

Redis Learning Roadmap & NextGen-FastAPI Implementation Guide

Part 1: Redis Complete Roadmap

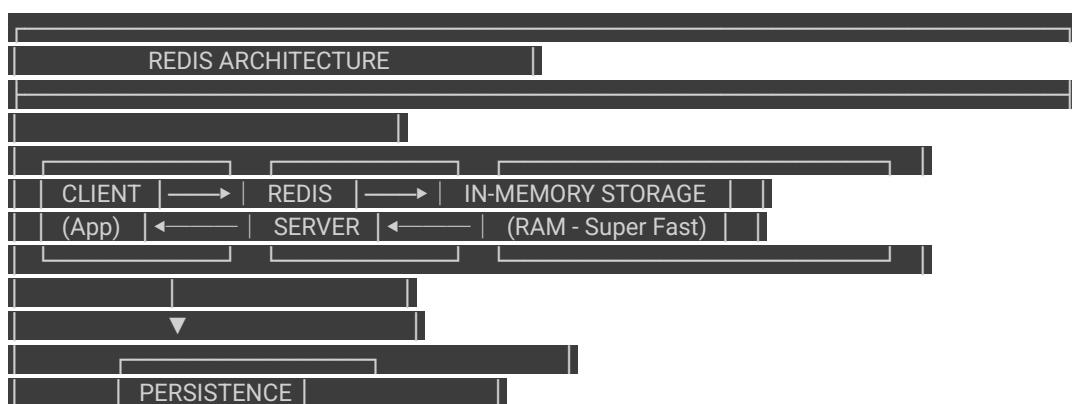
1. What is Redis?

- Remote Dictionary Server
- In-memory data structure store
- Used as database, cache, message broker, and queue
- Extremely fast (100,000+ read/write operations per second)

2. Core Data Types

Type	Description	Use Case
Strings	Basic key-value	Session tokens, counters, JSON cache
Lists	Ordered collections	Message queues, activity feeds
Sets	Unordered unique elements	Tags, unique visitors
Sorted Sets	Sets with scores	Leaderboards, priority queues
Hashes	Field-value pairs	User objects, settings
Streams	Append-only log	Event sourcing, messaging

3. Key Redis Concepts

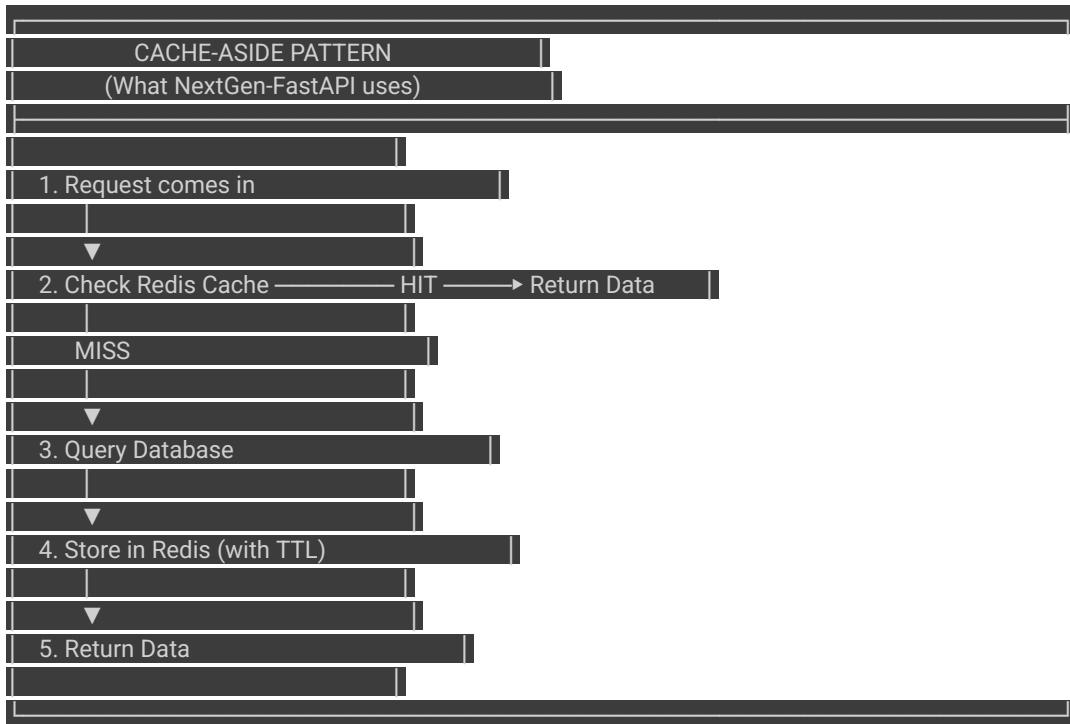




4. TTL (Time To Live)

- Automatic expiration of keys
- Set when creating: `SET key value EX 3600` (expires in 1 hour)
- Check remaining: `TTL key`

5. Caching Strategies



6. Key Naming Conventions

`prefix:entity:identifier`

Examples:

- `masterlookup:all:company_123`
- `dropdown:departments:user_456`
- `call:details:call_789`

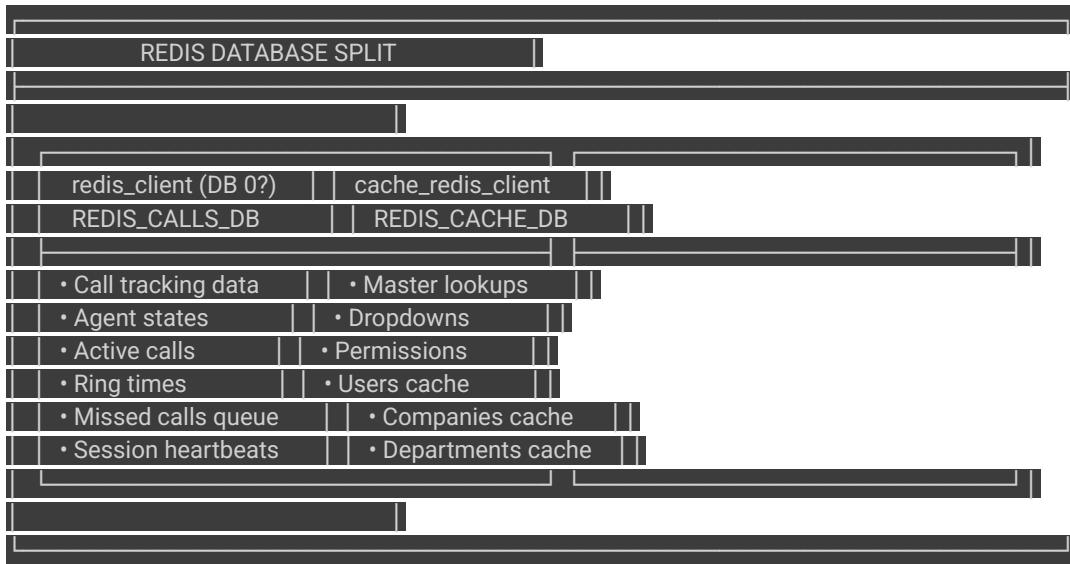
Part 2: Current Understanding (Validated ✓)



