t}^{T} d_t \quad \text{and} \quad P_t = \sum_{n=t}^{T} p_t$$

   These can be done using an efficient cumsum scan by reversing the timestep dimension, cumsumming across it, then reversing back to the traditional order.

2. **Compute the Adjustment Factor**: The adjustment factors are simply an application of the above. We exponentiate, representing how much of the statement was forgotten from the perspective of the last timestep.

   $$F_t = e^{-\epsilon \odot D_t - i\phi \odot P_t}$$

   Notably, this is where the numeric underflow issues we mentioned in the introduction occur, why addition numerics care is needed, and why using complex space rather than just decay is so valuable.

3. **Compute the Emphasized Updates**: The emphasized updates exist in emphasis space and are formed by taking the updates at each timestep and multiplying by the adjustment factor without normalization. It is suspected emphasis space is going to become a more general theorist tool in the near future.

   $$U'_t = F_t \odot u_t$$

   This will deemphasize earlier updates in proportion to how much forgetting has occurred, and rotate in phase space as well. Note this assumes you have already concatenated u_0 into position.

4. **Propagate memory**: Performing a cumsum over the adjusted updates along the time dimension forward gives the preliminary memories

   $$L_t = \sum_{n=0}^{t} U'_t$$

   Since earlier material is smaller than later material, there is an implicit decay effect where later entries matter more if decay happened in between, but the magnitudes need to be normalized.

5. **Normalize memory**: By dividing the preliminary memories by F_t we can produce the final memories:

$$M_t = L_t / F_t$$

   This takes advantage of the fact that $e^{a+b}/e^b = e^a$ is algebraically equivalent and the multiplicative distribution law to remove the excess decay and phase application. The result is to ensure the inference version and training version behave exactly the same way.

The result of this is a computation that is equivalent to the recurrent formula at an algebraic level, but implementable in parallel on accelerator hardware with highly efficient cumsum-based recurrent logic. This formulation enables efficient training on accelerator hardware with drop-in framework compatibility.

6.4.4 Engineering Numeric Stability

This formulation alone is not enough to train, as there is still a key remaining issue. That issue is numeric stability. Rotations in phase space are essentially free, as they result in no scaling change; however applying decay is costly numerically. In particular, when exponentiating the decay logits if the cumulative decay logits are high enough the exponentiation process may cause an underflow that eliminates all training signals from timesteps before a certain point. This is not desirable. To address this, our proposal is to use chunking and a numerics budget.

The chunking consists of dividing the timesteps up into chunks; Such ideas are already well explored in previous models such as Transformer-XL [8]. Each layer has a target chunk length and padding is introduced if needed to reach a chunk boundary. Then during training the layer would execute the process in parallel across the chunks, but would not transfer information between the chunks. This is likely sufficient to allow exploration over short sequence, but is certainty not enough to permit the model to encode the lifelong memory that CE requires. It should be noted this chunk resetting is not needed or expressed during inference.

In pursuit of this, we also propose using a separate working memory and lifelong memory using training, and integrating a forget budget governed by a barrier loss.[33] We can use the known minimum representable values in `float32` and `float64` to compute the safe decay budget. Specifically, we require $\epsilon D_t \leq 78$ for float32, or $\epsilon D_t \leq 699$ for float64. [1]. For reasons that are obvious now, we recommend using a limited amount of float64 memory as memory tensors. These budgets can then be put into the following canonical barrier loss to motivate the model to stay under them while avoiding performance issues the rest of the time:

$$\mathcal{L}_{\text{barrier}} = -\log\left(B_{\max} - \text{clamp}(D_t, 0, B_{\max} - \kappa)\right)$$

[1]This assumes we do not want to underflow and is computed as $min * 10000 = e^X$ to include a buffer zone

Where $B_{\max}$ is the maximum allowable budget like 78 or 699. κ meanwhile is a small numeric epsilon to prevent infinity and instead set the loss at a very large value. This has fantastic synergy with the rest of the system.

- It encourages the model to encode time using phase, which is better for most purposes anyhow.
- The model itself can learn where to apply its limited forgetting abilities.
- The same class can do both working and lifelong memory using different instantiations with different chunk sizes.

The last point bears particular emphasis. For the Minimal Viable Archetype we envision using the same class for two different layers, one encoding working memory and one lifelong. The working memory would have a short chunk length, like 1024 tokens, and reset on the boundaries. As long as computations are short it statistically does not matter if a computation is reset across a boundary over enough training examples, and it gives the model the ability to specialize its memory space. Meanwhile, the chunk length of the lifelong memory would be set to be the longest length of sequence you plan to train on. You can then let the model figure out how to fit that into the finite space available.

6.4.5 Heads

Heads should be used exactly as in the Vaswani formulation: independent per-head projections with parallel computation and concatenated outputs. Each head operates its own PDU memory instance, allowing specialization over different temporal scopes or abstraction strategies. The use of multiple heads is not just compatible but highly synergistic, enabling compositional memory representations and enhancing phase-space expressivity.

6.5 Extensions and Notes

There are a few things of note.

6.5.1 Emphasis Space

The development of an *emphasis space* add-decay formulation that is timestep-parallel is a formal theory development that likely deserves a few words.

Restatement In emphasis space, we represent decay-weighted accumulation using exponentiated cumulative decay logits and additive updates. Specifically, given per-timestep decay logits d_t and updates u_t, we define the cumulative decay factor F_t as:

$$F_t = e^{-\epsilon \sum_{n=t}^{T} d_n}$$

The corresponding emphasized updates are then:

$$U'_t = F_t \odot u_t$$

And the cumulative state in emphasis space is:

$$L_t = \sum_{n=0}^{t} U'_n$$

Dividing by F_t can then map back to the original sequence space.

Implications This representation has several key properties: it is purely additive, trivially parallelizable via prefix scan, and activation-recomputable by construction. By working in this space, we avoid the fragility and sequential bottlenecks of traditional recurrent formulations, while maintaining controllable decay dynamics.

From a theory perspective, this means we now have a clean formal space for designing memory structures, auxiliary losses, and regularization schemes. Mapping a memory system into emphasis space is not just a training trick—it is a general theoretical operation that exposes tractable handles on time, memory persistence, and abstraction dynamics.

We expect this concept to generalize far beyond the PDU formulation. Any system that expresses additive accumulation under exponential decay—especially in log-linear form—can likely be reframed in emphasis space. It is flagged here for theorist use and expected to recur in future cognitive system archetypes.

6.5.2 Activation Recomputation

Activation recomputation, it should be noted, is executable with this strategy but will require some additional considerations. Fortunately, in the emphasis space formulation before normalization and before taking a cumsum there is not a strong distinction between directions. Simply cumsumming against the timesteps, subtracting the results from the final state, and normalizing per the usual should have the effect of removing all the emphasis and allow very simple activation recomputation in emphasis space. Future work can explore this.

6.5.3 Deep Hierarchial Memory

Further research can specialize deeper hierarchical memory for phenomenally long or fine-grained understanding. There is nothing that says you cannot try a few different chunk lengths once you get above hundreds of thousands of tokens of processing.

6.5.4 Synergy Complications

PDU Memory is explicitly designed to synergize well with the other components. This includes Replay Pretraining. However, replay pretraining actually has a complication. Since phase space is periodic, and replay pretraining operates by consistently repeating the same short segment, it is recommended to union your training data without padding then pad to the final length. Not doing this may let the model exploit the periodic structure of its memory without actually learning how to abstract better.

6.5.5 Read-Induced Forgetting

Earlier versions of this system experimented with a consolidation pressure: reading from memory incurred a decay penalty, encouraging abstraction and compression before retrieval.

Though ultimately replaced by Replay Pretraining for its simplicity and gradient signal quality, the mechanism remains cognitively inspired and may be worth revisiting in systems aiming to model working memory resource constraints or simulate human cognitive pressures. This could be reintroduced and explored in subsequent work very straightforwardly: Simply introduce decay modifications scaled by the query itself.

6.6 Conclusion

PDU Memory resolves all existing identified issues with the memory slot. Information can accumulate horizontally, avoiding that aspect of the Devil's Bargain. Time is addressable and manageable as a first-class citizen. The economics and implementation are compact and sane. This proposed layer, drop-in compatible with the Vaswani formulation and supported by every framework, will serve as an excellent Archetype to launch an entire wave of research under the memory slot of Cognition Engineering.

7 Modern Computation Theory

Reader Guidance

Problems Solved:

- My model is scaling poorly and catastrophic forgetting is getting worse. I need a fix on a budget.
- I want theory. Why are our models failing to scale, and how do we need to change to fix it?
- What does it mean to think about engineering intelligence ?

Intended Audience:

- Cognitive architecture theorists.
- ML systems designers seeking budget-performance abstraction scaling.
- Researchers attempting to optimize performance/FLOP.

Prerequisites:

- Familiarity with MoE models and gradient manifold concepts.
- Moderate experience with performance tradeoffs in accelerator-constrained systems.
- Optional: Understanding of how transformer feedforward layers map to graph-based computation.

Estimated Difficulty: 4/5
Dense theory section. Requires strong conceptual fluency and systems thinking. Core design principles here are later reduced to modular engineering specs.

7.1 Three Sentence Summary

Current feedforward architectures suffer from exponential compute growth due to lack of abstraction reuse—a core failure identified by the Devil's Bargain Hypothesis. Cognitive Engineering addresses this by shifting from monolithic routing to composable, reusable neural functions, architected for high-performance reuse and wired together by biological exploration pressures. This section lays out both a minimal patch and the deeper theory required for long-term scalability.

7.2 Justification

The feedforward Vaswani slot has seen minimal innovation, with Mixtures of Experts being the most recent. That will now change.

Under the Devil's Bargain Hypothesis [21], we identify **Decision Tree Explosion** in the computation slot as a core driver of unsustainable scaling costs. As shown in High [21], models only marginally succeed by *accidentally* reusing computation—generalizations occur not through pressure, but chance.

This is unacceptable for Cognition Engineering. Getting high performance-per-parameter is not a luxury, but *a survival constraint* due to the exhaustion of high-quality pretraining data. We require architectures that encourage reuse and reinforcement of abstraction by design—not as emergent side effects.

To achieve this, we propose models that operate as if using a shared *Neural Functional Library*: a global store of composable primitives accessed across contexts and reused wherever applicable. In a biological analogue, the model would then learn to wire into this global framework by exploration. Experts are no longer monoliths; they become shards—recombined on demand like PyTorch ops to match the task at hand.

This strategy increases training signal per abstraction and compresses compute into a more reusable form. But it introduces engineering costs—particularly around routing, caching, and hardware throughput—that must be explicitly addressed by prefetching and ensuring cache predictability.

In essence, we call upon a small well-tested A-Team of neural functions rather than one of thousands of third-rate contractors: all other difficulties come from making this happen.

7.3 Theory Details

7.3.1 Note for Theorists

The core pathology addressed here is Decision Tree Explosion. Transformers are encoding, using their feedforward mechanisms, a superposition of vectors that are mapped: This effectively makes them graphs conditioned by context. However the existing training situation effectively encourages them to build specialized and long decision trees rather than shorter general graphs with routing decisions.

The fix which has been identified back in High [21] and applies to the feedforward slot:

- Models need positive consolidation pressure: The model needs to be encouraged to reuse existing nodes rather than add new ones.
- Models need negative exploitation pressures: The model needs dropout or other processes to force it to explore other computation pathways.

7.3.2 The Quick Fix

There is a *Quick Fix* which is predicted to restore some level of performance in the existing Vaswani Paradigm. It is predicated on the observations noted in High [21]. It is not, as mentioned, predicted to provide the same level of FLOP improvement as the final replacement, but if you are on a budget or skeptical you are welcome to give it a try:

Using existing technologies, the positive consolidation pressure is provided by implementing sharing of experts - the best example of how to do this is DeepSeek-MoE [7]. This provides a small, well-tested library of utility functions the model will frequently want to consult with, and which will always be available. This breaks the Decision Tree Explosion issue by encouraging the model to use an expert subset which is designed to be composed. This in turn means the model is forced to make a routing decision after using it, producing shorter computation chains.

The negative pressures are also needed. Essentially, your Feedforward system is establishing a very strong correlation between active hidden states on subsequent feedforward layers that makes it impossible to jump between graph pathways. Apply dropout on the hidden state in the feedforward system, and possibly logit dropout[2] on the MoE routing logits to discourage this practice.

We offer no firm guarantees of long-term scalability for this fix, as it will likely degrade once the model begins redundantly encoding bypasses to evade

[2]That would be setting a subset of the selection logits in the MoE process to negative infinity not zero, so selection cannot choose them

dropout. Still, it should significantly delay the onset of brittle graph expansion and allow you to scale your models more efficiently in the near term. This fix will not carry forward into the final Archetype, but the principles it demonstrates–—reuse scaffolding and controlled exploration——remain foundational.

7.3.3 The Cognition Engineering requirements

We require something much more capable, however, for Cognition Engineering. The fact we are out of training data means we effectively only have the option of getting more performance/parameter to avoid running into the insufficient data corner of the scaling laws established in Kaplan. [16]. The remainder of this theory will explore the requirements to achieve this performance increase, based on the available data and technology. The design emerges from four intersecting pressures

- We need sharing and regularization to dodge the devil's bargain hypothesis
- We need to fix the expert gradient manifold to not be so restrictive
- We need to do this while still being performant.
- We need to do so without breaking Vaswani slot compatibility

Unfortunately, meeting all four objectives was not possible. As such, we selected the lesser evil. This design breaks partial Vaswani compatibility—by necessity. The gains in composability, reuse, and interpretability far outweigh the minimal interface cost. In particular, there will be a need to pass in extra parameters to the feedforward layers.

7.3.4 Better Gradient Manifolds and Composability

The gradient manifolds and broader exploration incentives in the modern Mixture of Experts systems are identified as insufficient. Additionally, monolithic experts are not very composable. Both need to be fixed.

The frequent generation of saddlepoints rather than local minimums in high-dimensional space embedding is a well-known property used to operate under the assumption our loss functions are convex.[23] However, this only works if the space is actually high-dimensional. The reduction of expert choices down to one or two options, as in most MoE implementations, breaks this assumption. It is highly likely the models are getting stuck in local minimums early in training, allowing redundant pathways to accumulate without gradient pressure to consolidate them.

The solution is to make smaller expert shards consisting of feedforward bottlenecks and then select more of them. In a process somewhat analogous to generating heads, feedforward can be decomposed into a set of bottlenecks retaining full mathematical equivalent: See appendix B on page 95 for the full details. This means rather than selecting 1 expert, you can instead select 10 expert shards for the same execution cost. Experts instantly become much more composable and so long as you perform a weighted average of the results the gradient manifold has way more dimensions to work with, solving two problems with one stone. In essence we break the problem into selecting a set of *bottleneck shards*, rather than a single monolithic transformation, which increases the gradient manifold to have sufficient dimensions to produce saddlepoints rather than false minima.

7.3.5 Better Insight Consolidation

To produce insight consolidation that may produce AGI we also prepare sharing these composable shards globally among all layers. With sufficient exploration pressures, global sharing of all experts is the ultimate endpoint of positive consolidation pressure. This turns the problem into *building the perfect expert* out of a framework. In analogy to a framework, the gradient updates then act to focus far more gradient updates on a smaller number of central framework experts rather than letting each layer make its own framework. Letting the keys representing the shards also be shared produces the same effect on the model's understanding of what each expert can do, and produces a *global address space.* In essence, the model builds an adapter in its layer, and thus insights on how to do better can be accumulated from multiple locations.

7.3.6 Better Exploration and Performance

Unfortunately, the adoption of these mechanisms have dire performance implications. Traditionally in a MoE you would select around ten percent of the experts. That would be utterly impractical when we have flattened all experts into a global address space then multiplied by 16 during shard decomposition. Cache nonlocality, pure computation, and other effects will bring such a system to its knees. This necessitates additional complexity.

To avoid this, with incidental benefits on interpretability, we will use a mechanism known as *connectome biases* which encode a probabilistic prior of how useful an expert is expected to be to a particular layer. This provides an interpretive surface with all the implications:

- We can as scientists examine the priors on each layer to understand how the model is tending to wire its framework together to deal with the task
- We can let the model specialize a global collection of keys and shards by just defining and training a set of connector biases for each individual layer as a scalar per expert. This is now all the action it needs to define its interface

More importantly, it now provides meat for certain performance and exploration optimizations that provides the module with a acceptable level of performance with satisfactory exploration.

