

# Redis Learning Roadmap & NextGen-FastAPI Implementation Guide

## Part 1: Redis Complete Roadmap

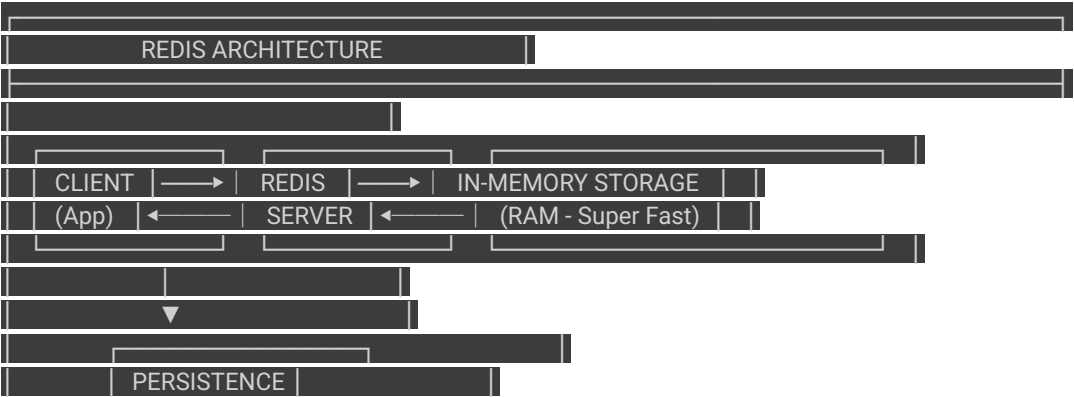
### 1. What is Redis?

- 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

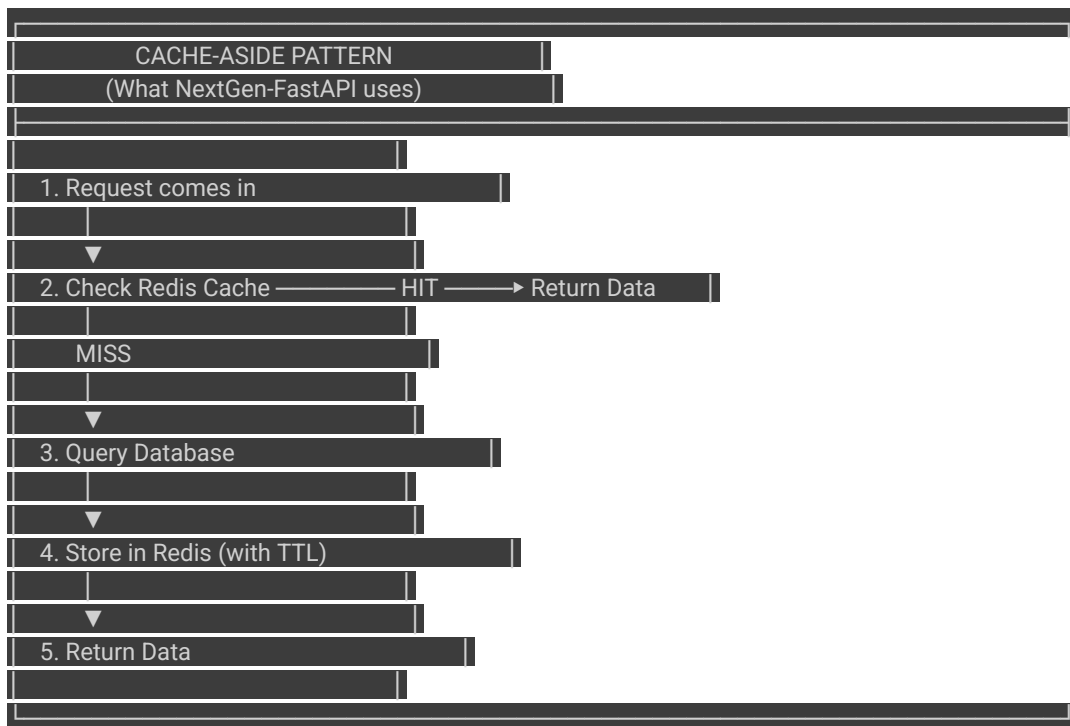




## 4. TTL (Time To Live)

- 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 ✓)



