Narrative Compression Engine

Applying "Semantic Compression" to Novel-to-Video generation. Treating narrative not as text, but as compile-able state data.

The Semantic Compression Pattern

Current LLMs suffer from "Context Rot" in long-form generation. To solve this, we are adopting an architectural pattern inspired by modern code repository packers[1].

The Core Analogy

The pattern solves context problems by "compressing" code: it parses the Abstract Syntax Tree (AST) and removes implementation details (function bodies), keeping only the definitions (signatures). We apply this exact logic to narrative.

💻 Code Repository

Strategy: AST Stripping

```
function calculatePhysics() {
  // detailed implementation...
  // math logic...
  // extensive comments...
}

>> COMPRESSED TO HEADER:
declare function calculatePhysics(): void;
```

The Insight: The "Truth" is the Logic Signature. Implementation details are stripped to save tokens.

📖 Narrative (Continuum Flow)

Strategy: Semantic Compression

~~He felt a wave of nostalgia as...~~
`Arjun picks up the glowing shard.`
~~It reminded him of the winters in...~~
`The shard pulses red.`

```
>> COMPRESSED TO STATE:
{ actor: "Arjun", action: "Take Shard", prop_state: "Red Pulse" }
```

The Insight: The "Truth" is the Visual State. Internal monologue is "whitespace" that wastes tokens.

The Narrative AST (Abstract Story Tree)

Transforming prose into a rigid JSON Schema.

INPUT: RAW TEXT

```
The cyber-rain poured down. "Wait!" she screamed. Her robotic arm sparked. She didn't want to fight, but she had to.
```

