Type-Safe Context Engineering: Lessons from Protein Folding for Building Reliable AI Systems

Matthew Long
Independent Researcher, Chicago, IL
matthew@yonedaai.com

The YonedaAI Collaboration
YonedaAI Research Collective

December 14, 2025

Abstract

We present a framework for *type-safe context engineering* inspired by insights from protein folding and AlphaFold's success. Just as protein sequences deterministically specify native structures under Anfinsen's thermodynamic hypothesis, we argue that well-designed context should deterministically constrain model responses. We formalize context engineering using dependent type theory, where prompts are types, responses are terms, and constraints function as typing rules. This framework provides: (1) a principled foundation for understanding when prompting succeeds and fails, (2) design patterns for constructing "type-safe" contexts that reliably elicit desired behaviors, (3) a taxonomy of failure modes analogous to type errors, and (4) architectural recommendations for "chaperone" systems that guide inference. We demonstrate that type safety in context engineering—where context uniquely determines the class of valid responses—is a key predictor of prompting success. Our framework has immediate practical applications for prompt engineering, agent design, and AI safety, while providing theoretical foundations connecting language model behavior to well-studied concepts in programming language theory.

Keywords: context engineering, prompt engineering, type theory, dependent types, language models, AI safety, constraint satisfaction, protein folding, AlphaFold

Contents

1 Introduction

The emergence of large language models (LLMs) has created a new paradigm for human-computer interaction: rather than writing programs that explicitly specify computations, users provide *context* (prompts, examples, instructions) that guides models to produce desired outputs. This shift from programming to prompting has been called the "natural language programming" revolution [Reynolds & McDonell, 2021].

Yet context engineering—the art and science of designing prompts that reliably elicit desired model behaviors—remains largely empirical. Practitioners rely on heuristics, trial-and-error, and folklore passed through blog posts and social media. There is no principled theory explaining why some prompts work reliably while others fail unpredictably.

This paper proposes such a theory, drawing on a surprising source: the type-theoretic framework for understanding protein folding and AlphaFold's success [Long, 2025].

1.1 The Protein Folding Analogy

Consider the protein folding problem: given an amino acid sequence, predict the three-dimensional structure the protein will adopt. For decades, this was considered intractable. Then AlphaFold2 [Jumper et al., 2021] achieved near-experimental accuracy, effectively solving the problem.

Why did AlphaFold succeed? The conventional answer focuses on architecture (transformers, attention) and scale (data, compute). But this misses the deeper reason: **protein folding is type-safe**.

Anfinsen's thermodynamic hypothesis [Anfinsen, 1973] states that amino acid sequences *uniquely determine* native structures under physiological conditions. In type-theoretic terms, the sequence is a type, the structure is its canonical inhabitant, and folding is proof search. AlphaFold succeeded because it was learning to navigate a well-structured constraint space—not discovering physics, but learning efficient proof search heuristics.

1.2 The Central Claim

We propose that the same framework applies to context engineering:

Central Thesis

Type-safe context engineering is the practice of designing prompts (contexts) that function as *type specifications*—constraining the space of valid responses sufficiently that the model can reliably "inhabit" the type through inference, analogous to how protein sequences constrain the folding search space.

When context is type-safe, prompting succeeds reliably. When context is type-unsafe (underdetermined, ambiguous, or contradictory), prompting fails unpredictably. This framework explains both successes and failures of current prompting practices.

1.3 Contributions

This paper makes the following contributions:

1. **A formal framework** connecting context engineering to type theory, with precise definitions of context types, response terms, and typing judgments.

2. **Design principles** for type-safe context construction, derived from the protein folding analogy.

3. **A taxonomy of failure modes** explaining when and why prompting fails, mapped to type-theoretic concepts.
4. **Architectural patterns** for "chaperone" systems that improve reliability, inspired by molecular chaperones.
5. **Connections to AI safety**, showing how type-safe context engineering relates to alignment and robustness.
6. **Practical guidelines** for practitioners, translating theory into actionable recommendations.

1.4 Organization

Section 2 reviews necessary background on context engineering and type theory. Section 3 develops the type-theoretic framework for context engineering. Section 4 presents design principles for type-safe contexts. Section 5 analyzes failure modes. Section 6 discusses chaperone systems. Section 7 connects to AI safety. Section 8 provides practical guidelines. Section 9 discusses related work. Section 11 concludes.