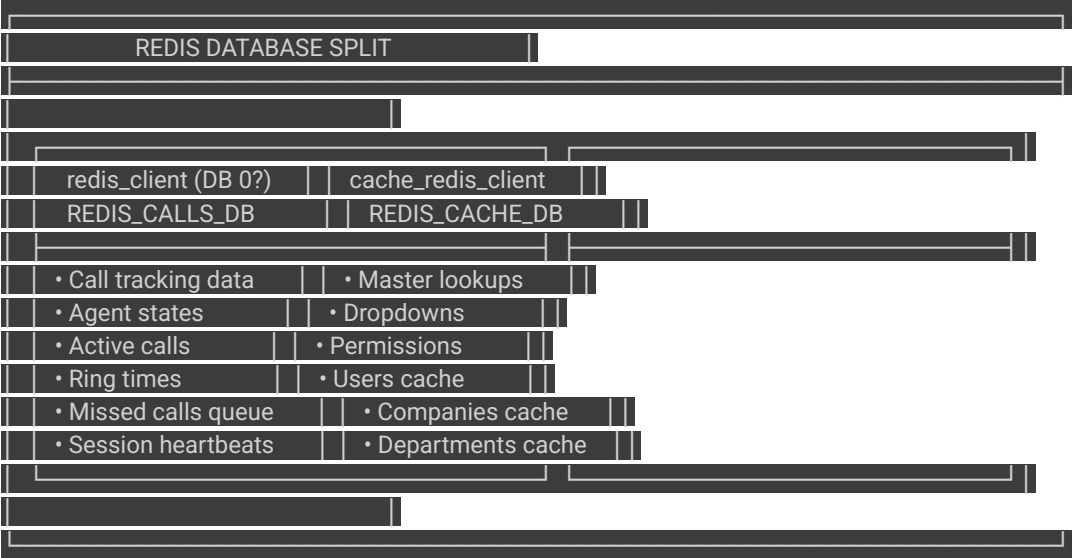
---

## Part 3: Call Cache Implementation

---

### Redis Database Separation

Our application uses **two separate Redis databases**:



Call Cache Architecture



```
graph TD
    subgraph 1_AGENTS [1. AGENTS]
        A1[agents:state:{user_id} → Hash with status, call details]
        A2[agents:available → Set of available user IDs]
        A3[agents:calls → Hash (user_id → remote_number)]
        A4[agents:metadata:{user_id} → Hash with name, extension]
        A5[agent:customer_call:{user_id} → Hash for consultation tracking]
    end

    subgraph 2_CALLS [2. CALLS]
        C1[calls:calls:{call_id} → Hash with call state (TTL: 1 hr)]
        C2[calls:active_calls:{user_id} → Set of call IDs (TTL: 1 hr)]
        C3[active_calls:{call_id} → Hash with full call data]
        C4[agent_active_call:{agent_id} → String call_id (lookup)]
    end

    subgraph 3_MULTI_LINE_SUPPORT [3. MULTI-LINE SUPPORT]
        C1
        C2
        C3
        C4
    end

    subgraph 4_MISSED_CALLS_QUEUE [4. MISSED CALLS QUEUE]
        M1[calls:missed:queue → List of JSON call data]
        M2[calls:missed:schedule → Sorted Set (call_id → process_time)]
    end

    subgraph 5_TRANSFER_DEVICE_TRACKING [5. TRANSFER & DEVICE TRACKING]
        T1[calls:transfer:{call_id} → Hash with transfer details]
        T2[calls:device_tracking:{call_id} → Hash with device tracking details]
    end
```

