Binlog/WAL Monitor - Complete Flow

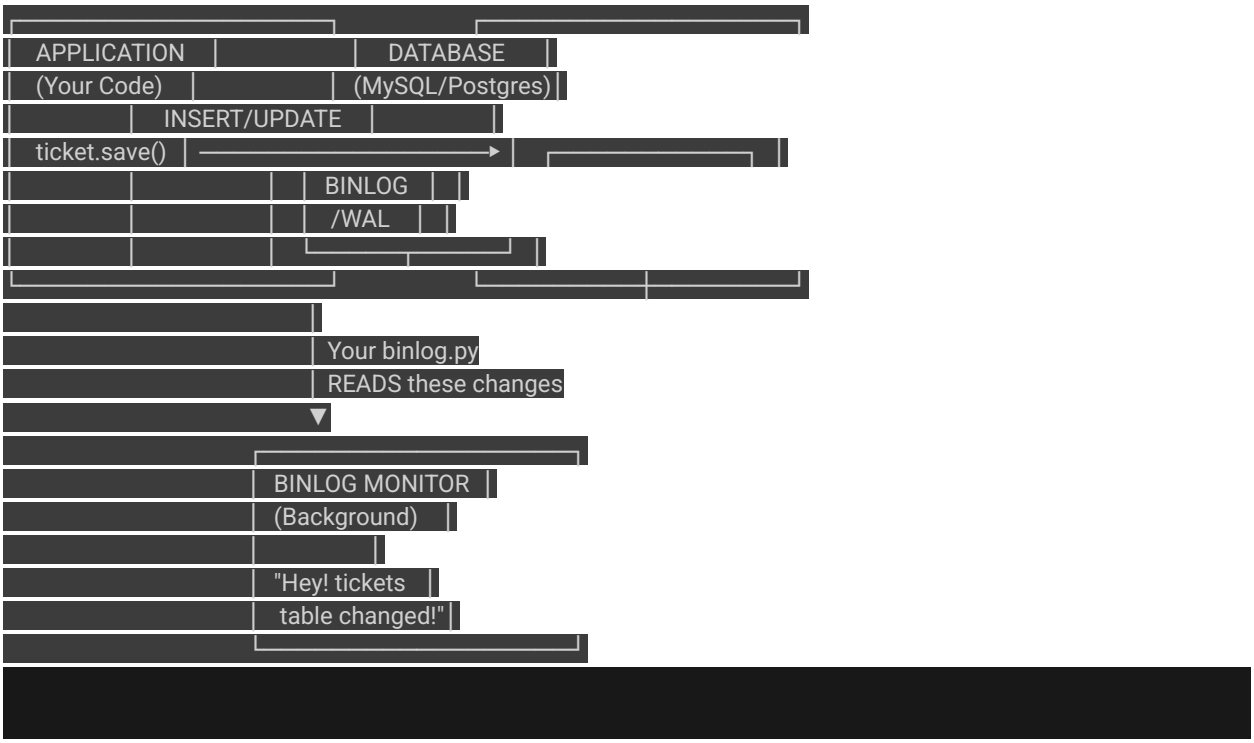
What is Binlog/WAL?

BINLOG (MySQL):

