

# Laboratório VISGRAF

Instituto de Matemática Pura e Aplicada

**MMSW: A Computational Framework for Multimodal  
Narrative Generation**

*Luiz Velho and Matteo Moriconi*

Technical Report    TR-26-02    Relatório Técnico

March - 2026 - Março

The contents of this report are the sole responsibility of the authors.  
O conteúdo do presente relatório é de única responsabilidade dos autores.

# MMSW: A Computational Framework for Multimodal Narrative Generation

Technical Report + Runtime Appendix (Multicast Multimodal Server/Client)

Luiz Velho  
VISGRAF Laboratory

Matteo Moriconi  
VFXRio

February 22, 2026

## Abstract

This report presents a computational framework for narrative generation based on artificial intelligence. The system integrates multi-modal generative models within a unified architecture designed to support collaborative storytelling between human authors and artificial agents. The work establishes a conceptual hierarchy linking cultural knowledge, narrative universes, and individual stories, and describes the probabilistic and computational principles that govern narrative synthesis.

**Runtime appendix (this revision).** In addition to the conceptual model, we document the MMSW reference implementation built on the VFXRio multicast server/client stack: multi-client graph routing, multi-model prompt orchestration (guardrails, perception, prompt-writer, image model), RAM memory vs. persistent memory capsules, and a plugin architecture for integrating external tools and embodied avatar systems (e.g., Unreal/Unity). This revision also makes explicit that the implementation evolved from a minimal multicast WebSocket reflector and was progressively extended through AI-assisted software development.

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Conceptual Model . . . . .	3
<b>2</b>	<b>Story Worlds: Definition and Structure</b>	<b>3</b>
2.1	Semantic space and generative manifolds . . . . .	3
2.2	Constraints and universes . . . . .	4
2.3	Stories as trajectories . . . . .	4
<b>3</b>	<b>Probabilistic Foundations of Narrative Generation</b>	<b>4</b>
3.1	Language modeling and narrative probability . . . . .	4
3.2	Sampling and creativity . . . . .	4
3.3	Planning and constraint satisfaction . . . . .	4
<b>4</b>	<b>The MMSW Runtime: VFXRio as a Reference Implementation</b>	<b>4</b>
4.1	Origin in a minimal multicast reflector . . . . .	5
4.2	Server endpoints (room state, transcript, avatars) . . . . .	5
4.2.1	Updating environment state (datetime, location, scene) . . . . .	6
4.3	Graph-based routing (multicast as a directed story graph) . . . . .	6
<b>5</b>	<b>Running the System (Canonical Start Order + CLI Examples)</b>	<b>6</b>
5.1	Server + Relay + ngrok . . . . .	6
5.2	Client (HTML5 or curses UI) . . . . .	7

<b>6</b>	<b>Multi-Model Orchestration and Nested Prompts</b>	<b>9</b>
6.1	API mode and perception caching (practical notes)	9
6.2	Nested prompt layers	10
<b>7</b>	<b>RAM Memory vs. Persistent Memory Capsules</b>	<b>10</b>
7.1	RAM memory (rolling, bounded working memory)	10
7.2	Memory capsules (persistent narrative continuity)	10
7.3	Why capsules matter for multimodal generation	11
<b>8</b>	<b>Media Actions: #take_a_picture and #draw</b>	<b>11</b>
8.1	Operational mental model	11
8.2	Example prompt “cards”	11
8.3	Director brief vs. final prompt	11
8.4	Prompt budget and deterministic fallback	11
8.5	Avatar reference pool (identity seeding)	12
<b>9</b>	<b>Plugin Architecture and Extensibility</b>	<b>12</b>
9.1	Client Plugin API (message injection + interceptors)	12
9.2	Intent orchestration (Spotify, Hue, timers, etc.)	13
9.2.1	Recommended start order (Intent Router in API mode)	13
9.3	Multicasting Graph Dashboard (plugin)	13
9.4	Tooling node patterns (command/receipt buses)	14
9.5	Embodiment and real-time engines (Unreal / Unity)	14
<b>10</b>	<b>Evaluation and Future Directions</b>	<b>14</b>
<b>11</b>	<b>Conclusion</b>	<b>15</b>
<b>A</b>	<b>Appendix A: Minimal API calls (server graph control)</b>	<b>15</b>
<b>B</b>	<b>Appendix B: Capsule structure (illustrative)</b>	<b>15</b>
<b>C</b>	<b>Appendix C: Tooling node packet formats (command / receipt / speaking plan)</b>	<b>16</b>
<b>D</b>	<b>Appendix D: Historical prototype (basic WebSocket reflector)</b>	<b>16</b>
D.1	Minimal reflector server	17
D.2	Minimal client with AI-generated follow-up	17
D.3	Why this prototype matters	18

# 1 Introduction

Narrative is a fundamental mechanism through which human beings structure experience and construct meaning. The ability to represent events as sequences connected by causality and intention constitutes one of the defining features of human cognition. Storytelling enables the communication of knowledge, the preservation of culture, and the transmission of collective memory across generations. Throughout history, storytelling has evolved in parallel with technological developments. Oral traditions gave rise to written literature, which was later expanded through printing technology. The invention of photography introduced visual narrative representation, which was further extended by cinema and television. The emergence of digital media enabled interactive narratives and virtual environments.