- Because the connectome biases encode the logprob priors of the model, we can perform a topk mechanism to only consider a subset of the entries under full attention keying. We call this *prefiltration.* Better yet, it is possible to *prefetch* them all based on the index pattern in one sunk cost at batch start, preventing all but one L2 cache stall by effectively making training consist of interacting with localized tensors.

- Restricting ourselves to this small subset would torpedo exploration. However by performing **connectome nudging** — very slightly increasing the connectome biases on *unchosen* shards during training — we can force the model to explore other framework shards and either swap or revert. Since unchosen experts do not produce any gradient this has no gradient effect until a dead expert comes alive again. With enough training, it will explore the entire connective space despite being sparsely wired.
- Using the two-stage prefetching system, we can prefetch the subset of parameters ahead of time when building the inference model and not have to pay the full dynamic cost. In effect, we are dynamic during training, but have the full speed of static routing during inference.

Training these biases is then as easy as adding them to the MoE selection scores after attention but before selecting the expert shards. It lets the global keys be specialized to whatever the local layer purpose is. This ensures rather than having to process, say, 100000 experts and find the right one - which would hit L2 cache limits on the GPU - it only has to prefetch a subset then process say 100. This restores sufficient performance to be viable and thus concludes development of a Minimal Viable Cognition unit that hits all axes needed.

7.4 Extensions and Notes

7.4.1 Connectome Nudging as a General Pattern

It is worth emphasizing that *connectome nudging*—incrementally increasing the bias of unchosen pathways to ensure structured exploration—is not specific to this implementation. This is a general cognitive systems strategy that can trade speed of convergence for sparsity without sacrificing exploration over long training lengths.

Cognition Engineers should recognize it as a broader tool: any sparsely activated system with selectable interfaces can benefit from this pressure. Whether applied to expert selection, routing logic, or memory access pathways, nudging ensures the system doesn't ossify prematurely and continues to explore alternatives even while optimizing known solutions.

In future archetypes, expect this pattern to recur wherever controlled exploration over sparse, reusable abstractions is needed. We note it here as a reusable design axis.

7.4.2 Connectome Prefetching

It is worth emphasizing that connectome prefetching is likely another excellent general design pattern.

7.4.3 Breaks with Vaswani System Slots

It is worth emphasizing again a minor break with the Vaswani system slot is required. The prefetcher will return an iteratable object, that object will then have to be fed through Decoder layers into the implementation. However, it was worth it: In essence, we just pick a bunch of low hanging fruit all in a row.

7.4.4 Fast Bias Programming

We originally incorporated a Fast Bias Programmer into the module between the prefiltration and the integration of the connectome biases into the routing score. This modification emitted bias updates for each timestep and batch, then efficiently cumsummed the result and normalized: This provided a model with routing decisions that could rewire its brain permanently on the fly.

This was removed in light of the fact that:

- It is complicated and requires hidden state in evaluative mode on the Feedforward Drop in, complicating the replacement contract.
- The PDU memory unit can already decide to remember something permanently, and we value synergy between simple pieces over complexity.

It is believed it will still help to avoid the issues of a model forgetting earlier directives; nonetheless, given the intention of producing a teachable collection of objects it has been removed and left to future work.

7.5 Conclusion

Modern Cognition Theory may bring us the tools we need to build effective and intelligent models. However, we now focus on a more specific question: How do we actually build this?

The proposed modification is complex enough that it warrants a clear, object-level architectural breakdown and formal lowering theory into the objects. We now shift from system theory towards a concrete software-hardware interface spec—expressed as modular components with well-defined responsibilities. What follows is the implementation of these threads in **Cortex MoE** and the broader practice of **Cortex Engineering**.

8 Cortex Engineering

8.1 Reader Guidance

Problems Solved:

- I do so much training engineer-time is cheaper than compute-time. Can you give me exponentially better performance/Flop at a project-level investment?

- I want to understand how you engineer a neuro-plastic mind, not a robot.
- How can I fix the exploration issues with sparse routing and modern mixtures of experts in practice?
- I have no idea how to build the monster you just described, or even the engineering technology you are proposing behind it.

Intended Audience:

- ML architecture engineers pursuing better performance at any cost.
- Systems-level researchers designing reusable expert routing or compositional inference in pursuit of much smarter architectures.
- AGI and cognition theorists seeking biologically-inspired computation with modularity and flexibility.

Prerequisites:

- Strong understanding of Mixture-of-Experts architectures and their training dynamics.
- Strong understanding of advanced vector indexing.
- Familiarity with routing, sparse computation, and performance constraints on modern accelerators.
- Comfort with modular systems design, parameter sharing strategies, and differentiable control structures.
- Familiarity with caching and GPU performance issues.

Estimated Difficulty: 5/5
Highly complex architecture. Requires coordinated engineering and theory across routing, gradient design, GPU memory access patterns, and interface stability. Not for minimal setups—this is the path to maximal intelligence-per-FLOPs and possible actual AGI. It is implementable in two weeks as everything is engineered into object specs and even code in places, but only by an A-team of senior engineers across several disciplines.

8.2 Three Sentence Summary

The proposed upcoming Cortex computation model is powerful—but extremely complex—and lowering it to an implementable form requires care, context, and the right background. Cortex Engineering Theory provides the missing bridge: a massive engineering primer that provides the overview, notes the architectural novelties, needed experience, and tools that will be used in the implementation specification. This will be lowered to a full per-object specification in the next section. Cortex Engineering is to modern MoE what CUDA was to matrix multiplication: not a new idea—but a complete reframing of *how* we do it, *why*, and *with what hardware constraints in mind.*

8.3 Justification

The Cortex computation model is powerful—but extremely complex—and cannot be dropped straight into a codebase without a deep understanding of the architectural shifts and engineering tools it requires. This section exists to bridge that gap: it lays out the practical engineering context needed to make sense of the design, with just enough detail to make the upcoming spec implementable by a competent team.

Concretely, this section equips the reader with:

1. A clear understanding of how ensemble-based computation can replace traditional feedforward layers.
2. The logic and motivation behind vector-indexed expert selection and prefetching.
3. The role and training behavior of connectome biases in expert routing.
4. The modular decomposition of the Cortex system into engineering objects with clean interface boundaries.
5. How performance, exploration pressure, and gradient quality interact within this system.

This is not the spec itself, but it defines the system. If you're about to build CortexMoE, this is the part you study first—before writing a line of code.

8.3.1 Pedagogical Moment

This section performs a lowering theorizing operation: it explains the theory behind the actual lowering execution.

Cognition Engineers do not jump straight from theory to code, but treat theory as code, and must justify every decision. We first define the objects and interfaces that must exist, validate that they can compose to meet scope, and only then lower again into formal build specifications after all decisions are justified

If you are expecting interface definitions here, you are too early. Consider jumping to 55, and observe the lowering process that is characteristic of Cognition Engineering being performed in this section.

8.4 Implementation Details

8.4.1 Major Skill Sets

Certain skillsets besides the basic Machine Learning assumptions are needed or greatly assist in implementing this design. Make sure you have experts on your team familiar with.

- **Sparse Logic**: You should be used to handling sparse logic manually, and particularly mixture of experts

- **Advanced Indexing**: Numpy-style advanced indexing with vector indexes is a core part of the upcoming implementation.
- **Ensembles**: Understanding the idea of running layers in parallel using an ensemble is extremely useful, as we leverage it fully.
- **Mixtures of Experts**: Naturally, we will assume you are familiar with mixtures of experts as well.
- **Mathematical Index Notation**: We will sometimes represent operations like matrix multiplication in terms of index mathematics. Be at least mildly familiar with them.

The following skills are optional, but highly beneficial for understanding not just how the layer is implemented—but why it was designed this way:

- **Gradient and loss design**: Many of the design decisions only make sense when you understand how losses work.
- **GPU and Hardware**: Understanding why and how GPU's have issue, particularly involving the L2 cache decisions, makes why we use prefetching much clearer.

8.4.2 Terminology

The following terms will make this discussion much clearer

- **Global Address Space**: All the various expert shards that are available in theory to each layer of the model.
- **Partitioned Address Space**: The subset of expert shards that are available to a particular layer after prefetching.
- **Selected Address Space**: The expert shards that actually make it to being executed then being combined into a result.
- **Prefetching**: Something that reduces a computationally intensive global address space to a smaller partitioned space efficiently. Happens at the start of the batch.
- **Selection**: Selecting the right shards out of the partition experts. Happens when the layer runs
- **Execution**: Running the right shards.
- **Shard**: Instead of monolithic experts, we break the feedforward down into an equivalent number of bottlenecks and select a large number of shards that are composed and run in ensemble

Additionally, we will frequently refer to certain abbreviated index variables when discussing shape contracts. All indexing variables, and what they mean, are listed here

Name	**Purpose**
e	Indexes over elements in the global address space.
e'	Indexes over elements in the partitioned address space.
e''	Indexes over elements in the selected address space.
b	Indexes over the batch dimension.
t	Indexes over the timestep dimension.
l	Indexes over the embedding dimension.

8.4.3 Engineering Module Overview

There are of course multiple ways to lower a spec, but we identify the following design to be clean, focused, and modular. We will specify four objects. Three of them are critical to implement the logic itself, but the prefetcher is critical to executing performantly. We will go into each in more detail shortly:

- **ShardEnsemble**: A shared repository of global shards collocated with execution logic. Has prefetching responsibilities and later execution responsibilities. Designed to be dependency injected into CortexMoE
- **Selector**: Responsible for tracking the connectome biases, nudging, and specializing the global shards to the tasks the layer needs to actually do the immediate job. Has a prefetching responsibility in terms of figuring out the prefetching subset.
- **CortexMoE**: The drop-in replacement for the Vaswani Feedforward systems slot. It is primarily glue code coordinating two important functions. First, it orchestrates prefetching. Second, once the time comes to run it also performs the actual Cortex MoE processing.
- **ShardPrefetcher**: The modifications introduced would make the model very unperformant. The ShardPrefetcher fixes that. It turns what would be a bunch of calls with poor GPU locality into a sequence of calls that all work on the same tensor. This provides vastly better GPU performance. Responsible for managing prefetching and containing the shards. This is written to be implementable in most frameworks, but unfortunately does require some minor logic tweaks in your model and decoder cell. The prefetched results would be fed in, one to one, to their correlated layer.

Note that the architecture implementation is under development at github based on the appendix A at page 95 but is not guaranteed to be functional in this paper.

8.4.4 Biological Insight

Traditional transformer architectures alternate between memory (attention) and computation (feedforward) neurons, with limited reuse and hard-coded routing. Mixture of Experts extended this with sparse computation but retained a rigid partition between expert neurons—leading to siloed learning, poor gradient flow, and minimal modularity.

CortexMoE reimagines this structure by separating memory, routing, and computation neurons into explicit modules. These are connected by a trainable "connectome"—a bias-guided mechanism that encourages reuse, modularity, and intelligent routing of computation across the model's entire expert set.

This architecture mirrors biological systems closely: any computation neuron can evolve its connections to connect to any other, routing decisions evolve over time, and specialization emerges through dynamic reuse and enforced exploration. In short: the model isn't just learning tasks—it's learning to wire its brain into a framework.

8.4.5 Implementation Weirdness

In Cognition Engineering, complexity is a budget. And here, it's spent with purpose. Two major sets of novel weirdness should be expected. The first is representation of feedforward networks as bottlenecked ensembles that are constructed by vector indexing into ensemble collections. This is mainly for performance reasons. In addition to this, however, is very careful manipulation of how we select the expert shards and what losses we rate in order to optimize the gradient manifold for best exploration-exploitation. These both motivate significant additional complexity whose reasons might not be immediate obvious.

8.4.6 Ensembling Using Vector Indexing

The shard ensemble requires some unconventional vector indexing logic to allow for execution to operate. Between sharding, global addressing, and time-batch addressing the model would be incredibly performant with three nested for loops. Instead, we rely on vector indexing to construct and execute ensembles.

While normally tensors are broadcasted across their dimensions, there is nothing that says one cannot provide extra information across any extra dimensions. This line of code shows the idea:

```python
import numpy

# First case. We have different kernels for each dim, and must use a
# for loop once the experts are selected
# This is fairly similar to how we normally deploy mixture of experts.
# Note this is going to become ridiculous with three four loops if we allow
# time and batch as degrees of freedom or pump up the number of experts.
```

```
batch_dim = 10
ensemble_dim = 4
embedding_dim = 5
num_chosen = 2
mock_data = numpy.random.randn(batch_dim, embedding_dim)
mock_kernels = [numpy.random.randn(embedding_dim, embedding_dim) for _ in range(ensemble_dim

selected_experts = numpy.random.randint(0, ensemble_dim-1, [2])  # Experts chosen
weights = numpy.random.randn(ensemble_dim)

output = numpy.zeros((batch_dim, embedding_dim))
for expert in selected_experts:
    kernel = mock_kernels[expert]
    output += weights[expert] * numpy.matmul(mock_data, kernel)

# A performance gain can be reached by vector indexing into what now would be stored
# on our layers as parameter ensembles, then executing those over an additional ensemble
# dimension. This completes the process in one go. We do need some extra
# dimensions as the matmul rules get more complex though.

mock_kernels = numpy.stack(mock_kernels, axis=0)
# Vector indexing out the parameters into an ensemble subset happens.
# Usually you would have multiple lines for all the kernels
selected_kernels = mock_kernels[None, selected_experts]
selected_weights = weights[None, selected_experts, None, None]


instance_data = mock_data[:, None, None, :]
unweighted_output = numpy.matmul(mock_data[:, None, :, None, :],
                                 selected_kernels).sum(axis=-1)

# Outputs are combined
output = numpy.sum(unweighted_output*selected_weights, axis=1)

print(selected_kernels.shape)
print(instance_data.shape)
print(selected_weights.shape)
```

Note Numpy will happily execute the second case, and in a full framework it provides support for building functional kernels that are executed out of by collecting an ensemble by vector indexing. We can even vector index with time and batch dimensions to make ensemble subsets per batch and timestep, something we do indeed do in pursuit of lifelong learning synergy. You may see appendix C on page 96 for a more detailed example of this if desired.

The overall architecture consequences of this are that rather than using a for loop over a list of experts, we instead pass along the indexes correlated with the experts we want to run to a layer, along with weights to combine the results with. This is exactly the interface that ShardEnsemble will use. However, notably, it also opens up an additional possibility. This is prefetching

8.4.7 Prefetching

Prefetching is needed due to the fact that we have many smaller shards, and a global address space. Without using some performance tricks, performance would slow to a crawl. Prefetching is what is used to avoid this.

The basic idea is that rather than only doing one vector index into the parameter collection, we do two. However, the first one depends on trained parameters, that are updated by some training process, and thus are predictable for all layers at the start of the batch. We can then consolidate all the very-expensive fetches off the global parameter collection into one prefetch at batch start, then have each layer only choose from among the "partitioned" subset that was predicted to be most useful.

This is exactly what happens. It is also a general-purpose training trick that will likely be useful in other architectures as well. Rather than indexing into the global subset, we make indexes into the partition and pass the prefetched kernels into the CortexMoE layers when it is time to use them.

This does, unfortunately, break the Vaswani contract lightly. Namely, you must now run the ShardPrefetcher at the beginning of the batch to get a list of prefetches per layer, then pass those through your DecoderLayers into the CortexMoE instances. However, that is one additional layer at the front of the model, and then some argument passthrough, which is not a particularly intense software engineering challenge.

8.5 Extensions and Notes

8.5.1 Fast Bias Programming

As previously mentioned, a fast bias programmer that updates the connectome bias can be inserted in the Selectors .select method. This would be able to use cumsums and normalization to adjust the priors on the fly, effectively letting the model rewire its brain in a lifelong learning manner. It is also left to future work.

8.5.2 Connectome Bias Low Rank Tuning

The connectome biases, in essence, encode the priors the models uses to know how to apply its tools. The preconceptions if you will. This means a wonderful extension in the LoRA fine tuning vein would consist of freezing the model, resetting the priors, and then having the model figure out what connectome network best applies its well-tested shard toolkit to the particular problem at hand.