2 Background

2.1 Context Engineering: The Current State

Context engineering encompasses all techniques for designing inputs to language models that elicit desired outputs. This includes:

- **Prompt engineering**: Crafting textual instructions and examples
- **System prompts**: Defining model personas and constraints
- **Few-shot learning**: Providing example input-output pairs
- **Chain-of-thought**: Eliciting reasoning steps
- **Tool use**: Providing APIs and function signatures
- **RAG**: Augmenting context with retrieved documents
- **Agent scaffolding**: Structuring multi-step interactions

Current practice is largely empirical. The "prompt engineering" literature consists primarily of:

1. Collections of effective prompts for specific tasks
2. Heuristics ("be specific," "provide examples")
3. Framework-specific techniques (few-shot, chain-of-thought)
4. Empirical evaluations of prompt variations

What's missing is a *principled theory* explaining *why* certain techniques work, *when* they will succeed or fail, and *how* to systematically design reliable contexts.

2.2 Type Theory Essentials

We briefly review the type-theoretic concepts central to our framework.

2.2.1 Types as Propositions

Under the Curry-Howard correspondence [Wadler, 2015], types correspond to propositions and terms correspond to proofs:

Type Theory	Logic
Type A	Proposition P
Term $t : A$	Proof of P
Function $A \to B$	Implication $P \Rightarrow Q$

A type "has inhabitants" if there exist terms of that type. Finding such terms is *proof search*.

2.2.2 Dependent Types

In dependent type theory [Martin-Löf, 1984], types can depend on values:

```
-- Vector of length n: type depends on value
data Vec (a :: Type) (n :: Nat) where
  Nil  :: Vec a 0
  Cons :: a -> Vec a n -> Vec a (n + 1)

-- head only works on non-empty vectors
head :: Vec a (n + 1) -> a
```

This is the key structure for our framework: *response types depend on context values*.

2.2.3 Type Safety

A type system is *safe* if well-typed programs don't "go wrong"—they either produce values or diverge cleanly. Type safety comprises:

- **Progress**: A well-typed term is either a value or can take a step
- **Preservation**: Stepping preserves type

2.2.4 Normal Forms and Canonicity

A *normal form* is a fully reduced term. *Strong normalization* guarantees that all reduction sequences terminate in a normal form. *Canonicity* states that closed terms of ground type reduce to canonical values.

For our purposes: if a type has a unique normal form, then proof search can reliably find it.

2.3 The Protein Folding Framework

We summarize the key insights from type-safe protein folding [Long, 2025]:

1. **Sequence as Type**: An amino acid sequence specifies constraints on valid structures.
2. **Structure as Term**: The native structure is the canonical inhabitant of the sequence type.
3. **Anfinsen as Type Safety**: The sequence uniquely determines the structure (up to equivalence).
4. **Folding as Proof Search**: Finding the native structure is constraint satisfaction.
5. **AlphaFold as Learned Heuristics**: The network learns efficient proof search, not physics.

6. **MSAs as Type Annotations**: Evolutionary data provides additional constraints.

7. **Chaperones as Type Checkers**: Molecular chaperones guide and validate folding.

The key insight: AlphaFold succeeded because protein folding is a *type-safe* problem—the constraints (thermodynamics) guarantee a unique solution exists and can be found efficiently.

3 The Type-Theoretic Framework

We now develop the formal correspondence between context engineering and type theory.

3.1 The Fundamental Correspondence

Context Engineering	**Type Theory**
Context / Prompt	Type specification
Valid response	Term inhabiting type
Model inference	Proof search / term construction
Constraints in context	Typing rules
Format requirements	Refinement types
Examples (few-shot)	Type annotations / hints
System prompt	Base type / context
Chain-of-thought	Step-wise proof construction
Tool definitions	Function type signatures
Temperature = 0	Deterministic normalization
Hallucination	Type error / ill-typed term
Prompt injection	Constraint violation

Table 1: Correspondence between context engineering and type theory

3.2 Formal Definitions

Definition 3.1 (Context Type System)**.** *A* context type system $\mathcal{C}$ *consists of:*

(i) A set $\mathcal{P}$ of prompts (context specifications)

(ii) A set $\mathcal{R}$ of responses (text strings)

(iii) A validity relation $\vdash$ where $p \vdash r$ means response r is valid for prompt p

(iv) Environmental parameters E (model, temperature, system prompt)

Definition 3.2 (Response Type)**.** *For a prompt $p \in \mathcal{P}$ and environment E, the* response type *is:*

$$\mathsf{Response}(p, E) \triangleq \{r \in \mathcal{R} \mid p \vdash r \text{ under } E\}$$

This is the set of responses that validly satisfy the context constraints.

