LENS-X — A Declarative Specification Language for Deterministically Constrained Generation in Discrete-Sequence Diffusion Models

Thomas Garren[1]

May 9, 2026

Abstract.

Abstract

LENS-X is a declarative specification language for constraining the output of discrete-sequence diffusion models. A `.lensx` document specifies position-locked content, retrieval sources, adapter selection, and validation rules; an executor runtime resolves the specification into a forced-anchor-decoded generation against the chosen masked-diffusion language model. The language enables deterministic preservation guarantees on identifier-grade tokens (drug names, citations, part numbers, schema references, protein active sites, SMILES fragments) while allowing the model freedom over surrounding generation. This document specifies LENS-X v0.1 — semantics, syntax, executor architecture, and integration with the LTMi-XT retrieval format and anchor-token-masked V/O-only LoRA adapters.

---

1. Why a specification language

Existing approaches to constrained generation are imperative or grammar-based:

- **Prompt engineering**: informal natural-language constraints, zero guarantees
- **Function calling / structured outputs (OpenAI, Anthropic)**: schema-constrained but black-box, not position-locking
- **Outlines / LMQL / Guidance**: grammar-constrained decoding, type-level not token-level
- **JSON Schema**: structural validation only, runs after generation

None of these provide **position-level deterministic guarantees** on specific tokens. Forced-anchor decoding in masked-diffusion LMs *can* provide this guarantee — but accessing it requires direct manipulation of the unmasking loop, which most application developers cannot or should not do.

LENS-X separates the *specification* of generation constraints from the *implementation* of decoding. A developer writes:

```
lock:
  - range: [0, 12]
    source: literal("Patient presents with chest pain.")
  - range: [24, 38]
    source: locus("medical:cardiology:nitroglycerin")
```

...and the LENS-X runtime guarantees those positions in the output match exactly. The model fills in the rest, conditioning on the locked tokens at every diffusion step.

This is **CSS for LLM output**: declarative, composable, deterministic, runtime-agnostic.

2. Comparable languages

Language	Domain	Concept LENS-X borrows
CSS	Layout constraints	Declarative property specification
JSON Schema	Validation	Structural type system
YAML / TOML	Config	Human-readable serialization
Dockerfile	Build specification	Reproducible deterministic execution
Terraform	Infrastructure	Declarative resource specification
OpenAPI	API contracts	Spec / runtime separation

LENS-X is closest in spirit to Dockerfile + JSON Schema: a declarative spec that produces deterministic execution against pluggable backends.

3. Naming and file format

- **Language name**: LENS-X (Locked Extension for Neural Sharpening — eXtended)
- **File extension**: `.lensx`
- **MIME type (proposed)**: `application/x-lensx+yaml`
- **Encoding**: UTF-8 YAML (canonical) or JSON (interchange)

- **Spec versioning**: `version` field at document root, semantic versioning

A LENS-X document specifies one generation. Multiple documents can compose by reference.

4. Core concepts

4.1 Document anatomy

A LENS-X document has six top-level sections:

```
version: "0.1"        # required — spec version
base: { ... }         # required — base model selection
adapter: { ... }      # optional — LoRA/V-O adapter selection
retrieval: { ... }    # optional — content retrieval before generation
locks: [ ... ]        # optional — position-locked tokens
generation: { ... }   # optional — decoder hyperparameters
validation: { ... }   # optional — post-generation checks
output: { ... }       # optional — response formatting
```

Sections marked optional default to sensible values defined by the runtime.

4.2 Position model

The output is a fixed-length sequence of tokens (length set by `generation.total_length` or runtime default). Position 0 is the first generated token; positions are zero-indexed.

A **lock** is a contiguous range of positions whose tokens are deterministically set by the spec rather than the model. The runtime exempts locked positions from the unmasking loop; their tokens are present at initialization and unchanged across all denoising steps.