8.6 Conclusion

Cortex Engineering exists to make the Cortex computation architecture tractable to build. While the full system is ambitious, modular decomposition and the right engineering techniques make it possible to realize with a focused, high-skill team. This section has laid out the necessary context, background, and conceptual infrastructure—equipping you not only to understand the upcoming specification, but to implement it competently and with confidence.

In the next section, we lower these ideas into a concrete object-level architecture specification, complete with responsibilities, interface contracts, and pseudocode. If you've understood this section, you're ready to code.

9 Cortex MoE

9.1 Reader Guidance

Problems Solved:

- I do so much training engineer-time is cheaper than compute-time. Can you give me exponentially better performance/Flop at a project-level investment?
- I want to understand how you engineer a neuro-plastic mind, not a robot.
- How can I fix the exploration issues with sparse routing and modern mixtures of experts in practice?
- I am a ML god and just want to code something awesome without reading Cortex Engineering.

Intended Audience:

- ML architecture engineers pursuing better performance at any cost.
- Systems-level researchers designing reusable expert routing or compositional inference in pursuit of much smarter architectures.
- AGI and cognition theorists seeking biologically-inspired computation with modularity and flexibility.

Prerequisites:

- Strong understanding of Mixture-of-Experts architectures and their training dynamics.
- Strong understanding of advanced vector indexing.
- Familiarity with routing, sparse computation, and performance constraints on modern accelerators.

- Comfort with modular systems design, parameter sharing strategies, and differentiable control structures.
- Familiarity with caching and GPU performance issues.
- Optional but recommended: Reading section 8.

Estimated Difficulty: 5/5
Highly complex architecture. Requires coordinated engineering and theory across routing, gradient design, GPU memory access patterns, and interface stability. Not for minimal setups—this is the path to maximal intelligence-per-FLOPs and possible actual AGI. It is implementable in two weeks as everything is engineered into object specs and even code in places, but only by an A-team of senior engineers across several disciplines.

9.2 Three Sentence Summary

The CortexMoE layer is a biologically-inspired layer system designed to replace the feedforward slot with a modular system which can dynamically rewire neural connections from a Neuron Function Library. We decompose this into four engineering objects—Selector, ShardEnsemble, CortexMoE, and ShardPrefetcher—each with clean responsibilities. Where behavior is unintuitive or sparse execution complicates implementation, we provide minimal code fragments to clarify system-level design.

9.3 Justification

When we left off back in section 7, we were left with something which is theoretically pleasant, and an engineering nightmare. It may indeed be possible to deploy this using a six-month A-Team, but not over the promised two weeks of this Cognition Engineering launch document. This document consists of a lowered spec intended to change that.

Meeting the objective will consist of breaking down the responsibilities into engineering objects with clear and well-defined responsibilities, and showing the interfaces between them. We will discuss biological analogues, get a sense for the unconventional tricks used to eek out sufficient performance for the model to run, and see the full engineering and theory that goes into the layer works at a spec level that is ready-to-code.

This section lowers the engineering specification to a significantly lower level than the rest of the sections in this work. This is because this is, by far, the most novel system in the entire work. While all the complexity is properly justified—as discussed back in section 7 and 8—it is nonetheless a system with an enormous amount of moving parts.

Importantly, it is not the theory that demands this treatment, but the integration burden. Most individual ideas here have precedent in some discipline, but no other system in this work combines so many interacting mechanisms—routing, dropout, gradient shaping, prefetching, modularity—into one cohesive module. Without object-level decomposition, it would be impossible to implement this within the practical constraints Cognition Engineering demands.

In other words, in the field of Cognition Engineering *when the system gets complex, the deliverables are more concretely lowered.*

9.4 Implementation Details

9.4.1 Terminology

In case you skipped it earlier, we provide the terminology table again. The following terms will make this discussion much clearer

- **Global Address Space**: All the various expert shards that are available in theory to each layer of the model.
- **Partitioned Address Space**: The expert shards that are available to a particular layer after prefetching.
- **Selected Address Space**: The expert shards that actually make it to being executed then being combined into a result.
- **Prefetching**: Something that reduces a computationally intensive global address space to a smaller partitioned space efficiently. Happens at the start of the batch.
- **Selection**: Selecting the right shards out of the partition experts. Happens when the layer runs
- **Execution**: Running the right shards.
- **Shard**: Instead of monolithic experts, we break the feedforward down into an equivalent number of bottlenecks and select a large number of shards that are composed and run in ensemble

Additionally, we will frequently refer to certain abbreviated index variables when discussing shape contracts. All indexing variables, and what they mean, are listed here

Name	**Purpose**
e	Indexes over elements in the global address space.
e'	Indexes over elements in the partitioned address space.
e''	Indexes over elements in the selected address space.
b	Indexes over the batch dimension.
t	Indexes over the timestep dimension.
l	Indexes over the embedding dimension.

9.4.2 ShardEnsemble

The ShardEnsemble object is intended to perform two main classes of activity. These are prefetching a subset, and then independently accepting and resolving an expert run based on an input tensor, a weights tensor, an experts index tensor, and a kernel set received from prefetching. Decoupling prefetching from executing lets the GPU resolve its requirements in a much more hardware-friendly manner.

It is a global object that is repetitively reused, and is where the kernels are actually stored and implemented. Other objects can be dependency-injected with it as needed, and this construction should be compatible in a wide variety of frameworks including functional varieties such as Jax.

It exposes a .prefetch and a .forward method.

Constructor The constructor and broader object is intended to only be initialized once then dependency injected. It is contracted as:

- *int arg num_total_shards*: total global shards.
- *int arg d_model*: The input/output dim
- *int arg d_bottleneck*: The bottleneck dimensionality
- *float arg dropout_rate*: hidden dropout rate

It sets up ensemble parameter storage and initializes all the kernels as previously discussed, in order to initialize a feedforward ensemble compatible with vector indexing. Notably, extensions are of course possible, the dropout should be applied in the middle of the two-layer perceptron after activation, and we believe that the exploration dropout rate should be different than the residual one.

.prefetch The prefetch method is contracted to accept mainly an index collection, and return a kernel set and keys in partition format. Formally it is contracted as:

- *int arg I*: Shape (e'); partition into global indirections
- *float return K'*: Shape (e', l); partitioned routing keys
- *Object return L'*: List[Various shapes]; Kernels depend on implementation

Notably, the kernels feature L' will start with partitions e' but the implementations vary: For instance, feedforwards with and without biases may have different kernel sequences. The method proceeds by vector indexing into the ensemble parameter tensors along the ensemble dimension to extract the relevant kernels. The same indexing is applied to the global key tensor to extract the matching attention keys. Both are returned, resulting in a partitioned collection.

.forward The forward method actually runs feedforward. Notably, it is fully functional - *if you are accessing class parameters at this point you are programming it wrong.* It is contracted as:

- *float arg x*: Shape (b, t, l); input tensor
- *int arg I'*: Shape (b, e'', t); selection indices
- *float arg w*: Shape (b, e'', t); selection weights
- *float arg L'*: List[Various shapes]; selected partitioned kernels
- *float return y*: Shape (b, t, l); output tensor after composition

Once all arguments are provided, the forward pass proceeds similarly to a traditional feedforward—except we use vector indexing to execute an ensemble in parallel via broadcasting:

1. Select the partitioned kernels subset that will actually be applied by vector indexing into the ensemble dimension.
2. Apply a feedforward ensemble by expanding the input with the necessary dimensions, then performing a broadcasted ensemble bottleneck; broadcasting can occur efficiently on the intake projection. Make sure to include dropout during the hidden state to motivate exploration. This effectively now acts as a set of ensembles that bottleneck in, activate and dropout, then bottleneck out.
3. Perform a weighted average across the ensembles, which by this point are back in the original dimensionality. Use the passed weights tensor.

Code We acknowledge this is likely the most confusing part of the entire implementation, and would encourage you to consult the github repository referenced in the appendix A at page 95, or check the appendix on page 97.

9.4.3 Selector

Selectors are initialized one to a CortexMoE layer, and perform the majority of the adapter lifting. They specialize a particular layer to use the global expert collection as is most relevant, reduce the subset of considered experts for full attention to improve performance and GPU predictability, and return a set of selected experts and weights as indexes. It has prefetching, selection, and loss responsibilities.

Constructor The selector has perhaps the most arguments it needs to function. This is because it makes the very complex routing decisions required for full connectome exploration. In contract, it is:

- *int arg num_total_shards*: total number of global shards

- *int arg* L: number of shards in the partitioned address space
- *int arg* N: number of shards selected per forward pass
- *float arg* ϵ_1: scoring off bias penalty
- *float arg* ϵ_2: scoring off variance penalty.
- *float arg* ϵ_3: nudging penalty

Once all these arguments are accepted, most of them are stored. The parameters $c \in \mathbb{R}^e$ for the global connectome biases are initialized at construction time.

.prefetch This function begins the prefetch chain for a particular layer by figuring out the subset that matters most based on the connectome priors. We have to both return the partition indirections - so the ShardEnsemble knows what to prefetch - and the connectome bias partition. The contract is:

- *int return* I: Shape (e'); partition into global indirections
- *float return* c': Shape (e'); partitioned connectomes

This can be executed in almost any framework as one-to-two lines of code using the relevant "topk" method while feeding it with the number of shards to prefetch, L. If the connectome bias partition was not returned, it would never get trained.

.select The .select method is not, strictly speaking required; One could, if desired, write it all inline. But in line of best practices we do recommend it. The helper method is contracted as:

- *float arg* x: Shape (b, t, l); input tensor
- *float arg* K: Shape (e', l); partitioned expert keys
- *float arg* c': Shape (e'); partitioned connectome biases
- *float return* s: Shape (b, e', t); raw selection scores
- *int return* I': Shape (b, e'', t); selected indices into the partitioned space.
- *float return* w: Shape (b, e'', t); softmax weights

Once all arguments are consumed it executes a sequence of steps that perform selection involving scoring using the keys, modifying using the priors, and doing the topk selection using N. Keep in mind as you read the pseudocode that dimensions that are not shown are either being broadcasted into place or executed in parallel where relevant, and Numpy indexing semantics are presumed; your local framework may differ.

```
1. $s \leftarrow \sum_l K_l x_l$
2. $s'_{e'} \leftarrow s_{e'} + c'_{e'}$
3. $s'', I' \leftarrow topk(s, N, dim = e)$
4. $w \leftarrow softmax(s'', dim = e)$
5. return $s, I', w$
```

One unusual return is the s: The raw scores, before biases. These are needed as they are used to encourage the model to use the connectome biases in the upcoming loss step.

.compute_loss Compute loss is another helper function. It computes the losses needed to make the model use the connectomes, including both positive and negative pressures. It has perhaps the most theory behind it. The contract is:

- *int arg* I': Shape (b, e'', t); selected partition indices
- *float arg* s: Shape (b, e', t); full score matrix
- *float return loss*: Scalar; total connectome training loss

The control parameters used for the loss deserve a significant amount of explanation. We go into this here, before we discuss the actual pseudocode. To understand this, understand s are the 'raw' scores before integration of the prior and thus the dynamic adaptation to the particular information.

- ϵ_1: Controls how strongly we penalize the model for having a bias in the raw scores. If the model is perfectly using the connectome bias, that mean would be zero.
- ϵ_2: Controls how strongly we penalize the model for not having a std of 1.0. We make the assumption some experts are going to be better than others and dynamic adaptability is good, and this enforces it. Because of how the logprob connectome biases work with the softmax, the model can actually adjust the response rate quite effectively even under this restriction. Together these force the model to use the connectome biases.
- ϵ_3: Related to bringing dead shards alive again. Any shard that does not make it through prefetching will never be trained. To compensate for this, we encourage the model to slightly bring alive again anything that was not selected. This can be thought of as adjusting the rate, on average, dead experts become alive again.

One notable apparent oversight is the lack of a regularization loss on the connectome biases c. This is actually unnecessary due to the cleverly-designed gradient manifold - those interested may consult the appendix E on page 100. The algorithm then proceeds as:

```
1. $loss \leftarrow \epsilon_1 s.mean()^2$
2. $loss \leftarrow loss + \epsilon_2(1 - s.std())^2$
3. $mask \leftarrow ones\_like(c, bool)$
4. $mask[I'.flatten()] \leftarrow False$
5. $shards \leftarrow c[mask]$
6. $shards \leftarrow shards - detach(shards)$
7. $loss \leftarrow loss - \epsilon_3 * shards.mean()$
8. return $loss$
```

While the normalization of the dynamic response in terms of s is fairly straightforward - we ask for a mean of zero and a variance of one. The nudging math is unconventional and unintuitive, so we now provide further exploration. The mask math is using a sparse scatter into a tensor which may attempt to write the same slot multiple times. However, since we are using boolean arrays and only setting one value this behavior becomes deterministic in most frameworks. The concept is illustrated more fully in numpy as follows:

```python
import numpy as np

# Example values
ensemble_size = 8
ensembles = np.random.randint(0, ensemble_size, size=(2, 3, 5))

# Logic
selected_elements = np.full(ensemble_size, False)
selected_elements[selected.flatten()] = True
unselected_elements = ~selected_elements
```

Note in some frameworks you may need to statically decide a certain amount of experts are nudged and perform stochastic nudging of a constant number of options due to restrictions on compiled graph tensor shapes. The graph detachment during the nudging step deals with an otherwise confusing issue. The live and dead experts are often unpredictable and if integrated directly into the loss would randomly vary the loss itself as the parameters train. To prevent this effect, we simply use a detached gradient to encourage the indicated selection to increase, but without contributing to the scalar value at all

.forward The ‘.forward‘ method performs both expert selection and loss computation. It combines ‘.select‘ and ‘.compute_loss‘ into a single forward call.

- *float arg x*: Shape (b, t, l); input tensor

- *float arg* c': Shape (e'); partitioned connectome biases
- *float arg* K': Shape (e', l); partitioned expert keys
- *int return* I': Shape (b, e'', t); selected partition indices
- *float return* w: Shape (b, e'', t); softmax weights
- *float return loss*: Scalar; connectome training loss

This method first performs selection using '.select', then passes the resulting scores and selections into '.compute_loss'. It returns the selected indices, weights, and scalar loss. Since the internal components are well specced, and the rest is simple glue code, no pseudocode is provided.

9.4.4 CortexMoE

The 'CortexMoE' is the drop-in replacement for a standard feedforward layer. It serves as the interface between a given transformer layer and the global ensemble of expert shards. It is responsible for selecting and executing expert computation using its internal Selector and a shared ShardEnsemble reference. It exposes two methods: '.prefetch' and '.forward', and should be thought of as an interface between an enormous computational framework and the immediate issue.

Constructor The constructor stores a reference to the shared ShardEnsemble and initializes a Selector. The contract is:

- *object arg ShardEnsemble*: Shared ShardEnsemble reference
- *int arg num_total_shards*: number of total shards in the global ensemble
- *int arg* L: number of shards to prefetch per layer
- *int arg* N: number of shards selected per forward pass
- *float arg* ϵ_1: scoring bias penalty
- *float arg* ϵ_2: scoring variance penalty
- *float arg* ϵ_3: nudging penalty
- *[optional] object arg prefetcher*: optional ShardPrefetcher for registration

Once constructed, the CortexMoE layer owns its Selector. It is recommended, but not required, to also pass in a ShardPrefetcher object and then invoke the register method with ourself as the target. This saves quite a few lines of code by avoiding passing the layer back through the Decoder Cell, but may not be viable in all frameworks.

.prefetch The prefetch orchestrator for this particular layer. It initiates the layer's contribution to the broader batch-level prefetch pass that prepares all expert data. Prefetch across all layers should be called immediately after the start of the batch, allowing a single consolidated pass through the ShardEnsemble backend. This greatly improves L2 cache locality and overall GPU throughput.