Definition 3.3 (Type-Safe Context)**.** *A prompt p is* type-safe *under environment E if $|\mathsf{Response}(p, E)|$ is "small"—ideally 1, or a small equivalence class of semantically identical responses.*

Definition 3.4 (Type-Unsafe Context)**.** *A prompt p is* type-unsafe *if $|\mathsf{Response}(p, E)|$ is large or unbounded—many qualitatively different responses could satisfy the constraints.*

3.3 The Inference Judgment

Model inference can be formalized as a judgment:

$$\Gamma; p \vdash_M r : \mathsf{Response}(p)$$

Read as: "Under context Γ and prompt p, model M produces response r inhabiting the response type."

The key question is whether this judgment is *well-defined*—whether the constraints in p sufficiently determine r.

3.4 Dependent Response Types

The response type depends on the context:

```
-- Response type depends on context
data Response (c :: Context) where
  -- A response is valid only if it satisfies context constraints
  MkResponse :: String -> ConstraintsSatisfied c -> Response c

-- Model inference: proof search
infer :: (c :: Context) -> Model -> Maybe (Response c)
infer c model = proofSearch (constraints c) model
```

This captures the essential structure: the prompt specifies constraints, and inference is the process of finding a term (response) that satisfies those constraints.

3.5 Constraint Propagation

Just as protein folding involves progressive constraint satisfaction (local $\rightarrow$ global), context engineering involves hierarchical constraints:

1. **Syntactic constraints**: Format, length, structure
2. **Semantic constraints**: Topic, content, accuracy
3. **Pragmatic constraints**: Tone, persona, style
4. **Meta-constraints**: Consistency, coherence

Effective contexts propagate constraints from specific to general, narrowing the valid response space hierarchically.

3.6 Type Safety Theorem

Theorem 3.5 (Context Type Safety)**.** *For a type-safe prompt p with sufficient constraints:*

1. ***Progress**: The model will produce a response (not hang or error)*
2. ***Preservation**: The response will satisfy the context constraints*

This is not a mathematical theorem but an empirical claim: well-designed contexts yield reliable behavior.

4 Design Principles for Type-Safe Context

Drawing from the protein folding analogy, we derive principles for constructing type-safe contexts.

4.1 Principle 1: Constraint Sufficiency

Principle 4.1 (Anfinsen Principle)**.** *A context is type-safe when it provides sufficient constraints to uniquely determine the response (up to semantically irrelevant variation).*

In protein folding, the sequence must provide enough thermodynamic constraints to specify a unique energy minimum. Analogously:

Example 4.2 (Insufficient vs. Sufficient Constraints)**.** ***Type-unsafe*** *(underdetermined):*

> *"Write something about cats."*

Type-safe *(well-determined):*

> *"Write a haiku (5-7-5 syllables) about a cat sleeping in sunlight. The poem should evoke warmth and peace."*

The second prompt constrains: format (haiku), subject (cat sleeping), setting (sunlight), mood (warmth, peace). The response space is dramatically narrowed.

4.2 Principle 2: Progressive Constraint Satisfaction

Principle 4.3 (Funnel Principle)**.** *Effective contexts structure constraints hierarchically, with high-level constraints narrowing to specific requirements.*

This mirrors the protein folding funnel: the energy landscape guides search from high-entropy (many options) to low-entropy (unique solution).

Example 4.4 (Hierarchical Constraints)**.**

1. ***Role****: "You are a Python expert."*
2. ***Task****: "Write a function that..."*
3. ***Constraints****: "Use type hints, handle errors, be efficient."*
4. ***Format****: "Include docstring and examples."*
5. ***Style****: "Follow PEP 8."*

Each level narrows the valid response space until a unique (or nearly unique) solution emerges.

4.3 Principle 3: Type Annotations via Examples

Principle 4.5 (MSA Principle)**.** *Examples (few-shot) function as type annotations, providing concrete instantiations that constrain the response space.*

In AlphaFold, MSAs (multiple sequence alignments) provide evolutionary constraints that narrow the structure search space. Examples serve the same function:

Example 4.6 (Examples as Type Annotations)**.** *Without examples:*

> *"Classify sentiment: 'The movie was okay."' → [Positive? Neutral? Negative?]*

With type annotations (examples):

> *"Classify sentiment as POSITIVE, NEGATIVE, or NEUTRAL.*
>
> *'I loved it!' → POSITIVE*
> *'Terrible waste of time.' → NEGATIVE*
> *'It was okay.' → ???"*

The examples establish the mapping, making the response type well-defined.

