International Journal of Science and Research Archive
eISSN: 2582-8185
Cross Ref DOI: 10.30574/ijsra
Journal homepage: https://ijsra.net/

(SHORT COMMUNICATION)

Check for updates

BUILDING A SCALABLE MULTI-PLATFORM CHATBOT WITH TELEGRAM, WEBSOCKET, AND REDIS PUB/SUB

Roman Yurievich Postovalov [1,*] and Anatoly Antonovich Bobrovsky [2]

[1] Individual Entrepreneur, 24rd Street, Building 35, Zovuni, Kotayk Province 2406, Armenia
[2] EORA, Bishkek 720000, Kyrgyzstan
* Corresponding Author

International Journal of Science and Research Archive, 2026, 20(03), 185–189

Publication history: Received on 26 July 2026; revised on 05 September 2026; accepted on 07 September 2026

Article DOI: https://doi.org/10.30574/ijsra.2026.20.3.1725

ABSTRACT

This short communication describes the design and functional verification of a multi-platform chatbot architecture in Python. The architecture separates platform-independent message processing from platform adapters and uses Redis publish/subscribe (Pub/Sub) as the communication layer. The prototype comprises a Core Bot, a Telegram adapter, a WebSocket adapter, and a Redis service deployed with Docker Compose. A shared message model and a callback-channel attribute enable the Core Bot to return a response through the same adapter that received the original user message. Functional tests using Telegram and WebSocket clients confirmed the expected routing of messages through the Core Bot and platform-specific Redis channels. The implementation provides a compact pattern for extending a chatbot to additional communication platforms while retaining a single message-processing component. The reported tests demonstrate functional routing only; they do not quantify throughput, latency, or fault tolerance.

Keywords: Multi-Platform Chatbot, Redis Pub/Sub, Telegram Bot, WebSocket, Docker Compose, Message Routing

1 INTRODUCTION

Chatbots frequently need to serve users through more than one communication channel. A design in which platform-specific code is tightly coupled to the message-processing logic becomes harder to maintain as new channels are added. This work presents a compact architecture that separates a shared Core Bot from platform adapters for Telegram and WebSocket clients.

WebSocket provides bidirectional client-server communication over a persistent connection [1]. Redis Pub/Sub allows publishers and subscribers to communicate through named channels without direct knowledge of one another [2]. In the proposed design, these mechanisms support a platform-neutral message flow: adapters translate external messages into a shared internal format, while the Core Bot processes that format and returns the response through a callback channel.

2 MATERIALS AND METHODS

2.1 System Architecture

The prototype contains four Docker-managed services: telegram_bot, websocket_chat, core_bot, and redis. The Core Bot contains platform-independent processing logic. The Telegram and WebSocket services act as adapters that translate incoming messages into a shared Message object and send responses back to the appropriate client. All services communicate through Redis Pub/Sub on a Docker bridge network named chat.

The message-routing architecture is presented in Figure 1.