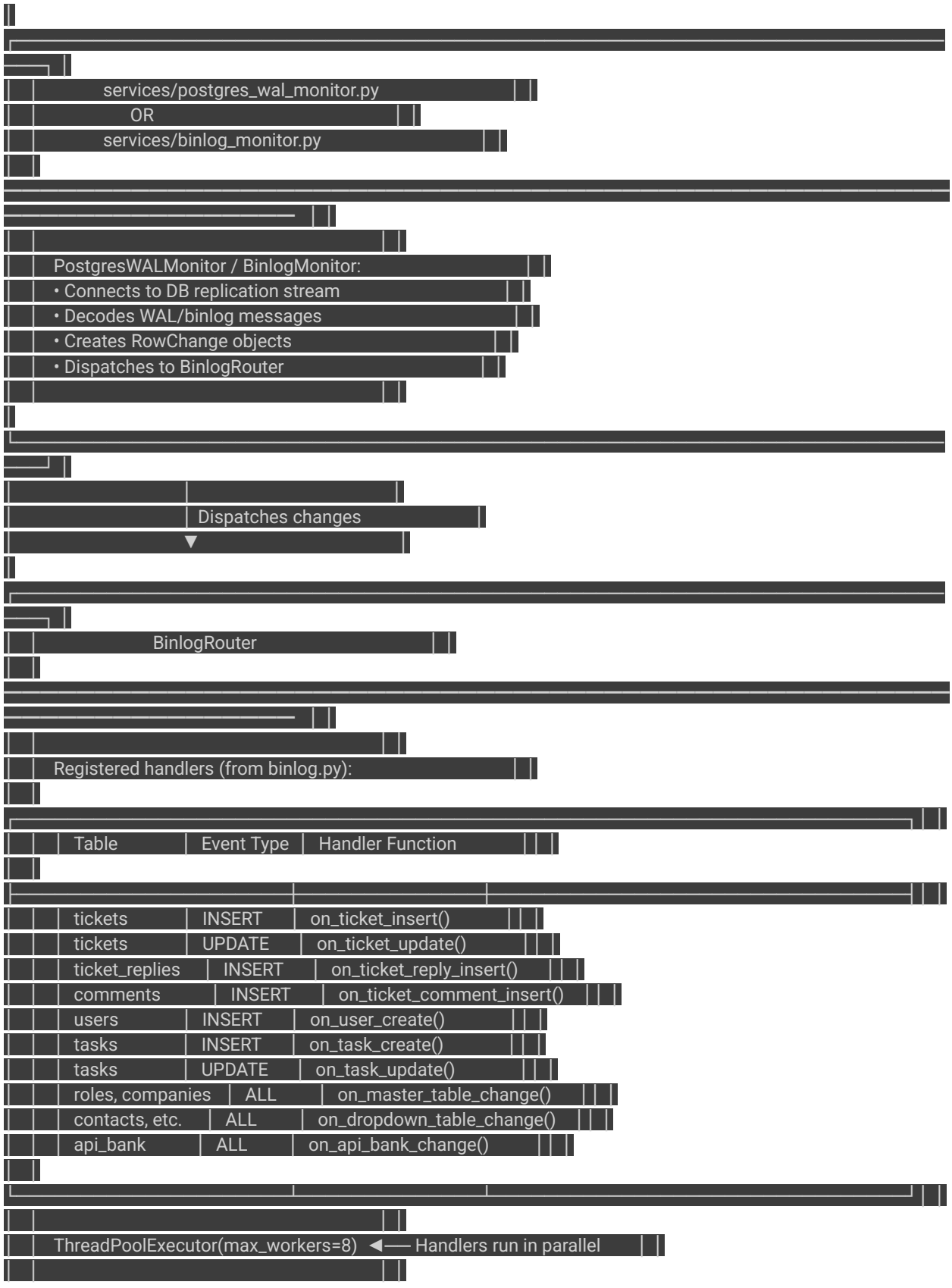
• Binary Log - records ALL changes to database
• Used for: replication, point-in-time recovery, change tracking
• Your app: reads binlog to detect INSERT/UPDATE/DELETE

