

# Binlog/WAL Monitor - Complete Flow

## What is Binlog/WAL?

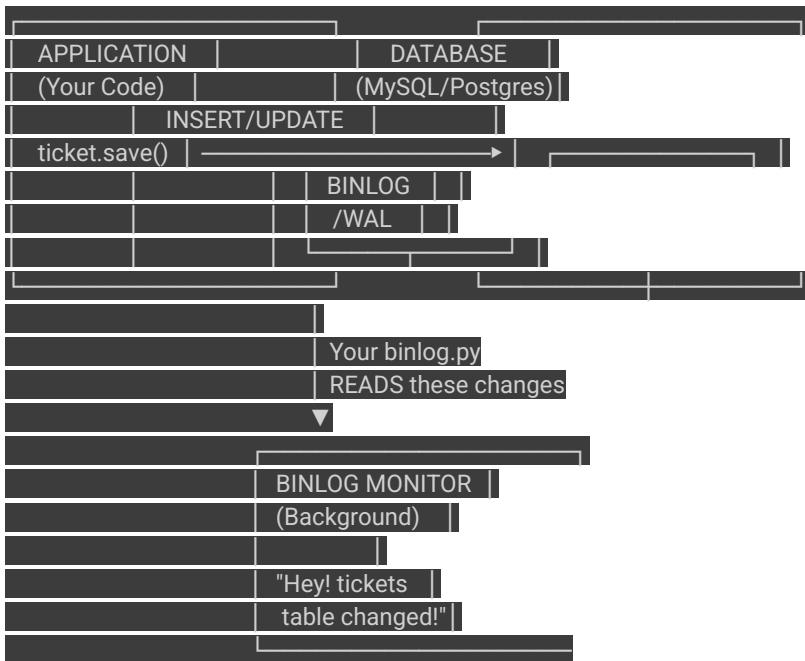


### BINLOG (MySQL):

- 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



### OverView:

overall logicalreplicationslot which lives live mean this is watcher which watches WAL stream and sends the stream which is basically a binary data, this data is converted to json using wal2json plugin (this is implemented in WALDecoder class which converts json → Rowchange object) and then this is sent over network to the app , this data is

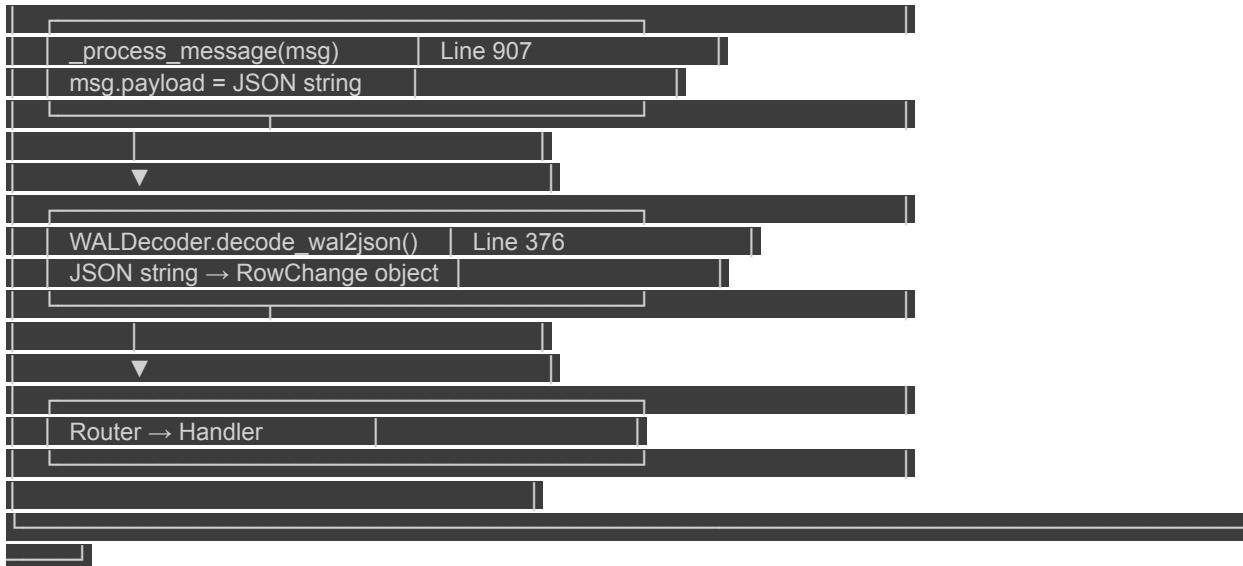
then processed in `_process_message` in `postgresqlwalmonitor` class where we also set the record of changes track in `offset.json` file so we can start from where we left, here after