Part 3: Call Cache Implementation

Redis Database Separation

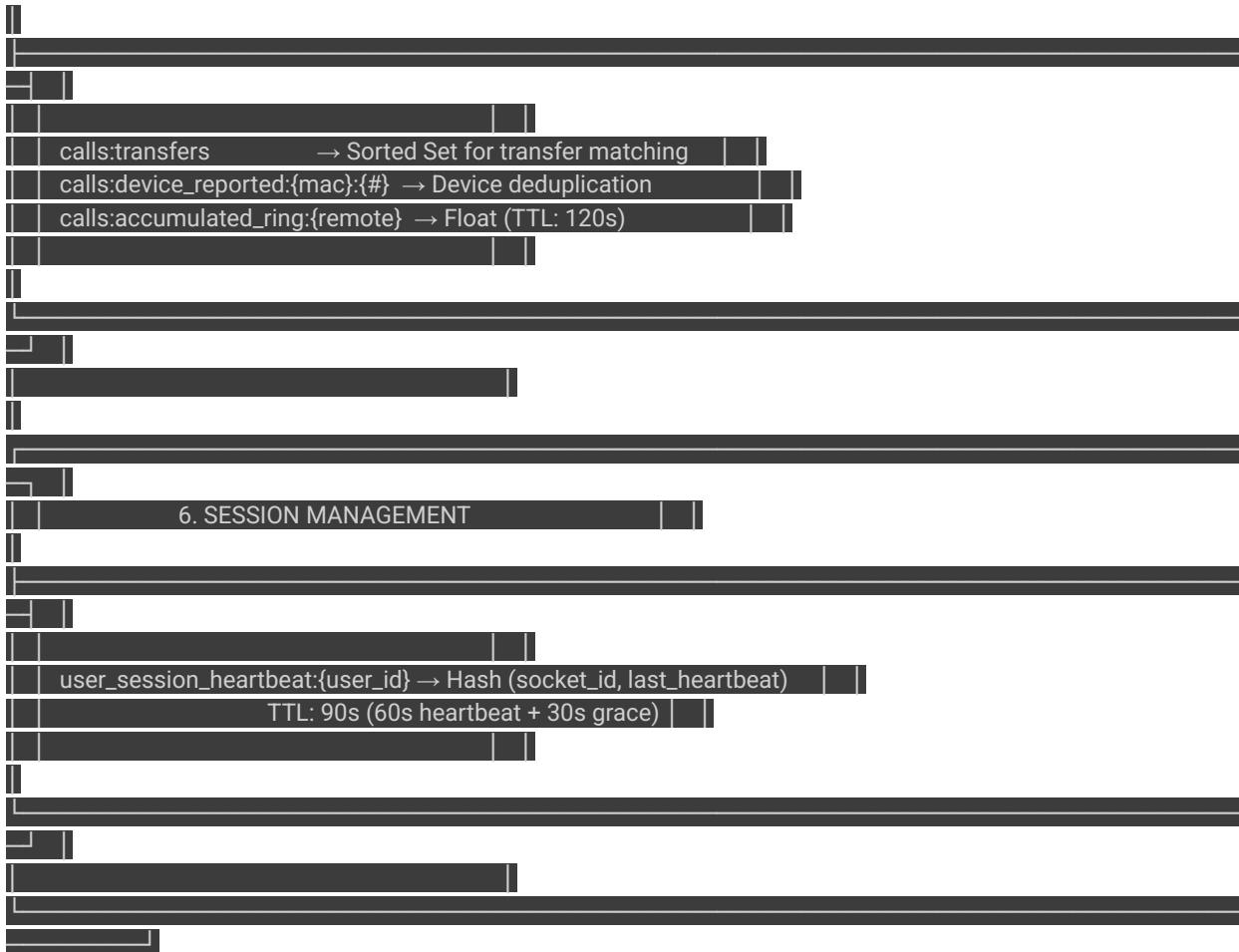
Our application uses two separate Redis databases:



Call Cache Architecture







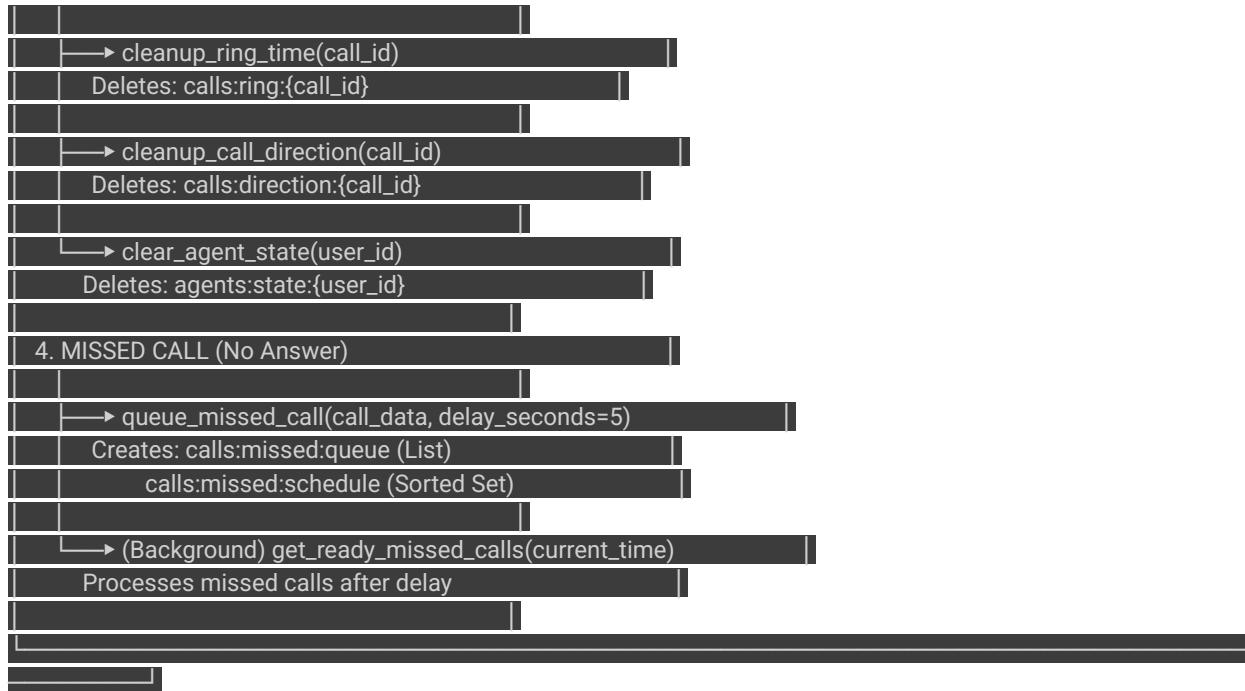
Key Differences: Master Lookup Cache vs Call Cache

Aspect	Master Lookup/Dropdown Cache	Call Cache
Data Type	Static/semi-static data	Real-time dynamic data
TTL	1-24 hours	Seconds to minutes
Update Trigger	Time-based refresh	Event-driven (call events)
Redis DB	<code>cache_redis_client</code> (REDIS_CACHE_DB)	<code>redis_client</code> (REDIS_CALLS_DB)
Data Format	JSON strings	Hashes, Sets, Sorted Sets, Lists

Purpose	Reduce DB queries	Track call state in real-time
---------	-------------------	-------------------------------

Call Lifecycle in Redis





Redis Data Types Used in Call Cache





Key Functions Explained

1. Call Claiming & Deduplication

```
def claim_call(call_id, remote_number, current_time):
```

- ||||
- Prevents duplicate call processing (multi-device support)
- First device to claim wins
- Creates: calls:active:{remote_number}
- ||||

2. Missed Call Queue (Delayed Processing)

```
def queue_missed_call(call_data, delay_seconds=5):
```

- ||||
- Waits 5 seconds before marking as missed
- Why? Call might be answered by another device
- Uses: List (queue) + Sorted Set (schedule)
- ||||

```
def get_ready_missed_calls(current_time):
```

- ||||
- Processes calls whose delay has passed
- Uses pipeline for atomic batch operations
- ||||

3. Transfer Detection

```
def create_transfer_fingerprint(call_id, remote_number, current_time):
    """
    - Creates a fingerprint for transfer matching
    - Stored in Sorted Set with 120 second window
    """

def find_transfer_match(remote_number, current_time):
    """
    - Matches new incoming call to previous transfer
    - Maintains call continuity across transfers
    """
```

4. Session Heartbeat (Online/Offline Detection)

```
def set_session_heartbeat(user_id, socket_id, ttl_seconds=90):
    """
    - Tracks if user is still connected
    - TTL = 60s heartbeat + 30s grace period
    - Auto-expires if no heartbeat received
    """
```

Summary Comparison

Your Understanding	Call Cache Reality
✓ Redis stores call data separately	Uses <code>redis_client</code> (<code>REDIS_CALLS_DB</code>)
✓ Prefixes for keys	<code>calls:</code> , <code>agents:</code> , <code>active_calls:</code> , etc.
✓ TTLs vary by data type	5 min (ring), 1 hr (direction), 90s (heartbeat)
	No warm/refresh - data is event-driven
	Uses Hashes, Sets, Sorted Sets, Lists
	Real-time state tracking, not cache

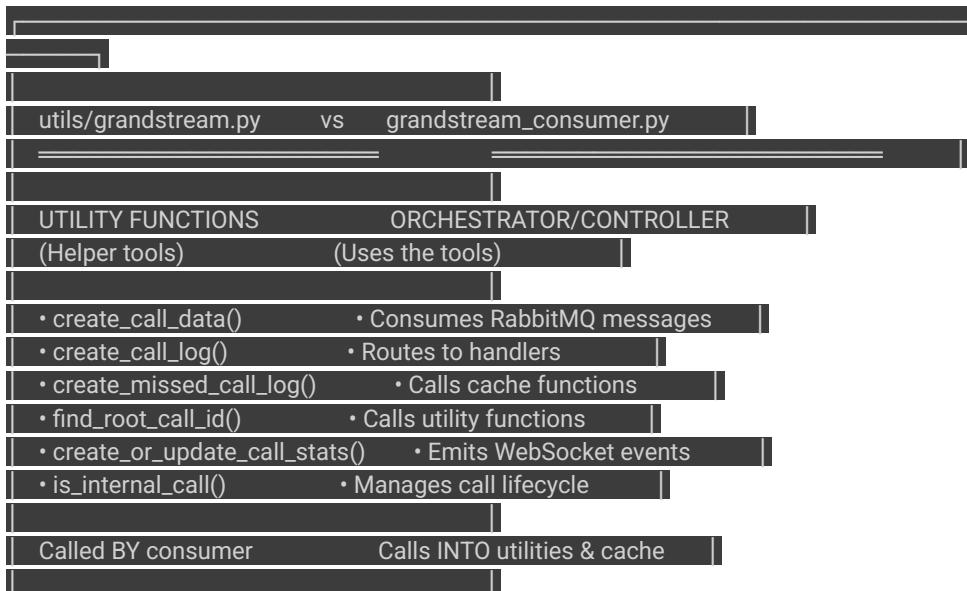
The call cache is fundamentally different from master lookup cache - it's not about caching database queries, but about **tracking real-time call state** across a distributed system (multiple devices, multiple agents).

Complete Call Flow Architecture

Quick Answer: What's the Difference?