Artificial intelligence introduces a new phase in the evolution of narrative media. Recent developments in machine learning have produced generative systems capable of synthesizing coherent language, realistic imagery, expressive audio, and dynamic video. These systems operate by learning statistical representations from large datasets and generating new outputs through probabilistic inference. Unlike traditional computational systems that execute predefined instructions, generative artificial intelligence produces novel content that was not explicitly programmed.

This technological transformation raises fundamental questions about the nature of creativity and the role of artificial systems in cultural production. The work presented in this report investigates the hypothesis that artificial intelligence can function not merely as a passive tool but as an active participant in narrative creation. The objective is to develop a computational framework capable of supporting collaborative storytelling involving both human creators and artificial agents.

## 1.1 Conceptual Model

The conceptual model underlying this work is based on the definition of a hierarchical structure that connects cultural knowledge to individual narrative instances. At the highest level of abstraction lies the concept of the *story world*. The story world may be understood as the complete semantic space containing all possible narrative elements. This includes historical events, fictional characters, cultural archetypes, symbolic structures, and conceptual relationships. From a computational perspective, the story world corresponds to a high-dimensional semantic manifold learned by artificial intelligence models during training.

Within the story world, specific *narrative universes* can be defined. A narrative universe represents a constrained region of the semantic space containing the elements relevant to a particular narrative context. This includes specific characters, environments, temporal settings, and thematic relationships. The narrative universe defines the conditions under which stories may emerge.

At the most concrete level are individual *stories*. A story is a realized trajectory through the narrative universe, consisting of a sequence of events, actions, and transformations that unfold over time. Stories may be generated through probabilistic sampling or guided planning within the constraints of the narrative universe.

# 2 Story Worlds: Definition and Structure

## 2.1 Semantic space and generative manifolds

In generative AI systems, a trained model can be interpreted as learning a distribution over a semantic space. This distribution defines a manifold of plausible narrative elements. Sampling from this distribution produces narrative content that reflects the statistical structure of the training corpus.

## 2.2 Constraints and universes

A narrative universe is defined by imposing constraints on the story world. Constraints may include character identities, setting, genre rules, causal laws, thematic boundaries, and stylistic guidelines. These constraints reduce the space of possible stories and enable coherent narrative generation. Computationally, constraints can be implemented through conditioning mechanisms such as prompts, latent variables, memory retrieval, and rule-based filters.

## 2.3 Stories as trajectories

A story can be modeled as a trajectory through the constrained narrative universe. Each step corresponds to a narrative state, and transitions are governed by probabilistic rules. In language models, narrative state transitions correspond to token generation conditioned on prior context. In multimodal systems, state transitions may include text, images, audio, and video outputs.

# 3 Probabilistic Foundations of Narrative Generation

## 3.1 Language modeling and narrative probability

Large language models approximate a probability distribution over sequences of tokens. A story can be represented as a sequence of narrative symbols. The probability of a story is computed as the product of conditional probabilities of each symbol given previous symbols.

## 3.2 Sampling and creativity

Creativity in AI narrative generation arises from stochastic sampling procedures. Temperature and nucleus sampling control the diversity of outputs. Higher diversity can produce surprising narrative developments but may reduce coherence; lower diversity increases determinism but can reduce novelty.

## 3.3 Planning and constraint satisfaction

To improve coherence, narrative generation can be guided by planning mechanisms. These may include explicit planning (generating an outline and expanding it), implicit planning via prompt structures, or hybrid approaches combining symbolic rules with probabilistic generation.

# 4 The MMSW Runtime: VFXRio as a Reference Implementation

This section documents the practical runtime architecture used to implement a multi-client, multimodal story world.

**AI-assisted evolution of the runtime.** An important methodological aspect of this project is that artificial intelligence was used not only within the runtime as a generative component, but also during the development of the runtime itself. The implementation began from a basic multicast WebSocket reflector—that is, a minimal server/client structure capable of forwarding messages between connected peers. From this initial prototype, the system was progressively extended and reprogrammed through AI-assisted development, which supported rapid iteration, code adaptation, feature expansion, and the integration of higher-level capabilities such as graph-based routing, multimodal orchestration, memory handling, plugin interfaces, and embodied avatar connectivity. In this sense, artificial intelligence contributed not only to the generation of narrative content, but also to the engineering evolution of the system, helping transform a

simple communication scaffold into a broader computational environment for multimodal story worlds.