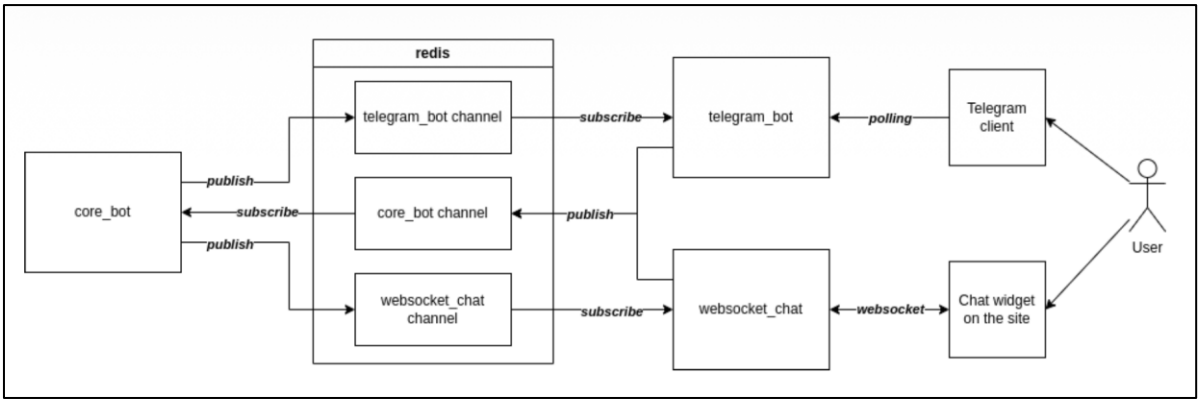

Figure 1: Message-routing architecture of the multi-platform chatbot.

2.2 Deployment Configuration

Docker Compose was used to define and run the multi-container application [3]. The WebSocket service exposes port 8000, Redis exposes port 6379, and Redis data are stored in an external volume named redis-data.

```
services:
 telegram_bot:
  build:
   context: .
   dockerfile: ci/telegram_bot/Dockerfile
  environment:
   TELEGRAM_BOT_TOKEN: ${TELEGRAM_BOT_TOKEN}
 websocket_chat:
  build:
   context: .
   dockerfile: ci/websocket_chat/Dockerfile
  ports:
   - "8000:8000"
 core_bot:
  build:
```

```
    context: .
    dockerfile: ci/core_bot/Dockerfile
  redis:
    image: redis:7.4
    ports:
    - "6379:6379"
```

Listing 1. Docker Compose configuration for the multi-platform chatbot.

2.3 Message Routing and Platform Adapters

The services were built from the python:3.12-slim image. The shared module used redis 5.2.1; the Telegram adapter used aiogram 3.15.0 and pydantic-settings 2.7.0; and the WebSocket adapter used FastAPI 0.115.6, Uvicorn 0.33.0, and websockets 14.1 [4-6].

```
class Channel(Enum):
    TELEGRAM_BOT = "telegram_bot"
    CORE_BOT = "core_bot"
    WEBSOCKET_CHAT = "websocket_chat"
@dataclass
class Message:
    text: str
    chat_id: str
    callback_channel: Channel
    def to_json(self) -> str:
        data = asdict(self)
        data["callback_channel"] = self.callback_channel.value
        return json.dumps(data)
async def publish(channel: Channel, message: Message) -> None:
    await redis_client.publish(channel.value, message.to_json())
async def subscribe(channel: Channel, process_message) -> None:
    async with redis_client.pubsub() as pubsub:
        await pubsub.subscribe(channel.value)
        while True:
            event = await pubsub.get_message(ignore_subscribe_messages=True          )
            if event:
```

```
        data = json.loads(event["data"])
        data["callback_channel"] = Channel(  data["callback_channel"]          )
        await process_message(Message(**data))
```

Listing 2. Channel-agnostic message model and Redis Pub/Sub transport.

The callback_channel attribute identifies the adapter to which the Core Bot must return a response. Telegram communication uses the Bot API [5], while the WebSocket adapter is implemented with FastAPI [4].

```
@dp.message()
async def message_handler(message: TelegramMessage) -> None:
  await publish(
    Channel.CORE_BOT,
    Message(
      text=message.text,
      chat_id=str(message.chat.id),
      callback_channel=Channel.TELEGRAM_BOT,
    ),
  )
async def process_message(message: Message) -> None:
  response = get_echo_response(message)
  await publish(message.callback_channel, response)
await subscribe(Channel.CORE_BOT, process_message)
```

Listing 3. Telegram adapter and platform-independent Core Bot processing.

3 RESULTS AND DISCUSSION

3.1 Functional Verification

The prototype was functionally verified using Telegram and WebSocket clients. In the WebSocket scenario, a client message was published to the CORE_BOT channel with WEBSOCKET_CHAT specified as the callback channel. After processing, the Core Bot returned the response through WEBSOCKET_CHAT. In the Telegram scenario, incoming messages were published to CORE_BOT with TELEGRAM_BOT as the callback channel, and the response was returned to the relevant chat.

