Redis Learning Roadmap & NextGen-FastAPI Implementation Guide

---

Part 1: Redis Complete Roadmap

1. What is Redis?

- **R**emote **Di**ctionary **S**erver
- 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

```
┌─────────────────────────────────────────────────────────┐
│              REDIS ARCHITECTURE                         │
├─────────────────────────────────────────────────────────┤
│                                                         │
│  ┌──────────┐    ┌──────────┐    ┌────────────────────┐ │
│  │ CLIENT   │───▶│  REDIS   │───▶│ IN-MEMORY STORAGE  │ │
│  │ (App)    │◀───│  SERVER  │◀───│ (RAM - Super Fast) │ │
│  └──────────┘    └──────────┘    └────────────────────┘ │
│                       │                                 │
│                       ▼                                 │
│                ┌─────────────┐                          │
│                │ PERSISTENCE │                          │
│         │  (RDB/AOF)  │                 │
│         └─────────────┘                 │
└─────────────────────────────────────────┘
```

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

```
┌─────────────────────────────────────────┐
│           CACHE-ASIDE PATTERN           │
│       (What NextGen-FastAPI uses)       │
├─────────────────────────────────────────┤
│                                         │
│  1. Request comes in                    │
│      │                                  │
│      ▼                                  │
│  2. Check Redis Cache ────── HIT ─────► Return Data │
│      │                                  │
│     MISS                                │
│      │                                  │
│      ▼                                  │
│  3. Query Database                      │
│      │                                  │
│      ▼                                  │
│  4. Store in Redis (with TTL)           │
│      │                                  │
│      ▼                                  │
│  5. Return Data                         │
│                                         │
└─────────────────────────────────────────┘
```

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 ✓)

```
┌─────────────────────────────────────────────────────────────┐
│            NEXTGEN-FASTAPI REDIS STRUCTURE                    │
├─────────────────────────────────────────────────────────────┤
│                                                               │
│  ┌──────────────┐      ┌──────────────────────────┐           │
│  │ redis_client │─────▶│ Configuration & Connection│          │
│  │              │      │ - Host, Port, Password   │           │
│  │              │      │ - Prefixes & TTLs        │           │
│  └──────────────┘      └──────────────────────────┘           │
│                                                               │
│  ┌─────────────────────────────────────────────┐              │
│  │              DATA PARTITIONS                │              │
│  ├──────────────────────┬──────────────────────┤              │
│  │  CALL DATA           │  OVERALL DATA        │              │
│  │  - Call details      │  - Master lookups    │              │
│  │  - Call history      │  - Dropdowns         │              │
│  │                      │  - Permissions       │              │
│  │                      │  - Users, Companies, Depts │        │
│  └──────────────────────┴──────────────────────┘              │
│                                                               │
│  ┌─────────────────────────────────────────────┐              │
│  │              DATA FRESHNESS                 │              │
│  ├──────────────┬──────────────┬───────────────┤              │
│  │  STATIC      │ SEMI-STATIC  │  DYNAMIC      │              │
│  │  24 hrs      │ 6 hrs        │  1 hr         │              │
│  │ (rarely      │ (occasionally│ (frequently   │              │
│  │  changes)    │  changes)    │  changes)     │              │
│  └──────────────┴──────────────┴───────────────┘              │
│                                                               │
│  ┌──────────────┐      ┌──────────────────────────┐           │
│  │ cache_warmer │─────▶│ Loads data on app startup │          │
│  └──────────────┘      └──────────────────────────┘           │
│                                                               │
│  ┌──────────────┐      ┌──────────────────────────┐           │
│  │cache_refresher│────▶│ Refreshes data after TTL │           │
│  └──────────────┘      └──────────────────────────┘           │
│                                                               │
└─────────────────────────────────────────────────────────────┘
```