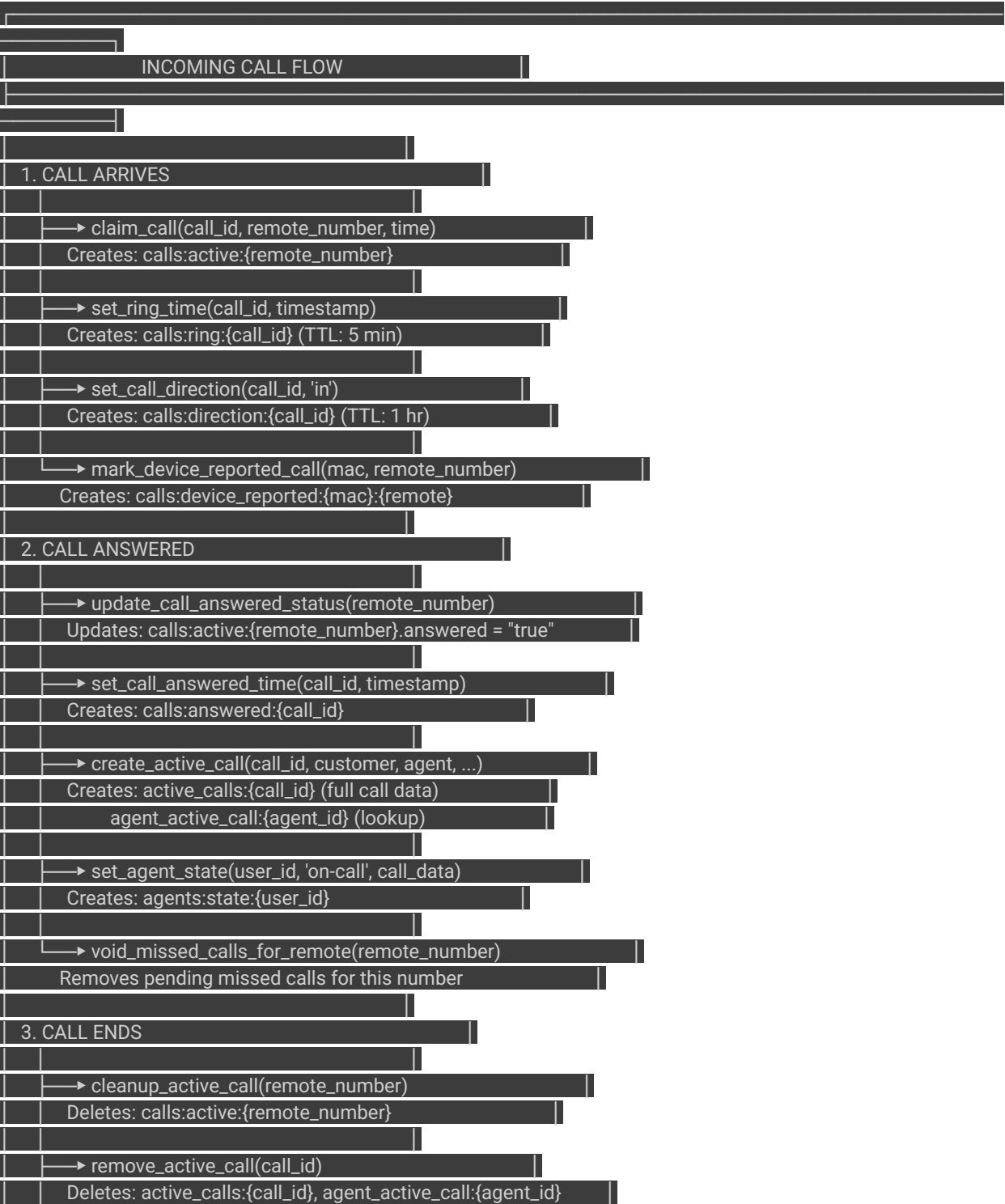
The diagram illustrates the database structure for a call center system, organized into five main sections:

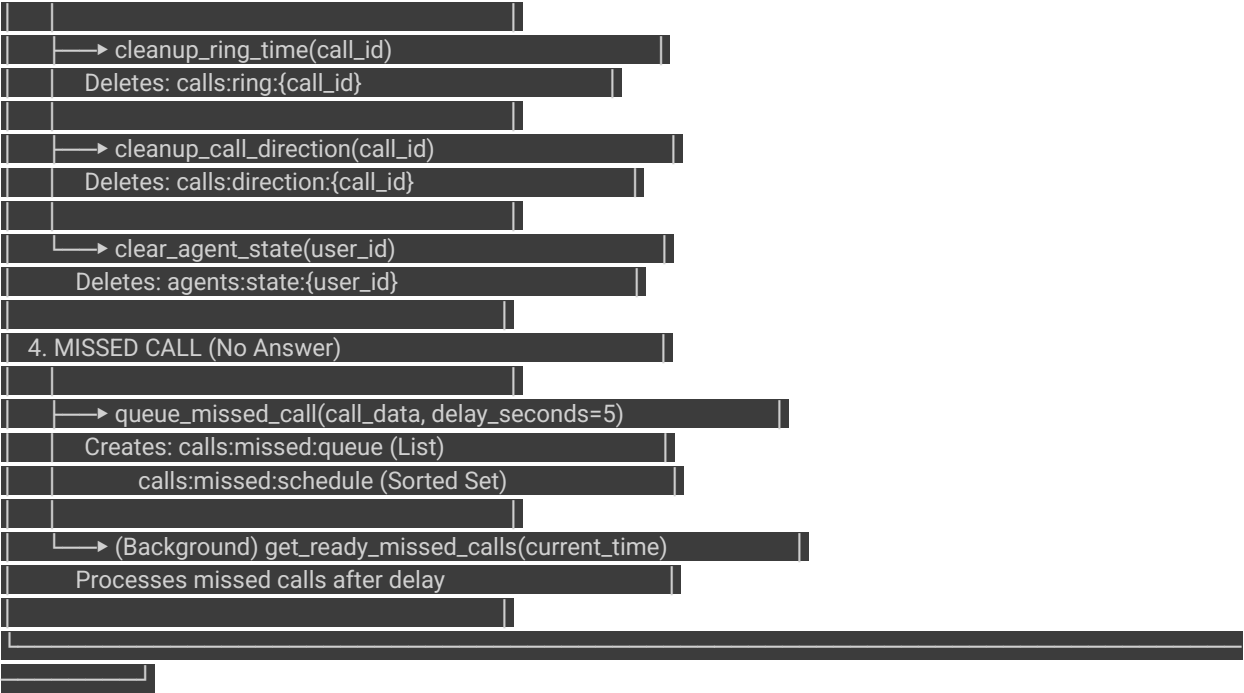
- 1. AGENTS**
  - `agents:state:{user_id}` → Hash with status, call details
  - `agents:available` → Set of available user IDs
  - `agents:calls` → Hash (`user_id` → `remote_number`)
  - `agents:metadata:{user_id}` → Hash with name, extension
  - `agent:customer_call:{user_id}` → Hash for consultation tracking
- 2. CALLS**
  - `calls:calls:{call_id}` → Hash with call state (TTL: 1 hr)
  - `calls:active_calls:{user_id}` → Set of call IDs (TTL: 1 hr)
  - `active_calls:{call_id}` → Hash with full call data
  - `agent_active_call:{agent_id}` → String `call_id` (lookup)
- 3. MULTI-LINE SUPPORT**
  - `calls:calls:{call_id}` → Hash with call state (TTL: 1 hr)
  - `calls:active_calls:{user_id}` → Set of call IDs (TTL: 1 hr)
  - `active_calls:{call_id}` → Hash with full call data
  - `agent_active_call:{agent_id}` → String `call_id` (lookup)
- 4. MISSED CALLS QUEUE**
  - `calls:missed:queue` → List of JSON call data
  - `calls:missed:schedule` → Sorted Set (`call_id` → `process_time`)
- 5. TRANSFER & DEVICE TRACKING**
  - `calls:transfer:{call_id}` → Hash with transfer details
  - `calls:device_tracking:{call_id}` → Hash with device tracking details