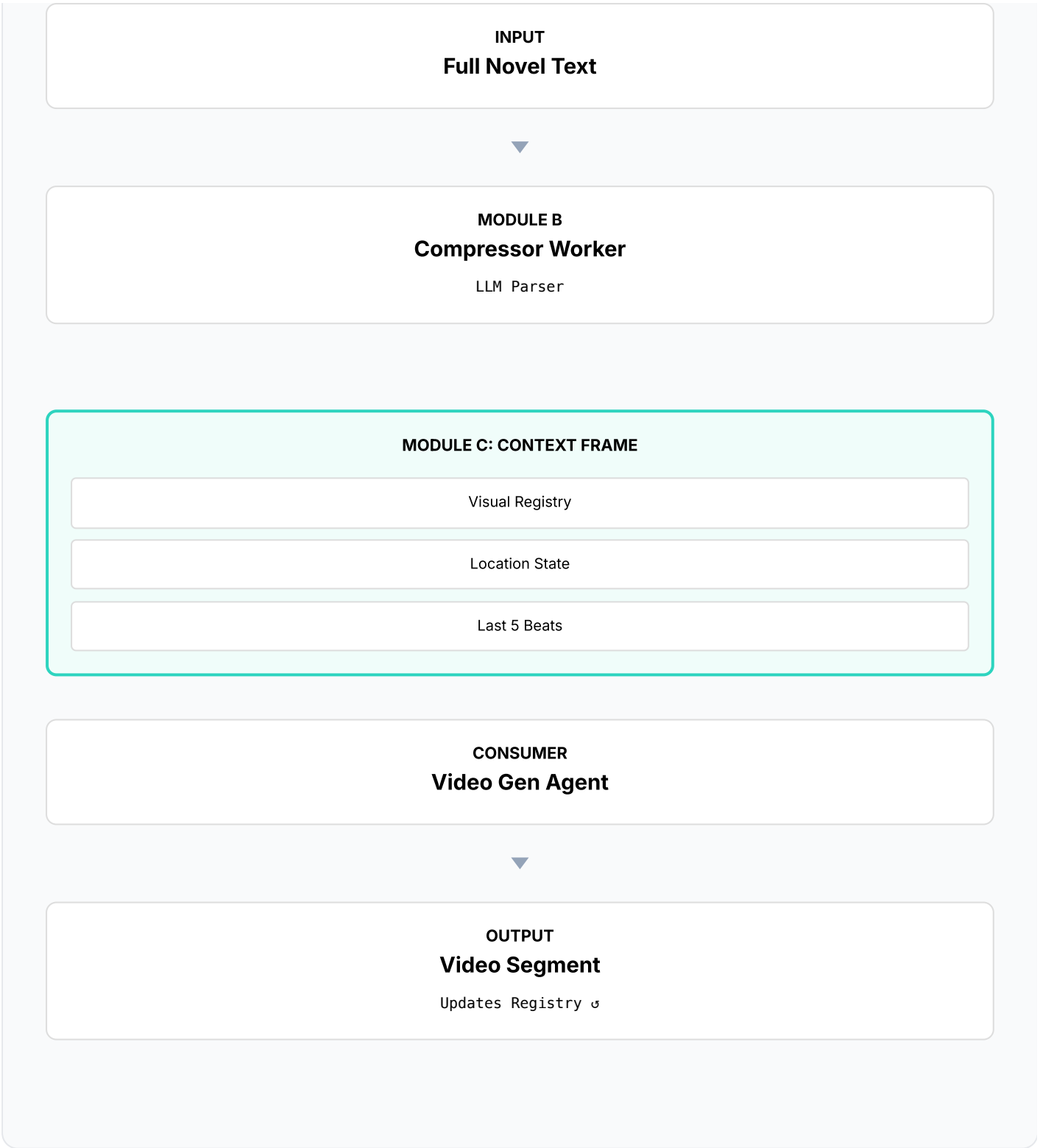
→

OUTPUT: SCENENODE JSON

```
{
  "env": "Rain (Cyber)",
  "audio": "Scream: 'Wait!'",
  "visual_fx": "Sparking Arm",
  "subtext": "Reluctant"
}
```

Network Architecture: Visualized

The following diagram illustrates the flow of data through the Semantic Compression Engine. It visualizes how raw text is ingested, parsed by the "Compressor Worker," structured into a rigid JSON Context Frame (Module C), and finally consumed by the Video Generation Agent.



Module Breakdown

Module A: The "Compressor" Worker (The Parser)

- **Role:** Acts as the parsing engine (similar to an AST strategy).

- **Algorithm:**

    1. **Ingest:** Takes a 5-page raw text buffer.
    2. **Entity Extraction (NER):** Identifies all Proper Nouns (Characters) and Physical Objects (Items).
    3. **State Differential Check:** Compares the current object description with the `GlobalRegistry`.
    4. **Action Distillation:** Summarizes 500 words of dialogue/action into a single atomic "Beat".
    5. **Discard:** Removes all "flavor text" (internal monologue, metaphors).

Module B: The "State Manager" (The Context Guard)

Instead of feeding the LLM "The last 10,000 words," it constructs a synthetic context frame containing:

1. **The "World State":** Current immutable facts (Time: Night, Weather: Rain, Health: 50%).
2. **The "Compressed Context":** The summary of previous chapters (Level 2 Context).
3. **The "Active Chunk":** The raw text of the current scene being generated.

Module C: The "Lookahead" Buffer

- **Role:** Parallel processing for temporal consistency.
- **Purpose:** To detect **Future State Changes** (e.g., a character losing an arm in Chapter 3) ensuring temporal consistency.

The Protocol Layer (TOON)

To combat "Context Rot," Continuum Flow abandons JSON for the Context Window. We utilize **TOON (Token-Oriented Object Notation)** for all state tracking.

Narrative consistency requires tracking hundreds of state variables (wounds, inventory, relationships). JSON's verbosity limits how much history we can retain. By switching to TOON, we fit **3x more chapters** into the same context window, allowing for 'novel-length' memory retention rather than just 'chapter-length'.

The "Sweet Spot" Analysis

TOON shines in one specific area: **Uniform Arrays of Objects**. In a novel, you track lists of characters, active props, and environmental states. JSON repeats the keys for every single item, wasting tokens. TOON defines the schema once.

🐌 Standard JSON

Overhead: High (Repeated Keys)

```
[
  { "name": "Arjun", "status": "injured" },
  { "name": "Mira", "status": "alert" },
  { "name": "Kael", "status": "asleep" }
]
```

Result: ~50 Tokens. Keys "name" and "status" repeated 3x.

⚡ TOON Protocol

Overhead: Minimal (Schema Header)

```
characters[3]{name,status}:
Arjun,injured
Mira,alert
Kael,asleep
```

Result: ~20 Tokens. 60% Reduction in Context Load.

Revised "TOON-Native" Workflow

We implement a safe pipeline to use TOON without risking LLM syntax hallucination.

1. **Extract (Safety):** The Agent reads Chapter 1 and outputs `scene_data.json`. We keep the LLM output as JSON because models are trained heavily on it.

2. **Compress (Worker):** A Node.js worker converts the JSON into `` `context_history.toon` ``. This ensures perfect syntax.
3. **Inject (Context):** When generating Chapter 2, we load the TOON file into the System Prompt: *"Here is the current state of the world in TOON format."*

---

[1] Reference Architecture: This pattern is inspired by RepoMix (https://github.com/yamadashy/repomix), a tool for packing codebases into LLM contexts using Tree-sitter for semantic understanding.