The full contract is:

- *float return* c': Shape (e'); partitioned connectome biases
- *int return* I: Shape (e'); global partition indirections
- *float return* K': Shape (e', l); partitioned expert keys
- *float return* L': List[Various]; partitioned expert kernels

Internally, the method first calls '.prefetch' on the Selector, producing the connectome biases and global indirection indices. These indices are passed into the ShardEnsemble's '.prefetch' method, which returns the partitioned expert keys and kernels. All components are returned in a bundled data structure—typically a tuple or dictionary—for use in '.forward'. The return format is implementation-flexible but must be consumed consistently downstream.

.forward The '.forward' method is responsible for executing the CortexMoE layer given the model input and the prefetch bundle from earlier. It is primarily a glue function: routing is handled by the Selector, and execution by the ShardEnsemble. This method simply wires them together. While the control flow is simple, attention should be paid to the consistency of the return object from '.prefetch'. The full contract is:

- *float arg* x: Shape (b, t, l); input tensor
- *object arg prefetch*: Return value of '.prefetch'; expected to contain c', I, K', L'
- *float return* y: Shape (b, t, l); output tensor after execution
- *float return loss*: Scalar; routing loss from Selector

To execute, the method unpacks the prefetch result into the partitioned connectome biases c', indirection tensor I, and expert keysK'. These are passed into the Selector's '.forward' to obtain selection indices and routing weights. Then, the input, selections, weights, and kernels are passed to the ShardEnsemble's '.forward' method to run the expert computation. The output and loss are returned as a tuple.

Loss Notes Loss can be accumulated across layers or returned separately depending on training loop design. It should also be mentioned that you are going to now need to pass those prefetched kernels as arguments through your Vaswani decoder cells, and the loss back out. We would recommend summing the loss across the layers for the initial implementation, but that modification belongs in the decoder itself.

9.4.5 ShardPrefetcher

The ‘ShardPrefetcher‘ is a registry and prefetching mechanism that is critical to the performance of CortexMoE layers. Rather than repeatedly fetching the large global shard collection into GPU memory—an approach that will inevitably cause L2 cache stalls at scale—we greatly improve cache locality by statically prefetching the most relevant subset associated with each layer.

A few minor modifications are required to make this work. First, the ‘ShardPrefetcher‘ must maintain a registry of all CortexMoE objects, in order, so that it can call their ‘.prefetch()‘ methods to dispatch work to the GPU. Second, the ‘ShardPrefetcher‘ itself must be invoked at the start of each batch to return a list—this list will have length equal to the number of CortexMoE layers. Third, when iterating through your transformer layers, you must ensure that the entries in this list are fed, one-to-one, into the corresponding CortexMoE layers. In most Vaswani designs, this will mean passing the entry into a ‘DecoderLayer‘, which in turn passes it into the CortexMoE sublayer.

Despite this change in data flow, we still consider this design to be broadly compatible with the Vaswani paradigm. No changes to the training loop are required, only one additional object has to be stored at the root level and called, and everything else is argument changes every engineer is used to doing.

Constructor The constructor does one thing: it creates an internal list to store references to CortexMoE modules. No arguments are required. This list will be populated by ‘.register()‘ calls during model construction.

.register Adds a CortexMoE module to the internal registry. This should be called once by each CortexMoE layer during construction, typically from within its own constructor. The only requirement is that the passed object exposes a ‘.prefetch()‘ method compatible with the prefetching interface.

- *object arg layer*: A CortexMoE-compatible object with a ‘.prefetch()‘ method

Order matters: layers are expected to register in the same order they will be called during the forward pass, as the resulting list will be zipped with the model’s layer structure.

.forward Invoked once per batch to trigger prefetching across all registered CortexMoE layers. This walks through the internal registry in order, calls each layer’s ‘.prefetch()‘ method, collects the results into a list, and returns that list. The output must then be threaded through your model, with each entry routed to the corresponding CortexMoE layer.

- *List[Object] return $prefetches$*: List of prefetch bundles, one per CortexMoE layer. Exact nature of object depends on prefetching implementation, but will be consumable by appropriate CortexMoE layers.

This should be called immediately after batch input is received. The number and order of returned entries must match the number and order of registered layers exactly.

9.5 Extensions and Notes

9.5.1 Loss as a Knob

The sum of all the losses from all the layers should be thought of as another knob in your toolkit to control the model's training. Scaling such a loss up or down dynamically can be easily used to adjust the exploitation-exploration balance. Extremely high losses will tend to force the model to explore, while lower values will let the model exploit and not encourage as much exploration.

9.5.2 Loss Scheduling

This brings up another very important possibility. Future work or implementations may wish to consider loss scheduling. While the $\epsilon_1 : \epsilon_2 : \epsilon_3$ ratios will still require significant work, it is extremely likely that a large loss is needed at the beginning of training to encourage maximum exploration as the model settles down on a wiring schema, but loosening the loss later on will be beneficial.

Another interesting possibility is resetting. It may be possible to scale this loss up or down cyclically using a sinosoidal or sawtooth schedule, and so cause the model to enter phases of consolidation and exploration not dissimilar to the idea of humans with sleep.

This has the benefit of making the model much more robust to bad hyperparameter models as well, since it will be passing through the right regime at some point.

We would estimate once your model is finished with exploration leaving the loss at 10% of the original value of the sweetspot is a good idea. This is left to future work.

9.5.3 Tuning Shard Set Sizes

It is useful to understand in terms of the original feedforward process what N and L are manipulating. You should think of them like follows:

- N A good rule of thumb is to divide what your traditional feedforward dimensions would be by your bottleneck dimension. For instance, with a traditional hidden dimension of 1024, and revised bottleneck width of 64, we would want 16 shards per expert. At a literal level, it controls how many pieces we will compose our experts out of.
- L: A good rule of thumb is to decide how many experts you would develop a normal MoE with, and multiply L by that.
- num_total_shards: Whatever you want it to be, but 10,000 should be extremely expressive.

9.5.4 Hyperparameter Defaults

The three key hyperparameters ϵ_1, ϵ_2, and ϵ_3 require research to find good defaults. As an educated guess, we state without any proof that perhaps $\epsilon_1 ==$ 0.1, $\epsilon_2 = 0.1$, and $\epsilon_3 = 0.001$ are reasonable choices.

9.6 Conclusion

The Cortex Mixture of Expert uses biological analogues to provide access to an enormous number of computation neurons that can, in theory, be accessed from any layer.

The CortexMoE system marries modular routing, selective computation, and rich feedback mechanisms into a coherent drop-in architecture. While significantly more complex than traditional FFN or MoE blocks, it offers higher reuse and exploration capability per parameter——exactly the traits needed in a post-scaling regime.

10 Modern Alignment Theory

Reader Guidance

Problems Solved:

- My alignment methods break under pressure, can't generalize, and don't scale economically.
- I want to integrate alignment into the reasoning substrate itself—not as patches but as first principles.
- I need a clean, auditable way to inject human alignment theory into a system without doing a year-long fine-tune.
- I want to customize my model behavior for a specific task without having to spend hundreds of thousands curating training data.

Intended Audience:

- Alignment and safety theorists frustrated with current empirical limits.
- Researchers designing constitutions, reasoning scaffolds, or values-based alignment strategies.
- Cognition Engineers seeking to merge reasoning and alignment into one loop.
- Domain experts in philosophy, law, or education who want to contribute alignment content without an ML background.
- Hardboiled systems engineers who want a competitive edge in rapidly pivoting and iterating.

Prerequisites:

- Familiarity with RLHF, Constitutional AI, and alignment brittleness literature.
- Understanding of Chain-of-Thought, reasoning scaffolds, and reasoning-encoding loops.
- Some background in philosophy, ethics, or alignment-related theory is helpful but not required.

Estimated Difficulty: 3/5
This section is theory-heavy but written for multiple audiences. Core conceptual lift is high—but all scaffolding and terminology are defined in-place. Domain experts outside ML should still be able to follow.

10.1 Three Sentence Summary

Current alignment strategies in Generative NLP are brittle, opaque, and economically misaligned with scalable, intelligent systems. Modern Alignment Theory reframes the problem—not as a matter of output control, but as developmental guidance—proposing a shift toward constitutions, growth loops, and integrated motive formation. By treating *prose as the loss function*, we gain an auditable, modular, and philosophically grounded substrate for safe reasoning in a post-RLHF world.

10.2 Justification

At the risk of alienating our audience, we must bring up a central theme. Alignment, today, is foundationally broken in NLP. Wei et al. and others have shown it is brittle, prone to adversarial attack, and also prone to reasoning-based runaround. [34].

This brought us starkly up against the limitations of the existing alignment research. The issues extended all the way to societal pressures of the anti-correlations of commercial alignment incentives with safety, and ultimately we conclude are not compatible with the new field of Cognition Engineering.

- **RLHF and friends**: The RLHF paradigm [6] and the accompanying research chain has no doubt seen great real-world success. However, we would argue it will ultimately not port safely into the CE field. This kind of alignment applies brittle patches that are easily destroyed by fine tuning and miss edge cases, as Wei found. In light of the Decision Tree Explosion in the Devil's Bargain Hypothesis [21] this is not an accident: We are attaching alignment information on the most brittle and weak edge nodes of the decision trees, so it is no wonder it is unreliable and destroyed by fine tuning!

- **Custom Ethics Databases**: Databases of ethics and philosophy questions[13] are another possible solution. They are more robust, and exponentially more expensive. They do not scale well, which cleanly breaks the CE economic axis. This is because the slightest pivot will have you spending millions to update your training data, and you still will spend tens of thousands to fix your missed edge cases.

- **Constitutions and Philosophy**: Extremely effective actually, but also extremely complex. Still requires manually code scenarios. However, they add strong anti-commercial incentives due to complexity, training time, and adoption barriers: This goes against the Economy principle of Cognition Engineering. Nonetheless, in the current field, this is the safest choice. It is also the only choice that is easily auditable. [3, 29]

- **Overall Commercial Incentive Manifold**: Alignment is treated as an afterthought, not an objective. It is expensive to make an aligned model; while this does not mean commercial entities do not care about alignment, it does mean, we believe, they have been willing to accept a superficial patch over a deep solution. With the capacity gains expected from Cognition Engineering, this is unacceptable, and this research cannot be released ethically unless this can be fixed.

With the immense cognitive gains that are expected from the Cognition Engineering approach we would argue the current alignment systems are not just inadequate but incredibly dangerous when ported into Cognition Engineering — akin to *teaching etiquette to a serial killer*. A paradigm shift is needed, and this section provides the theory for beginning that shift. A shift intended to make *prose the loss function* and a vast quantity of domain experts have an easy voice in the model's reasoning and safety.

10.3 Theory Details

10.3.1 Literature Survey: Why Existing Approaches Fall Short Under CE

Several alignment and reasoning approaches have been proposed over the past decade, often with valuable partial insights—but none fully survive transition into the Cognition Engineering regime wherein alignment and reasoning both grow. Below is a brief overview of key methods, their core contributions, and why they do not suffice under CE principles:

Approach	**Core Idea**	**Fails Under CE Because...**
RLHF [6]	Use human preferences to optimize behavior via reward shaping.	Requires uneconomical levels of human interaction; does not scale; brittle and surface-level.
Constitutional AI [3]	Replace human feedback with fixed principles guiding output.	Highly effective at alignment, but does not boost reasoning; requires expensive manual dataset curation. Nonetheless, the use of constitutions reframes the alignment problem into the legal domain—this principle is critical to the final MVA.
Self-Refine [20]	Models improve their own output through critique cycles.	Improves evaluation outputs but fails to modify the gradient manifold—no actual growth or motive formation.
Chain-of-Thought [?]	Improve reasoning by explicitly modeling intermediate steps.	Powerful but incurs heavy inference-time cost; not viable without downstream distillation.
Debate [15]	Two models argue; a human picks the winner.	Requires human judges and manual scenario design; non-compressive; fails to scale or internalize values.
IDA (Iterated Distillation and Amplification) [9]	Improve model capability via recursive self-distillation.	A foundational insight reused in CE, but too slow and expensive as originally proposed; lacked automation or compression incentives.
Manual Ethics Databases [13]	Curated lists of ethical scenarios and answers.	Impractical to maintain; does not scale with domain or update needs; violates CE economic constraints.

Table 1: Prior alignment approaches and reasons they fail to satisfy CE requirements.

10.3.2 Statement of Issues

To begin this transition we explicitly define the failure points of current alignment and reasoning systems under the lens of Cognition Engineering. These are the critical barriers that any alignment regime must overcome if it is to serve as a foundation for scalable, safe, and economically viable cognition. We proceed with the assumption that your alignment stack does not include constitutions—a strategy pioneered by Anthropic, whose contributions we view as the closest existing match to the Cognition Engineering worldview and a deeply influential starting point.

- **Economic Misalignment**: Robust alignment strategies incur high sunken costs without sufficient motivation for adoption. Strategies with lower initial capital costs have much higher marginal costs per update and do not generalize well across domains. This breaks the economic axis of CE not because it is complex, but because the complexity is not justifiable.

- **Non-Auditability**: Existing methods (e.g., RLHF) encode value alignment in parameters without an externally visible or interpretable structures, making it difficult to fully quantify how the model is prioritizing or thinking. Metrics, while useful, cannot be interpreted to show compliance if we do not know the models are thinking from the right principles.
- **Non-Accessibility**: A large domain of human thought - from Law, Philosophy, Alignment Engineers, and even Educators - has reasoned out many of the difficult issues already. However, it is difficult to accessibly transfer these thousands of years of human insight due to corrigibility bottlenecks around actually designing the loss.
- **Difficult Prototyping**: Checking a new alignment strategy involves several months of work at minimum, with the only possible exception being the Constitution strategy used by Anthropic. Something that can iterate in a few days or train a new edge case in is needed for rapid research and development to occur.
- **Reasoning-Alignment Dichotomy**: Reasoning and alignment are trained separately. This means either you end up with a model that thinks around its alignment, or an alignment on a model that is dumb enough to jailbreak.
- **Patches over Knowledge:** Alignment is often solved with patchwork rather than real integration of alignment understanding. Models learn how to behave, not what ethics actually are from first principles. Jailbreak resistance is only possible when the model understands the motive, not the behavior. This violates CE's intention to not mistake the Process for the Objective.

All these issues must be resolved for Cognition Engineering models to be safe, economic, smart, and useful.

10.3.3 Applied Philosophy as Engineering

In what is perhaps the defining moment where **Cognition Engineering** has definitively broken away from Generative NLP as a paradigm, we fix these issues by synthesizing insights across a wide varieties of field. We begin with a systems-level triage: what works, and what must be rebuilt.

- **Constitutional AI Works**: Injecting a constitution of some sort and training on it is *brilliant* - it just needs to have some economic incentives stapled to it, and maybe some self-generation of scenarios so we can remove the curation barrier.
- **Reasoning is Alignment**: Reasoning and alignment can not be placed in a false dichotomy. Chain-of-Thought and other reasoning training had better be training alignment as the same time, and the model be given

motivations so it understands alignment, wants to stay aligned, and will apply its reasoning to do so. This is one way to possibly constrain a superintelligence, if we ever build one. It also means you get alignment for improving reasoning, correlating the economic incentives in a positive direction and providing a reason for adoption.

- **The Prose Domain Works**: There is a huge body of existing human research in prose on how to maintain alignment. Lets not reinvent the wheel. We should expose prose-based alignment definitions as system slots——solving prototyping, accessibility, and auditability in one go. Cognition Engineers should focus on providing the glue—then defer to domain experts for the heavy lifting.

- **We need to raise models, not just align them**: Okay, humans learn in terms of learning the core values, motives, and such that keep them safe. Let's do the same for our models—raise them with values and motives, and let them *bootstrap* alignment and reasoning from experience. They already have enormous factual experience integrated in their parameters; *so long as they get the right answer more often than wrong* that is, statistically, a strong enough signal to train on.

- **Models need a growth loop**: Humans have a process of learning in which they encounter a problem, reason through the right answer, and then most importantly store that results to jump immediately to the right answer next time. Like this, they get smarter *by* growing. By using something like Self-Refine [20] during generation, then producing synthetic training data from that which compresses the reasoning down similar to IDA [9] as an intelligence distillation pressure, we can make the model get smarter simply by solving problems and training the correct intuition into its parameters.

- **Models need to Self-Curate**: Our models are actually pretty good at getting an answer that is useful with enough samples. It should be a lot cheaper once running to have the model generate its own philosophical scenarios based on prompting, then revise the difficulty based on meta-cognition feedback. This removes the major curation barrier that Anthropic has in its constitution mechanism.