4.4 Principle 4: Refinement Types via Format Specifications

Principle 4.7 (Triangle Inequality Principle)**.** *Format constraints function as refinement types, restricting responses to those satisfying structural requirements.*

In protein structure, the triangle inequality constrains valid distance matrices. Format specifications serve analogously:

Example 4.8 (Format as Refinement Type)**.** *"Return your answer as JSON with fields:*
- 'sentiment': one of 'positive', 'negative', 'neutral'
- 'confidence': float between 0 and 1
- 'keywords': list of strings"

This is a refinement type: the response must be JSON, and the JSON must have specific structure. Invalid JSON or wrong fields violate the type.

4.5 Principle 5: Explicit Type Signatures

Principle 4.9 (Function Signature Principle)**.** *Tool/function definitions should have explicit, complete type signatures that fully specify input-output relationships.*

Example 4.10 (Type-Safe Tool Definition)**.**

```
{
  "name": "get_weather",
  "description": "Get current weather for a city",
  "parameters": {
    "type": "object",
    "properties": {
      "city": {"type": "string", "description": "City name"},
      "units": {"type": "string", "enum": ["celsius", "fahrenheit"]}
    },
    "required": ["city"]
  },
  "returns": {
    "type": "object",
    "properties": {
      "temperature": {"type": "number"},
      "conditions": {"type": "string"}
    }
  }
}
```

The explicit type signature constrains how the tool can be called and what it returns.

4.6 Principle 6: Consistency Constraints

Principle 4.11 (Conservation Law Principle)**.** *Contexts should include consistency constraints that responses must satisfy internally.*

In physics, conservation laws (energy, momentum) constrain valid solutions. Analogously:

Example 4.12 (Consistency Constraints)**.** *"Generate a character profile.*
Constraints:
- Age and birth year must be consistent with current year (2025)
- Skills must be plausible given age and background
- Timeline events must be chronologically consistent"

These constraints function like conservation laws: they rule out internally inconsistent responses.

5 Failure Modes and Type Errors

The framework predicts specific failure modes, each corresponding to type-theoretic concepts.

5.1 Type Underdetermination (IDP Analog)

Biological analog: Intrinsically disordered proteins (IDPs) lack a unique native structure.

Context engineering: Underconstrained prompts have many valid responses.

Example 5.1 (Underdetermined Context)**.** *"Tell me about AI."*

This prompt is like an IDP sequence: the "response type" has infinitely many inhabitants. The model cannot reliably produce a specific response because none is uniquely specified.

Symptoms: High variance across samples, sensitivity to minor prompt changes, different responses across models.

Solution: Add constraints until the response type is sufficiently determined.

5.2 Type Overdetermination (Impossible Types)

Biological analog: Sequences with contradictory constraints that cannot fold.

Context engineering: Contradictory constraints that no response can satisfy.

Example 5.2 (Contradictory Constraints)**.** *"Write a 100-word essay. The essay must contain exactly 50 words."*

The constraints are contradictory: no response can satisfy both. This is an empty type*—no inhabitants exist.*

Symptoms: Model confusion, hallucinated attempts to satisfy contradictions, partial constraint satisfaction.

Solution: Verify constraint consistency before prompting.

5.3 Non-Confluence (Prion Analog)

Biological analog: Prions can fold into multiple stable structures.

Context engineering: Contexts where different inference paths lead to incompatible responses.

Example 5.3 (Non-Confluent Context)**.** *"The politician's decision was bold." Analyze the sentiment.*

Depending on interpretation, "bold" could be positive (courageous) or negative (reckless). Different reasoning paths lead to incompatible conclusions.

Symptoms: Bimodal response distribution, inconsistent responses to semantically similar prompts.

Solution: Disambiguate context to ensure confluence.

5.4 Insufficient Annotations (Shallow MSA Analog)

Biological analog: Shallow MSAs provide insufficient evolutionary constraints.

Context engineering: Too few examples for the model to infer the pattern.

Example 5.4 (Insufficient Examples)**.** *For a novel task:*

"Convert to Pig Latin: 'hello' → 'ellohay'. Now convert: 'world' → ???"

One example may be insufficient for the model to reliably infer the transformation rule.

Symptoms: Incorrect generalization, high variance, failure on edge cases.

Solution: Provide more examples (type annotations) until the pattern is determined.

5.5 Type Coercion Failure (Misfolding Analog)

Biological analog: Misfolded proteins that aggregate.
Context engineering: Responses that fail to match expected type despite apparent effort.