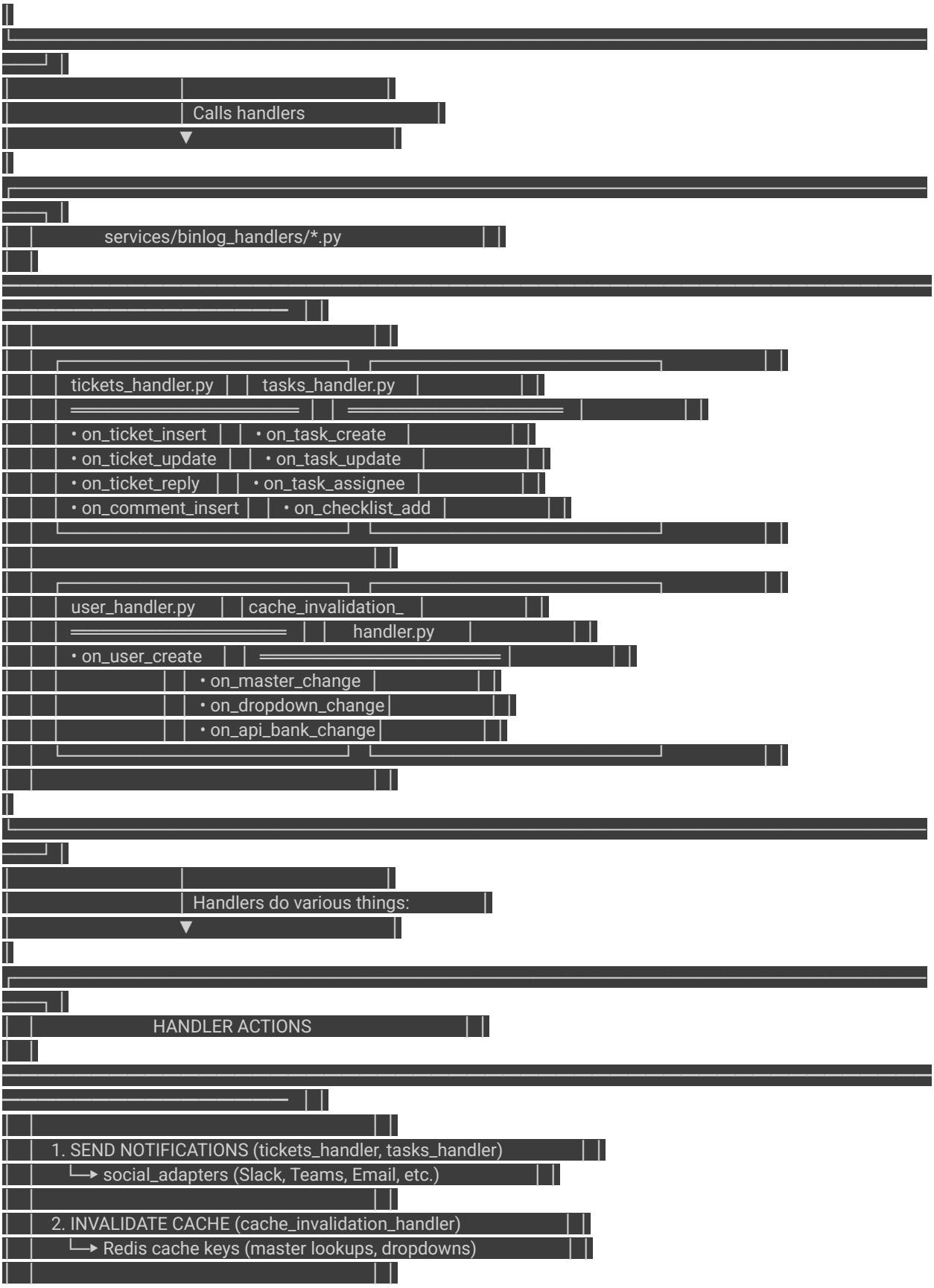
WAL (PostgreSQL):

• Write-Ahead Log - records ALL changes before they're committed
• Used for: crash recovery, replication, change tracking
• Your app: uses "Logical Replication" to decode WAL into readable events