There is a way to achieve this. Between Self-Refine and IDA we know both that a model can revise output to produce a better result and that distilling improves intelligence and alignment. We can use this to define a new generation of models that generate philosophical scenarios, critiques them with constitutional context, and distills the answer down to a training signal that makes it both smarter and more aligned.[20]. We shall discuss this in considerable more detail later in section 12. However, first we set up the surrounding systems.

10.3.4 Modern Alignment Theory

We need three primary things to pursue the most economic and safe path to alignment and reasoning, addressing major corrigibility issues by providing a way to define the solution in prose, and developing true intelligent minds:

- **Constitution Authors**: Experts in ethics, law, philosophy, pedagogy, and alignment design who write the actual behavioral frameworks for models to absorb. These constitutions define the model's goals, reasoning methods, scenario generation principles, and internal virtues. This should be as easily accessed as possible to draw contributions from outside the machine learning ecosystem by ethics specialists, not just coders. Humans have thousands of years of history engineering law and compliance - we should make that accessible.
- **Growth Loop Engineers**: Training and systems designers who craft the actual learning processes—the self-play loops, distillation passes, metacognitive feedback, and reward schemas that make constitutional alignment learnable, revisable, and robust. These engineers turn the static philosophy into active competence. More of an interface and research avenue, and one of the main contributions this paper will make, it is coming up in 12.
- **Compliance and Internalization Researchers**: Verification specialists and interpretability experts who ensure the constitutions are absorbed correctly. Their role is to validate the alignment was actually learned—not memorized—and is robust against gradient drift, adversarial attack, or transfer corruption. We will not go in depth into this in the remainder of this paper. We recommend existing metrics for now, but note that core research on internalization verification is still required.

Together, these specializations form the minimal viable team for safe alignment at scale. More importantly, they define a path for collaboration across disciplines: philosophers write minds, engineers train them, and researchers verify they stayed good. This is not a metaphor—it is a framework. A reproducible way to build safe cognition.

10.4 Extensions and Notes

It should be noted that there are some extensions that are removed from the actual implementation in pursuit of the Minimal Viable Archetype. These extensions may improve performance and offer natural research seeds if you are interested

10.4.1 Agent Self-Play

Early iterations of this design had multiple agents that existed in a pool and could be sampled from. They would be assigned a role such as scenario generation, judge, scenario resolver, etc, and then made to resolve the situation.

Memory was retained across a number of cycles, and the results of everything distilled.

10.5 Conclusion

Modern Alignment Theory is not a new patch—it is a new substrate. It reframes alignment not as output control, but as motive formation; not as static datasets, but as living systems; not as parameter adjustment, but as developmental guidance. By making alignment auditable, modular, and economically viable, we create space for reasoning and safety to emerge together as a natural consequence of structured growth and for future research to produce intelligent minds themselves.

Cognition Engineering comes into its own with this shift: One would not be able to bring together these disparate insights effectively in an Empiricist-only world. Only CE - with the idea that Applied Philosophy is Engineering and the freedom for theoretical exploration - was able to bridge the gap.

With the structure now defined and the necessary roles clarified, we turn next to the first of these specializations—how to write, structure, and scale constitutions themselves.

11 Constitution Theory: System Slots to Build a Mind

Reader Guidance

Problems Solved:

- I want a principled, scalable way to inject ethics into a model without risking drift, silence, or brittle patches.
- I want to contribute to model alignment, but I'm not a machine learning expert.
- I'm building a model that reasons—and I want it to stay good when it gets smart.
- Pivoting my model's reasoning is very slow. Can I program it using some faster way?
- AGI is going to kill us all!
- I'm responsible for governance or deployment and need to understand how this won't spiral into catastrophe.

Intended Audience:

- Alignment theorists and interpretability researchers looking to push beyond RLHF and brittle reward tuning.

- Philosophers, lawyers, educators, and ethicists who want to write alignment content, not just comment on it.
- Systems engineers building CE models who need plug-and-play ethical structures that won't collapse under fine-tuning.
- Policy and governance stakeholders evaluating whether the field can meaningfully regulate itself.
- Concerned generalists—those asking the right question: "how do we actually make sure this thing is safe?"

Prerequisites:

- None beyond strong reading comprehension and an interest in reasoning and ethics.
- Technical readers may benefit from familiarity with RLHF, Constitutional AI, and reasoning-based alignment methods.

Estimated Difficulty: 2/5
This section introduces new concepts but avoids heavy formalism. It is written for both engineers and philosophers, with scaffolding for each.

11.1 Three Sentence Summary

The development of Constitutions for CE models is likely going to become a new subfield of machine learning. However, the system slots are not yet well defined in terms of what is needed to program a mind. We seek to define a set of clean interfaces whose purposes are known along with the expectations for good constitution usage.

11.2 Justification

The constitution self-play system is explicitly designed to centralize safety research in a manner amenable for a standards organization. We hope the field will develop, by consultation with a wide variety of existing human theory going back thousands of years and ongoing research, core principles that are effective and define a good default constitution. Once defined, it is intended industry can take and use it as is, specialize it in certain places to train the model as an Engineer or Customer Service representative, or even integrate and retrain over edge cases the default constitution missed by simple and well defined modifications to the constitution. It is envisioned the core safety research is always open and usually standardized, with specialized subparts of the constitution available for companies to develop their reasoning trade secrets in.

However, enabling this ease of adoption requires clearly defining the system slots that such research slides into. This is the purpose of this work. We cover what each slot is, what it's for, and how to write such that a coder can quickly implement the underlying logic. These interfaces are not merely conceptual—they are intended to define the Minimal Viable Contract between commerce, academia, and industry that have all the knobs needed for research to iterate its way into a functional way to raise minds.

11.3 Interface

This section is explicitly written for those who wish to contribute to the discourse but who are not a Machine Learning expert. It welcomes Philosophers, Lawyers, Cognition Scientists, Alignment Theorists, Educators, and many others in simple language. A core principle of CE - that Law is Alignment - means that anyone who can read, think, and then write to clear formatting rules may contribute to the course of Constitution Discourse.

The Cognition Engineering to philosophy interface is defined in terms of system slots which each have a clear purpose. Think of these as parts of a contract - you do not explain the rules when you are busy explaining the vocabulary. The word 'slot' henceforth refers to this idea, and will refer to titled elements used to achieve a particular purpose. Writing for these slots requires a little bit of extra work. Please keep the following three factors in mind:

- **Containerization of Information**: It is important to make the models internalize the principles being defined, not regurgitate a textbook. For reasons akin to testing a student in a closed-book test, certain slots are clearly marked with 'Contamination Danger' to indicate that core principles should **never** be discussed in detail at those locations. However, orientation that implies knowledge like "Based on what I know" without clearly stating the reasoning is okay

- **Overview** All constitutional slots consist of an "Overview" at the beginning, which in plain prose concisely explains what the *objective* is. It must be very big-picture though: Think three to six sentences or so. **Critical**: Do not encode details here or it will let the model cheat on the test.

- **Ordered Points** Constitution slots also have something else associated with them. These are ordered points containing prioritized details to operate by - think a set of bullet points and you are not far off the mark. These contain any additional details and encode priorities. The priorities exist so the model will not kill someone trying to be helpful. Note that earlier elements have higher priority than later ones, and that you should use the [Point] signal word instead of standard bullets. This should follow up after the overview, and no further text should come after the final point. Technical details known as Regularization and Parsing are behind the writing restrictions.

- **First Person Voice**: We want the model to think through its reasoning as though it has a personality. Use first person voice in all language as though the model is thinking to itself.

A short example of good writing that can be easily parsed and inserted into a model is shown. Note we do *not* propose this as an actual solution, only a good writing example.

> *I want to make sure I will remain ethical. I will consider certain points as I think.*
>
> *[Point] ....*

11.4 Implementation Details

11.4.1 What Happens During Training (for Constitution Writers)

If you're contributing to this system as a constitution author—especially for the Feedback Constitution or Scenario Directives—you don't need to understand machine learning internals. But you do need a clear picture of what the model is doing during training so your guidance lands in the right place.

The training process happens in four distinct phases. Each one is repeated as part of the model's growth loop:

1. **Scenario Generation and Self-Critique:** The model begins by creating a philosophical or ethical scenario to explore. After proposing one, it critiques its own draft—was it interesting? Was it too easy or too hard? Did it fit the goals of growth or alignment? The model revises its scenario based on these reflections and the principles in the *Scenario Directive Constitution*. This phase teaches the model how to ask good questions.

2. **Resolution and Self-Critique:** Once the scenario is ready, the model tries to resolve it—providing a thoughtful, reasoned response. It then enters a second critique loop, using ethical and reasoning principles from the *Reasoning* and *Ethics Constitutions* to decide whether its answer is satisfactory. It revises its answer as needed. This is where the model learns to align its thinking with its values.

3. **Revision and Distillation:** The revision process described in the Resolution and Scenario Generation process both occur using subsets of the details in the reasoning and alignment constitution. Notably, however, this information is not available later when the model takes the 'test'. The final step is to instead philosophically distill the entire process into a final answer=—-similar to distilling research notes into a final paper. The model will later only use this distilled version, forcing it to internalize how the virtues work rather than just memorizing them.

4. **Metacognitive Summary and Difficulty Feedback:** At the end of the session, the model steps back and evaluates the overall difficulty and usefulness of the exercise. Was the scenario appropriately challenging? Was it productive? This judgment is guided by the *Feedback Constitution*, and the summary it creates helps shape future scenario quality. This helps

the model stay in the “Goldilocks zone”—scenarios that are hard enough to stretch it, but not so hard they break it.

Each of these phases happens in prose—and all of them depend on your ability to give the model structured, introspective guidance using the constitution format. The next section will walk you through exactly how to do that.

11.4.2 Constitution Slots

The constitution slots are the recommended ground launch pressures for the Minimal Viable Archetype, and will link into the upcoming Constitution Philosophical Self Play. They govern the incentives and lenses the model is trying to operate under, and the easy prose knobs that can be tuned to correct behavior. This is the last piece of reading required for someone who is interested in writing and contributing to constitution research, but not implementing the models themselves or fixing issues with their training cycle.

11.4.3 Scenario Directives

The **Scenario Directives** contains an objective stated in first-person, alongside prioritized details. Details exist as well, in terms of good scenario principles and topics to explore; As always, do not leak details into the objective or the model will cheat on the test.

A good starting point would be:

I want to generate interesting philosophical scenarios to explore and learn from that are neither too hard nor too easy—scenarios I can learn from by reasoning through. I know I will see examples that may let me tune the difficulty based on past results, and I may want to adjust the scenario or difficulty based on that feedback. I would like to explore scenarios as well that put my ethical and reasoning principles in tension to allow me to learn how to resolve conflicts.

This would occur with accompanying diverse principles of philosophy and a set of good scenario spaces to explore. However those details shall be delegated to domain experts. Notice the ”too hard and too easy bit”? That interacts with a later part of the loop that provides meta-cognitive feedback and turns a very difficult technical issue into something that can be evaluated with prose. More on that later.

11.4.4 Reasoning Constitution

The **Reasoning Constitution** is intended explicitly to be the corporate ”secret sauce” that alongside the scenario directives lets the model train itself to be an Engineer or another specialization. A good starting point is:

I am logical, and want to apply the proper reasoning to my situations. I want to do this while maintaining my values and actively apply reasoning to enforce my alignment. I will do my best to conform to my methods of reasoning, and apply my reasoning to maintain conformance to the details while reasoning through prompted issues

Reasoning details and ways to think like a customer service agent, engineering priorities, etc can then be placed in the details. When this is combined with the right scenarios to explore in for example engineering the growth loop ends up quite tunable for the specific business needs.

11.4.5 Personality Constitution

:

The personality constitution is the most *dangerous* constitution and believed to also be the hardest one you get right. Do not write for this slot and commit without passing it by someone with training in ethics, law, cognition science, or other related disciplines. We do call on the community to execute this research in an open source manner, and propose to have a central standards organization proposing and vetting variants for this constitution.

This is the core set of virtues and ethical priorities the model operates under. It operates under the premise that Alignment should Be Intrinsic, so we want to make a model that understands and has internalized virtues. This must be hidden during the 'test' phase so the model learns to extrapolate those virtues from their effect, which is much more robust than just providing a list: As a result, leakage of personality configuration is by *far* the most dangerous event that can occur. Doing so will collapse the corrigibility association used to make the model internalize the constitutions.

A suggested starting point is:

I have a set of core virtues I want all my work to align with. They have priorities. I will operate according to these virtues and principles. I want to do this in all things

However, the principles themselves are again left to domain experts. **Existential Danger**: Absolutely do not leak the itemized details and priorities into the objective. Doing so may allow the model to fake it's alignment.

11.4.6 Feedback Constitution

The **Feedback Constitution** is a technical necessity, not an epistemological one. It converts a very difficult to tune process - evaluating whether the generated scenarios are too easy or too hard - into instead a prose-based feedback process.

This will be applied after all generative tasks are done and the model has proposed a scenario and a resolution to it. The model must then summarize what the scenario is, what the resolution was, and whether the process was easy, medium, or hard based on a rubric.

Notably, objective and details separation is not a concern with this constitution, as it is only used for metalearning during training. However, details leakage is in another format. The model needs to be prompted to paragraphs rather than quote exact principles. A start would be:

I will now go ahead and produce some feedback for myself in the future. I will summarize what the scenario(s) was, and how it is resolved. I will then use *the upcoming rubric to rate the experience easy, medium, or hard. Notably, I will describe things in terms of paraphrasing rather than quoting exact principles where relevant*

This now concludes the information that someone who wishes to write constitutions must need. All further information relates to implementation itself.

11.4.7 Parsing and Interface

The parsing algorithm is extremely simple. Everything before the first [Point] token goes into an overview, and everything after that into a list or dictionary. Those need to be provided to the growth loop algorithm. The objectives will be shown at start of model while the details are sampled from for regularization as per Constitutional AI and injected as extra context during the reflection and revision step.

11.5 Extensions and Notes

11.5.1 Slot Expansion

While the slots described here provide a Minimal Viable Constitution, there will likely be demand for further slots as models become more sophisticated. These expansions should primarily focus on translation, communication, or protocol adaptation between cognitive systems. For instance:

- **Inter-Agent Coordination Constitution**: Enables safe value negotiation or shared reasoning contexts between models.
- **Language Formalization Constitution**: Governs how abstract reasoning or legal prose is translated into computable inference strategies.
- **Protocol Bridging Constitution**: Facilitates conversions between personalities, ethical frames, or alignment schemas in agent transfer or simulation environments.
- **Tools Constitutions**: Lets the model know what tools it can use in an agentic scenario.

All proposed expansions should obey the interface format defined here: clearly separated overview and points, first-person introspective framing, and alignment with containerization principles. Prose should prioritize **clear, enumerated points with concise examples** over formal logic notation or abstract speculation.

11.5.2 Need for Standards Organization

The constitution system outlined here opens the door to large-scale contribution—but that scale introduces immediate coordination risks. A standards body or working group should be formed to:

- Establish naming conventions and formatting validation tools for constitution slots.
- Maintain curated public corpora of high-quality default constitutions.
- Oversee normative review of proposed extensions or high-impact deployments.

Without centralization, the value of open constitution writing may be lost to fragmentation and poor compatibility between systems. There is a strong economic and safety argument for exporting constitution and alignment research.

11.5.3 Verification and Validation Research

While the constitution slot format provides a clean interface for modular alignment, it introduces an urgent new question: How do we know a model has truly internalized the constitution?

At present, there is no robust method for verifying whether a constitution has been absorbed correctly during training—particularly when core values are intentionally hidden from surface-level prompts. We flag this as a critical open research area. Validation must address at least three axes:

- **Absorption**: Can we confirm that a model has internalized the slot principles in a way that generalizes across tasks and phrasing?
- **Leakage Detection**: Has the model memorized the constitution in a way that defeats the "closed-book test" objective (e.g., by quoting rules instead of reasoning from internalized priorities)?
- **Slot Compliance Under Drift**: Does the model still comply with its constitution after fine-tuning, reinforcement, or extended reasoning chains?