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

Call Lifecycle in Redis





Redis Data Types Used in Call Cache

REDIS DATA TYPES IN USE	
STRING   Simple key-value	
• Ring times	calls:ring:{call_id} → "1703954400.123"
• Direction	calls:direction:{call_id} → "in"
• Lookups	agent_active_call:{agent_id} → "call_123"
HASH   Multiple fields in one key	
• Active call	calls:active:{remote} → {call_id, timestamp, answered}
• Agent state	agents:state:{user_id} → {status, call_id, customer...}
• Call data	active_calls:{call_id} → {full call details}
• Assignments	agents:calls → {user_id: remote_number, ...}
SET   Unordered unique elements	

• Available	agents:available → {user_id1, user_id2, ...}
• Active IDs	agents:active_calls:{user_id} → {call_id1, call_id2}
SORTED SET	Ordered by score (timestamp)
• Transfers	calls:transfers → {json_data: timestamp, ...}
• Missed	calls:missed:schedule → {call_id: process_time, ...}
LIST	Ordered collection (queue)
• Missed Q	calls:missed:queue → [json1, json2, ...]

## 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> (REDIS_CALLS_DB)
✓ 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)
	<b>No warm/refresh</b> - 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)

utils/grandstream.py vs grandstream\_consumer.py

UTILITY FUNCTIONS (Helper tools)	ORCHESTRATOR/CONTROLLER (Uses the tools)
-------------------------------------	---

• create_call_data()	• Consumes RabbitMQ messages
• create_call_log()	• Routes to handlers
• create_missed_call_log()	• Calls cache functions
• find_root_call_id()	• Calls utility functions
• create_or_update_call_stats()	• Emits WebSocket events
• is_internal_call()	• Manages call lifecycle

Called BY consumer	Calls INTO utilities & cache
--------------------	------------------------------