REST OF THE FLOW

RowChange → Router.dispatch() → Handler function → Notifications/Cache processing

## Complete Architecture





### 1. REPLICATION SLOT (Lives on PostgreSQL server)

Line 754-756: pg\_create\_logical\_replication\_slot('nextgen\_slot\_xxx', 'wal2json')

- Created ONCE on PostgreSQL server
- Uses "wal2json" plugin to decode binary WAL → JSON
- Remembers: "What position have I sent to the client?"
- Keeps WAL data until client acknowledges receipt

### 2. BINARY → JSON CONVERSION (Done by PostgreSQL + wal2json plugin)

PostgreSQL WAL (binary): 0x00 0x1F 0xA3 0x7B ...

↓  
▼ (wal2json plugin on server)

JSON over TCP: {"kind": "insert", "table": "tickets", "columnvalues": [...]}

### 3. OUR CODE RECEIVES JSON (Not binary!)

Line 918: payload = json.loads(msg.payload) ← Already JSON string!

### 4. WALDecoder CONVERTS JSON → RowChange Object

Line 919: changes = self.decoder.decode\_wal2json(payload)

```

Input: {"kind": "insert", "schema": "public", "table": "tickets", ...}
Output: RowChange(schema="public", table="tickets", event=INSERT, row={...})

```

## 5. REST OF THE FLOW

RowChange → Router.dispatch() → Handler function → Notifications/Cache

## Key Correction

The **binary → JSON decoding** happens on the PostgreSQL server side (via wal2json plugin), NOT in our Python code. Our **WALDecoder** does **JSON → RowChange object** conversion (not binary decoding):

```

# Line 376-395 in postgres_wal_monitor.py
def decode_wal2json(self, payload: Dict[str, Any]) -> List[RowChange]:
    """
    Decode wal2json format messages.

```

Input (already JSON, decoded from binary by PostgreSQL):

```

{
  "change": [
    {
      "kind": "insert",
      "schema": "public",
      "table": "tickets",
      "columnnames": ["id", "title"],
      "columnvalues": [11, "test"]
    }
  ]
}

```

Output: List[RowChange] objects

```

"""

```

So the flow is:

Location	What happens	Format
PostgreSQL WAL files	Raw database changes	Binary
Replication Slot + wal2json	Decodes binary	Binary → JSON
TCP Stream to our app	Sends changes	JSON string
_process_message()	Parses JSON	JSON string → Dict
WALDecoder.decode_wal2json()	Creates objects	Dict → RowChange
Router + Handlers	Business logic	RowChange objects

## THE 4 CONVERSION STEPS - EXPLAINED

### Step 1: TCP Stream → JSON String

PostgreSQL sends this raw text over the network:

```
'{"change": [{"kind": "insert", "schema": "public", "table": "tickets", "columnnames": ["id", "title"], "columnvalues": [123, "Help needed", 5]}]}'
```

This arrives in `msg.payload` - it's just a **string** (text).