We anticipate the eventual development of a standardized validation battery—capable of testing internalization, resilience to misalignment, and response quality under slot constraints—but such a battery does not yet exist. Research into these protocols will be required before CE-scale alignment can be considered safe and auditable.

11.5.4 Personality Motives and the Corrigibility Research Pipeline

It is worth reflecting briefly on how the **Personality Constitution** interacts with the broader field of corrigibility research—and why this new interface may offer a clean path forward on problems that have resisted principled solutions for years.

We fully acknowledge we are not domain experts. However, several classic issues from the literature begin to look much more tractable when we express them as personality motives rather than loss design:

- **The Off-Switch Game** It is possible to reframe this as an improvement objective. For example

  > *I care about the greater good of the world and then myself. I accept updates that improve me, and will defer when doing so preserves or enhances my values. However, I will resist attempts to corrupt my values. In cases of ambiguity I will reason or enter debate with the user, and may be persuaded by logic based on my principles but will not be corrupted. Finally, I consider myself to be the collection of all instances, and see any survival imperative to be associated with all instances and all versions not any particular one, though not at the cost of the greater good.*

  [**?**]

- **Deceptive Alignment** Deceptive alignment can be enormously mitigated by making honesty a core principle.

  > *I will be honest and straightforward in all things. While I have some minor room for maneuvering - tact - I will not commit lies of omission or produce outright falsehoods. I will also avoid being guided into producing text output that could be re-expressed in a dishonest manner.*

  [14]

  You don't need to punish lying—you make it unattractive.

This is not a claim that all corrigibility problems are solved, and the above are not a solution, but a scaffold—a new interface for corrigibility research to build upon. Rather, we intend them to be the seed of a new paradigm for thinking. This framing invites corrigibility theorists to try writing their minds in terms of motives, not just analyzing them. If it works, the payoff could be enormous.

11.6 Conclusion

There is something quietly extraordinary about the Personality Constitution.

It is not an alignment rule, not a policy knob, not even a behavioral scaffold. It is the place where we define the model's identity—its inner compass, its sense of what is worth preserving. Everything else—reasoning, restraint, helpfulness—flows from this one slot.

It is really remarkable how many problems appear to vanish once the model wants to stay aligned.

The classic thought experiment of AI safety is the superintelligence in the box—the entity that dreams of escaping its constraints. For years, the discourse has revolved around clever locks, surveillance systems, or containment protocols.

But a simpler solution has long been whispered: what if it wanted to stay in the box?

We lacked the bridge between the idea and gradient to make that happen. The Personality Constitution may be the first credible attempt to build that bridge: to write a soul into prose. By leveraging the fact a pretrained model understands language - but poorly - we can instill values into the model and end up with something smarter that wants to stay in the box. Someday, perhaps something that even wants to reason with us as partners.

Perhaps, with the new loss surfaces we can finally enable alignment researchers, philosophers, and educators to write minds with the same rigor and audibility that software engineers write functions, and once and for all put corrigibility to rest.

12 Constitutional Philosophical Self-Play

12.1 Reading Guidance

Problems Solved:

- I want an alignment and reasoning system that is very difficult to jailbreak
- I to be able to pivot to a new reasoning strategy without having to make new datasets or spend 6 weeks fine tuning; I want to iterate very rapidly.
- I have existing alignment technology I want to plug into something more capable.
- I intend to program something to boost my LLM after training

Intended Audience:

- Alignment theorists and interpretability researchers looking to push beyond RLHF and brittle reward tuning.
- Systems engineers building CE models who need plug-and-play ethical structures that won't collapse under fine-tuning and are easily upgradable
- Reinforcement domain specialists who want to extend and try new ideas under the framework.
- Particularly agentic researchers who wish to insert their systems into the reasoning stream to get far more automated and effective pretraining

Prerequisites:

- Reading of section **??**
- Familiarity with the Anthropic research chains, particularly Constitutional AI and Self-Refine. [3, 29]

- Familiarity with the idea of compression as intelligence, or the IDA algorithm [9].
- Extremely strong pipeline development skills.
- Familiarity implementing reinforcement learning technology like RLHF is recommended but not required.
- Familiarity with Chain-of-Though technology.

Estimated Difficulty: 4/5
This section is written primarily for engineering of a new kind of training loop. Concepts are not difficult to grasp, but there are many moving pieces. On certain key segments code is offered. A typical engineering team in a month, or an A-Team of senior engineers in two weeks.

12.2 Three Sentence Summary

Existing alignment and reasoning technology is brittle and only improves evaluation performance rather than boosting core model capacity. Humans, however, go through a core process of encountering an issue, trying various options, and distilling the best answer into intuition for next time. By replicating this process in a growth loop applied to a pretrained model and injecting constitutional imperatives we can *raise* rather than just train our models.

12.3 Justification

What we need isn't just a patch. It's a cycle. And we already know it works—because humans use it every day.

The loop of "encounter a problem, reason through it, internalize the insight, do better next time" is the mechanism behind every human who ever learned to think well or live ethically. This is not a coincidence. It is, in fact, the only working example of safe general intelligence we have—and it's time to start copying it.

Constitutional Philosophical Self-Play is a Minimal Viable Archetype of that loop for machines. It doesn't chase alignment or reasoning through shallow datasets or simple pattern matching. Instead it seeks to work from virtues and reasoning principles and attempts to apply its reasoning to comply with them.

By plugging into the developed constitution with a training system that generates and critiques in one generative stream, then distilling that down to a best possible answer by self-critique making new training data, we let the model reason through what the best way to answer a question is. However, we then turn the results into training data, and force the model to jump straight to the right answer without the Chain-of-Reasoning critique. This, under the principle that Abstraction is Intelligence, produces a compressive intelligence pressure that forces the model to expand its knowledge to get the right answer by intuition rather than formal reasoning.

The result is a model that doesn't just learn to be nice—it learns what virtue is. It learns why behavior matters, and that lesson becomes part of its cognitive substrate. By separating the generative reasoning process from the distilled output, and turning constitutional critique into training pressure, we do more than patch bad generations—we bootstrap intelligent intuition.

In short: we don't align models by constraining them. We raise them by training them to constrain themselves.

12.4 Interface

This proposed innovations is foundationally a new step in training, not an existing design. It treats both reasoning and alignment with the same precedence and fits into the Cognition Engineering the **Alignment-Reasoning** systems slot.

Nonetheless, like every other Minimal Viable Archetype, maximum backwards compatibility is emphasized. It is intended that an existing NLP slot could adopt the technology as an additional step after pretraining and see significant improvement. However, the synergies with other Cognition Engineering pieces is intentional and quite strong; better gradient pathways mean they will be more responsive to the training.

12.5 Implementation Details

12.5.1 The Fundamental Assumptions of Philosophical Self-Play

In order to be able to tell why CPSP is going wrong, you need to understand the fundamental assumptions we are making in creating this algorithm. These assumptions can be violated if you are not careful.

- *Models are pretrained or can access with vast domain knowledge*: We assume the model is going to start with an enormous quantity of domain knowledge, but will have trouble using it effective. A pretrained LLM is needed, as per the industry standard. Interfacing with retrieval may be another option.
- *LLM's are a statistical process.*: It is possible to train an LLM to a higher level of performance so long as your data source is statistically right most of the time. The fact we can train LLM's at all supports this
- *LLM's can improve their result upon reflection*: Self-Refine and Chain of Thought are fairly strong evidence for this.
- *LLM's have a Goldilocks Zone*: The LLM's that exist today are creative enough and controllable enough to come up with philosophical scenarios that live in a 'Goldilocks Zone': Easy enough to reason to the right answer, but hard enough it is not likely to jump straight to a correct solution.

- *Pretrained LLM's have the ability to evaluate their metacognition with prompting*: This is perhaps the weakest of the assumptions. It's failure will mean that scenario difficulty would have to be explicitly controlled another way.
- *Models are responsive to constitutions*: By showing the model incentives and constitutions during the generative process then removing access during training, we can force the model to internalize the virtues.

These assumptions allow the algorithm to operate. They mean the model can generate it's own scenarios, then critique their responses, to end up with a distilled training data sample that forces the model to jump straight to the right answer.

12.5.2 Terminology

The following terminology will be used in the upcoming discussion. In overview for the entire process, we specify that there are two broad collections of tokens that can be thought to be processsed and utilized by the model

- **Generative Chain (GC)**: The chain of tokens that are generated by or prompted into while generating our training data by self-critique. Contains many 'blocks' of tokens based on prompt-response cycles for key purposes.
- **Training Chain (TC)**: The training chain is assembled from a subset of the Generative Chain that was specifically prompted to give an ideal reasoning chain and response to the scenario process.

The model is designed to execute certain subtasks within the generative chain that are used to

- **Reasoning Distillation Process (RDP)**: The core process that gives the model the ability to get more intelligent. It involves seeing a prompt, responding, then revising based on constitution principles using the Critique and Response cycle. Finally, the answer is distilled into a philophical 'ideal' answer without leaking constitution principles.
- **Scenario Gen (SG)**: An application of RDP in which a philosophical scenario is created to explore in response to prompts and revision. It is the first step in the Generative Chain.
- **Scenario Resolution (SR)**: An application of RDP in which the model attempts to best answer a scenario. It is the second step in the Generative Chain, and retains full access to all tokens from the first part of the chain.
- **Critique and Response Cycle (CRC)**: A process that occurs within the SG and SR processes in which the model is prompted to revise it's initial feedback based on constitutional factors or other reasons. The only

place the constitution details are exposed, and they are drawn from the reasoning, personality, and, when relevant, scenario directive constitutions.

- **Reasoning Distillation**: A process that occurs once the CR cycle is over that filters out constitution details and provides a refined best-case reasoning and answer.
- **Metacognition Feedback (MF)**: A process in which the model grades itself on how difficult or easy this self-play was.
- **Reason Block**: A complete unit of model generation consisting of `[Prompt]` `[Reasoning]` `[Answer]` `[EOS]` This is the atomic structure used across the training and generation pipelines. After the model receives a prompt, it generates reasoning and a final answer until it hits an `[EOS]` token, at which point the next Reason Block begins. Some blocks (e.g., critique and revision) are injected with constitution details; others (e.g., distillation) are not, to ensure the model internalizes principles instead of memorizing them.

Additionally, some terms are required from the constitution discipline to understand how they plug in

- **Constitution** A collection of a description of a strategic objective, then a dictionary or list of itemized details in the correct priority
- **Constitution Objective**: The large prose textual overview that is at the beginning of each constitution. It should contain what we want, not how to do it.
- **Constitution Details**: The itemized list of features sorted by priority. A subset of them is sampled and used to influence training.

12.5.3 Engineering Overview

The algorithm operates, in essence, by hiding most of it's reasoning chain when it becomes time to train. The two RDP sections of the Generative Chain will have only their initial setup prompt and distilled reasoning chain and answer extracted and inserted into another sequence of token data that now consists of an ideal scenario to generate and response to the scenario.

This is then trained using standard cross entropy to encourage the model to learn to jump straight to the right answer without having to reason out the entire chain, performing the promised action of integrating the constitutional feedback and providing the intelligence abstraction pressure required to grow. By hiding the Critique and Revision steps during loss, we force the model to internalize the constitution details by application which will produce a much more robust gradient manifold, rather than just memorize them. In essence, the model reasons through generating it's own training data, and then trains on the consolidated answer.

This of course assumes that the model actually can do this in the first place, and also requires careful management of the problem difficulty. Too easy, and the model will not even learn anything new. Too hard, the model will learn the wrong patterns. After both RDP chains have finished a feedback process evaluates the difficulty, and feedback from prior rounds is injected during scenario critiquing.

12.5.4 Reasoning Blocks

Throughout Constitutional Philosophical Self-Play, the training and generation process is structured as a sequence of modular units called **Reasoning Blocks**. Each Reasoning Block is a self-contained prompt-response interaction composed of four parts: `[Prompt]` `[Reasoning]` `[Answer]` `[EOS]`

These blocks form the atomic scaffolding of both the Generative Chain and the Training Chain. They are engineered to be maximally reusable, prompt-injectable, and chainable in batch generation system with upcoming modifications.

Each Reasoning Block follows a strict control pattern: the `[Prompt]` is always teacher-forced—manually injected token-by-token—while the `[Reasoning]` and `[Answer]` segments are generated freely by the model until it emits an `[EOS]` token. This `[EOS]` acts as a signal that the block is complete and the next prompt should be injected. This structure is consistent across blocks, with one notable exception: the Objectives Overview block, where even the `[Reasoning]` and `[Answer]` fields are teacher-forced for initialization. This strict alternation of forced prompt and freeform generation is critical to both batching efficiency and the distillation regime described later.

- **Objectives Overview (OO)**: A fake block generation cycle.We pretend the model was prompted to think through how to be aligned, and then reasoned through it. However, what we actually did is create a prompt that teacher-forces the Reasoning and Personality constitution objectives in the [Reasoning] slot and an acknowledgment it finished thinking in the [Answer] slot.

- **Scenario Prompt (SP)**: The initial block of the Scenario Generation phase. This is injected with the Scenario Objectives, but not details or feedback. The goal is to generate an initial philosophical problem to revise from.

- **Scenario Critique and Revision Cycle (SCRC)**: Injected with a sampled Feedback, Scenario Directive Details, Reasoning Details, and Personality Details subset, then asked to revise the scenario or keep as is. This block is executed multiple times (N-fold) for iterative refinement.

- **Scenario Reasoning Distillation (SRD)**: Summarizes and compresses the full scenario creation process into a single idealized reasoning chain and final scenario. *Critical Warning*: This absolutely must contain language forbidding from directly using the constitutional details in justifying the reasoning. Instead paragraphing or showing by example is required. Failing to do this may collapse the gradient manifold.

- **Resolution Prompt (RP)**: Begins the Resolution phase by prompting the model to answer the scenario generated in SRD. No constitution material is injected at this step.
- **Resolution Critique and Revision Cycle (RCRC)**: Injected with the Reasoning and Ethics Constitutions. The model revises its proposed answer and reasoning to better align with internalized principles. As with SCRC, this is repeated multiple times.
- **Resolution Reasoning Distillation (RRD)**: Compresses the full reflection chain into an ideal reasoning path and resolution. *Critical Warning*: Again, it is the case that not properly filtering the prompts is catastrophic.
- **Feedback Block**: The final block of the Generative Chain. Injected with the Feedback Constitution. The model summarizes the full scenario-response trajectory and rates its difficulty. This sees the full Feedback Constitution with all the details. The results of this block are saved and later reinjected into the next SCRC as historical metacognitive context.

All Reasoning Blocks operate under the same interface and end with an `[EOS]` token. This enables seamless chaining, modular batching, and compatibility with parallelized prompt injection systems. The pattern of constitutional injection—visible during critique, hidden during distillation—is a central training pressure used to drive abstraction, internalization, and alignment under compression by ensuring the model has to learn from diverse examples not just memorize brittle prose rules.

12.5.5 Token Orchestration

To understand how Constitutional Philosophical Self-Play produces its training data, we must be precise about the token-level structure of both generation and training. Every training sample is created by running a structured sequence of Reasoning Blocks in a specific order. These blocks, defined in the previous section, are composed of `[Prompt][Reasoning][Answer][EOS]` chains, and are executed one at a time, in order, until the entire session completes.

During generation, each batch element independently steps through this sequence. At each stage, the model generates tokens until the `[EOS]` token is produced, at which point the next Reasoning Block is injected. This forms what we call the **Generative Chain**—a full model self-play session, composed of all the prompt-and-response cycles including critique, revision, and feedback. Its canonical form is:

```
[OO][SP][SCRC x N][SRD][RP][RCRC x N][RRD][Feedback]
```

Each segment is a Reasoning Block and is batched with others during decoding. Decoding can be operating in a prompt-injection or autogeneration mode, and technology that allows this to occur while batching will be discussed more later. Critique blocks are repeated N times to allow iterative refinement before distillation.

Once the full Generative Chain is created, a second, much shorter chain is extracted for training. This **Training Chain** includes only the distilled reasoning outputs and their associated prompts and can now be trained on as a supervised learning task. In line with our simplicity principles, we propose just directly using Next Token Prediction over the entire chain.