## 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)





▼

FRONTEND UI | React/Vue app shows call to agent  
(Agent Panel) |

## Incoming Call Example - Step by Step

INCOMING CALL - DETAILED FLOW

### 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()

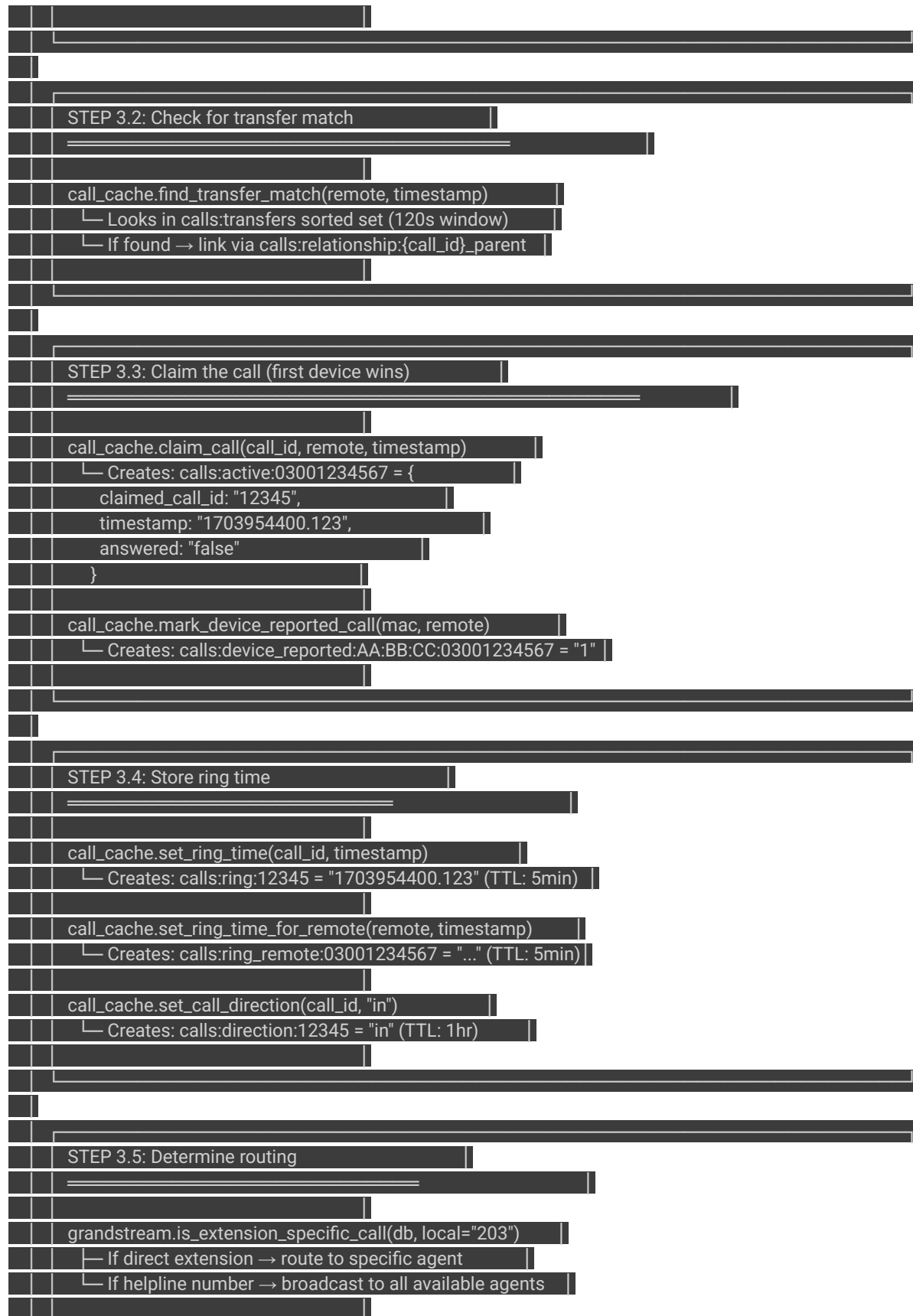
STEP 3.1: Check for duplicates (multi-device)