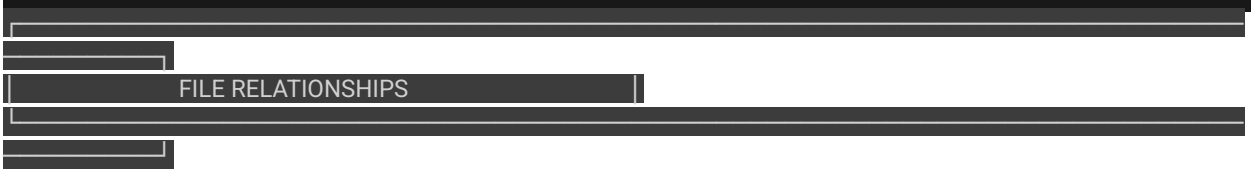


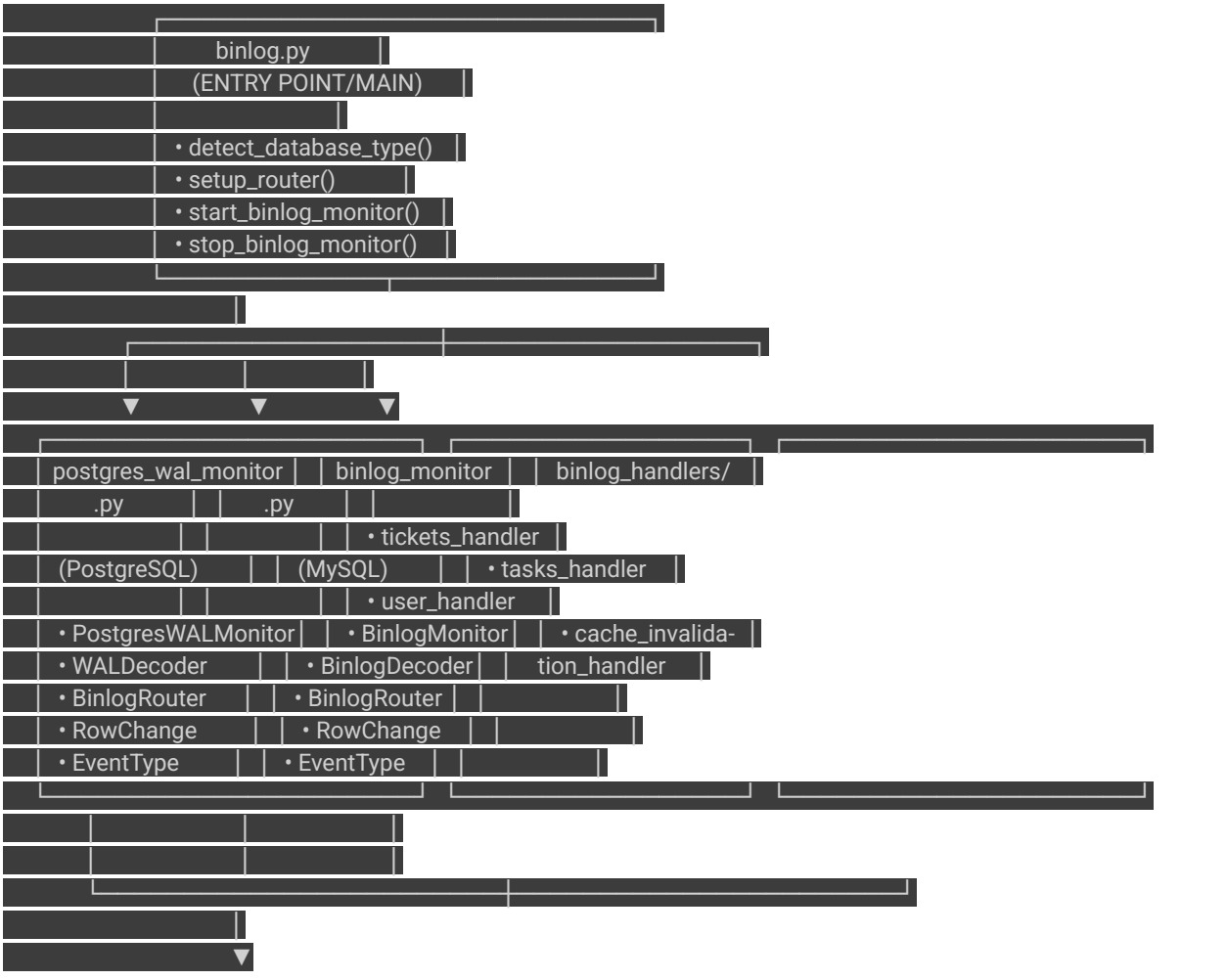
Complete Architecture


services/postgres_wal_monitor.py
OR
services/binlog_monitor.py
PostgresWALMonitor / BinlogMonitor:
• Connects to DB replication stream
• Decodes WAL/binlog messages
• Creates RowChange objects
• Dispatches to BinlogRouter
Dispatches changes
BinlogRouter
Registered handlers (from binlog.py):
Table
Event Type
Handler Function
tickets
INSERT
on_ticket_insert()
tickets
UPDATE
on_ticket_update()
ticket_replies
INSERT
on_ticket_reply_insert()
comments
INSERT
on_ticket_comment_insert()
users
INSERT
on_user_create()
tasks
INSERT
on_task_create()
tasks
UPDATE
on_task_update()
roles, companies
ALL
on_master_table_change()
contacts, etc.
ALL
on_dropdown_table_change()
api_bank
ALL
on_api_bank_change()
ThreadPoolExecutor(max_workers=8)
Handlers run in parallel


Calls handlers
services/binlog_handlers/*.py
tickets_handler.py
• on_ticket_insert
• on_ticket_update
• on_ticket_reply
• on_comment_insert
tasks_handler.py
• on_task_create
• on_task_update
• on_task_assignee
• on_checklist_add
user_handler.py
• on_user_create
cache_invalidation_handler.py
• on_master_change
• on_dropdown_change
• on_api_bank_change
Handlers do various things:
HANDLER ACTIONS
1. SEND NOTIFICATIONS (tickets_handler, tasks_handler)
social_adapters (Slack, Teams, Email, etc.)
2. INVALIDATE CACHE (cache_invalidation_handler)
Redis cache keys (master lookups, dropdowns)