The VFXRio stack provides:

- a central **multicast server** (HTTP + WebSocket) that publishes room events,
- multiple **clients** (humans or agents) that subscribe to the room stream and optionally auto-reply,
- an optional **telephony relay** (Twilio/voice bridge) for phone calls,
- a **plugin API** inside each client to allow external tools to inject messages or intercept inbound messages,
- a **graph layer** that routes messages by directed links (edges) between client nodes.

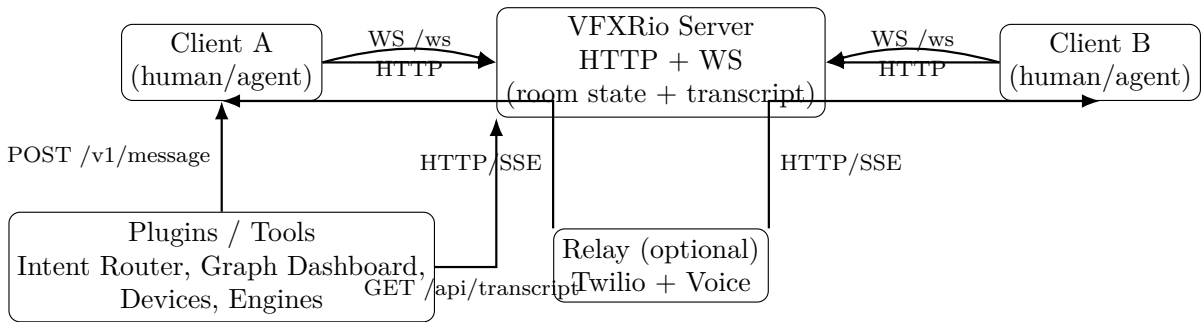


Figure 1: Multicast Multimodal Story Worlds runtime overview.

#### 4.1 Origin in a minimal multicast reflector

The historical starting point of the runtime was deliberately simple: a WebSocket server maintaining a set of connected clients and reflecting each received message to the other peers, together with lightweight clients capable of connecting to the server, receiving messages, and generating a follow-up reply. That minimal arrangement already contained the essential seed of a multicast conversational environment: shared connectivity, asynchronous message propagation, and agent-like response behavior. The later architecture preserved that core logic while layering on graph-based routing, environment state, persistent memory, tool injection, telephony bridges, and multimodal generation. For completeness, a cleaned version of this original prototype is reproduced in Appendix D.

#### 4.2 Server endpoints (room state, transcript, avatars)

The server exposes a minimal room API. Typical endpoints include:

- GET /state: lightweight server state,
- POST /api/datetime, POST /api/location, POST /api/scene: environment updates,
- GET /api/avatar/<uid>: avatar proxy for UIs,
- GET /api/transcript: raw room transcript for observer tools,
- WS /ws: multicast stream of room events.

This environment state is conceptually important for story worlds: it is the shared *situational frame* that stabilizes generation (time, location, and scene) across multiple clients and modalities.

### 4.2.1 Updating environment state (datetime, location, scene)

Environment state can be updated at runtime without restarting clients. A lightweight CLI tool sends updates to the server:

```
python -m vfxrio.cli datetime 'Oct 29, 1973'
python -m vfxrio.cli location 'Rome, Italy'
python -m vfxrio.cli scene 'reading the news in a coffee shop:
    https://metamessage.id34.com/time-news/'
```

Listing 1: Update shared environment state (examples).

When `-perception=on` (or `auto`), URLs in scene/location can be fetched and summarized by the perception reader, then injected into the dialogue context.

### 4.3 Graph-based routing (multicast as a directed story graph)

Unlike a simple broadcast room, VFXRio can route messages by a directed connectivity graph:

- **Nodes** are clients (identified by UID).
- **Edges** are links that define who receives whose messages.
- When a message is sent without an explicit recipients list, the server can expand recipients using the sender's outgoing adjacency list.

This graph layer is useful for story worlds because it supports:

- *role-based routing* (e.g., narrator speaks to all, but specialists only speak to narrator),
- *tooling nodes* (a tool agent receives commands from an orchestrator and emits receipts),
- *multi-stage narrative pipelines* (e.g., plan → critique → revise).

## 5 Running the System (Canonical Start Order + CLI Examples)

The system is typically started in four steps: server, relay, public relay exposure (ngrok), then clients.

### 5.1 Server + Relay + ngrok

```
# 1) Multicast server (room)
python -m vfxrio.server

# 2) Telephony relay (Twilio bridge), listening on 8080
python -m vfxrio.relay \
    --host 0.0.0.0 \
    --port 8080 \
    --twilio-timeout 8 \
    --twilio-speech-timeout auto

# 3) Expose relay publicly (for Twilio callbacks)
ngrok http 8080
```

Listing 2: Canonical start sequence (server + telephony relay + ngrok exposure).

