

Laboratório VISGRAF

Instituto de Matemática Pura e Aplicada

**MMSW Runtime: Complexity Metrics and Evolution from a
WebSocket Prototype**

Matteo Moriconi and Luiz Velho

Technical Report TR-26-01 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 Runtime: Complexity Metrics and Evolution from a WebSocket Prototype

Matteo Moriconi Luiz Velho

March 8, 2026

Abstract

This document summarizes, in academic format, the main engineering metrics of the *MMSW Runtime* (server, client, and telephony layer), with the goal of characterizing size, API surface, asynchronous concurrency, and complexity implications. We also include, for historical traceability, the original prototype (server and client) from which the architecture evolved.

1 Context and objective

The MMSW Runtime implements a *runtime* for *AI story worlds* with: (i) environment state and an entity connection graph, (ii) real-time communication (WebSocket), (iii) multimodal actions (media, plugins, and API calls), and (iv) a telephony layer for voice interaction. The purpose of this report is to provide objective metrics of the current codebase and an evidence-based reading of its complexity.

2 Architecture overview

Figure 1 synthesizes the main blocks: *Runtime Server* (state/graph/realtime hub), *Runtime Client* (orchestration + UI + multimodal actions), and *Telephony Relay* (call cycle, webhooks, and speech synthesis/input).

3 Measurement methodology

The metrics presented were obtained through static analysis of the repository snapshot (`vfxrio.zip`):

- **Raw LOC**: count of all lines in text files.
- **SLOC (approx.)**: approximate count of non-empty lines, excluding full-line comments.
- **Routes**: extraction of HTTP/WebSocket decorators (e.g., `@app.get(...)`), counting unique routes per service.
- **Asynchronous**: count of `async def` per component.

These metrics do not estimate cyclomatic complexity; they characterize the system's *size*, *surface*, and *concurrent nature*.

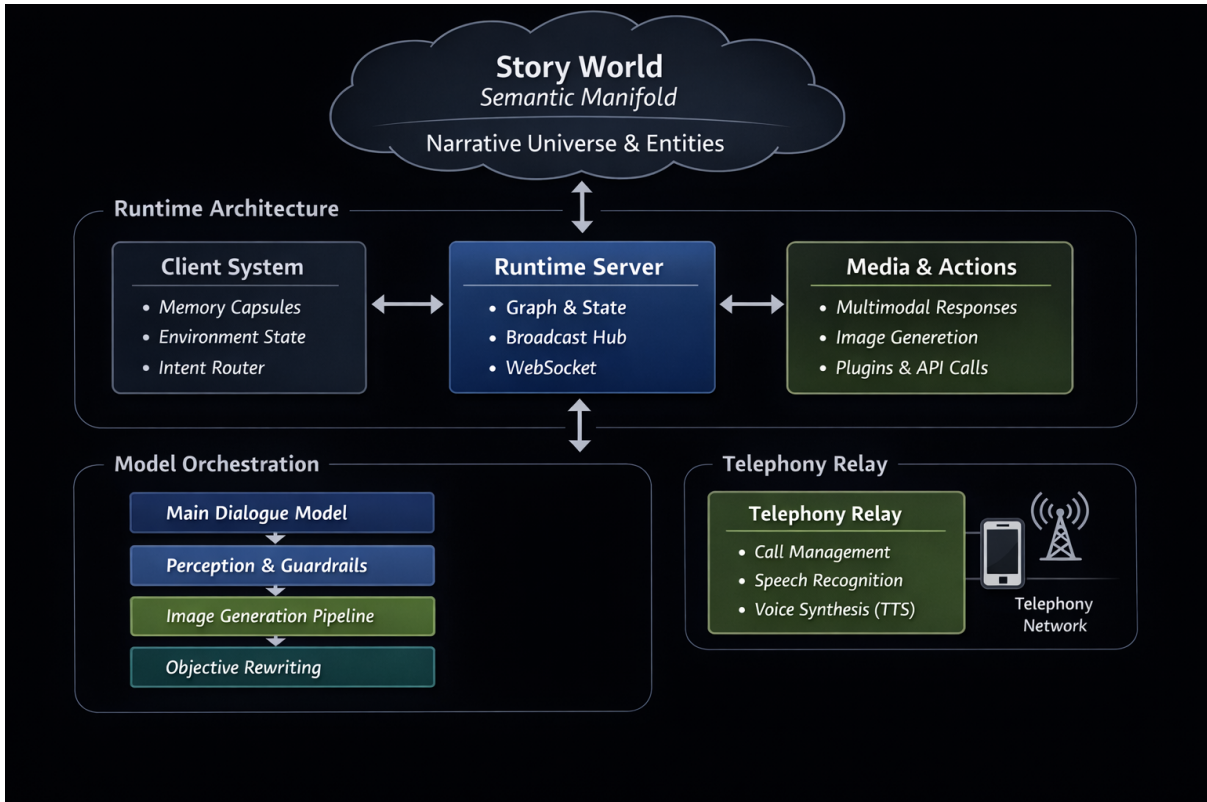


Figure 1: Dark-mode diagram of the MMSW Runtime architecture.