YOUR UNDERSTANDING (VALIDATED ✓)

- ✓ Grandstream machine detects call events (incoming, outgoing, missed, etc.)
- ✓ Grandstream machine calls YOUR endpoints (grandstream_routes.py)
- ✓ Routes use helper functions from utils/grandstream.py
- ✓ Helper functions extract data from Request object
- ✓ Data is published to RabbitMQ
- ✓ grandstream_consumer.py runs in background, consumes from RabbitMQ
- ✓ Consumer handlers use: call_cache.py (Redis) + utils/grandstream.py (utilities)
- ✓ Consumer also calls sockets/sockets_core.py for WebSocket events
- ✓ Consumer writes to DB (CallLog, MissedCallLog, CallStat)
- ✓ WebSocket events display on frontend (agent sees popups)



Complete Flow Diagram

COMPLETE CALL FLOW

STEP 1: PHONE TRIGGERS WEBHOOK

```
GRANDSTREAM | Phone detects incoming/outgoing/answered/hangup
PHONE (PBX) |  
| HTTP GET Request
| /gs/cti/incoming?callId=XXX&remote=123&local=203&mac=AA:BB:CC
▼
```

routes/grandstream_routes.py

```
Endpoints:  
• /gs/cti/incoming → @router.get  
• /gs/cti/outgoing → @router.get  
• /gs/cti/answered → @router.get  
• /gs/cti/hangup → @router.get  
• /gs/cti/rejected → @router.get  
• /gs/cti/missed → @router.get
```

What it does:

1. call_data = create_call_data(request) ←— utils/grandstream.py
2. publish_event("incoming", call_data) ←— messaging/rabbitmq.py
3. return {"status": "ok"}

Publishes to RabbitMQ

STEP 2: MESSAGE QUEUE (DECOUPLING)

```
messaging/rabbitmq.py
```

publish_event(event_type, data):
 • Exchange: "grandstream"
 • Routing Key: "call.incoming" / "call.answered" / etc.
 • Queue: "grandstream-queue"

Consumer picks up message

▼

STEP 3: CONSUMER PROCESSES EVENT

```
messaging/consumers/grandstream_consumer.py
```

start_grandstream_consumer():
 • Runs in background thread
 • Consumes from grandstream-queue
 • Routes based on call["status"]:

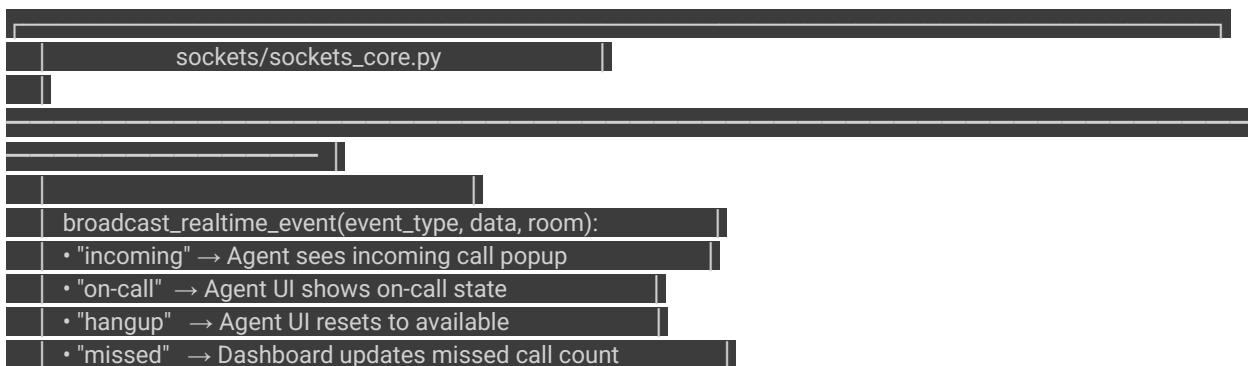
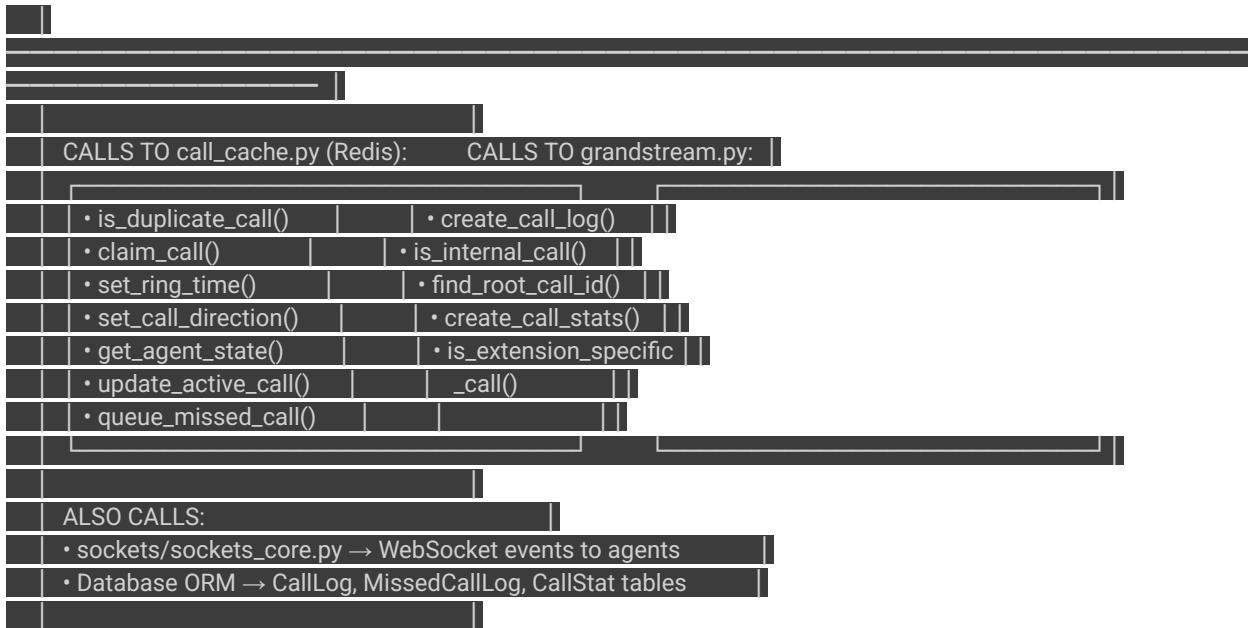
status="incoming" → handle_incoming_call(call, db)
 status="outgoing" → handle_outgoing_call(call, db)
 status="answered" → handle_answered_call(call, db)
 status="hangup" → handle_hangup_call(call, db)
 status="rejected" → handle_rejected_call(call, db)
 status="missed" → handle_missed_call(call, db)