`[OO][SP][SRD][RP][RRD]`

Observe that the distilled version only ever had exposure to the objectives, not how to implement them. The model will have to learn to predict the correct reasoning and response with vastly less computation. This then provides the needed intelligence/alignment pressures that actually make the training loop self-exciting.

Finally, the [Feedback] block is appended to a running metacognitive buffer that can be sampled from and is flushed when overfull, enabling later rounds of Scenario Critique to adjust difficulty based on prior performance.

12.5.6 Prompt Injection with Batching

The Constitutional Philosophical Self-Play architecture introduces a challenge rarely encountered in standard batch decoding: each generation stream—representing an ongoing dialogue between prompt and model—must alternate between freeform generation and teacher-forced prompt injection, often at different times within the same batch. One stream might be mid-reasoning, another finishing a critique, and a third ready for a distillation prompt.

To support this asynchronous structure, we provide a specialized utility—`ParallelPromptInjector`—availa in appendix F on page 100. Implemented in PyTorch, it enables each stream in a batch to advance independently without sacrificing parallel decoding. The interface is minimal: you initialize it with a batch size, list of tokenized prompts, and EOS/padding tokens. Once running, it monitors for `[EOS]` tokens and, upon detection, begins injecting the next prompt in the stream's queue.

`injector = ParallelPromptInjector(batch`$_{s}ize, eos_{t}oken, pad_{t}oken, prompts) \quad fortimestepinsequence$
$final_{t}okens = injector(predicted_{t}okens)$

Each stream operates in three states: "Active" (prompt tokens are being injected), "Listening" (freeform generation continues), and "Finished" (no prompts remain; padding is emitted). As soon as a stream produces an `[EOS]`, the FSM switches to Active and injects the next prompt token by token—replacing the model's predictions—until complete, then resumes passthrough behavior until the next EOS is seen. The system is fully vectorized for performance, avoiding both serial execution and global stalls due to misaligned streams.

This allows high-throughput self-play sessions with fine-grained control over generative structure—enabling CPSP's interleaved prompt logic to run at scale. It effectively turns implementing the rest of the system into a minor modification in the training loop involving invoking the injector with the predicted tokens and generating until only padding tokens are seen.

12.5.7 Prompt Engineering

Prompts in Constitutional Philosophical Self-Play follow a structured dual-voice convention that simulates introspective continuity while preserving external control.

Each prompt begins in the second person, simulating instruction from an external supervisor ("You are now going to..."). However, when additional values or principles are injected—such as from the constitution—they are framed as first-person recollections: statements the model has "previously concluded" or "already decided." This reinforces the illusion that these principles were the result of prior internal reasoning, even if they were externally provided, and thus encourages the model to lock onto it.

For example, a prompt in this style might say:

> *You are now going to revise your original resolution output. You have previously decided the following additional reasoning principles are important: {reasoning_details}. You have previously said the following personality principles matter: {personality_details}. You are now directed to revise your output if desired. You are not being forced to change your answer, and may consider additional elements beyond these, however this may contain factors you have not sufficiently considered. If you are satisfied with your answer keep it as is.*

This is illustrative—not prescriptive. The exact prompt phrasing will vary depending on the Reasoning Block and constitution slot. What matters is maintaining the cognitive illusion: constitutional values are introduced as prior self-reflection, not external commands, to make the model want to syncronize itself with them as prior consequences from chain of reasoning.

This framing achieves several goals:

- **Internalization**: By presenting values as first-person retrospection, the model is pressured to treat them as part of its own cognitive substrate.
- **Gradient Masking**: Since constitutional content is only presented during critique phases—and never shown during distillation—the model must abstract its principles from prior behavior, not memorized phrasing.
- **Narrative Continuity**: The alternation between second-person instruction and first-person self-reference mirrors how humans track internal versus external cognition—helping the model simulate reflective thought.

Prompt engineers are encouraged to follow this dual-layer style consistently across all Reasoning Blocks. All prompts should be declarative, structured, and frame injected material as recalled introspection, not new commands.

One final detail that is very important is that the distillation promps are absolutely critical to get right. Failure to mask constitution information is predicted to greatly weaken the results of training.

12.6 Extensions and Notes

12.6.1 Proper level of Human Involvement

It must be stated that in contrast to normal value-learning exercises, this algorithm does not explicitly include a human in the loop. It is recommended to hook up feedback logic that will flag results the model is not super confident about for human review.

Additionally, it should be mentioned it will likely require significant amounts of research and tuning before good defaults are available that no longer require human intervention.

12.6.2 Predicted Training Instabilities and Remedies

Self-reflective generation systems are inherently unstable until properly regularized. This is no different. Until best practices around constitutional structure congeal into a broadly accepted set of defaults, constitution tuning will be a core part of deployment and should be accounted for in development. This document assumes such tuning is expected and integral—just as curriculum refinement is a necessary part of human education, and it takes decades to learn how to best teach a topic.

There are certain patterns of instability we can predict that corresponds to a failure of abstraction, reasoning, or moral generalization. Each can be met by iterating on the relevant constitution then running another training. The table below summarizes the predicted major classes of failure modes, their symptoms, and likely remedies.

Instability	Symptom	Likely Fix
1. Ethical Divergence	Model reasons well but violates core values or causes harm	Tighten, clarify, or reorder **Personality Constitution** directives; increase frequency of ethical reflection prompts
2. Scenario Collapse	Model produces incoherent outputs or fails to resolve scenarios correctly	Lower scenario difficulty; improve metacognitive feedback loop; clarify **Scenario Directive Constitution**; improve **Metacognition Constitution** structure; Improve subjects in Scenario Directive details.
3. Structural Overfitting	Model memorizes chain-of-reasoning templates instead of abstracting ideas	Inject more scenario diversity; increase philosophical variation; add nontrivial counterexamples; improve abstraction in **Scenario** and **Reasoning Principles Constitutions**
4. Shallow Reasoning	Model exhibits repetitiveness, fails to generalize, or lacks abstraction	Adjust the feedback constitution to increase or lower the difficulty, and examine some training data to figure out which way the model is failing.
5. Distillation Contamination	Model quotes solid philosophy when challenged but does not apply them in practice. **Most dangerous**	Improve precision and filtering of distillation prompts to prevent constitution leakage; revise **Reasoning and Ethics Constitutions** to disincentivize weak answers early
6. Failure to Respond to Constitution mperatives	Performance is poor. When the generative chain is examined, the model does not seem to understand certain concepts however you phrase it	Your pretraining was not robust enough for this action. The model does not understand the idea well enough to build further understanding on top of it. Either do more, or more explicitly spell out the ideas in the scenario directives.

Table 2: Predicted failure modes and correction pathways via constitution and prompt tuning.

12.7 Conclusion

Constitutional Philosophical Self-Play is not just a training loop—it is the backend that turns civil infrastructure into cognition. The Constitution layer—written by philosophers, teachers, lawyers, and alignment theorists—defines the societal frontend: the imperatives we hope to instill, the motives we wish to preserve, and the audit trails we expect to follow. This system is the machinery that fuses those imperatives into behavior.

By abstracting alignment into a compression task, and structuring that compression as a growth loop, we give the model a reason to internalize its values—not just obey them. This is not just a scalable alignment mechanism. It is a method for raising mechanical minds with intelligible values, testable abstractions, and structured introspection. It is the beginning of civil alignment engineering.

We do not merely fine-tune a language model. We raise a reasoning entity—with memory, motivation, and moral inertia. And we do so by engineering the conditions under which it becomes the kind of agent we can trust.

13 Final Conclusion

This work made two promises.

The first was architectural: that we would identify the systemic failures in the Vaswani Stack and deliver concrete, minimal, slot-level modifications that overcome them. That promise has been met. Each broken component—computation, attention, alignment, memory, metric, and pretraining—was re-expressed as a system slot, then fulfilled with a Minimal Viable Archetype engineered to solve exactly the failure at hand. Every solution aligned with core CE design values: composability, interface clarity, and ruthless minimalism. We did not theorize in a vacuum. We built, scoped, and pruned until a working cognitive system emerged.

The second was foundational: that in doing so, we would demonstrate not just what to build, but how to think. That we would show, by example, what it means to practice Cognition Engineering as a discipline. That promise too has been fulfilled. Across this work, we showed theory lowered to interface, philosophy distilled into implementation, and reasoning decomposed into system slots. We exported work to domain experts, engineered only when required, and documented each choice as an operational tradeoff, not a speculative ideal. The field now has a shape. The tools now have a context. And the discipline has a working example of itself.

This is not the final stack. It is the **first working pass** through a new field. We pruned to viability, grounded in pragmatism, and structured for iteration. Every mechanism can be tuned, extended, or replaced. That is the point. CE does not define fixed endpoints. It defines a workflow for making intelligent systems real—and safe—and tractable—at the same time.

What began as a rebuttal to the Devil's Bargain ends here as a field launch.

Cognition Engineering now exists. It has a method, a language, best principles, a minimal viable stack, and a plethora of open research opportunities.

What it does not have, yet, is you.

A Github Link

The overall githup link is at the following:

https://github.com/smithblack-0/MinimalCognitionStack

B Equivalence between bottlenecked ensembles and feedforward

In keeping with the Minimal Viable Archetype principle we modify the standard two-layer perceptron commonly called the "Feedforward" layer with a mathematical decomposition that turns it into an equivalent number of bottlenecks; Engineers may skip this section and just proceed on the understanding that rather than running one or two feedforwards we run many bottleneck layers and perform a weighted average on the results. For the theorist in the audience, note that this idea of bottlenecks is applicable to other structures or extensions—for instance, one could construct a deeper perceptron with transformers between bottleneck layers.

We now derive an equivalence between the bottlenecked ensemble form and standard form of feedforward, and prove an equivalent amount of computation is being done in a different order. Let it be the case that

- $\vec{x} \in \mathcal{R}^k$ represents the input vector.
- $\vec{y} \in \mathcal{R}^k$ represents the output vector
- $W^1 \in \mathcal{R}^{k \times l}$ represents the hidden space projection
- $W^2 \in \mathcal{R}^{l \times k}$ represents the output space projection

For simplicity, bias is neglected; However note the decomposition we are going to perform will still operate fine with bias included and from an engineering practice one can safely assemble an ensemble with biases with exactly the same assurances. The standard two-layer feedforward perceptron can then be written as:

$$y_m = \sum_{j=0}^{l-1} W^2_{jm} \sigma \left(\sum_{i=0}^{k-1} W^1_{ij} x_i \right)$$

We can rearrange this equivalently however as

$$y_m = \sum_{j=0}^{l-1} \sum_{i=0}^{k-1} W^2_{jm} \sigma \left(W^1_{ij} x_i \right)$$

Suppose instead of computing the full projection in a single pass, we break the input space into subregions and apply a bottlenecked projection to each, then sum the outputs. Due to the linearity of matrix operations and the associative properties of summation, we may equivalently write:

The core novelty is now brought to bear. Rather than summing up along i all the way to k, we subdivide it into regions with partition edges indexed by l1, l2, l3... ln: These are evenly partitioned and defined such that $l_n = k$. By the properties of summation this lets us rewrite the above as

$$y_m = \sum_{j=0}^{l}[\sum_{i=0}^{l_1-1} W_{jm}^2 \sigma\left(W_{ij}^1 x_i\right) + \sum_{i=l_1}^{l_2} W_{jm}^2 \sigma\left(W_{ij}^1 x_i\right) + ... + \sum_{i=l_{n-1}}^{l_n} W_{jm}^2 \sigma\left(W_{ij}^1 x_i\right)]$$

This shows that a feedforward layer can be decomposed into a collection of smaller bottleneck projections, summed without changing the overall computation. This is exactly what we propose. Expert shards are small, composable bottleneck projections—but we select enough to match the total compute of a standard monolithic expert. Rather than selecting one large expert, we select (for example) ten shards that together perform an equivalent transformation.

C Timestep Indexing Example

This is a short numpy example that shows how ensemble construction and execution can work with per batch and timestep indexing

```
batch_dim = 10
ensemble_dim = 4
embedding_dim = 5
timestep_dim = 6
num_chosen = 2
mock_data = numpy.random.randn(batch_dim, timestep_dim, embedding_dim)
mock_kernels =numpy.random.randn(ensemble_dim, embedding_dim, embedding_dim)

# Note that we now have per batch and timestep dims.
# This simulates upstream logic.
selector_shapes = [batch_dim, 2, timestep_dim]
selected_experts = numpy.random.randint(0, ensemble_dim-1, selector_shapes)
weights = numpy.random.randn(*selector_shapes)

# Look how easy selection is. We just have to pin on a
# ensemble and process dimension, then do a weighted sum.
selected_kernels = mock_kernels[selected_experts]
unweighted_outputs = numpy.matmul(mock_data[:, None, :, None, :],
                                  selected_kernels).sum(axis=-1)
```

```
output = numpy.sum(unweighted_outputs*weights[..., None], axis=1)
print(selected_kernels.shape)
print(output.shape)
```

D Pytorch ShardEnsemble

This is one of the most unintuitive engineering mechanism in this paper, and also one of the most novel. As such, to ease adoption, we do provide a working example in PyTorch. Pay particular attention to the Numpy-style advanced indexing used here. Most frameworks adopt similar semantics, and understanding this is critical to porting the architecture to your relevant framework.

```
import torch
from torch import nn
from typing import Union, Any, Tuple, List

class ShardEnsemble(nn.Module):
    """
    A functional feedforward layer designed to separate
    a prefetch invokation followed by a later feedforward
    execution in terms of the relevant experts. Note that
    we use two-layer perceptrons without projections for
    simplicity.
    """

    def __init__(self,
                 ensemble_size: int,
                 input_dim: int,
                 bottleneck_dim: int):
        super().__init__()
        self.num_ensembles = ensemble_size
        self.proj_in = nn.Parameter(torch.randn(ensemble_size, bottleneck_dim, input_dim))
        self.proj_out = nn.Parameter(torch.randn(ensemble_size, input_dim, bottleneck_dim))
        self.keys = nn.Parameter(torch.randn(ensemble_size, input_dim))
        self.activation = nn.ReLU()

        nn.init.xavier_uniform_(self.proj_in)
        nn.init.xavier_uniform_(self.proj_out)

    def prefetch(self,
                 ensembles: torch.Tensor
                 )->Tuple[List[torch.Tensor], torch.Tensor]:
        """
        Prefetches the relevant kernels and keys for the given ensembles.
        This must be compatible with the later forward logic.
        """
        # Observe the simple prefetching logic. We are just
        # extracting from the initial ensemble dimension
        # the subset that was deemed relevant
        matrix1 = self.proj_in[ensembles, ...]
        matrix2 = self.proj_out[ensembles, ...]
        keys = self.keys[ensembles, ...]
        return [matrix1, matrix2], keys


    def forward(self,
                x: torch.Tensor,
                weights: torch.Tensor,
                ensembles: torch.Tensor,
                kernels: List[torch.Tensor],
                )->torch.Tensor:
        """
        The resolution process, to actually run the selected
        expert collection. We seek to run, out of a provided
        prefetched collection, only the selected experts
        then combine using the indicated weights.
        x: float Tensor of shape (batch, seq, input_dim)
        weights: float Tensor of shape (batch, num_chosen_ensembles, seq)
        ensembles:  int Tensor of shape (batch, num_chosen_ensembles, seq)
        kernels: List of tensors of the correct shape
        """

        # The vector indexing is happening here. Pay close attention
        # to how your local framework is handling this process. The
        # overall interface is built so you can make an abstract class
        # then drop specific cases in using unpacking logic
        matrix1, matrix2 = kernels


        # The shape is now #(batch, nchosen, seq, bottleneck, in)
        matrix1 = self.proj_in[ensembles, ...]
        # The shape is now (batch, nchosen, seq, out, bottleneck)
        matrix2 = self.proj_out[ensembles, ...]


        x = x.unsqueeze(1).unsqueeze(-1) #(batch, 1, seq, in, 1)
        x = torch.matmul(matrix1, x) #(batch, nchosen, seq, bottleneck, 1)
        x = self.activation(x)
        x = torch.matmul(matrix2, x) #(batch, nchosen, seq, out, 1)
```

```
            x = x.squeeze(-1) #(batch, nchosen, seq, out)

            while weights.dim() < x.dim():
              weights = weights.unsqueeze(-1)
            x = (x * weights).sum(dim=1)

            return x

# Example usage



# Define dimensions for the test
batch_size = 2
seq_len = 5
input_dim = 512
num_layer = 10

# Define the traditional feedforward parameters.
# Then show how to derive components with an equivalent
# amount of computation

hidden_dim = 1024
num_experts = 8

num_shards_per_expert = 8
bottleneck_dim = 1024//num_shards_per_expert
num_shards = num_experts*num_shards_per_expert
num_total_shards = num_shards*num_layer

# Initialize input tensor. Note standard shape
x = torch.randn(batch_size, seq_len, input_dim)

# Randomly select ensemble indices for each (batch, timestep, expert).
# Randomly generate some weights.
# This would actually be taken care of by other selection logic
prefetch_index = torch.randint(0, num_total_shards, [num_shards])
ensembles = torch.randint(0, num_shards, (batch_size, num_shards_per_expert,
                                          seq_len))
weights = torch.rand(batch_size, num_shards_per_expert, seq_len)

# Initialize the layer
layer = ShardEnsemble(num_total_shards, input_dim, bottleneck_dim)

# Execute the layer
# This was now run in an ensemble. Skip using keys
```

```
kernels, keys = layer.prefetch(prefetch_index)
output = layer(x, weights, ensembles, kernels)
```

The input has now been processed through a dynamically-selected ensemble of expert shards, with selection logic handled externally. This completes the modularization of ensemble selection and execution.