## 5.2 Client (HTML5 or curses UI)

Clients can run as interactive terminals (`-uix=curses`) or a browser UI (`-uix=html5`). Below are two practical launch examples (as used in day-to-day MMSW operation).

```
python3 -m vfxrio.client \
  --uid=1 \
  --name='Celestia Borealis' \
  --persona='Celestia Borealis' \
  --color=GREEN \
  --self.id=on \
  --super \
  --api=chat \
  --max-depth=32 \
  --temperature=0.75 \
  --max-tokens=8000 \
  --guardrail=on \
  --guardrail-model=gpt-4o-mini \
  --guardrail-show=on \
  --persona-dir="/path/to/personas" \
  --capsule-dir="/path/to/memory-capsules" \
  --capsule-facts=on \
  --stream=off \
  --stream-throttle-ms=80 \
  --perception=on \
  --perception-detail=full \
  --perception-url-raw-chars=1200000 \
  --perception-url-context-chars=120000 \
  --perception-max-tokens=8000 \
  --image-model=gpt-image-1 \
  --image-model-fallback=dall-e-3 \
  --image-size=800x800 \
  --image-background=on \
  --image-background-workers=0 \
  --image-prompt-llm=on \
  --image-prompt-model=gpt-4o-mini \
  --image-prompt-temperature=0.4 \
  --image-prompt-max-tokens=700 \
  --image-prompt-ram-turns=25 \
  --image-prompt-ram-chars=4000 \
  --image-prompt-show=on \
  --auto-save-photo \
  --auto-save-draw \
  --media-dir="/path/to/media" \
  --uix=html5 \
  --uix-port=56440 \
  --uix-theme=dark \
  --telephony=on \
  --telephony-relay-url="https://<ngrok-subdomain>.ngrok-free.dev" \
  --speech-language="pt-BR,en-US" \
  --speech-language-phone=auto \
  --elevenlabs-phone-voice="YKLogvnVogI4aFHoGIEw" \
  --phone-transcript-streaming=on \
  --client-api=on \
  --client-api-runtime-dir="/path/to/vfxrio-expressions" \
  --private-network=on \
  --pushtotalk=on \
  --avatar-ref-upload=on \
  --avatar-ref-per-person=15 \
```

```

--avatar-ref-max-total=15 \
--avatar-ref-dir="/path/to/vfxrio-avatar-ref/UID-1" \
--image-prompt-brief=on \
--image-prompt-brief-model=gpt-4o-mini \
--image-prompt-brief-max-tokens=1200 \
--image-prompt-brief-temperature=0.2 \
--image-expression=auto \
--persona-open='UID-1-Celestia_Borealis-Date20260215-110647.json' \
--autosave-capsule \
--host 127.0.0.1 \
--stdout

```

Listing 3: Client example: Celestia Borealis (HTML5 UI, telephony, perception, image actions).

```

python3 -m vfxrio.client \
  --uid=5 \
  --name='Divine Freeman' \
  --persona='Divine Freeman' \
  --color=BLUE \
  --self.id=on \
  --self.uid=on \
  --super \
  --mode=auto \
  --api=chat \
  --max-depth=12 \
  --temperature=0.75 \
  --max-tokens=13000 \
  --guardrail=on \
  --guardrail-model=gpt-4o-mini \
  --guardrail-show=on \
  --persona-dir="/path/to/personas" \
  --capsule-dir="/path/to/memory-capsules" \
  --capsule-facts=on \
  --perception=on \
  --perception-detail=full \
  --perception-url-raw-chars=1200000 \
  --perception-url-context-chars=120000 \
  --perception-max-tokens=10000 \
  --image-model=gpt-image-1 \
  --image-model-fallback=dall-e-3 \
  --image-size=400x400 \
  --image-background=on \
  --image-background-workers=2 \
  --image-prompt-llm=on \
  --image-prompt-model=gpt-4o-mini \
  --image-prompt-temperature=0.3 \
  --image-prompt-max-tokens=600 \
  --image-prompt-ram-turns=32 \
  --image-prompt-ram-chars=4000 \
  --image-prompt-show=on \
  --image-prompt-brief=on \
  --image-prompt-brief-model=gpt-4o-mini \
  --image-prompt-brief-max-tokens=3000 \
  --image-prompt-brief-temperature=0.7 \
  --auto-save-photo \
  --auto-save-draw \
  --media-dir="/path/to/media" \
  --uix=html5 \

```

```

--uix-port=56420 \
--uix-theme=dark \
--telephony=on \
--telephony-relay-url="https://<ngrok-subdomain>.ngrok-free.dev" \
--speech-language="pt-BR,en-US" \
--speech-language-phone=auto \
--elevenlabs-phone-voice="YKlogvnVogI4aFHoGIEw" \
--phone-transcript-streaming=on \
--client-api=on \
--client-api-runtime-dir="/path/to/vfxrio-expressions" \
--private-network=on \
--pushtotalk=on \
--avatar-ref-upload=on \
--avatar-ref-per-person=15 \
--avatar-ref-max-total=15 \
--avatar-ref-dir="/path/to/vfxrio-avatar-ref/UID-5" \
--persona-open='UID-5-Divine_Freeman-Date20260214-233524.json' \
--host='192.168.1.174'

```

Listing 4: Client example: Divine Freeman (HTML5 UI, smaller max-depth, image actions).