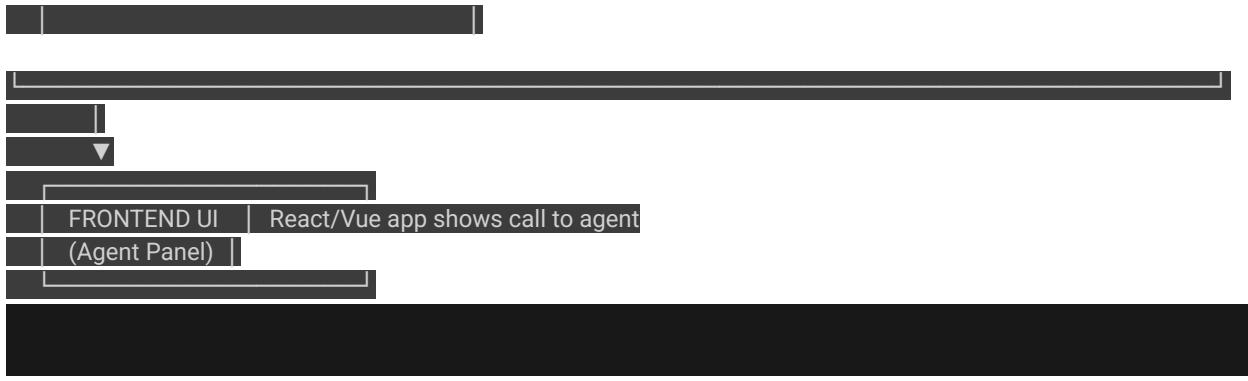
Handler function executes

▼

STEP 4: HANDLER USES CACHE + UTILITIES

```
HANDLER FUNCTION (e.g., handle_incoming_call)
```





Incoming Call Example - Step by Step

1) PHONE RINGS

```

Grandstream phone sends:
GET /gs/cti/incoming?callId=12345&remote=03001234567&local=203&mac=AA:BB:CC

```

2) grandstream_routes.py :: incoming()

```

call_data = {
    "call_id": "12345",
    "remote": "03001234567",
    "local": "203",
    "mac": "AA:BB:CC",
    "status": "incoming",
    "timestamp": 1703954400.123
}
publish_event("incoming", call_data) —► RabbitMQ

```

3) grandstream_consumer.py :: handle_incoming_call()

STEP 3.1: Check for duplicates (multi-device)

```

call_cache.is_duplicate_call(remote, mac)

```

```

Check: calls:active:03001234567 exists?

```

```

Check: calls:device_reported:AA:BB:CC:03001234567 exists?

```

```

If YES → return (ignore duplicate)

```

STEP 3.2: Check for transfer match

```
call_cache.find_transfer_match(remote, timestamp)
```

- └ Looks in calls:transfers sorted set (120s window)
- └ If found → link via calls:relationship:{call_id}_parent

STEP 3.3: Claim the call (first device wins)

```
call_cache.claim_call(call_id, remote, timestamp)
```

└─ Creates: calls:active:03001234567 = {

claimed_call_id: "12345",

timestamp: "1703954400.123",

answered: "false"

}

```
call_cache.mark_device_reported_call(mac, remote)
```

└ Creates: calls:device_reported:AA:BB:CC:03001234567 = "1" |

STEP 3.4: Store ring time

```
call_cache.set_ring_time(call_id, timestamp)
```

↳ Creates: calls:ring:12345 = "1703954400.123" (TTL: 5min)

call cache set ring time for remote(remote timestamp)

↳ Creates: calls:ring remote:03001234567 = "..." (TTI : 5min)

cell-to-cell contact, cell division (cell division)

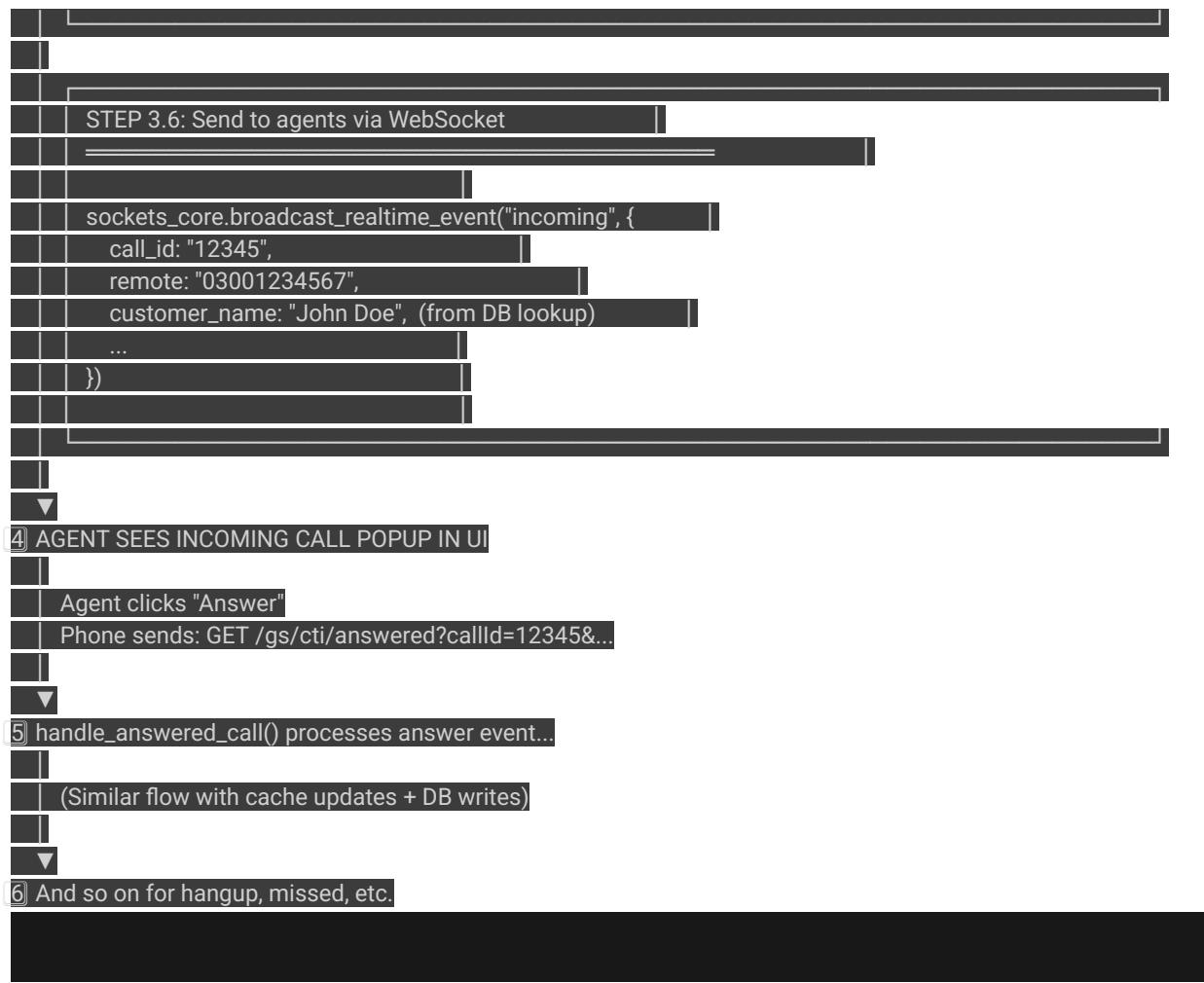
Creates: call:direction:12345 = "in" (TTL: 1hr)