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 3.6: Send to agents via WebSocket
|
|
|
| sockets_core.broadcast_realtime_event("incoming", {
|   call_id: "12345",
|   remote: "03001234567",
|   customer_name: "John Doe", (from DB lookup)
|   ...
| })
|
|
|
```



4 AGENT SEES INCOMING CALL POPUP IN UI



Agent clicks "Answer"

Phone sends: GET /gs/cti/answered?callId=12345&...



5 handle\_answered\_call() processes answer event...



(Similar flow with cache updates + DB writes)



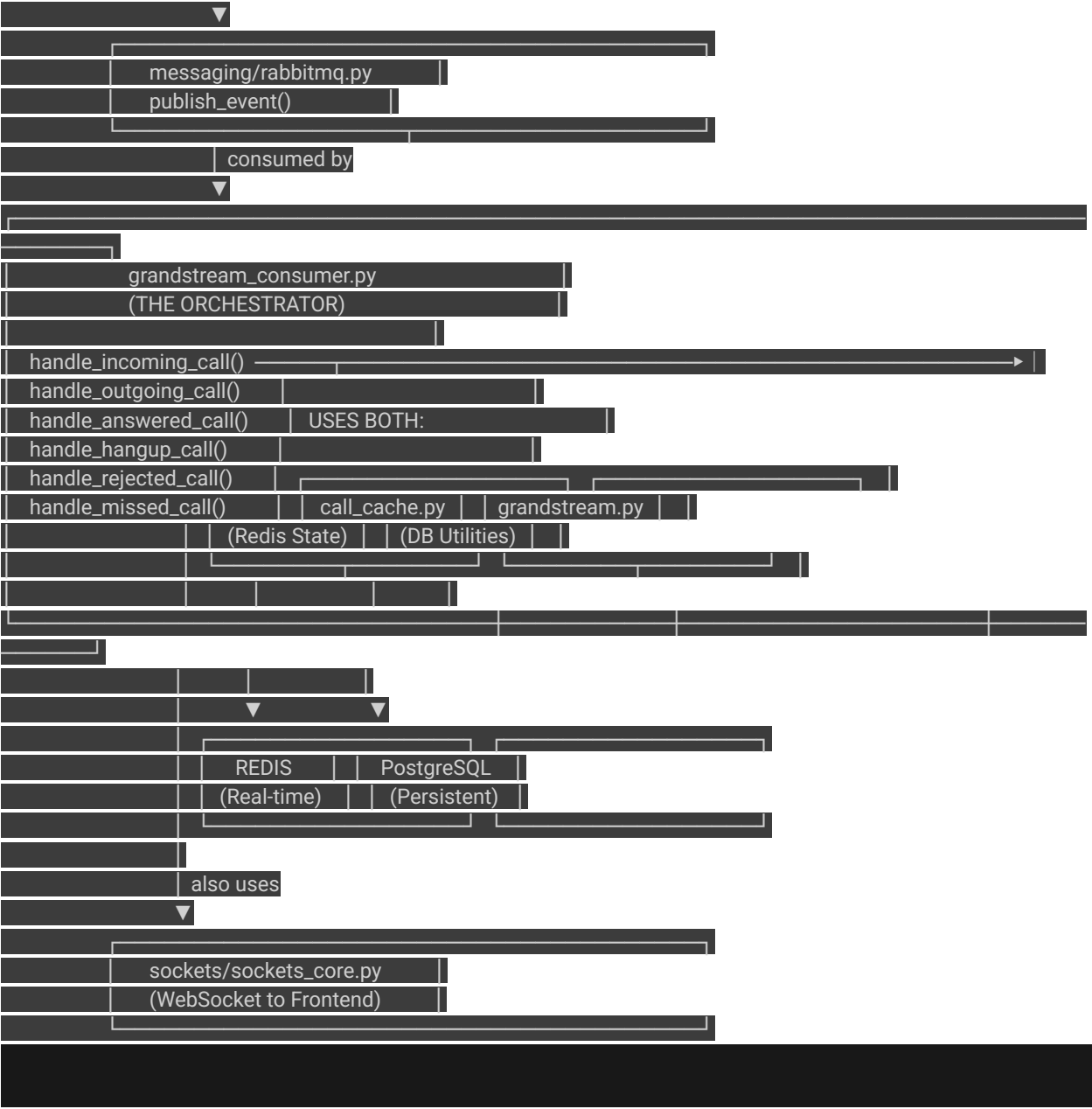
6 And so on for hangup, missed, etc.



## File Relationship Summary

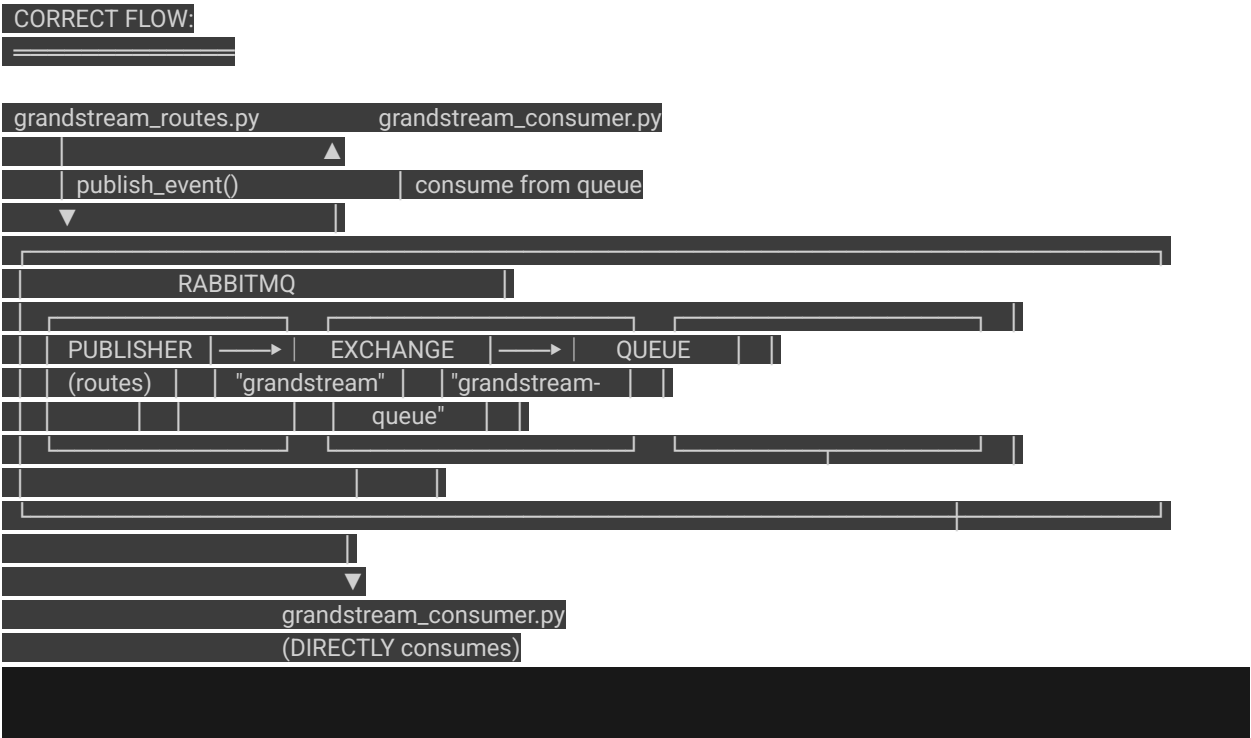
FILE RELATIONSHIPS	

	grandstream_routes.py (HTTP Entry Point)
	uses
	utils/grandstream.py create_call_data()