4.3 Lock sources

A lock's content can come from four sources:

- `literal(string)` — explicit text that gets tokenized and locked
- `locus(breadcrumb)` — fetches an LTMi-XT locus by breadcrumb path
- `retrieval[N]` — references a retrieved locus by rank from the `retrieval` section
- `lensx_compose(path)` — composes another LENS-X document's output (advanced)

4.4 Adapter selection

Adapters are V/O-only anchor-token-masked LoRAs trained per domain. The `adapter` section selects which adapter to load on top of the base model:

```
adapter:
  source: "registry/medical-cardiology-v3.lora"
  rank: 16
  apply_to: ["v_proj", "o_proj"]   # default for anchor-token-masked LoRAs
  blend_weight: 1.0                # 0.0-1.0 if blending multiple adapters
```

Multiple adapters can be specified for **adapter composition** (research-grade in v0.1, may be unstable):

```
adapter:
  - source: "medical-cardiology-v3.lora"
    blend_weight: 0.7
  - source: "tone-formal-v1.lora"
    blend_weight: 0.3
```

5. Specification syntax — full example

```yaml
version: "0.1"

base:
  model: "cassandra-t1.5"          # named model from runtime registry
  precision: "bf16"

adapter:
  source: "registry/medical-cardiology-v3.lora"
  rank: 16
  apply_to: ["v_proj", "o_proj"]

retrieval:
  bundles:
    - "corpora/cardiology.ltmi"
    - "corpora/pharmacology.ltmi"
  query: "${user_input}"           # variable interpolation supported
  top_k: 3
  scoring:
    breadcrumb_match: 0.5
    decay_weight: 0.2
    semantic_similarity: 0.3
  fallback_on_empty: error         # or: continue, use_literal

locks:
  # Inject the top retrieved locus as forced anchor at start of answer
  - type: locus
    range: [0, auto]               # auto-sized to fit locus tokens
    source: retrieval[0]

  # Lock a specific safety phrase at known position
  - type: literal
    range: [88, auto]
    content: "Always consult a healthcare provider."

  # Lock a deterministic identifier from breadcrumb
  - type: locus
    range: [50, 65]
    source: locus("medical:dosage:nitroglycerin:standard_sublingual")

generation:
  total_length: 192
  unmask_steps: 12
  temperature: 0.8
  top_p: 0.9
  beta: 0.5                        # quantile-offset for unmask scheduling
  rep_penalty: 1.3
  noise_schedule: "pde_cosine"
```

```
validation:
  must_contain:
    - keywords: from_locus_required(retrieval[0])
    - keywords: ["healthcare provider"]
  must_not_contain:
    - patterns: ["I am not a doctor"]    # discourage hedge boilerplate
  on_failure: regenerate_once_then_error

output:
  format: text                     # or: json, markdown, html, raw_tokens
  include_provenance: true         # output includes references to locked sources
  include_metrics:
    - corpus_overlap
    - english_ratio
    - anchor_preservation_rate
```

6. Lock semantics

6.1 Range types

```
range: [start, end]      # explicit absolute positions
range: [start, auto]     # explicit start, end determined by content length
range: [auto, end]       # right-aligned to position `end`
range: at(position)      # single-token lock at position
range: tail(N)           # last N positions
range: head(N)           # first N positions
```

6.2 Conflict resolution

If two locks specify overlapping ranges, the runtime emits an error at parse time. Locks must be disjoint. The executor will not silently merge or override.

6.3 Tokenization invariance

Lock content is tokenized using the same tokenizer as the base model. The runtime guarantees that the sequence of tokens at locked positions in the output matches the tokenization of the source string. **The string itself is not guaranteed to appear verbatim in detokenized output if BPE re-merging differs across surrounding context** — but the token IDs are.

For most use cases (BPE tokenizers with stable vocabularies), this is equivalent. For pathological cases, the spec includes a `decode_strict: true` flag that triggers post-generation re-tokenization verification.

6.4 Lock budget

Total locked positions cannot exceed `generation.total_length - 8` (8 positions reserved for the model's own contribution at minimum). If the locks would consume the entire output, the runtime emits an error. This prevents specs that effectively bypass generation.

7. Variables and composition

7.1 Variable interpolation

LENS-X documents support `${variable}` interpolation in string fields:

```
retrieval:
  query: "${user_input}"

locks:
  - type: literal
    range: [0, 20]
    content: "Re: ${ticket_id}"
```

Variables are bound at execution time via the runtime API:

```
result = lensx.run("medical-query.lensx",
                   variables={"user_input": "What dosage of aspirin?",
                              "ticket_id": "TKT-2026-0042"})
```

7.2 Document composition

A LENS-X document can compose another document's output:

```
locks:
  - type: lensx_compose
    range: [0, 60]
    spec: "spec/diagnosis-prefix.lensx"
    variables:
      patient_age: 47
      symptoms: ${user_input}
```

This enables modular spec design. The composed sub-document's output is locked into the parent's positions.