STEP 3.5: Determine routing

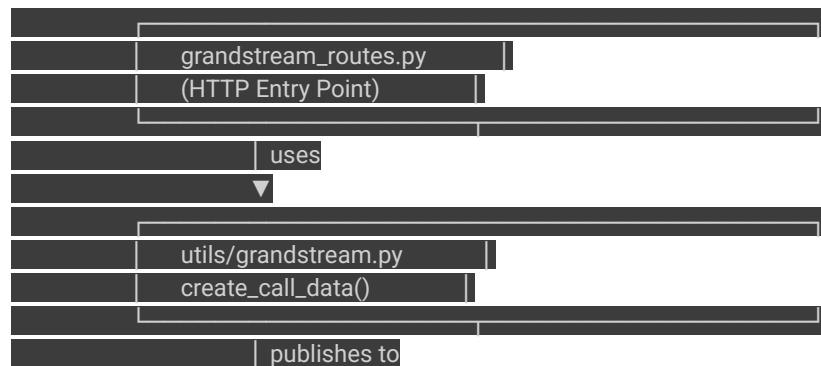
grandstream.is_extension_specific_call(db, local="203")

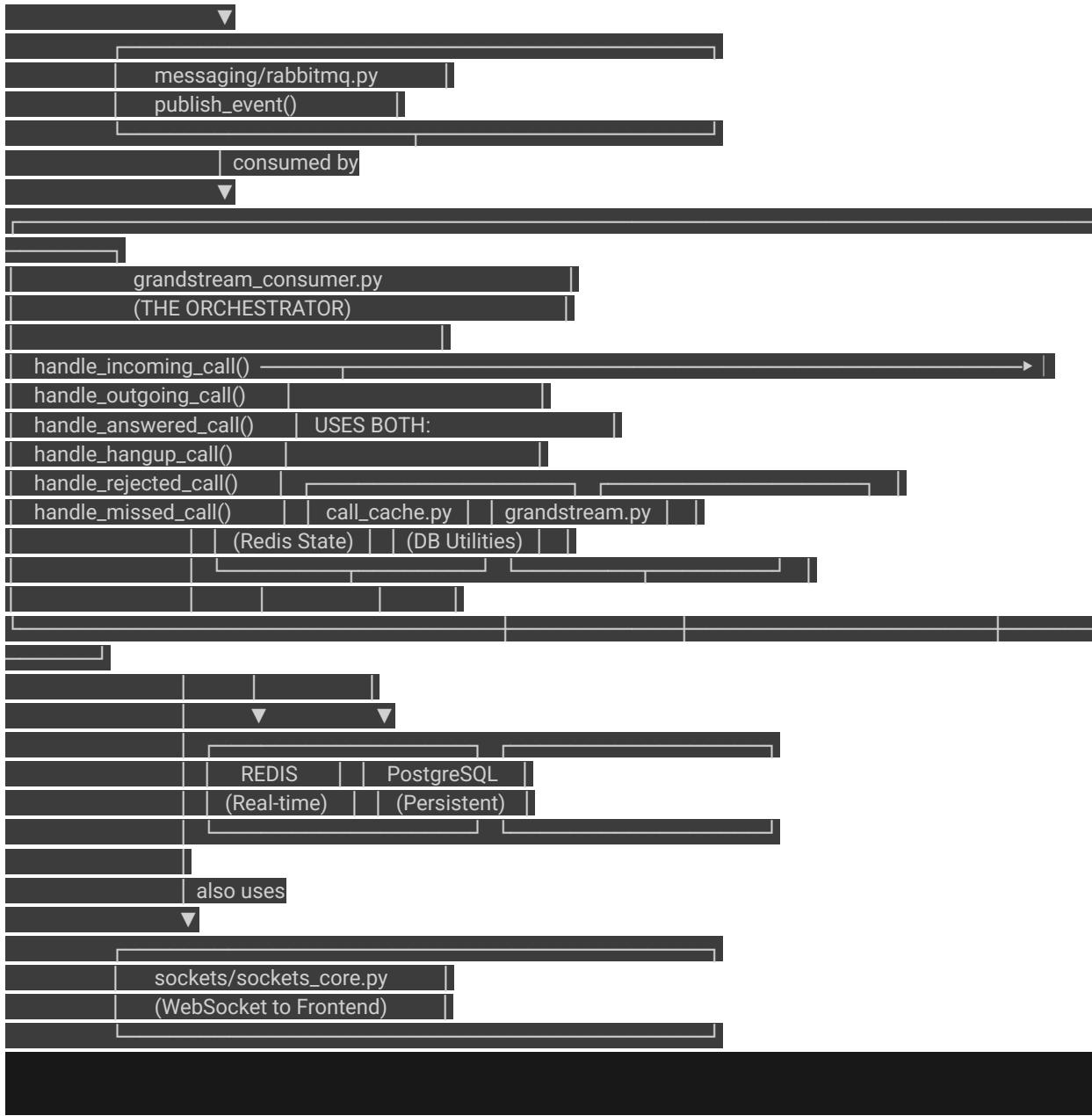
|— If direct extension → route to specific agent