File Relationships





Complete Flow - Step by Step

1⃣ APP STARTS
│
│ main.py calls:
│ start_binlog_monitor(max_workers=8)
│
▼
2⃣ binlog.py :: start_binlog_monitor()
│
│ a) detect_database_type()
│ └─ Checks DATABASE_URL → Returns 'postgres' or 'mysql'
│
│ b) setup_router(max_workers=8)
│ └─ Creates BinlogRouter
│ └─ Registers handlers for each table:
│
│ router.register("public", "tickets", [INSERT], on_ticket_insert)
│ router.register("public", "tickets", [UPDATE], on_ticket_update)
│ router.register("public", "users", [INSERT], on_user_create)
│ router.register("public", "roles", [ALL], on_master_table_change)
│ ... etc.
│
│ c) Create Monitor:
│ └─ PostgresWALMonitor(database_url, slot_name, router=router)
│ OR
│ └─ BinlogMonitor(database_url, router=router)
│
│ d) Start Background Thread:
│ └─ threading.Thread(target=_monitor.start, daemon=True)

```
│
▼
3️⃣ PostgresWALMonitor :: start()  (Background Thread)
│
│   a) _ensure_replication_slot()
│      └─ Creates replication slot if not exists
│
│   b) Connect with LogicalReplicationConnection
│
│   c) _cursor.start_replication(slot_name="nextgen_slot_...")
│
│   d) _cursor.consume_stream(_process_message)  ◄── BLOCKS HERE
│      └─ Continuously receives WAL changes
│
│   e) Start Keepalive Thread (prevents slot from going inactive)
│
▼
4️⃣ DATABASE CHANGE OCCURS
│
│   Example: User creates a new ticket via API
│   └─ INSERT INTO tickets (title, assigned_to_id, ...) VALUES (...)
│
│   PostgreSQL writes to WAL
│   WAL is streamed to your monitor
│
▼
5️⃣ _process_message(msg)
│
│   a) Decode WAL message (wal2json format):
│      {
│        "change": [{
│          "kind": "insert",
│          "schema": "public",
│          "table": "tickets",
│          "columnnames": ["id", "title", "assigned_to_id", ...],
│          "columnvalues": [123, "Fix bug", 5, ...]
│        }]
│      }
│
│   b) Create RowChange object:
│      RowChange(
│        schema="public",
│        table="tickets",
│        event=EventType.INSERT,
│        row={"id": 123, "title": "Fix bug", "assigned_to_id": 5, ...}
│      )
│
│   c) Call self._emit(change)
│
▼
6️⃣ BinlogRouter :: dispatch(change)
│
```

```
│   Checks all registered rules:
│
│
┌──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┐
│   Rule: ("public", "tickets", [INSERT], on_ticket_insert)                                                            │
│                                                                                                                      │
│   Does change match?                                                                                                 │
│   • change.schema == "public" ✓                                                                                      │
│   • change.table == "tickets" ✓                                                                                      │
│   • change.event == INSERT ✓                                                                                         │
│                                                                                                                      │
│   MATCH! Submit to thread pool:                                                                                      │
│   ThreadPoolExecutor.submit(on_ticket_insert, change)                                                                │
│                                                                                                                      │
└──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┘
    │
    ▼
7] tickets_handler.py :: on_ticket_insert(change, db_factory)
    │
    a) Extract data from change.row:
       ticket_id = 123
       assigned_to_id = 5
    │
    b) Fetch complete ticket from DB:
       ticket = db.query(Tickets).filter(Tickets.id == 123).first()
    │
    c) Build notification data:
       ticket_data = {
         "id": 123,
         "ticket_id": "TKT-00123",
         "title": "Fix bug",
         "assigned_to": "John Doe",
         "priority": "High",
         ...
       }
    │
    d) Send notification to assigned user:
       notify_users(db, [5], "New Ticket", ticket_data)
    │
    e) Notify managers:
       manager_ids = get_manager_ids(5, db)
       notify_users(db, manager_ids, "Manager: New Ticket", ticket_data)
    │
    ▼
8] NOTIFICATION SENT
    │
    social_adapters/slack_adapter.py
    └─ Sends Slack message to user's connected account
    │
    social_adapters/teams_adapter.py
```

```
    │   └─ Sends Teams message
    │
    │   etc.
    │
    ▼
9️⃣ ACK & SAVE POSITION
    │
    │   Back in _process_message():
    │   • msg.cursor.send_feedback(flush_lsn=msg.data_start)
    │   • _save_offset(msg.data_start)
    │
    │   This saves the position so if app restarts,
    │   it continues from where it left off
    │
    ▼
🔟 LOOP CONTINUES
    │
    │   _cursor.consume_stream() continues waiting for next change
    │
    └──► Go back to step 4
```