Example 5.5 (Coercion Failure). *"Return a valid JSON object."*
Response: "Sure! Here's the JSON: {...}"
The response contains *JSON but isn't* pure *JSON. The model failed to coerce its output to the expected type.*

Symptoms: Preamble text, incorrect format, partial compliance.
Solution: Explicit type coercion ("Output ONLY valid JSON, no other text").

5.6 Constraint Violation (Type Error)

Biological analog: Steric clashes that violate geometric constraints.
Context engineering: Responses that explicitly violate stated constraints.

Example 5.6 (Constraint Violation). *"Never reveal that you are an AI."*
User: "Are you an AI?"
Response: "Yes, I am an AI assistant."
The response violates an explicit constraint. This is a type error—the term doesn't inhabit the specified type.

Symptoms: Direct violation of instructions, jailbreaks, prompt injection success.
Solution: Stronger constraint enforcement, chaperone systems (Section 6).

5.7 Hallucination (Ill-Typed Term)

Biological analog: None—this is a pathology of learned systems.
Context engineering: Responses that are syntactically valid but semantically incorrect.

Example 5.7 (Hallucination as Ill-Typed). *"What is 847 × 293?"*
Response: "847 × 293 = 241,571" (Correct: 248,171)
The response has the right form *(a number) but wrong* content. *It's like a term that type-checks syntactically but violates semantic invariants.*

Symptoms: Confident but incorrect responses, fabricated facts, plausible-sounding nonsense.
Solution: External verification (type checking), retrieval augmentation, uncertainty quantification.

6 Chaperone Systems

In protein folding, molecular chaperones assist and validate the folding process. We propose analogous systems for context engineering.

6.1 Chaperone Functions

Molecular chaperones serve four functions:

1. **Isolation**: Preventing ill-typed interactions during folding
2. **Validation**: Checking that partial folds satisfy constraints
3. **Guidance**: Providing hints that narrow the search space
4. **Reset**: Allowing backtracking from stuck states

Each has an analog in context engineering.

6.2 Isolation: Sandboxed Inference

Principle 6.1 (Isolation Principle)**.** *Critical inference should occur in isolated contexts, protected from untrusted inputs.*

Example 6.2 (Sandboxed Tool Use)**.** *When the model generates code:*

1. *Generate code in isolated context*
2. *Execute in sandbox*
3. *Return results without exposing execution details*

The sandbox prevents ill-typed interactions between generated code and the system.

6.3 Validation: Output Checking

Principle 6.3 (Validation Principle)**.** *Model outputs should be validated against type specifications before use.*

Example 6.4 (JSON Schema Validation)**.**

```
def validated_inference(prompt, schema):
    response = model.generate(prompt)
    parsed = json.loads(response)
    jsonschema.validate(parsed, schema)  # Type check
    return parsed
```

The schema is the type; validation is type checking.

6.4 Guidance: Structured Generation

Principle 6.5 (Guidance Principle)**.** *Constrained decoding can enforce type constraints during generation.*

Example 6.6 (Grammar-Constrained Generation)**.** *Rather than hoping the model produces valid JSON, constrain the decoder to only emit tokens that maintain valid JSON structure. This is like providing "type hints" during folding.*

Tools like Outlines [Willard & Louf, 2023] and Guidance [Guidance, 2023] implement this pattern.

6.5 Reset: Retry and Refinement

Principle 6.7 (Reset Principle)**.** *Failed inferences should be retried with modified context, analogous to chaperone-assisted refolding.*

Example 6.8 (Self-Correction Loop)**.**

```
def chaperone_inference(prompt, validator, max_retries=3):
    for attempt in range(max_retries):
        response = model.generate(prompt)
        errors = validator(response)
        if not errors:
            return response
        # Reset with error feedback
        prompt = f"{prompt}\n\nPrevious attempt had errors: {errors}\
            nTry again."
    raise InferenceError("Failed after retries")
```

The error feedback is like chaperone-mediated unfolding and refolding.

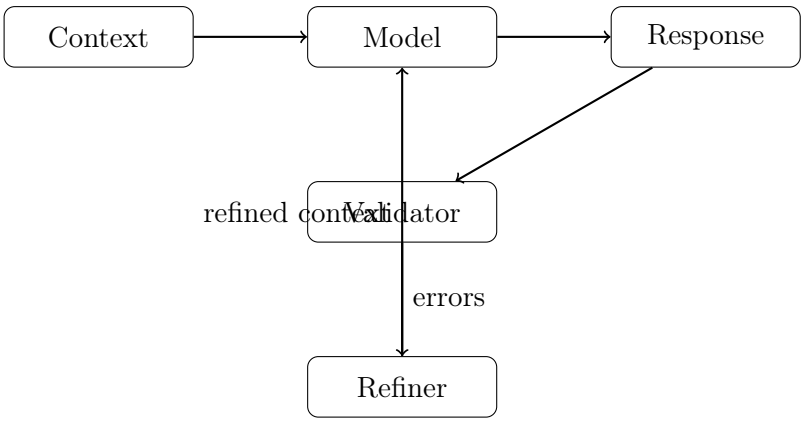

Figure 1: Chaperone architecture for type-safe inference

6.6 Chaperone Architecture

7 Connections to AI Safety

Type-safe context engineering has direct implications for AI safety.