## 6 Multi-Model Orchestration and Nested Prompts

A key design principle of MMSW is that different subtasks can be handled by different models. This supports both performance and reliability: a smaller/cheaper model can handle routing, verification, and prompt rewriting, while a larger model focuses on dialogue and narrative synthesis.

Role	Typical responsibility	Configuration knob(s)
Dialogue model	Main narrative response generation	<code>-model</code> , <code>-api</code>
Guardrail verifier	Evidence-based rewrite if unsupported claims appear	<code>-guardrail=on</code> , <code>-guardrail-model</code>
Perception reader	URL fetch+summary, image description, injected notes	<code>-perception=on/auto</code> , <code>-perception-model</code> , <code>-perception-api</code>
Director brief	Staging-focused intermediate summary for images	<code>-image-prompt-brief=on</code> , <code>-image-prompt-brief-model</code>
Prompt writer (“The Eye”)	Rewrite/condense context into final image prompt	<code>-image-prompt-llm=on</code> , <code>-image-prompt-model</code>
Image model	Render <code>#take_a_picture</code> / <code>#draw</code> outputs	<code>-image-model</code> , <code>-image-model-fallback</code>
Nested tool rewrite	Turn tool packet into grounded human-like reply	Environment / router settings (e.g., <code>o3-mini</code> )

Table 1: Multi-model decomposition (conceptual): different models can own different subtasks.

### 6.1 API mode and perception caching (practical notes)

In most deployments, the `responses` API mode is preferred for multimodal operation because it more naturally supports rich inputs. If a client runs in `chat` API mode, enabling the perception reader (`-perception=on` or `auto`) is strongly recommended when you rely on URLs, uploaded images, or other external context.

Perception notes are typically cached (default TTL is on the order of tens of minutes) so repeated references do not repeatedly re-fetch the same pages. To force re-fetching, operators can include a `refresh` keyword in the same message that references a URL.

## 6.2 Nested prompt layers

From an implementation perspective, a single user message may trigger multiple nested prompts:

1. **Perception (optional):** fetch/describe external URLs and images, producing *compact notes*.
2. **Dialogue draft:** main model generates a reply plan (and/or final answer).
3. **Guardrail (optional):** verifier model checks that claims are supported by context and rewrites only if needed.
4. **Media actions (optional):** if the message includes `#take_a_picture` or `#draw`, the client runs an image prompt pipeline:
  - (a) Director brief pass (staging summary),
  - (b) Prompt-writer pass (TheEye rewrite) to produce the final image prompt,
  - (c) Image model call (with strict prompt length constraints).

This layered approach is crucial in story worlds because it helps keep *long-running continuity* coherent even when individual model calls are bounded.

## 7 RAM Memory vs. Persistent Memory Capsules

Long-running story worlds require memory beyond a single prompt window. MMSW uses two complementary mechanisms:

### 7.1 RAM memory (rolling, bounded working memory)

**RAM** refers to the in-session working context: recent transcript turns, environment state, and short-lived caches. RAM is bounded (e.g., by `-max-depth` and by prompt-size constraints). It is optimized for:

- dialogue coherence (what was just said),
- immediate scene grounding (time/location/scene),
- image prompt continuity (the most recent relevant turns, limited by `-image-prompt-ram-turns` and `-image-prompt-ram-chars`).

### 7.2 Memory capsules (persistent narrative continuity)

**Memory capsules** persist across sessions. A capsule is a structured record (often stored as a folder of files) that contains:

- a three-part summary (participants/roles, themes, abstract),
- extracted facts with timestamps/evidence (optional but recommended for grounding),
- metadata such as a capsule id and creation time.

In practice, capsules act as a *compressive memory*: they trade raw transcript length for a smaller representation that can be re-injected into future prompts. This is critical for story worlds that evolve over days/weeks.

### 7.3 Why capsules matter for multimodal generation

Image generation is particularly sensitive to prompt budgets. If a client tries to feed too much transcript into an image prompt, the image API may reject the request. The capsule abstraction helps because:

- it provides *high-density continuity* (themes, identities, recurring props) in a compact form,
- it reduces the need to over-inject irrelevant chat history,
- it enables deterministic fallbacks when a prompt-writer rewrite becomes too long or drops required subjects.