---

### Step 2: `_process_message()` - JSON String → Dict

Read d:\NextGenCC\NextGen-FastAPI\services\postgres\_wal\_monitor.py (lines 908-932)

```
Line 918: payload = json.loads(msg.payload)
# BEFORE (string):
msg.payload =
'{"change": [{"kind": "insert", "schema": "public", "table": "tickets", "columnnames": ["id", "title"], "columnvalues": [123, "Help"]}]}'
```

```
# AFTER (Python dictionary):
payload = {
    "change": [
        {
            "kind": "insert",
            "schema": "public",
            "table": "tickets",
            "columnnames": ["id", "title"],
            "columnvalues": [123, "Help"]
        }
    ]
}
```

`json.loads()` converts **text** → **Python dict** (so we can access keys like `payload["change"]`)

---

### Step 3: `WALDecoder.decode_wal2json()` - Dict → RowChange

Read d:\NextGenCC\NextGen-FastAPI\services\postgres\_wal\_monitor.py (lines 377-456)

Line 429: `row_data = dict(zip(columnnames, columnvalues))` Line 430-438: Creates RowChange object

```
# INPUT (Python dict from json.loads):
payload = {
    "change": [
        {
            "kind": "insert",
            "schema": "public",
            "table": "tickets",
            "columnnames": ["id", "title", "assigned_to_id"],
            "columnvalues": [123, "Help needed", 5]
        }
    ]
}

# PROCESSING (Line 429):
columnnames = ["id", "title", "assigned_to_id"]
columnvalues = [123, "Help needed", 5]
row_data = dict(zip(columnnames, columnvalues))
# row_data = {"id": 123, "title": "Help needed", "assigned_to_id": 5}

# OUTPUT (RowChange object - Line 430-438):
RowChange(
    schema="public",
    table="tickets",
    event=EventType.INSERT,
    row={"id": 123, "title": "Help needed", "assigned_to_id": 5},
    timestamp=1234567890.123
)
```

---

## Step 4: Router + Handlers - Use RowChange Object

```
# In tickets_handler.py Line 92-100:
def on_ticket_insert(change, db_factory):
    row = change.row # {"id": 123, "title": "Help needed", "assigned_to_id": 5}

    ticket_id = row.get("id")      # 123
    assigned_to_id = row.get("assigned_to_id") # 5

    # Now do business logic...
```

---

## Visual Summary

STEP 1: Network receives text

`'{"change": [{"kind": "insert", ...}]}'` ← Just a STRING (text)

#### STEP 2: json.loads() - String → Dict

```
{                                     ← Now a DICT (can use [keys])
  "change": [
    "kind": "insert",
    "columnnames": ["id", "title"],
    "columnvalues": [123, "Help"]
  ]
}
```

#### STEP 3: decode\_wal2json() - Dict → RowChange

```
RowChange(          ← Now an OBJECT (can use .attributes)
  schema="public",
  table="tickets",
  event=EventType.INSERT,
  row={"id": 123, "title": "Help"}
)
```