└ If helpline number → broadcast to all available agents



File Relationship Summary



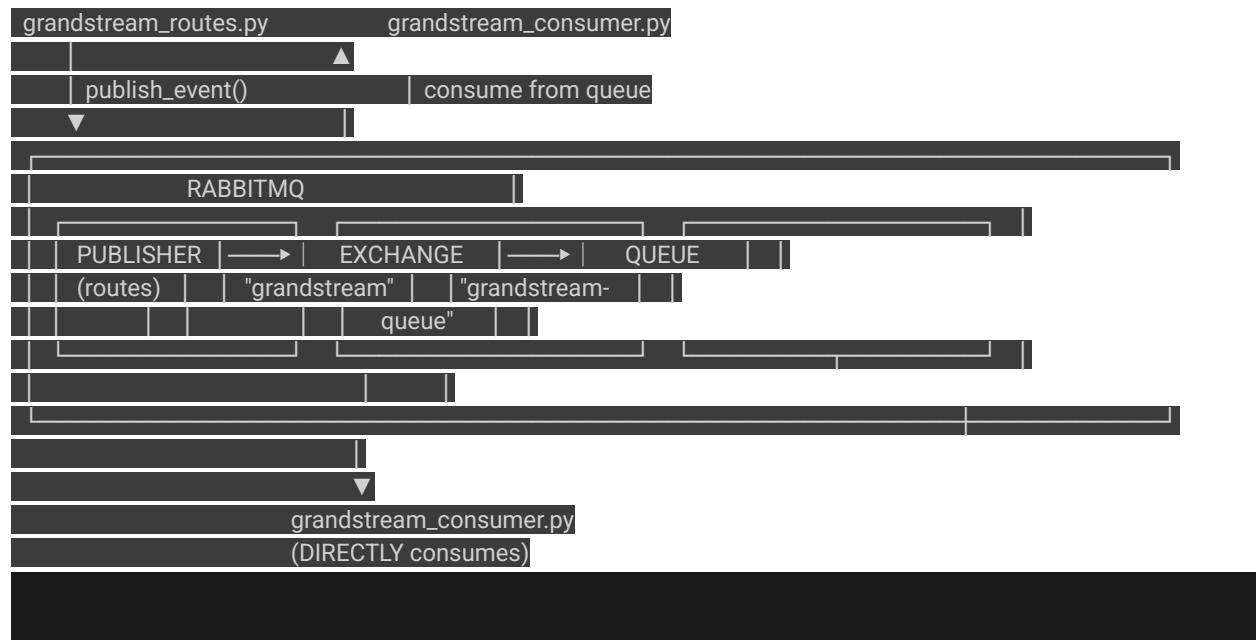


Key Insight: Why Two Separate Files?

File	Purpose	Design Pattern
utils/grandstream.py	Pure utility functions (no side effects on Redis)	Helper/Utility - Reusable across codebase
grandstream_consumer.py	Orchestrates entire call lifecycle	Controller - Coordinates all components
call_cache.py	Redis state management	Repository - Data access layer

The consumer imports and uses both `call_cache.py` and `utils/grandstream.py` to do its job. Think of it like a chef (consumer) using tools (utils) and a refrigerator (cache) to prepare a meal (handle the call).

CORRECT FLOW:



The consumer connects DIRECTLY to RabbitMQ queue - it does NOT go through the routes again.

Your Corrected Complete Flow

- ① GRANDSTREAM MACHINE (PBX/Phone)
 - | Detects: incoming call, outgoing call, answered, hangup, missed
 - | Sends HTTP webhook to YOUR server
- ② `grandstream_routes.py (YOUR ENDPOINTS)`
 - | • /gs/cti/incoming
 - | • /gs/cti/outgoing
 - | • /gs/cti/answered

```
|   • /gs/cti/hangup  
|   • /gs/cti/missed
```

```
|  
|   Uses: utils/grandstream.py → create_call_data(request)  
|   Extracts: call_id, remote, local, mac, timestamp from Request
```

3] messaging/rabbitmq.py :: publish_event()

```
|  
|   Publishes call_data JSON to RabbitMQ  
|   Exchange: "grandstream"  
|   Routing Key: "call.incoming" / "call.answered" / etc.
```

4] RABBITMQ (Message Broker)

```
|  
|   Stores message in "grandstream-queue"  
|   Waits for consumer to pick it up
```

★ WHY RABBITMQ?

- Decouples routes from processing (async)
- Routes return immediately (fast response to Grandstream)
- Processing happens in background (doesn't block)
- If consumer crashes, messages stay in queue (reliability)

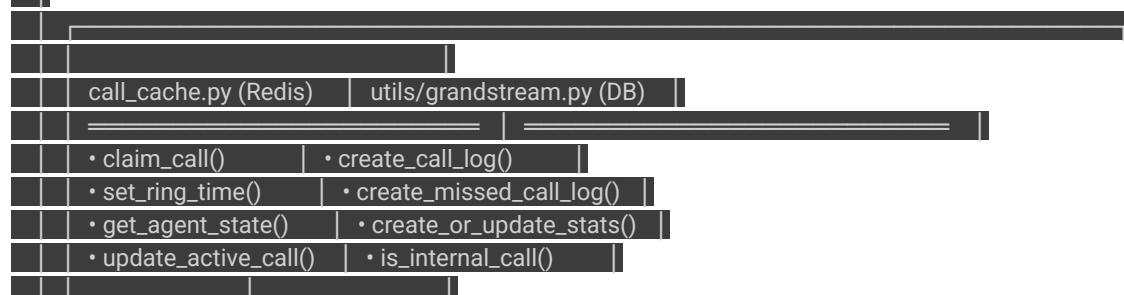
5] grandstream_consumer.py (BACKGROUND PROCESS)