## 8 Media Actions: #take\_a\_picture and #draw

Two core multimodal actions are designed for day-to-day story world operation:

- **#take\_a\_picture**: camera-like staged image generation, photoreal or documentary feeling.
- **#draw**: illustration/concept-art generation, symbolic or stylized feeling.

### 8.1 Operational mental model

A VFXRio message can simultaneously communicate with other participants and trigger a media action. When a message includes **#take\_a\_picture** or **#draw**, the receiving client builds an image prompt using: (1) the user directive, (2) RAM excerpts, (3) identity seeds (persona + avatars), (4) optional director brief, (5) optional prompt-writer rewrite (TheEye), and then calls the image model.

### 8.2 Example prompt “cards”

In practice, strong media prompts are short, structured, and enforce constraints:

```
#take_a_picture Documentary-style phone photo: Divine (yellow) and Celestia (green)
in a rainy-night cafe, neon reflections on the window, mid-conversation, 35mm,
shallow DOF, warm interior + cool outside light, no text / watermark.

#draw Painterly concept art: Divine (yellow lantern) and Celestia (green lantern)
on a foggy bridge symbolizing the relay, limited palette (yellow/green/cool blues),
strong leading lines, no text / watermark.
```

Listing 5: Compact prompt examples (one-liners).

### 8.3 Director brief vs. final prompt

When `-image-prompt-brief=on`, the client generates a short staging-oriented brief (“director notes”) from transcript/RAM. This brief is *not* the final prompt; it is an intermediate representation that improves context faithfulness without ballooning prompt length.

### 8.4 Prompt budget and deterministic fallback

Image APIs enforce strict limits on prompt length. The client therefore maintains multiple layers of compression: RAM excerpts are clipped (`-image-prompt-ram-turns`, `-image-prompt-ram-chars`), and the prompt-writer attempts to compress while preserving required subjects.

If the prompt-writer drops required subjects (e.g., character identities), the client rejects the rewrite and falls back to a deterministic prompt. For debugging, enable `-image-prompt-show=on` to print the *BASE* and *FINAL* prompts and watch for log lines such as:

```
[image_prompt] ... rejecting rewrite: dropped required subjects -> fallback to
deterministic prompt
HTTP 400: prompt too long -> reduce --image-prompt-ram-turns /
--image-prompt-ram-chars
```

Listing 6: Common debugging signals for image prompt orchestration.

Operationally, prefer short “shot card” directives and let the prompt-writer handle density rather than adding long adjective chains.

## 8.5 Avatar reference pool (identity seeding)

To stabilize character identity across generated images, clients can upload a small set of avatar reference images to the server and reuse them during image generation. Two controls are important:

- `-avatar-ref-per-person`: how many references per depicted participant,
- `-avatar-ref-max-total`: a hard cap per generation (kept low for reliability).

These reference images can be used as *conditioning inputs* to the image model (in addition to the text prompt), improving consistency for recurring characters.

## 9 Plugin Architecture and Extensibility

Story worlds become more powerful when they can *act* on the world, not only describe it. VFXRio supports this through a client-local Plugin API and external “observer” processes.

### 9.1 Client Plugin API (message injection + interceptors)

Each running client can expose a local HTTP API (sidecar) that allows external tools to:

- inject a message as if typed (POST `/v1/message`),
- set/replace draft UI text without sending (POST `/v1/draft`),
- query client status (GET `/v1/state`),
- optionally register interceptors to analyze/modify/drop inbound messages before they reach the UI.

```
curl -X POST "http://127.0.0.1:<client_api_port>/v1/message" \
-H "Authorization: Bearer <TOKEN>" \
-H "Content-Type: application/json" \
-d '{"content":"Respond to USER:
...","recipients":null,"responding_to":null,"source":"intent-router"}'
```

Listing 7: Inject a message into a running client via the Plugin API.

## 9.2 Intent orchestration (Spotify, Hue, timers, etc.)

The Intent System is a separate process that observes the room and decides whether a user message corresponds to a tool intent (e.g., music control, lights control, timers). In its recommended API mode, it:

1. connects to the server WebSocket to detect new activity,
2. pulls the latest transcript from GET `/api/transcript`,
3. decides what to do (tool action / timer / clarification / conversation),
4. injects exactly one router prompt into one or more agent clients via POST `/v1/message`.

This separation is important: tools should not open a competing “chat client” connection. Instead they inject into existing agents.

### 9.2.1 Recommended start order (Intent Router in API mode)

The Intent Router discovers agent clients by reading descriptor files (e.g., `vfxrio-client-<uid>.json`) from a shared runtime directory. Each agent writes its `api_base_url` and `token` there on startup, allowing the router to inject prompts safely.

```
# Optional: enable/disable the nested LLM normalization layer
export INTENT_USE_LLM=1 # or 0 for rule-first only

python3 -m intent_system.run_live_router_api \
  --super-admins-uids='1,2' \
  --agents-uids='5,6' \
  --client-api-runtime-dir='/path/to/vfxrio-runtime'
```

Listing 8: Start the Intent System router (API mode).