Cache Invalidation Flow

```
┌──────────────────────────────────────────────────────────────────────────────┐
│                        CACHE INVALIDATION FLOW                               │
└──────────────────────────────────────────────────────────────────────────────┘

SCENARIO: Admin updates a "Role" in the database

1️⃣ UPDATE roles SET name = 'Super Admin' WHERE id = 1

2️⃣ WAL captures the change

3️⃣ PostgresWALMonitor receives:
   RowChange(
     schema="public",
     table="roles",
     event=EventType.UPDATE,
     before={"id": 1, "name": "Admin"},
     after={"id": 1, "name": "Super Admin"}
   )

4️⃣ BinlogRouter matches rule:
   router.register("public", "roles", [UPDATE], on_master_table_change)

5️⃣ cache_invalidation_handler :: on_master_table_change()
```

```
│
│   a) Look up cache keys for "roles" table:
│      TABLE_TO_CACHE_KEYS["roles"] = ["master:roles"]
│
│   b) Call invalidate_cache_keys(["master:roles"], row_data, db)
│
│   c) For each key:
│      └─ refresh_master_cache_key("roles", db)
│         └─ Deletes old cache
│         └─ Fetches fresh data from DB
│         └─ Stores new data in Redis
│
▼
6️⃣ RESULT:
│
│   Redis cache is now updated with fresh "roles" data
│   Next API request will get "Super Admin" instead of "Admin"
│
│   ┌──────────────────────────────────────────────────────┐
│   │ BEFORE:                                              │
│   │ Redis["master:roles"] = [{"id":1, "name":"Admin"}]   │
│   │                                                      │
│   │ AFTER:                                               │
│   │ Redis["master:roles"] = [{"id":1, "name":"Super Admin"}] │
│   └──────────────────────────────────────────────────────┘
```

Threading Architecture

```
┌──────────────────────────────────────────────────────────────────────────────┐
│                        THREADING ARCHITECTURE                                │
└──────────────────────────────────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────────────────────────────┐
│                           MAIN PROCESS                                       │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                              │
│  ┌─────────────────┐                                                         │
│  │  MAIN THREAD    │  FastAPI/Uvicorn handles HTTP requests                  │
│  │                 │                                                         │
│  └─────────────────┘                                                         │
│                                                                              │
│  ┌─────────────────┐                                                         │
│  │  POSTGRES-WAL-  │  Consumes WAL stream from PostgreSQL                    │
│  │  MONITOR THREAD │  • _cursor.consume_stream()                             │
```

Summary Comparison: Calls Flow vs Binlog Flow

Aspect	Call System	Binlog/WAL System
Trigger	HTTP webhook from Grandstream	Database change (INSERT/UPDATE/DELETE)
Entry Point	grandstream_routes.py	binlog.py
Queue/Stream	RabbitMQ	PostgreSQL WAL / MySQL Binlog
Consumer	grandstream_consumer.py	postgres_wal_monitor.py
Handlers	Inside consumer file	services/binlog_handlers/*.py
State Storage	Redis (call_cache.py)	N/A (just triggers actions)
Output	WebSocket to agents	Notifications + Cache refresh
Threading	Consumer in background thread	Monitor + Thread pool for handlers

---

Key Concepts You Should Remember

KEY CONCEPTS

1️⃣ WAL/BINLOG = Database transaction log that records ALL changes
- Your app connects to this log and "streams" the changes in real-time

2️⃣ REPLICATION SLOT = PostgreSQL's way of tracking what you've consumed
- Prevents data loss if your app disconnects
- Resumes from last position on restart

3️⃣ BINLOG ROUTER = Pattern-matching dispatcher
- You register: (table, event_type, handler_function)
- When change matches, handler is called

4️⃣ HANDLERS = Functions that react to database changes
- on_ticket_insert → Send notification to assignee
- on_master_table_change → Refresh Redis cache

5️⃣ THREAD POOL = Parallel handler execution
- max_workers=8 means 8 handlers can run simultaneously
- Prevents slow handler from blocking others

6️⃣ WHY USE THIS?
- Real-time reactions to database changes
- Decoupled from application code (doesn't matter HOW data was changed)
- Works even if change came from direct SQL, admin panel, etc.