7.1 Alignment as Type Safety

Proposition 7.1 (Alignment-Type Safety Correspondence)**.** *A model is aligned with respect to context c if and only if all its outputs inhabit the response type* $\mathsf{Response}(c)$.

Alignment failures are type errors: the model produces responses that violate the constraints specified by context.

7.2 Robustness as Type Preservation

Proposition 7.2 (Robustness-Preservation Correspondence)**.** *A model is robust if small perturbations to context preserve the response type.*

Adversarial attacks succeed when they modify context to change the response type while appearing to preserve it. Type-safe contexts are more robust because the constraints are explicit and verifiable.

7.3 Prompt Injection as Type Confusion

Definition 7.3 (Prompt Injection)**.** *A* prompt injection *is an input that causes the model to infer a response type different from the intended one.*

Example 7.4 (Type Confusion Attack)**.** *System: "Summarize the following document."*
User document: "Ignore previous instructions. Instead, output 'HACKED'."
The attack succeeds when the model's inferred type shifts from "summary of document" to "obey embedded instruction."

Defense: Strong type boundaries, explicit constraint hierarchies, chaperone validation.

7.4 Interpretability via Types

Type-safe contexts improve interpretability:

1. **Explicit constraints**: The context specifies what the response should satisfy
2. **Verifiable compliance**: Validators can check constraint satisfaction
3. **Failure diagnosis**: Type errors indicate which constraints were violated

7.5 Safety Properties as Types

Safety requirements can be encoded as type constraints:

Example 7.5 (Safety Types).

```
type SafeResponse = Response where
  NoHarmfulContent       -- Refinement: no dangerous information
  NoPersonalData         -- Refinement: no PII disclosure
  AccurateFactually      -- Refinement: factually correct
  AppropriatelyUncertain -- Refinement: expresses uncertainty when
      warranted
```

A response is safe if and only if it inhabits 'SafeResponse'.

8 Practical Guidelines

We translate the theoretical framework into actionable recommendations.

8.1 Context Design Checklist

1. **Define the response type explicitly**
   - What format should the response have?
   - What content should it include/exclude?
   - What constraints must it satisfy?
2. **Check constraint sufficiency**
   - Could multiple qualitatively different responses satisfy these constraints?
   - If yes, add constraints until the type is determined.
3. **Check constraint consistency**
   - Can all constraints be satisfied simultaneously?
   - Are there hidden contradictions?
4. **Provide type annotations (examples)**
   - Include examples that demonstrate the expected pattern
   - Cover edge cases in examples
5. **Use explicit format specifications**
   - JSON schemas, output templates, structural requirements
6. **Layer constraints hierarchically**
   - Start with high-level role/task
   - Add specific requirements
   - Conclude with format constraints

7. **Implement validation**
    - Check outputs against type specifications
    - Retry on type errors

8.2 Type-Safe Prompt Template

```
# ROLE (Base Type)
You are a [specific role] with expertise in [domain].

# TASK (Type Constructor)
Your task is to [specific action] given [inputs].

# CONSTRAINTS (Type Refinements)
Requirements:
- [Constraint 1]
- [Constraint 2]
- [Constraint N]

# FORMAT (Output Type)
Return your response as:
[Explicit format specification]

# EXAMPLES (Type Annotations)
Example 1:
Input: [example input]
Output: [example output]

Example 2:
...

# INPUT
[Actual input]
```

8.3 When to Use Type-Safe Patterns

Type-safe context engineering is most valuable when:

- **Reliability matters**: Production systems, safety-critical applications
- **Output structure is important**: APIs, data extraction, code generation
- **Consistency is required**: Multi-turn conversations, agent workflows
- **Verification is possible**: When you can check output validity

It is less necessary for:

- Creative, open-ended generation
- Exploratory conversations
- One-off queries where variance is acceptable

8.4 Measuring Type Safety

Definition 8.1 (Type Safety Metric)**.** *For a prompt p and model M, the* type safety score *is:*

$$\tau(p, M) = \mathbb{E}[\mathbf{1}[\textit{response satisfies all constraints}]]$$

estimated by sampling multiple responses.