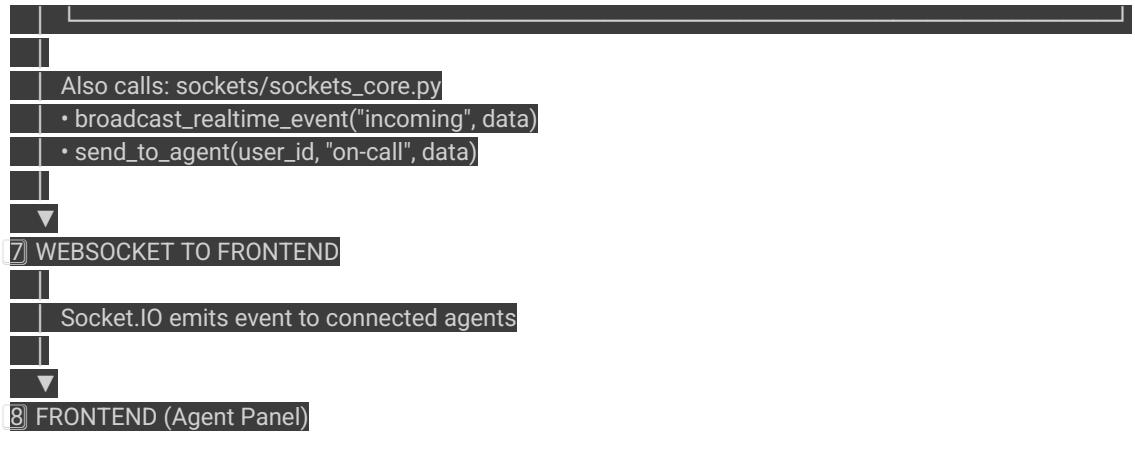
```
|  
|   Runs in separate thread on app startup  
|   DIRECTLY connects to RabbitMQ (NOT through routes!)  
|   Consumes messages one by one
```

```
|  
|   Routes to handler based on status:  
|   • "incoming" → handle_incoming_call()  
|   • "outgoing" → handle_outgoing_call()  
|   • "answered" → handle_answered_call()  
|   • etc.
```

6] HANDLER FUNCTIONS (inside consumer)

```
|  
|   Uses THREE things:
```

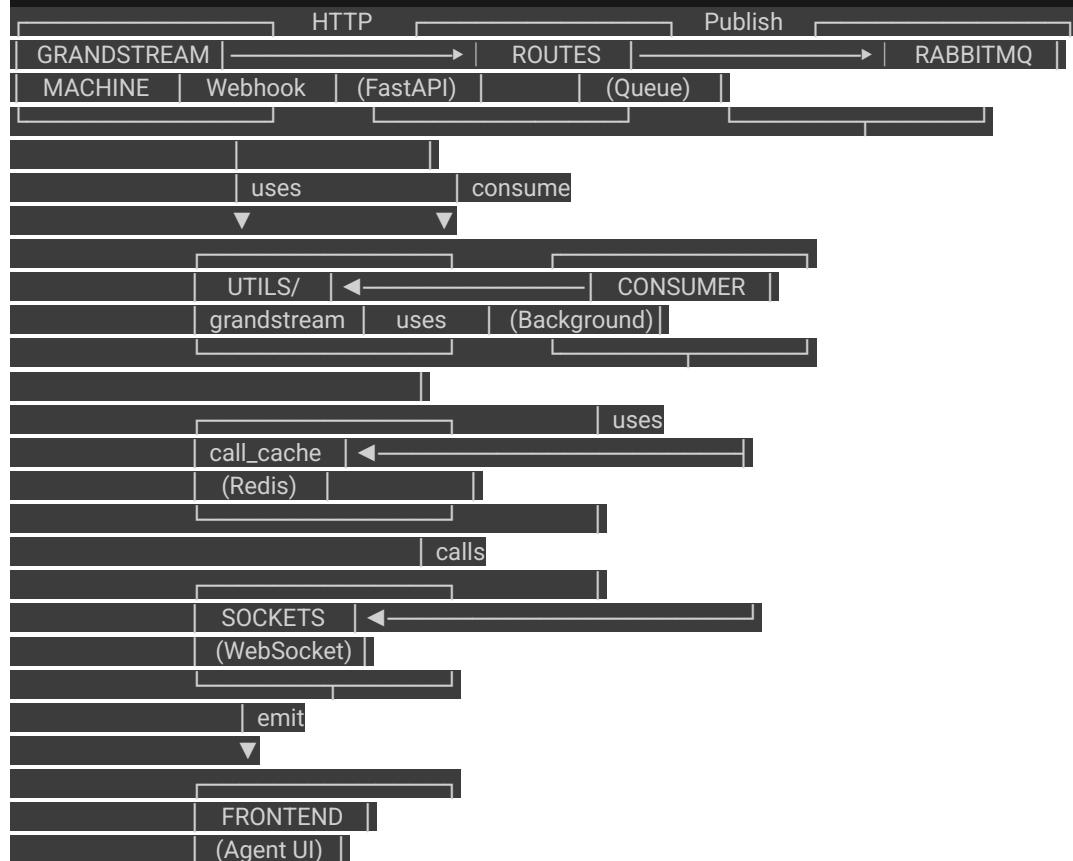




Agent sees:

- Incoming call popup
- On-call status
- Hangup notification
- Missed call alert

Visual Summary





Key Point You Should Remember

Component	Role	Connection
<code>grandstream_routes.py</code>	ENTRY POINT (receives webhooks)	HTTP from Grandstream machine
<code>rabbitmq.py</code>	MESSAGE BROKER (decoupling)	Routes publish, Consumer consumes
<code>grandstream_consumer.py</code>	PROCESSOR (background)	Directly connects to RabbitMQ queue
<code>call_cache.py</code>	STATE MANAGER (Redis)	Called by consumer handlers
<code>utils/grandstream.py</code>	DB UTILITIES	Called by routes AND consumer
<code>sockets_core.py</code>	REAL-TIME PUSH	Called by consumer to notify frontend

Your overall understanding is excellent! The only correction is that RabbitMQ does NOT route back through the endpoints - the consumer connects directly to the queue.