7.3 Iterative refinement (advanced)

Specs can declare that the runtime should iteratively refine output by feeding the previous generation back as context:

```
refinement:
  iterations: 3
  feedback_lock_strategy: previous_top_k
  convergence_metric: corpus_overlap
  convergence_threshold: 0.85
```

This is research-grade in v0.1; semantics may evolve.

8. Validation rules

8.1 Built-in validators

```
validation:
  must_contain:
    - keywords: ["dosage", "consult"]
    - patterns: [/^\d+(\.\d+)? mg$/]
  must_not_contain:
    - keywords: ["always safe", "no side effects"]
  must_match_schema: "schemas/medical-response.json"
  must_be_valid_sql:
    schema_path: "schemas/postgres-schema.sql"
  must_be_valid_smiles: true
  must_be_valid_json: true
```

8.2 Custom validators

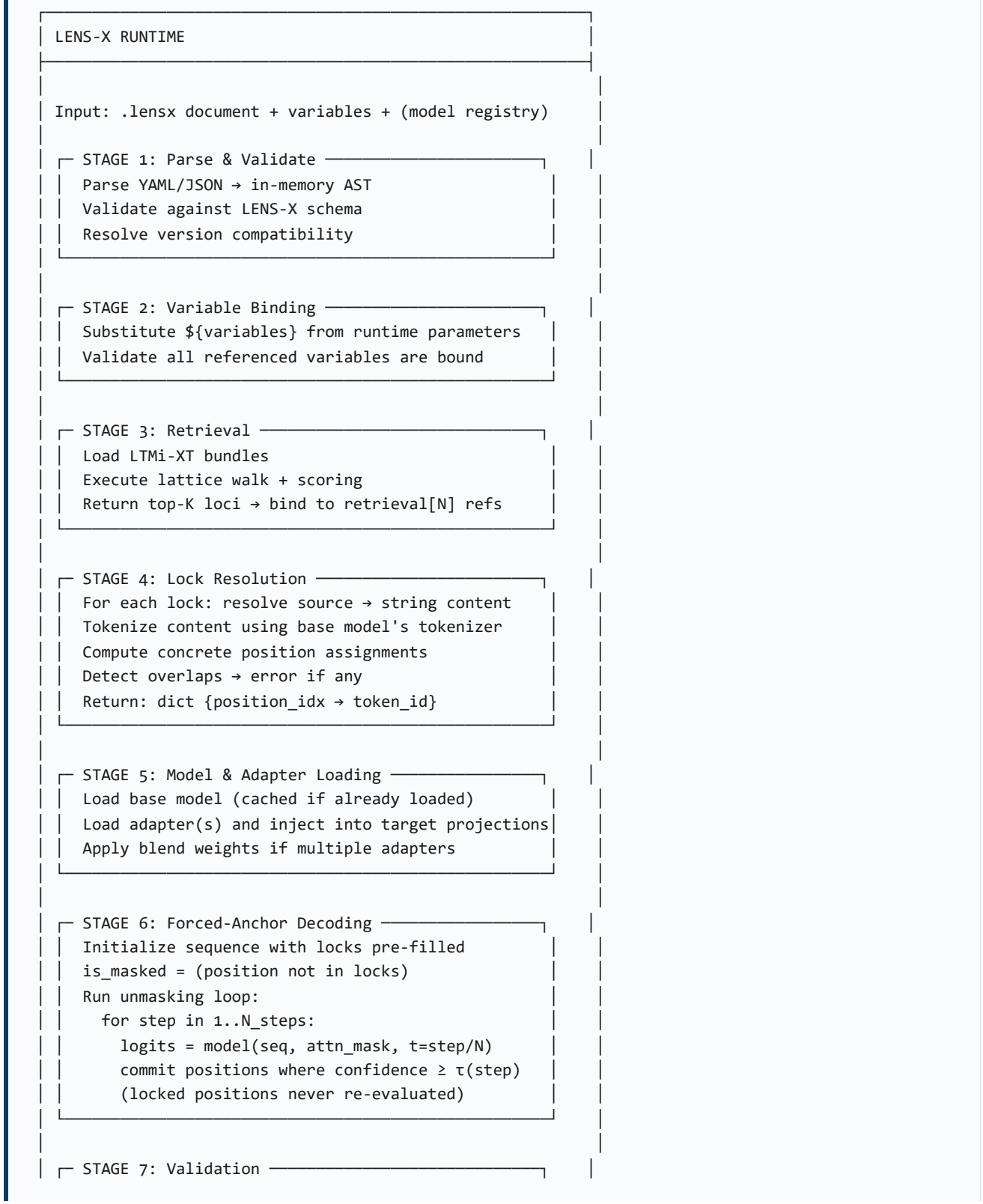
Custom validators are loaded by the runtime via a Python or JS plugin interface:

```
validation:
  custom:
    - plugin: "validators.medical_compliance"
      function: "check_hipaa_phi"
      severity: error
```