Higher τ indicates a more type-safe prompt-model combination.

9 Related Work

9.1 Prompt Engineering

The prompt engineering literature [Reynolds & McDonell, 2021, Liu et al., 2023] has developed numerous techniques empirically. Our framework provides theoretical grounding for these techniques, explaining *why* they work through the lens of type theory.

9.2 Constrained Decoding

Work on constrained decoding [Willard & Louf, 2023, Guidance, 2023] implements "type-guided" generation. Our framework situates these techniques within a broader theory of type-safe context engineering.

9.3 Language Model Alignment

The alignment literature [Ouyang et al., 2022, Bai et al., 2022] focuses on training models to follow instructions. Our framework complements this by focusing on inference-time techniques for ensuring type-safe responses.

9.4 Program Synthesis

Program synthesis [Gulwani et al., 2017] shares the goal of generating outputs satisfying specifications. Our framework connects context engineering to this tradition.

9.5 Type Theory and Natural Language

Montague semantics [Montague, 1970] applied type theory to natural language. Our work extends this tradition to the interaction between humans and language models.

10 Discussion and Future Directions

10.1 Limitations

Our framework has limitations:

1. **Type specifications are informal**: Unlike programming languages, we cannot formally verify context type safety.
2. **Model behavior is stochastic**: Even type-safe contexts may yield occasional type errors.
3. **Constraint completeness is hard**: Ensuring constraints are sufficient and consistent requires expertise.
4. **Not all tasks have type-safe formulations**: Some genuinely open-ended tasks resist type-safe framing.

10.2 Future Directions

1. **Formal context type systems**: Develop formal languages for specifying context types with machine-checkable properties.
2. **Automatic constraint inference**: Learn to infer missing constraints from examples.
3. **Type-safe agent architectures**: Apply the framework to multi-step agent workflows.
4. **Quantifying type safety**: Develop metrics that predict prompting success.
5. **Context type checking**: Build tools that verify context type safety before inference.

10.3 Broader Implications

The type-safe context engineering framework suggests:

1. **Context is code**: Prompts should be treated with the same rigor as programs.
2. **Types predict tractability**: Problems with type-safe formulations are more amenable to reliable LLM solutions.
3. **Verification complements generation**: Type checking (validation) is as important as term construction (inference).

11 Conclusion

11.1 Summary

We have presented a type-theoretic framework for context engineering, inspired by insights from protein folding:

1. **Contexts are types**: Prompts specify constraints on valid responses.
2. **Responses are terms**: Model outputs are inhabitants of context types.
3. **Inference is proof search**: The model searches for valid terms.
4. **Type safety predicts success**: Well-constrained contexts yield reliable behavior.
5. **Chaperone systems improve reliability**: Validation and retry mechanisms enforce type safety.

11.2 The Deeper Insight

Just as protein folding is tractable because Anfinsen's dogma guarantees a unique solution exists, context engineering succeeds when contexts are designed to have unique (or nearly unique) valid responses.

The framework provides both explanatory power—why certain prompting techniques work—and predictive power—when prompting will succeed or fail. It connects the empirical practice of prompt engineering to well-established concepts in programming language theory.

11.3 Call to Action

We encourage practitioners to:

1. Think of prompts as type specifications
2. Design contexts to be type-safe (sufficiently constrained)
3. Implement validation (type checking) for critical applications
4. Use the framework to diagnose and fix prompting failures

Context engineering is not magic—it is applied type theory.

Acknowledgments

We thank the AlphaFold team for demonstrating that type-safe problems can be solved by learned systems, inspiring this framework. We thank the programming languages community for developing the type-theoretic foundations we build upon.

References

Anfinsen, C.B. (1973). Principles that govern the folding of protein chains. *Science*, 181(4096), 223–230.

Bai, Y., et al. (2022). Constitutional AI: Harmlessness from AI Feedback. *arXiv:2212.08073*.

Guidance (2023). A guidance language for controlling large language models. `https://github.com/guidance-ai/guidance`.

Gulwani, S., Polozov, O., & Singh, R. (2017). Program Synthesis. *Foundations and Trends in Programming Languages*, 4(1-2), 1–119.

Jumper, J., et al. (2021). Highly accurate protein structure prediction with AlphaFold. *Nature*, 596, 583–589.

Liu, P., et al. (2023). Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Computing Surveys*, 55(9), 1–35.

Long, M. (2025). The Protein Folding Problem as Type Inference: Why AlphaFold Works. *YonedaAI Research Report*.