Part 3: Call Cache Implementation

Redis Database Separation

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

```
┌──────────────────────────────────────────────────────────────┐
│                 REDIS DATABASE SPLIT                          │
├──────────────────────────────────────────────────────────────┤
│                                                              │
│  ┌──────────────────────────┐  ┌──────────────────────────┐  │
│  │  redis_client (DB 0?)    │  │  cache_redis_client      │  │
│  │  REDIS_CALLS_DB          │  │  REDIS_CACHE_DB          │  │
│  ├──────────────────────────┤  ├──────────────────────────┤  │
│  │ • Call tracking data     │  │ • Master lookups         │  │
│  │ • Agent states           │  │ • Dropdowns              │  │
│  │ • Active calls           │  │ • Permissions            │  │
│  │ • Ring times             │  │ • Users cache            │  │
│  │ • Missed calls queue     │  │ • Companies cache        │  │
│  │ • Session heartbeats     │  │ • Departments cache      │  │
│  └──────────────────────────┘  └──────────────────────────┘  │
│                                                              │
└──────────────────────────────────────────────────────────────┘
```

Call Cache Architecture

```
┌──────────────────────────────────────────────────────────────────────────┐
│                    CALL CACHE STRUCTURE                                  │
├──────────────────────────────────────────────────────────────────────────┤
│                                                                          │
│ ┌──────────────────────────────────────────────────────────────────────┐ │
│ │                    1. CALL TRACKING                                  │ │
│ ├──────────────────────────────────────────────────────────────────────┤ │
│ │                                                                      │ │
│ │  calls:active:{remote_number}    → Hash with call_id, timestamp      │ │
│ │  calls:ring:{call_id}            → Float timestamp (TTL: 5 min)      │ │
│ │  calls:ring_remote:{remote}      → Float timestamp (TTL: 5 min)      │ │
│ │  calls:direction:{call_id}       → String 'in' or 'out' (TTL: 1 hr)  │ │
│ │  calls:answered:{call_id}        → Float timestamp                   │ │
│ │  calls:relationship:{key}        → Call ID mapping                   │ │
│ │  calls:assignments               → Hash (call_id → user_id)          │ │
│ │                                                                      │ │
│ └──────────────────────────────────────────────────────────────────────┘ │
│                                                                          │
│ ┌──────────────────────────────────────────────────────────────────────┐ │
│ │                    2. AGENT STATE                                    │ │
```

```
║
║ ├──────────────────────────────────────────────────────────────────────────────
─┤ ║
║ │                                             │ ║
║ │  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    │ ║
║ │                                             │ ║
║
║ └──────────────────────────────────────────────────────────────────────────────
─┘ ║
║                                                  ║
║
║ ┌──────────────────────────────────────────────────────────────────────────────
─┐ ║
║ │              3. MULTI-LINE SUPPORT                        │ ║
║
║ ├──────────────────────────────────────────────────────────────────────────────
─┤ ║
║ │                                             │ ║
║ │  agents:calls:{user_id}:{call_id} → Hash with call state (TTL: 1 hr)   │ ║
║ │  agents: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:transfers              → Sorted Set for transfer matching   ││
││  calls:device_reported:{mac}:{#}  → Device deduplication            ││
││  calls:accumulated_ring:{remote}  → Float (TTL: 120s)              ││
││                                                  ││
││
│└──────────────────────────────────────────────────────────────────────────────
─┘│
│                                                   │
││
┌───────────────────────────────────────────────────────────────────────────────
─┐│
││               6. SESSION MANAGEMENT                 ││
││
│├──────────────────────────────────────────────────────────────────────────────
─┤│
││                                                  ││
││  user_session_heartbeat:{user_id} → Hash (socket_id, last_heartbeat)   ││
││                        TTL: 90s (60s heartbeat + 30s grace) ││
││                                                  ││
││
│└──────────────────────────────────────────────────────────────────────────────
─┘│
│                                                   │
└───────────────────────────────────────────────────────────────────────────────
────────────┘
```