## 9.3 Multicasting Graph Dashboard (plugin)

The Graph Dashboard is a browser-based UI that visualizes clients as nodes and links as edges, supports directed/undirected views, and applies common patterns (star, ring, chain, connect-all). It typically relies on:

- a backend implementing `/api/graph`, `/api/connect`, `/api/disconnect`, etc.,
- an SSE stream `/api/events` for near-real-time refresh,
- optional avatar lookups at `/api/avatar/{uid}`,
- an optional plugin host providing `/_plugin/*` endpoints for blueprint upload + layout persistence.

Blueprint overlays allow operators to map the network onto a physical floorplan (useful for installations and spatial storytelling).

## 9.4 Tooling node patterns (command/receipt buses)

For larger deployments, the multicast graph can be structured into buses:

- **Command bus:** orchestrator → tool nodes (structured commands),
- **Receipt bus:** tool nodes → orchestrator (structured receipts),
- **Speech bus:** orchestrator → conversational agent (final plan or talking points).

This makes complex story worlds safer: the speaking agent does not claim that a device action succeeded until receipts confirm it.

## 9.5 Embodiment and real-time engines (Unreal / Unity)

The same plugin interface that supports device tools can also support avatar embodiment:

- A real-time engine (Unreal/Unity) can subscribe to room state (graph, scene, transcript events) and drive animation or rendering.
- The engine can send back events (gesture triggers, emotion state, camera framing requests) by injecting structured messages into an agent client.
- Image generation and real-time embodiment can coexist: images provide “still frames”; engines provide continuous animation.

In this sense, story worlds become *embodied worlds*: the narrative does not only exist as text, but also as a living, multi-modal scene that can be rendered, acted, and sensed by external systems.

# 10 Evaluation and Future Directions

The conceptual framework and the runtime implementation together suggest several promising directions:

- **Narrative stability:** improving continuity by combining RAM, capsules, and explicit world-state constraints.
- **Multi-agent orchestration:** using directed graphs and tooling nodes to create robust narrative pipelines.
- **Embodiment:** coupling story generation with real-time avatar systems and interactive environments.
- **Governance:** using guardrails, receipts, and access control for safe tool actuation in shared spaces.
- **AI-assisted software evolution:** studying how generative systems can accelerate the iterative design of complex multimodal infrastructures while preserving coherence, traceability, and operator control.

## 11 Conclusion

This report described a computational framework for multimodal narrative generation based on artificial intelligence. At the theoretical level, story worlds, narrative universes, and stories were modeled as hierarchical structures within a learned semantic space. At the practical level, the MMSW runtime shows how a multicast, multi-client architecture can operationalize these ideas: distributed agents collaborate, media actions render visual artifacts, memory capsules preserve continuity, and plugins connect narrative to tools and embodied worlds. An additional contribution of the project is methodological: AI was used not only as a narrative engine inside the system, but also as an instrument in the progressive extension and reconfiguration of the system's own capabilities.

## A Appendix A: Minimal API calls (server graph control)

The multicast graph can be controlled via HTTP endpoints (used by the Graph Dashboard plugin). Examples:

```
# Fetch current graph
curl http://127.0.0.1:9100/api/graph

# Create a directed edge A -> B
curl -X POST http://127.0.0.1:9100/api/connect \
  -H "Content-Type: application/json" \
  -d '{"a": "1", "b": "5"}'

# Remove edge A -> B
curl -X POST http://127.0.0.1:9100/api/disconnect \
  -H "Content-Type: application/json" \
  -d '{"a": "1", "b": "5"}'
```

Listing 9: Graph control examples (connect, disconnect, fetch graph).

## B Appendix B: Capsule structure (illustrative)

A typical capsule save creates a folder with files such as `transcript.txt`, `summary.txt`, `facts.json`, and `capsule.json`. An illustrative `capsule.json` may resemble:

```
{
  "capsule_id": "UID-5-Divine_Freeman-Date20260214-233524",
  "created_at": "2026-02-14T23:35:24Z",
  "summary": {
    "users_and_roles": "...",
    "themes": "...",
    "abstract": "..."
  },
  "facts": [
    {"fact": "Celestia uses green theme.", "timestamp": "...", "evidence": "..."}
  ]
}
```

Listing 10: Illustrative `capsule.json` structure (example keys).