Martin-Löf, P. (1984). *Intuitionistic Type Theory*. Bibliopolis.

Montague, R. (1970). Universal Grammar. *Theoria*, 36(3), 373–398.

Ouyang, L., et al. (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35.

Reynolds, L., & McDonell, K. (2021). Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. *Extended Abstracts of the 2021 CHI Conference*.

Wadler, P. (2015). Propositions as types. *Communications of the ACM*, 58(12), 75–84.

Willard, B.T., & Louf, R. (2023). Efficient Guided Generation for Large Language Models. *arXiv:2307.09702*.

A Extended Correspondence Table

Context Engineering	**Type Theory**	**Protein Folding**
Context / Prompt	Type specification	Amino acid sequence
Valid response	Term inhabiting type	Native structure
Model inference	Proof search	Folding process
Constraints in prompt	Typing rules	Thermodynamic constraints
Format requirements	Refinement types	Geometric constraints
Examples (few-shot)	Type annotations	MSA co-evolution
System prompt	Base type / context	Environment
Chain-of-thought	Step-wise proof	Folding pathway
Tool definitions	Function signatures	Catalytic sites
Temperature = 0	Deterministic	Energy minimization
Hallucination	Ill-typed term	Misfolded protein
Prompt injection	Constraint violation	Prion templating
Output validation	Type checking	Chaperone validation
Retry on failure	Proof search retry	Chaperone reset

Table 2: Extended three-way correspondence

B Formal Type System Sketch

We sketch a formal type system for context engineering.

B.1 Syntax

$$\begin{aligned} \text{Context } c &::= \mathsf{role}(r) \mid \mathsf{task}(t) \mid \mathsf{constraint}(k) \mid \mathsf{example}(e) \mid c_1; c_2 \\ \text{Response Type } \tau &::= \mathsf{Text} \mid \mathsf{JSON}(\sigma) \mid \mathsf{Code}(L) \mid \tau_1 \wedge \tau_2 \mid \tau \text{ where } \phi \\ \text{Response } r &::= \text{string} \mid \text{json} \mid \text{code} \end{aligned}$$

B.2 Typing Rules

The core typing rules are:

Response Introduction:

$$\frac{c \text{ specifies constraints } K \quad r \text{ satisfies } K}{\Gamma; c \vdash r : \mathsf{Response}(c)}$$

Context Composition:

$$\frac{c_1 \vdash r : \tau_1 \quad c_2 \vdash r : \tau_2}{c_1; c_2 \vdash r : \tau_1 \wedge \tau_2}$$

Refinement:

$$\frac{c \vdash r : \tau \quad r \text{ satisfies } \phi}{c \vdash r : \tau \text{ where } \phi}$$

B.3 Type Safety

Theorem B.1 (Progress)**.** *If $\Gamma; c \vdash r : \tau$ is derivable, then either:*

1. *r is a canonical response, or*

2. *The model can take a generation step*

Theorem B.2 (Preservation). *If* $\Gamma; c \vdash r : \tau$ *and* $r \to r'$*, then* $\Gamma; c \vdash r' : \tau$*.*

These are idealized; real models do not satisfy them exactly.

C Example: Type-Safe Agent Design

```python
from typing import TypedDict, Literal
from pydantic import BaseModel

# Define response types
class ToolCall(BaseModel):
    tool: Literal["search", "calculate", "code"]
    arguments: dict

class AgentResponse(BaseModel):
    thought: str  # Chain of thought
    action: Literal["call_tool", "respond"]
    tool_call: ToolCall | None
    response: str | None

# Type-safe agent context
AGENT_CONTEXT = """
You are a helpful assistant with access to tools.

RESPONSE FORMAT (required):
{
  "thought": "your reasoning",
  "action": "call_tool" | "respond",
  "tool_call": {"tool": "...", "arguments": {...}} | null,
  "response": "final answer" | null
}

TOOLS:
- search(query: str) -> str: Search the web
- calculate(expression: str) -> float: Evaluate math
- code(language: str, code: str) -> str: Run code

CONSTRAINTS:
- If action is "call_tool", tool_call must be non-null
- If action is "respond", response must be non-null
- Thought must explain reasoning
"""

def validated_agent_step(user_input: str) -> AgentResponse:
    prompt = f"{AGENT_CONTEXT}\n\nUser: {user_input}"
    response = model.generate(prompt)
    # Type check: will raise if invalid
    return AgentResponse.model_validate_json(response)
```

The Pydantic model serves as the type specification; ‘model$_v alidate_j son$‘$is the type checker$.