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	`cache_redis_client` (REDIS_CACHE_DB)	`redis_client` (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

```
┌───────────────────────────────────────────────────────────────────────
──────┐
│                    INCOMING CALL FLOW                    │
├───────────────────────────────────────────────────────────────────────
──────┤
│                                                    │
│  1. CALL ARRIVES                                          │
│     │                                                    │
│     ├──→ claim_call(call_id, remote_number, time)            │
│     │   Creates: calls:active:{remote_number}               │
│     │                                                    │
│     ├──→ set_ring_time(call_id, timestamp)                   │
│     │   Creates: calls:ring:{call_id} (TTL: 5 min)           │
│     │                                                    │
│     ├──→ set_call_direction(call_id, 'in')                  │
│     │   Creates: calls:direction:{call_id} (TTL: 1 hr)        │
│     │                                                    │
│     └──→ mark_device_reported_call(mac, remote_number)         │
│         Creates: calls:device_reported:{mac}:{remote}          │
│                                                    │
│  2. CALL ANSWERED                                          │
│     │                                                    │
│     ├──→ update_call_answered_status(remote_number)           │
│     │   Updates: calls:active:{remote_number}.answered = "true"  │
│     │                                                    │
│     ├──→ set_call_answered_time(call_id, timestamp)           │
│     │   Creates: calls:answered:{call_id}                  │
│     │                                                    │
│     ├──→ create_active_call(call_id, customer, agent, ...)      │
│     │   Creates: active_calls:{call_id} (full call data)       │
│     │           agent_active_call:{agent_id} (lookup)       │
│     │                                                    │
│     ├──→ set_agent_state(user_id, 'on-call', call_data)        │
│     │   Creates: agents:state:{user_id}                   │
│     │                                                    │
│     └──→ void_missed_calls_for_remote(remote_number)          │
│         Removes pending missed calls for this number          │
│                                                    │
│  3. CALL ENDS                                          │
│     │                                                    │
│     ├──→ cleanup_active_call(remote_number)                 │
│     │   Deletes: calls:active:{remote_number}              │
│     │                                                    │
│     ├──→ remove_active_call(call_id)                      │
│     │   Deletes: active_calls:{call_id}, agent_active_call:{agent_id}   │
│    │                                           │
│    ├──→ cleanup_ring_time(call_id)                        │
│    │   Deletes: calls:ring:{call_id}                    │
│    │                                           │
│    ├──→ cleanup_call_direction(call_id)                     │
│    │   Deletes: calls:direction:{call_id}                  │
│    │                                           │
│    └──→ clear_agent_state(user_id)                       │
│        Deletes: agents:state:{user_id}                   │
│                                              │
│  4. MISSED CALL (No Answer)                            │
│    │                                           │
│    ├──→ queue_missed_call(call_data, delay_seconds=5)              │
│    │   Creates: calls:missed:queue (List)                   │
│    │          calls:missed:schedule (Sorted Set)             │
│    │                                           │
│    └──→ (Background) get_ready_missed_calls(current_time)            │
│        Processes missed calls after delay                  │
│                                              │
└──────────────────────────────────────────────────────────────────────────
──────────┘
```

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 `redis_client` (REDIS_CALLS_DB)
✓ Prefixes for keys	`calls:`, `agents:`, `active_calls:`, 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)

```
┌══════════════════════════════════════════════════════════════════
═════┐
║                                                      ║
║  utils/grandstream.py      vs      grandstream_consumer.py      ║
║  ═════════════════════════         ═════════════════════════════      ║
║                                                      ║
║  UTILITY FUNCTIONS                 ORCHESTRATOR/CONTROLLER      ║
║  (Helper tools)                (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)
   ════════════════════════════════════════════
```

```
┌──────────────────────────────────────────────────────────────────────┐
│                    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)           │
```

```
│
══════════════════════════════════════════════════════════════════
════════════════════│
│                                        │
│  CALLS TO call_cache.py (Redis):      CALLS TO grandstream.py:  │
│  ┌────────────────────────────┐      ┌────────────────────────┐│
│  │ • is_duplicate_call()      │      │ • create_call_log()    ││
│  │ • claim_call()             │      │ • is_internal_call()   ││
│  │ • set_ring_time()          │      │ • find_root_call_id()  ││
│  │ • set_call_direction()     │      │ • create_call_stats()  ││
│  │ • get_agent_state()        │      │ • is_extension_specific││
│  │ • update_active_call()     │      │   _call()              ││
│  │ • queue_missed_call()      │      │                        ││
│  └────────────────────────────┘      └────────────────────────┘│
│                                        │
│  ALSO CALLS:                                   │
│  • sockets/sockets_core.py → WebSocket events to agents       │
│  • Database ORM → CallLog, MissedCallLog, CallStat tables     │
│                                        │

└───────────────────────────────────────────────────────────────┘
      │
      ├────────────────────────────────┐
      ▼                                ▼
┌─────────────────────┐    ┌─────────────────────┐
│  REDIS (Cache)      │    │  PostgreSQL (DB)    │
│  ─────────────────  │    │  ─────────────────  │
│                     │    │                     │
│ • Call state        │    │ • CallLog           │
│ • Agent state       │    │ • MissedCallLog     │
│ • Ring times        │    │ • CallStat          │
│ • Active calls      │    │ • User              │
│ • Transfer links    │    │ • Company           │
│                     │    │                     │
└─────────────────────┘    └─────────────────────┘
      │
      ▼
  STEP 5: WEBSOCKET TO FRONTEND
  ═══════════════════════════════


┌───────────────────────────────────────────────────────────────┐
│                sockets/sockets_core.py            │
│
══════════════════════════════════════════════════════════════════
════════════════════│
│                                        │
│  broadcast_realtime_event(event_type, data, room):      │
│  • "incoming" → Agent sees incoming call popup          │
│  • "on-call"  → Agent UI shows on-call state            │
│  • "hangup"   → Agent UI resets to available            │
│  • "missed"   → Dashboard updates missed call count     │
```

```
  │                                        │
  └──────────────────────────────────────────────────────────────────────────┘
        │
        ▼
  ┌──────────────────────────┐
  │  FRONTEND UI   │  React/Vue app shows call to agent
  │  (Agent Panel)  │
  └──────────────────────────┘
```