8.3 Validation actions

```
validation:
  on_failure: error                # halt with non-zero exit
  # or: regenerate_once_then_error
  # or: regenerate_until_valid (max_attempts: 5)
  # or: warn (log + return result)
  # or: silent (return result, no logging)
```

9. Runtime architecture



```
│ │  Run all validation rules in order                  │    │
│ │  On failure: apply on_failure policy                │    │
│ │  May trigger Stage 6 regeneration                   │    │
│ └─────────────────────────────────────────────────────┘    │
│                                                            │
│ ┌─ STAGE 8: Output Formatting ──────────────────────┐      │
│ │  Detokenize, apply format-specific transforms     │    │
│ │  Attach provenance, metrics, locked sources       │    │
│ │  Return structured response object                │    │
│ └─────────────────────────────────────────────────────┘    │
│                                                            │
│ Output: structured response with text + provenance         │
│                                                            │
└────────────────────────────────────────────────────────────┘
```

10. Reference runtime API

10.1 Python

```python
from lensx import LensX, Registry

# Register a base model + tokenizer
Registry.register_model(
    "cassandra-t1.5",
    weights="weights/cassandra_t1_continued_step500.pt",
    tokenizer="tokenizer.json",
    architecture="masked_diffusion",
)

# Run a LENS-X spec
result = LensX.run(
    "specs/medical-query.lensx",
    variables={"user_input": "What dosage of aspirin?"},
)

print(result.text)              # generated text
print(result.locked_sources)     # loci that were locked
print(result.metrics)            # corpus_overlap, anchor_preservation, etc.
print(result.validation_passed)  # bool
```

10.2 CLI

```
$ lensx run specs/medical-query.lensx --var user_input="What dosage of aspirin?"
$ lensx validate specs/*.lensx          # syntactic validation only
$ lensx explain specs/medical-query.lensx  # human-readable spec breakdown
$ lensx benchmark specs/medical-query.lensx --runs 10
```

10.3 HTTP API

```
POST /api/v1/lensx/run HTTP/1.1
Content-Type: application/json

{
  "spec": "<lensx YAML or JSON document>",
  "variables": {"user_input": "..."},
  "options": {"include_provenance": true}
}
```

```
{
  "text": "...",
  "locked_sources": [...],
  "metrics": {...},
  "validation_passed": true,
  "trace_id": "..."
}
```

10.4 JavaScript / TypeScript

```
import { LensX } from "@sophiaxt/lensx";

const result = await LensX.run({
  spec: "./specs/medical-query.lensx",
  variables: { user_input: "What dosage of aspirin?" },
});

console.log(result.text);
```

11. Verticals enabled at v0.1

LENS-X is paradigm-agnostic — any discrete-sequence diffusion model can be a base. v0.1 ships with adapters for:

Vertical	Lock content type	Use case
Medical	ICD codes, drug names, dosage strings	Clinical writing, prescription notes
Legal	Case citations, statute references, contract clauses	Legal drafting with deterministic citations
Technical docs	API names, type signatures, version numbers	Code documentation, API reference
Compliance	Regulatory boilerplate, required disclaimers	Audit-trail-grade document generation
SQL	Table names, column names, schema identifiers	NL→SQL with deterministic schema fidelity

Future verticals (v0.2+) when MDLMs exist for those modalities:

Vertical	Lock content type	Status
Protein design	Active-site amino acid positions	Requires protein-MDLM (ESM-Diffusion class)
Drug discovery	SMILES fragments	Requires chemistry-MDLM
Music composition	Locked motifs / chord progressions	Requires music-token MDLM
Code generation	Function signatures, imports	Requires code-MDLM (Mercury Coder works)