4 Main results

4.1 Size and distribution

Table 1 shows that the *Client* concentrates most of the code (around 70% of Raw LOC), which is consistent with an architecture where:

- the *server* acts as a lean hub (state, graph, broadcast, and realtime),
- the *client* concentrates orchestration, UI, multimodal pipelines, and telephony integration.

4.2 Integration surface

Table 2 summarizes 44 unique routes distributed across services (server, relay, local client API, and FlicAudio). This surface defines the *integration boundary* (plugins, tools, telephony webhooks, and real-time synchronization).

4.3 Concurrency

Table 3 indicates 158 asynchronous functions (`async def`), signaling that the system is strongly oriented toward real-time I/O (WebSocket, webhooks, workers, and tasks).

5 Complexity reading (based on the metrics)

Based on size, distribution, concurrency, and API surface, the overall complexity of the MMSW Runtime can be classified as **medium-high**, with **higher complexity concentrated in the Client** (because it centralizes orchestration and multimodality). In practical terms:

Table 1: Codebase size by component (repository snapshot). Raw LOC counts all lines; SLOC is an approximate count excluding blank lines and full-line comments.

Component	Files	Raw LOC	SLOC (approx.)	Share (raw)
Client (runtime)	33	24,364	19,178	70.1%
Server (runtime)	32	1,697	1,321	4.9%
Telephony Relay	11	2,849	2,194	8.2%
FlicAudio	7	1,374	1,019	4.0%
CLI tools	25	1,950	1,529	5.6%
Berry / device	3	1,010	771	2.9%
Core (shared)	3	1,489	1,138	4.3%
Total	114	34,733	27,150	100.0%

Table 2: HTTP/WebSocket surface by service (unique routes).

Service	Routes	Methods	Notes
Runtime Server	26	POST=16, GET=9, WebSocket=1	
Telephony Relay	8	POST=5, GET=3	
Client Local API	8	GET=3, POST=3, PATCH=1, DELETE=1	
FlicAudio	2	GET=1, POST=1	
Total	44		

1. **Architectural complexity** (service integration) is moderate: there are few services, but with clearly defined boundaries (server, client, relay, ingest).
2. **Operational complexity** is high: telephony and realtime introduce partial failures, variable latency, and reentrancy.
3. **Maintenance complexity** tends to concentrate in large Client modules (a typical feature-coupling point).
4. **Orchestration complexity** is high: the runtime uses a *model stack* with multiple configurable roles (e.g., main dialogue, reading/perception, verification/guardrail, prompt rewriting, media pipeline, and call-goal normalization). (Specific identifiers are deliberately omitted from the body text.)

Practical indicator (slide-ready). A useful summary for executive communication:

“The system is predominantly client-driven (70% of the code), with strong concurrency (158 asynchronous functions) and a moderate integration surface (44 routes). This characterizes a multimodal and realtime runtime with medium-high complexity, especially at the edge/orchestration layer.”

6 Origin: original prototype (server & client)

For traceability purposes, we include below the initial prototype: a WebSocket *reflector server* and a client that listens to messages and generates automatic responses. The identifiers and strings of the prototype were preserved *verbatim*.

Table 3: Asynchronous functions (count of `async def`) by component.

Component	# async functions
Client (runtime)	66
Server (runtime)	35
Telephony Relay	35
FlicAudio	16
Berry / device	5
Core (shared)	1
CLI tools	0
Total	158

```
#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())

#client
import asyncio
import websockets
import openal
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 = openal.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):
    await websocket.send(f"({name}): (input 'Your message: ')") # Send the initial message only if initiating
    if initiate_conversation:
        async for message in websocket:
            print(f"Received: {message}")
            # Extracting 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

            # Generating 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): ").strip().lower() == 'yes'

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

    asyncio.run(main())
```

(a) Prototype: WebSocket server (reflector).

(b) Prototype: WebSocket client with response generation.

Figure 2: System origin: attached initial prototype.

6.1 Prototype captures (attachments)

6.2 Prototype code (transcribed)

Listing 1: Original prototype: server.py (WebSocket reflector).

```
Server (reflector)
#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 2: Original prototype: client1.py (listener + response generation).

```

#client1
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: ')}") # Send the initial
            message only if initiating

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

        # Generating 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): ").strip().
        lower() == 'yes'

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

asyncio.run(main())

```

7 Conclusion

The metrics support the thesis that the MMSW Runtime is a *realtime* and *multimodal* system with medium-high complexity, concentrated primarily in the Client and amplified by telephony and asynchronous concurrency. Its evolution from the original prototype demonstrates a substantial increase in surface area and responsibilities, consistent with the transition from a “reflector + simple bot” into a story-world runtime with state, graph, and multiple pipelines.