Incoming Call Example - Step by Step

```
┌─────────────────────────────────────────────────────────────────────────────
────────┐
│               INCOMING CALL - DETAILED FLOW                    │
└─────────────────────────────────────────────────────────────────────────────
────────┘

1️⃣ PHONE RINGS
  │
  │  Grandstream phone sends:
  │  GET /gs/cti/incoming?callId=12345&remote=03001234567&local=203&mac=AA:BB:CC
  │
  ▼
2️⃣ 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
  │
  ▼
3️⃣ 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 = "..." (TTL: 5min) │
│ │ │                                                         │
│ │ │ call_cache.set_call_direction(call_id, "in")            │
│ │ │   └─ Creates: calls: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 │
│ │ │                                                         │
```

```
   │  └──────────────────────────────────────────────────────────────┘
   │
   │  ┌──────────────────────────────────────────────────────────────┐
   │  │ 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)        │
   │  │   ...                                                 │
   │  │ })                                                    │
   │  │                                                       │
   │  └──────────────────────────────────────────────────────────────┘
   │
   ▼
4️⃣ AGENT SEES INCOMING CALL POPUP IN UI
   │
   │  Agent clicks "Answer"
   │  Phone sends: GET /gs/cti/answered?callId=12345&...
   │
   ▼
5️⃣ handle_answered_call() processes answer event...
   │
   │  (Similar flow with cache updates + DB writes)
   │
   ▼
6️⃣ 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
```

```
                   ▼
          ┌──────────────────────────────────────┐
          │   messaging/rabbitmq.py              │
          │   publish_event()                    │
          └─────────────────┬────────────────────┘
                            │ consumed by
                            ▼
┌──────────────────────────────────────────────────────────────────────┐
│                                                                      │
│          grandstream_consumer.py                                     │
│          (THE ORCHESTRATOR)                                          │
│                                                                      │
│  handle_incoming_call() ─────┬─────────────────────────────────────► │
│  handle_outgoing_call()      │                                       │
│  handle_answered_call()      │ USES BOTH:                            │
│  handle_hangup_call()        │                                       │
│  handle_rejected_call()      │  ┌──────────────┐  ┌──────────────┐   │
│  handle_missed_call()        │  │ call_cache.py│  │grandstream.py│   │
│                              │  │ (Redis State)│  │(DB Utilities)│   │
│                              │  └──────┬───────┘  └──────┬───────┘   │
│                              │         │                 │           │
└──────────────────────────────┼─────────┼─────────────────┼───────────┘
                               │         │                 │
                               │         ▼                 ▼
                               │  ┌──────────────┐  ┌──────────────┐
                               │  │    REDIS     │  │  PostgreSQL  │
                               │  │  (Real-time) │  │ (Persistent) │
                               │  └──────────────┘  └──────────────┘
                               │
                               │ also uses
                               ▼
          ┌──────────────────────────────────────┐
          │   sockets/sockets_core.py            │
          │   (WebSocket to Frontend)            │
          └──────────────────────────────────────┘
```