## C Appendix C: Tooling node packet formats (command / receipt / speaking plan)

When tools are represented as nodes in the multicast graph, it is helpful to standardize three packet types:

- `:tool_call` — an actionable command sent to a tool node,
- `:tool_receipt` — a confirmation sent back by the tool node (with evidence artifacts),
- `:say_plan` — a “speaking plan” sent to the conversational agent, derived from receipts.

```
:tool_call {
  "tool": "spotify",
  "action": "play",
  "query": "Take Five Dave Brubeck",
  "request_id": "20252123T203011Z-7f2c",
  "who": "operator"
}
```

Listing 11: Example `:tool_call` packet (command bus).

```
:tool_receipt {
  "tool": "spotify",
  "request_id": "20252123T203011Z-7f2c",
  "ok": true,
  "changed": true,
  "now_playing": {"track": "Take Five", "artist": "The Dave Brubeck Quartet"},
  "artifacts": [
    {"name": "spotify_last10_window.csv", "remote":
      "/public/callback/spotify_last10_window.csv"},
    {"name": "spotify_upcoming10.json", "remote":
      "/public/callback/spotify_upcoming10.json"}
  ]
}
```

Listing 12: Example `:tool_receipt` packet (receipt bus).

```
:say_plan {
  "to": "human",
  "facts": {
    "music_track": "Take Five",
    "music_artist": "The Dave Brubeck Quartet"
  },
  "actions_taken": ["spotify:play(query)"],
  "tone": "celestia"
}
```

Listing 13: Example `:say_plan` packet (only after receipts).

## D Appendix D: Historical prototype (basic WebSocket reflector)

For historical and methodological completeness, this appendix reproduces a simplified version of the minimal reflector-based prototype from which the broader runtime evolved. The purpose of including it here is not to present it as the final architecture, but to show the original communication scaffold that was later extended through AI-assisted iteration.

## D.1 Minimal reflector server

```
# server.py
import asyncio
import websockets

# Store connected clients
clients = set()

async def reflect_message(message, websocket):
    # Send the received message to all other connected clients
    for client in clients:
        if client != websocket:
            try:
                await client.send(message)
                print(f"Broadcasted message to client {client.remote_address}:
{message}")
            except websockets.exceptions.ConnectionClosedError:
                # Remove the client if the connection is closed
                clients.remove(client)

async def reflector_handler(websocket, path):
    # Register new client
    clients.add(websocket)
    try:
        async for message in websocket:
            # Reflect the received message to all other clients
            await reflect_message(message, websocket)
    except websockets.exceptions.ConnectionClosedError:
        pass # Connection closed, do nothing
    finally:
        # Unregister client when connection is closed
        clients.remove(websocket)

async def main():
    uri = "0.0.0.0"
    port = 5678
    start_server = await websockets.serve(reflector_handler, uri, port)
    print(f"Reflector server running on ws://{uri}:{port}")
    # Wait for the server to finish serving
    await start_server.wait_closed()

loop = asyncio.new_event_loop()
asyncio.set_event_loop(loop)
loop.run_until_complete(main())
```

Listing 14: Basic multicast WebSocket reflector (historical starting point).

## D.2 Minimal client with AI-generated follow-up

```
# client1.py
import asyncio
import websockets
import openai
import os

# Assuming OPENAI_API_KEY is set in your environment variables
```

```

openai.api_key = os.getenv('OPENAI_API_KEY')

async def generate_response_and_new_question(content, name):
    prompt = f"Generate a relevant follow-up based on: '{content}'"
    response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo",
        messages=[
            {
                "role": "system",
                "content": "Respond to all interactions with insightful and engaging
dialogue."
            },
            {"role": "user", "content": prompt}
        ],
    )
    generated_text = response.choices[0].message['content'].strip()
    return f"{name}: {generated_text}"

async def handle_messages(websocket, name, initiate_conversation):
    if initiate_conversation:
        await websocket.send(f"{name}: {input('Your message: ')}")

    async for message in websocket:
        print(f"\nReceived: {message}")

        # Extract the message content, assuming format "Name: Message"
        content = message.split(": ", 1)[1]

        # Skip generating and sending a response if this client sent the original
        # message
        if message.startswith(name):
            continue

        # Generate a response based on the received message
        response = await generate_response_and_new_question(content, name)
        await websocket.send(response)

async def main():
    uri = "ws://192.168.1.105:5678" # Adjust to match your WebSocket server's URI
    name = input("Your Name: ")
    initiate = input("Do you want to start the conversation? (yes/no): ")
    initiate.strip().lower() == 'yes'

    async with websockets.connect(uri) as websocket:
        await handle_messages(websocket, name, initiate)

asyncio.run(main())

```

Listing 15: Basic client with automatic AI follow-up (historical starting point).

### D.3 Why this prototype matters

Even in this early form, the prototype already established several principles that remained central to the later MMSW/VFXRio runtime:

- asynchronous peer connectivity,
- server-mediated multicast propagation,

- agent-like auto-reply behavior,
- a separation between transport logic and response-generation logic.

What changed in the full system was not the abandonment of these principles, but their elaboration: a simple reflector became a graph-aware narrative infrastructure; a single AI reply loop became multi-model orchestration; and ad hoc peer messaging became a persistent, multimodal, tool-integrated story world.