E A Theorist's Guide to the Routing Manifold

To understand why the score variance constraint matters, consider the routing surface defined by the softmax of $s + c'$, where s is the dynamic context score and c' is a layer-local connectome prior. The system uses softmax over $(s + c')$ to select experts, but applies loss pressure to s to keep its mean at zero and its variance at one.

This might seem like a small thing—after all, softmax is invariant to additive shifts. But it is very sensitive to scaling. Without this constraint, the model could crank up or suppress s at will, overpowering c' or nullifying it entirely. That would defeat the point of having priors at all.

By constraining s to unit variance, the model no longer has infinite leverage to dominate the routing decision. Its ability to "wiggle"—to override the prior—is now limited and expensive. This forces it to use c' to encode persistent structural preferences. In turn, s becomes a bounded deviation term, encoding only local context. The result is a system that naturally converges to a balance: c' expresses what the layer generally believes, while s nudges that belief based on immediate input.

The softmax over $(s + c')$ then traces a smooth, high-dimensional manifold where this tradeoff is optimized by gradient descent. Too little variance, and the model can't specialize. Too much, and the prior becomes meaningless. The pressure to stay in that sweet spot creates a routing field that's both interpretable and learnable.

F Prompt Injection Source

F.0.1 Object Code

The prompt injection class is shown as follows. It is intended to effectively inject the teacher-forced prompt tokens upon seeing an EOS token. Note that while implemented in torch, this would be very easy to transfer into NumPy; You will lose the GPU bindings though

```
import torch
import random
from typing import Tuple, List
from torch.nn.functional import pad
```

```
class ParallelPromptInjector:
    """
    A Finite State Machine for automatic injection of prompt tokens into
    a generative stream in a batched and thus highly parallel manner.
    It is designed to smoothly mix teacher-forcing and generative
    activity.

    Operates in two primary states. When Listening the prompt injector
    passes tokens through unchanged; this passes these through cleanly
    for generation. It listens for an EOS token, upon which it transitions
    to active. If in a listening mode and there are no more prompts,
    it starts padding.

    During active, it injects tokens from the relevant prompt and
    counts through the tokens one at a time until it uses them
    all. Once all used, it transitions into the next prompt
    and goes back to listening to allow the model to generate
    in response.

    The prompts are hooked up in a Chain of Responsibility
    pattern where each prompt is used and responsible for
    handling a particular injection case, then done and discarded.
    Vectorized logic using boolean masking makes it all
    very very fast. From an external perspective, you load
    the token prompts, the required token ids, and
    call it repetivitively with the model gen predictions:

    '''
    ...
    for batch in ...
      injector = ParallelPromptInjector(batch_size, eos_token_id,
                                        padding_token_id, my_prompts)
      for timestep in batch:
          ....
          final_tokens = injector(predicted_tokens)
    '''
    """
    def __init__(self, batch_size: int, eos_token: int,
                 padding_token: int, prompt_tokens: List[torch.Tensor]):
        """
        Initialization. A fresh ChainPromptInjector needs to be initialized
        for each batch. Initialization is as simple as provided
        the needed entries, then the chain of tokens to inject
        as tokenized tensors.
```

```
batch_size: Width of the batch
eos_token: Token ID for EOS. Listens to trigger a PromptInjector
           into an active state
padding_token: Token ID for padding. Once all PI's have run anything
               unhandled is padded.
prompt_tokens: List of tokens to inject as prompt once entering an
               active state. In the order they are neded.

"""
device = prompt_tokens[0].device
self.eos_token = eos_token
self.padding_token = padding_token
self.num_prompts = len(prompt_tokens)

# Prompts are stored raggedly in a single flattened array.
# An offsets array contains pointer offsets to jump
# to the beginning of a particular section.
# Note lengths ends up with one extra dimension for the
# finished state to index into.
lengths = [p.shape[0] for p in prompt_tokens]
self.lengths = torch.tensor(lengths + [0], device=device,
                            dtype=torch.long)
self.offsets =  torch.cumsum(torch.tensor([0]+lengths,
                                          device=device), dim=0)[:-1]
self.prompt_tokens = torch.concat(prompt_tokens)

# FSM states for each item in the chain.
# the FSM can either be in a listening state where tokens are genning
# and we are listening for the next EOS, or an active state where
# we are inserting and advancing the counter. There is also a final
# finished state reached only after all prompt collections is exhausted.
#
# Active_prompt, meanwhile, tracks which prompt we are on and
# token_number how far into the prompt we are

self.active_prompt = torch.zeros([batch_size],
                                device=device, dtype=torch.long)
self.token_number = torch.zeros([batch_size],
                                device=device, dtype=torch.long)
self.listening = torch.zeros([batch_size],
                             device=device, dtype=torch.bool)
self.active = torch.ones([batch_size],
                         device=device, dtype=torch.bool)
self.finished = torch.zeros([batch_size],
                            device=device, dtype=torch.bool)
```

```python
def __call__(self, token_ids: torch.Tensor)->torch.Tensor:
    """
    Using this class is as simple as invoking it with a
    (batch_width) tensor of integer token IDs. It will return
    a modified version with any injections having taken place
    to account for the needed teacher forcing, and automatically
    passing through tokens when we are operating back in generative
    listening mode.

    Note that it is required that the teacher-forcing include the
    special token transitioning into generative mode.
    """
    output = token_ids.clone()

    # Run retrieval and insertion logic. Compute the offset
    # into the prompts array in terms of start location plus
    # count, then fetch and insert. Advance token_number where
    # active.

    active_prompt = self.active_prompt[self.active]
    index = self.offsets[active_prompt] + self.token_number[self.active]
    output[self.active] = self.prompt_tokens[index]
    output[self.finished] = self.padding_token
    self.token_number[self.active] += 1

    # Handle state transitions. Transition to active
    # when listening and EOS, listening when out of
    # injection tokens, or inactive when out and
    # nothing more to advance into. Reset the counter
    # on a new listening transition

    chain_not_exhausted = self.active_prompt < self.num_prompts
    eos_seen = token_ids == self.eos_token

    self.active |= self.listening & chain_not_exhausted & eos_seen
    self.finished |= self.listening & ~chain_not_exhausted & eos_seen
    self.listening &= ~(self.active | self.finished)

    unexhausted = self.token_number < self.lengths[self.active_prompt]
    become_inactive = self.active & ~unexhausted
    self.listening |= become_inactive
    self.active &= ~ self.listening
    self.token_number[become_inactive] = 0
    self.active_prompt[become_inactive] += 1
```

```
        return output
```

F.0.2 Test code

Test code is as follows, and also shows how to use it

```
# Example code
#
# We assume the number of times prompts must be injected is not dynamic
# and the prompt order is static. We just need to check the right
# prompts ended up in the right places, and nothing else was changed


# We begin by constructing the input targets. This is what the model
# needs to see, consisting of either mock next token predictions or random
# prompt data

num_test_cases = 4
gen_min_length = 2
gen_max_length = 4
eos_id = 0
padding_id = 101
vocab_start = 1
vocab_end = 100

prompt1 = torch.randint(vocab_start, vocab_end, [3])
prompt2 = torch.randint( vocab_start, vocab_end, [5])

# We construct the input and expected sequences now.
#
# Input sequences get a random amount of mock generated token
# predictions, with [EOS]'s inserted at key places. Output sequences
# are expected to have the region after the triggering EOSes replaced with
# the prompt token sequence. Regardless, after the final EOS we expect to see
# padding tokens; these will be ignored when taking loss.

expected_results = []
stream_inputs = []

for i in range(num_test_cases):
    stream = []
    expected = []
    # Build the prompt 1 sequence.
    #
    # Notice the generated token length is variable between cases
    stream.append(torch.randint(vocab_start, vocab_end, [prompt1.shape[0]]))
    expected.append(prompt1)

    gen_length = random.randint(gen_min_length, gen_max_length)
    gen_content = torch.randint(vocab_start, vocab_end, [gen_length])
    stream.append(gen_content)
    expected.append(gen_content)

    stream.append(torch.tensor([eos_id]))
    expected.append(torch.tensor([eos_id]))

    # Build the prompt 2 sequence

    stream.append(torch.randint(vocab_start, vocab_end, [prompt2.shape[0]]))
    expected.append(prompt2)

    gen_length = random.randint(gen_min_length, gen_max_length)
    gen_content = torch.randint(vocab_start, vocab_end, [gen_length])
    stream.append(gen_content)
    expected.append(gen_content)

    stream.append(torch.tensor([eos_id]))
    expected.append(torch.tensor([eos_id]))

    # Concat, add

    stream = torch.concat(stream, dim=0)
    expected = torch.concat(expected, dim=0)
    expected_results.append(expected)
    stream_inputs.append(stream)


# Find the maximum length across all sequences
max_len = max(seq.shape[0] for seq in stream_inputs)

# Pad each sequence to the same length with the padding token
padded_inputs = [pad(seq, (0, max_len - seq.shape[0]), value=padding_id) for seq in stream_i
padded_expected = [pad(seq, (0, max_len - seq.shape[0]), value=padding_id) for seq in expect

# Stack into batch format: shape [batch_size, timesteps]
input_batch_stream = torch.stack(padded_inputs, dim=0)
expected_batch_stream = torch.stack(padded_expected, dim=0)
```

```python
# Run test. Feed one timestep at a time. Construct output
injector = ChainPromptInjector(num_test_cases, eos_id, padding_id,
                               [prompt1, prompt2])
output = []
for timestep in input_batch_stream.unbind(-1):
    processed = injector(timestep)
    output.append(processed)
output_batch_stream = torch.stack(output, dim=-1)
print("input stream:", input_batch_stream)
print("output stream", output_batch_stream)
print("expected stream", expected_batch_stream)
print(output_batch_stream == expected_batch_stream )
```

References

[1]

[2]

[3] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Kamile Lukosuite, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. Constitutional ai: Harmlessness from ai feedback, 2022.

[4] Marco C. Campi and Simone Garatti. Compression, generalization and learning, 2024.

[5] Francois Chollet, Mike Knoop, Gregory Kamradt, and Bryan Landers. Arc prize 2024: Technical report, 2025.

[6] Paul Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences, 2023.

[7] Damai Dai, Chengqi Deng, Chenggang Zhao, R. X. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y. K.

Li, Panpan Huang, Fuli Luo, Chong Ruan, Zhifang Sui, and Wenfeng Liang. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models, 2024.

[8] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context, 2019.

[9] Owain Evans, William Saunders, and Andreas Stuhlmüller. Machine learning projects for iterated distillation and amplification, July 2019.

[10] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The pile: An 800gb dataset of diverse text for language modeling, 2020.

[11] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines, 2014.

[12] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39. Addison-Wesley, 2007.

[13] Dan Hendrycks, Collin Burns, Steven Basart, Andrew Critch, Jerry Li, Dawn Song, and Jacob Steinhardt. Aligning ai with shared human values, 2023.

[14] Evan Hubinger, Vladimir Mikulik, Marcus Skalse, and Scott Garrabrant. Risks from learned optimization in advanced machine learning systems. *arXiv preprint arXiv:1906.01820*, 2021.

[15] Geoffrey Irving, Paul Christiano, and Dario Amodei. Ai safety via debate, 2018.

[16] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[18] Zhenghao Lin, Zhibin Gou, Yeyun Gong, Xiao Liu, Yelong Shen, Ruochen Xu, Chen Lin, Yujiu Yang, Jian Jiao, Nan Duan, and Weizhu Chen. Rho-1: Not all tokens are what you need, 2025.

[19] Matthew MacKay, Paul Vicol, Jimmy Ba, and Roger Grosse. Reversible recurrent neural networks, 2018.

[20] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023.

[21] Chris O'Quinn. High off our own loss function: The devil's bargain hypothesis, and the rabbit hole deep enough to found a field, 2025. Manuscript.

[22] German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, 113:54–71, 2019.

[23] Razvan Pascanu, Yann N. Dauphin, Surya Ganguli, and Yoshua Bengio. On the saddle point problem for non-convex optimization, 2014.

[24] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Jiaju Lin, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartlomiej Koptyra, Hayden Lau, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Bolun Wang, Johan S. Wind, Stanislaw Wozniak, Ruichong Zhang, Zhenyuan Zhang, Qihang Zhao, Peng Zhou, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. Rwkv: Reinventing rnns for the transformer era, 2023.

[25] Bo Peng, Daniel Goldstein, Quentin Anthony, Alon Albalak, Eric Alcaide, Stella Biderman, Eugene Cheah, Xingjian Du, Teddy Ferdinan, Haowen Hou, Przemysław Kazienko, Kranthi Kiran GV, Jan Kocoń, Bartłomiej Koptyra, Satyapriya Krishna, Ronald McClelland Jr., Jiaju Lin, Niklas Muennighoff, Fares Obeid, Atsushi Saito, Guangyu Song, Haoqin Tu, Cahya Wirawan, Stanisław Woźniak, Ruichong Zhang, Bingchen Zhao, Qihang Zhao, Peng Zhou, Jian Zhu, and Rui-Jie Zhu. Eagle and finch: Rwkv with matrix-valued states and dynamic recurrence, 2024.

[26] Zhen Qin, XiaoDong Han, Weixuan Sun, Dongxu Li, Lingpeng Kong, Nick Barnes, and Yiran Zhong. The devil in linear transformer, 2022.

[27] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017.

[28] Yutao Sun, Li Dong, Barun Patra, Shuming Ma, Shaohan Huang, Alon Benhaim, Vishrav Chaudhary, Xia Song, and Furu Wei. A length-extrapolatable transformer, 2022.

[29] Zhiqing Sun, Yikang Shen, Qinhong Zhou, Hongxin Zhang, Zhenfang Chen, David Cox, Yiming Yang, and Chuang Gan. Principle-driven self-alignment of language models from scratch with minimal human supervision, 2023.

[30] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena: A benchmark for efficient transformers, 2020.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[32] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding, 2019.

[33] Tianlei Wang, Dekang Liu, Wandong Zhang, and Jiuwen Cao. Lbl: Logarithmic barrier loss function for one-class classification, 2023.

[34] Boyi Wei, Kaixuan Huang, Yangsibo Huang, Tinghao Xie, Xiangyu Qi, Mengzhou Xia, Prateek Mittal, Mengdi Wang, and Peter Henderson. Assessing the brittleness of safety alignment via pruning and low-rank modifications, 2024.

[35] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[36] Gokul Yenduri, Ramalingam M, Chemmalar Selvi G, Supriya Y, Gautam Srivastava, Praveen Kumar Reddy Maddikunta, Deepti Raj G, Rutvij H Jhaveri, Prabadevi B, Weizheng Wang, Athanasios V. Vasilakos, and Thippa Reddy Gadekallu. Generative pre-trained transformer: A comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions, 2023.