Redis Insight was used to observe the message flow. The events shown in Figure 2 include messages from both client types in core_bot, followed by responses in their corresponding platform-specific channels.

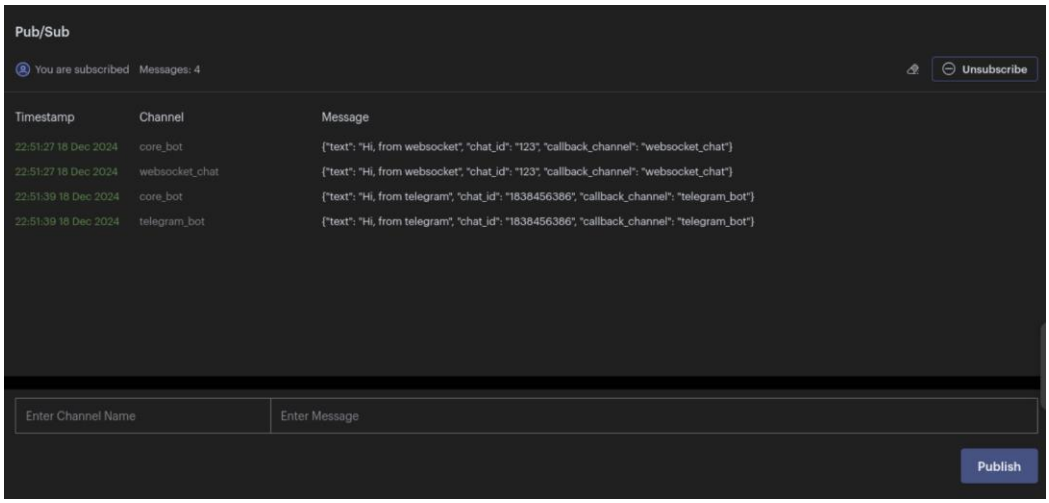

Figure 2: Redis Insight view of messages routed through the Core Bot during functional verification.

4 DISCUSSION

The functional test shows that a single Core Bot can process messages from different adapters without platform-specific changes to its processing logic. The callback-channel mechanism permits a response to be returned through the adapter from which the originating message was received.

The verification demonstrates correct routing for the implemented prototype. It does not measure throughput, response latency, fault tolerance, or behaviour under concurrent load. In addition, Redis Pub/Sub uses at-most-once delivery semantics; an unavailable subscriber can miss a published message [2]. These aspects should be evaluated before making quantitative claims about scalability.

5 CONCLUSION

A multi-platform chatbot prototype was implemented using Python, Docker Compose, Redis Pub/Sub, Telegram, and WebSocket. The design separates platform adapters from the Core Bot and uses a shared message format with a callback channel to route responses. Functional verification confirmed the intended message flow for Telegram and WebSocket clients. The approach provides a compact foundation for adding further communication channels while retaining centralized message-processing logic.

STATEMENTS ON COMPLIANCE WITH ETHICAL STANDARDS

Funding Source

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

Disclosure of conflict of interest

The authors declare that they have no competing interests.

REFERENCES

[1] Fette I, Melnikov A. The WebSocket Protocol. RFC 6455. Fremont (CA): RFC Editor; 2011 Dec. Available from: https://www.rfc-editor.org/rfc/rfc6455

[2] Redis. Redis Pub/sub [Internet]. Available from: https://redis.io/docs/latest/develop/pubsub/

[3] Docker Inc. Docker Compose [Internet]. Available from: https://docs.docker.com/compose/

[4] FastAPI. WebSockets [Internet]. Available from: https://fastapi.tiangolo.com/advanced/websockets/

[5] Telegram. Telegram Bot API [Internet]. Available from: https://core.telegram.org/bots/api

[6] Bobrovsky A. Multi-platform chatbot [Internet]. GitHub. Available from: https://github.com/anatoly-bobrovsky/multi-platform-chatbot