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:
═══════════════

grandstream_routes.py              grandstream_consumer.py
      │                                  ▲
      │ publish_event()                  │ consume from queue
      ▼                                  │
┌──────────────────────────────────────────────────────────────────┐
│                 RABBITMQ                        │
│  ┌──────────────┐    ┌────────────────────┐    ┌──────────────────┐ │
│  │ PUBLISHER │───→│ EXCHANGE │───→│  QUEUE  │ │
│  │ (routes)  │  │ "grandstream" │  │"grandstream-  │ │
│  │      │  │       │  │ queue"  │ │
│  └──────────────┘    └────────────────────┘    └────────┬─────────┘ │
│                                    │ │
└──────────────────────────────────────────────────────┼───────────┘
                                     │
                                     ▼
                           grandstream_consumer.py
                           (DIRECTLY consumes)
```

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

Your Corrected Complete Flow

```
┌──────────────────────────────────────────────────────────────────────
──────────┐
│              CORRECTED COMPLETE FLOW                │
└──────────────────────────────────────────────────────────────────────
──────────┘

1️⃣ GRANDSTREAM MACHINE (PBX/Phone)
  │
  │ Detects: incoming call, outgoing call, answered, hangup, missed
  │ Sends HTTP webhook to YOUR server
  │
  ▼
2️⃣ 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
  │
  ▼
3️⃣ messaging/rabbitmq.py :: publish_event()
  │
  │   Publishes call_data JSON to RabbitMQ
  │   Exchange: "grandstream"
  │   Routing Key: "call.incoming" / "call.answered" / etc.
  │
  ▼
4️⃣ 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)
  │
  ▼
5️⃣ 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.
  │
  ▼
6️⃣ HANDLER FUNCTIONS (inside consumer)
  │
  │   Uses THREE things:
  │
  │   ┌─────────────────────────────────────────────────────────┐
  │   │                                                          │
  │   │   call_cache.py (Redis)   │   utils/grandstream.py (DB)  │
  │   │   ═══════════════════════ │ ════════════════════════     │
  │   │   • claim_call()          │   • create_call_log()        │
  │   │   • set_ring_time()       │   • create_missed_call_log() │
  │   │   • get_agent_state()     │   • create_or_update_stats() │
  │   │   • update_active_call()  │   • is_internal_call()       │
  │   │                           │                              │
```

```
    │   └──────────────────────────────────────────────────────────┘
    │
    │   Also calls: sockets/sockets_core.py
    │   • broadcast_realtime_event("incoming", data)
    │   • send_to_agent(user_id, "on-call", data)
    │
    ▼
7️⃣ WEBSOCKET TO FRONTEND
    │
    │   Socket.IO emits event to connected agents
    │
    ▼
8️⃣ FRONTEND (Agent Panel)

    Agent sees:
    • Incoming call popup
    • On-call status
    • Hangup notification
    • Missed call alert
```

Visual Summary

```
┌──────────────┐     HTTP      ┌──────────────┐    Publish    ┌──────────────┐
│ GRANDSTREAM  │ ────────────► │    ROUTES    │ ────────────► │   RABBITMQ   │
│   MACHINE    │   Webhook     │  (FastAPI)   │               │   (Queue)    │
└──────────────┘               └──────────────┘               └──────┬───────┘
                                      │                              │
                                      │ uses                         │ consume
                                      ▼                              ▼
                               ┌──────────────┐               ┌──────────────┐
                               │   UTILS/     │ ◄──────────── │   CONSUMER   │
                               │ grandstream  │     uses      │ (Background) │
                               └──────────────┘               └──────┬───────┘
                                                                     │
                               ┌──────────────┐                      │ uses
                               │  call_cache  │ ◄────────────────────┤
                               │   (Redis)    │                      │
                               └──────────────┘                      │
                                                                     │ calls
                               ┌──────────────┐                      │
                               │   SOCKETS    │ ◄────────────────────┘
                               │ (WebSocket)  │
                               └──────┬───────┘
                                      │ emit
                                      ▼
                               ┌──────────────┐
                               │   FRONTEND   │
                               │  (Agent UI)  │
```

Key Point You Should Remember

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