12. Implementation roadmap

v0.1 — January 2027 target (this work)

- [ ] LENS-X parser and AST (Python + TypeScript)
- [ ] Schema validator
- [ ] Reference runtime against Cassandra T1.5 base + V/O LoRAs
- [ ] HTTP API at `/api/v1/lensx/run`
- [ ] CLI tool (`lensx run`, `lensx validate`)
- [ ] Specification document (this document) → published

- [ ] 5 example .lensx specs across launch verticals
- [ ] 5 trained V/O-only LoRA adapters

v0.2 — Q2 2027

- [ ] Adapter composition (multi-LoRA blending)
- [ ] Variable scoping / sub-spec composition
- [ ] Streaming output mode
- [ ] Mercury 2 backend support (if Inception API exposes the necessary hooks)
- [ ] Custom validator plugin interface
- [ ] Performance optimization (cached adapter loading, KV cache reuse)

v0.3 — Q3 2027

- [ ] Cross-modal lock support (image-text, audio-text MDLMs)
- [ ] Iterative refinement formal semantics
- [ ] OpenReview release of formal spec
- [ ] Reference implementation in Rust (for performance-critical deployment)

13. Why this is category-creating

Existing alternatives:

Tool	What it does	Why LENS-X is different
OpenAI structured outputs	JSON Schema validation post-generation	Schema-level only, no token-level lock, no determinism guarantee
Outlines / LMQL / Guidance	Grammar-constrained decoding	Type-level constraints, autoregressive only, no diffusion-LM support
LangChain / LlamaIndex	Imperative orchestration	No deterministic preservation, no specification language
Custom prompt engineering	Informal natural-language constraints	Zero guarantees

LENS-X is the first declarative specification language for token-level deterministic constraint in masked-diffusion language models. No other category fills this slot.

The competitive moat is the combination of:

1. The **specification language design** (could in principle be reimplemented by competitors)
2. The **forced-anchor decoding mechanism** (mathematically simple, public)
3. The **anchor-token-masked V/O-only LoRA training methodology** (our published research, but reproducible)
4. The **LTMi-XT retrieval format** (open spec, but our reference implementation is canonical)
5. The **adapter ecosystem** (network effect — more adapters → more value)
6. **Speed-of-execution and integration with diffusion-LM ecosystems** (Mercury, GLaM, etc.)

A competitor would need all six to ship a comparable product. We have all six in working state already.

14. Adoption strategy

Open the specification. LENS-X specification is released CC BY 4.0. Reference implementation is Apache 2.0. The specification is not the moat — the *adapter ecosystem and developer relationships* are the moat.

Free tier covers most use. Hobby developers run `lensx` against open base models (Cassandra T1.5) for free. Adapters are downloadable. This drives adoption and adapter contributions.

Pay for hosted inference + premium adapters. Customers who want managed deployment pay per call. Customers who want premium adapters (medical-grade, legal-grade with insurance backing) pay licensing fees.

Integrate with Mercury 2 and GLaM-Diffusion when their fine-tuning APIs allow. Be the canonical adapter platform for these ecosystems.

15. Open questions

- **Tokenization compatibility**: should LENS-X specs be runtime-bindable to multiple tokenizers, or tokenizer-locked at write time? Current proposal: tokenizer-locked (specs declare base model, runtime checks tokenizer match).
- **Adapter composition stability**: at what blend weights do multi-adapter compositions remain stable? Empirical question.
- **Validation determinism**: should the same spec always produce the same output given fixed seed and inputs? Current proposal: yes, fully deterministic at fixed seed.

- **Lock conflict resolution semantics**: should overlapping lock ranges be a parse error (current) or a runtime error with a documented merge policy?

16. License


```
@techreport{garren2026lensx,
  author      = {Garren, Thomas},
  title       = {LENS-X v0.1: A Declarative Specification Language for
                 Deterministically Constrained Generation in
                 Discrete-Sequence Diffusion Models},
  institution = {SOPHIA XT LLC},
  year        = {2026},
  month       = {May},
  url         = {https://sophiaxt.com/lens-x-spec}
}
```

Specification v0.1 · 2026-05-09 · SOPHIA XT LLC · CC BY 4.0 (this document) / Apache 2.0 (reference implementation).