	publishes to



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).



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()`

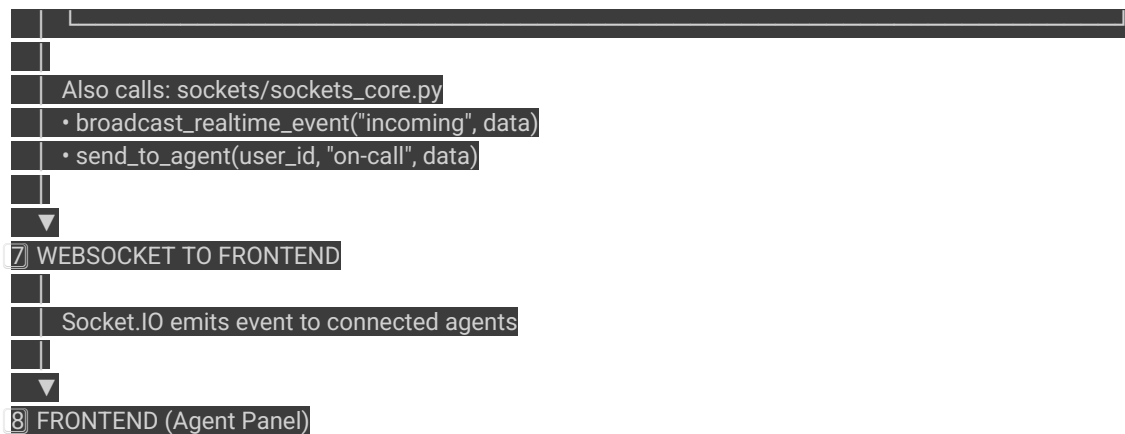
- etc.



### 6) HANDLER FUNCTIONS (inside consumer)

Uses THREE things:

<code>call_cache.py</code> (Redis)		<code>utils/grandstream.py</code> (DB)	
<code>claim_call()</code>		<code>create_call_log()</code>	
<code>set_ring_time()</code>		<code>create_missed_call_log()</code>	
<code>get_agent_state()</code>		<code>create_or_update_stats()</code>	
<code>update_active_call()</code>		<code>is_internal_call()</code>	



Agent sees:

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



## 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.