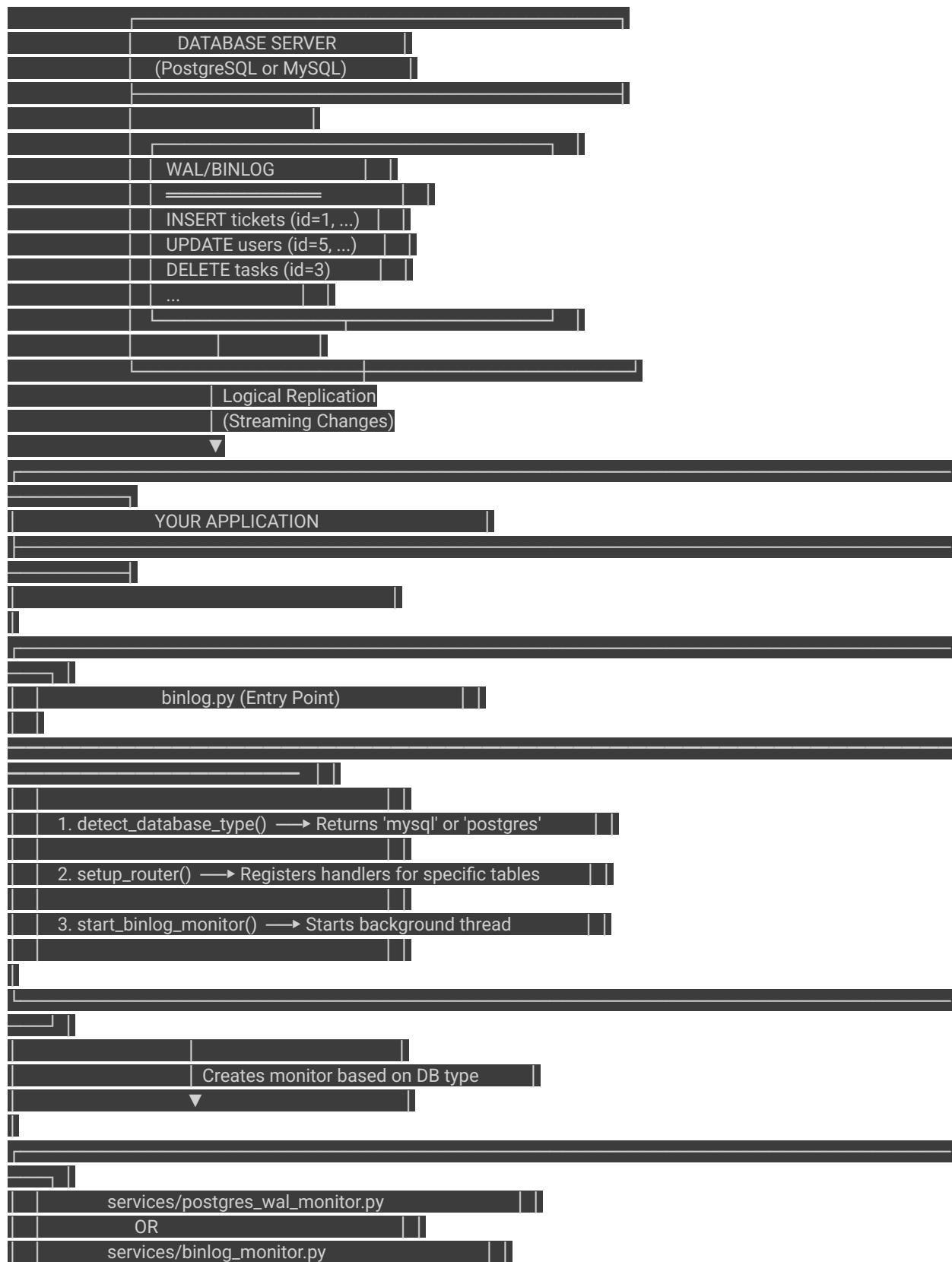
#### STEP 4: Handler uses RowChange

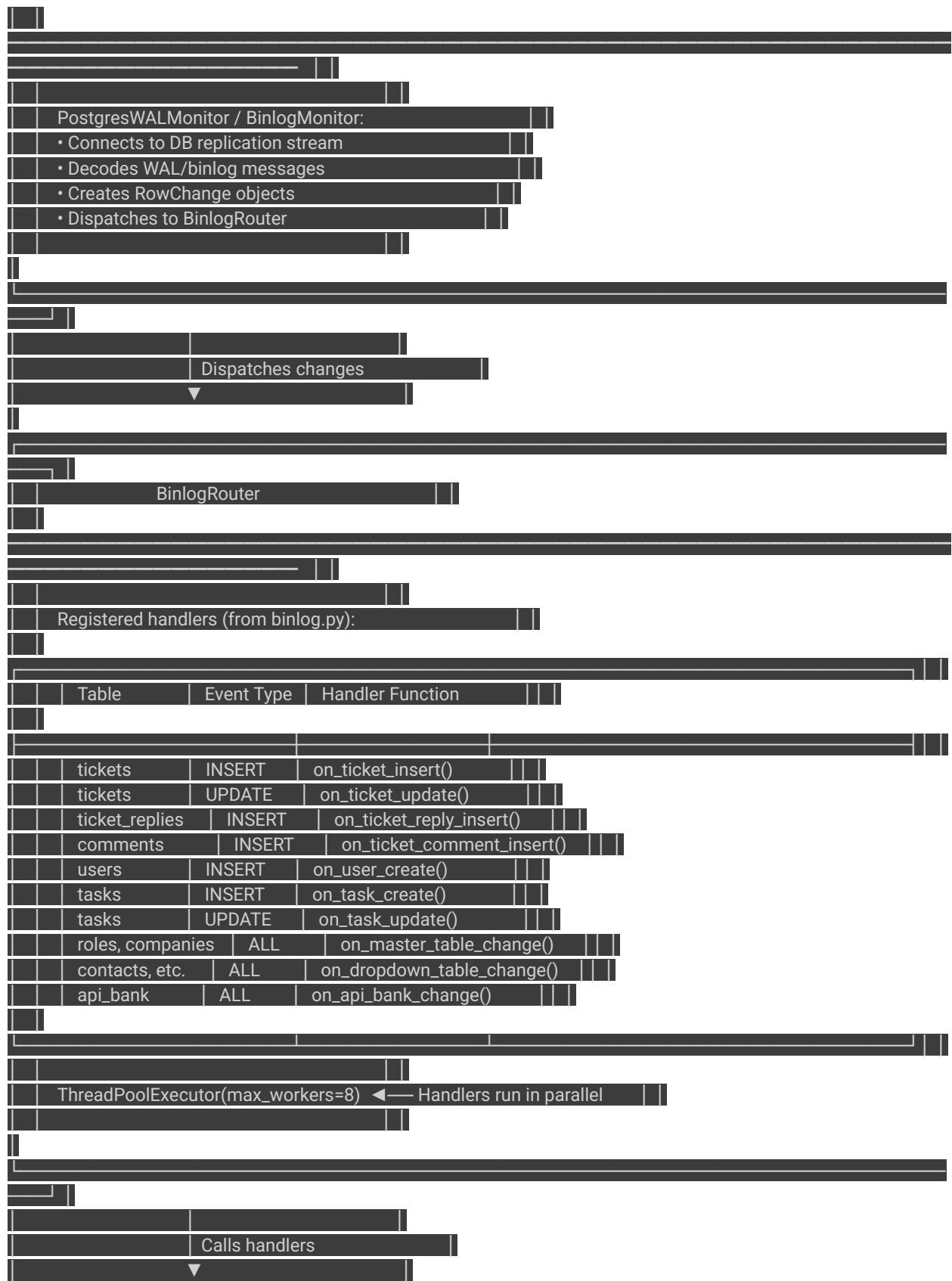
```
change.table → "tickets"
change.event → EventType.INSERT
change.row["id"] → 123
```

## Why 3 Conversions?

Format	Why needed
String	Network can only send text/bytes
Dict	Python can access data by keys ( <code>payload[ "change" ]</code> )
RowChange	Clean object with named attributes ( <code>change.table</code> , <code>change.row</code> ) - easier to use in handlers







services/binlog\_handlers/\*.py

tickets\_handler.py | tasks\_handler.py  
===== | |  
• on\_ticket\_insert | • on\_task\_create |  
• on\_ticket\_update | • on\_task\_update |  
• on\_ticket\_reply | • on\_task\_assignee |  
• on\_comment\_insert | • on\_checklist\_add |

user\_handler.py | cache\_invalidation\_ |  
===== | | handler.py |  
• on\_user\_create | |  
	• 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)

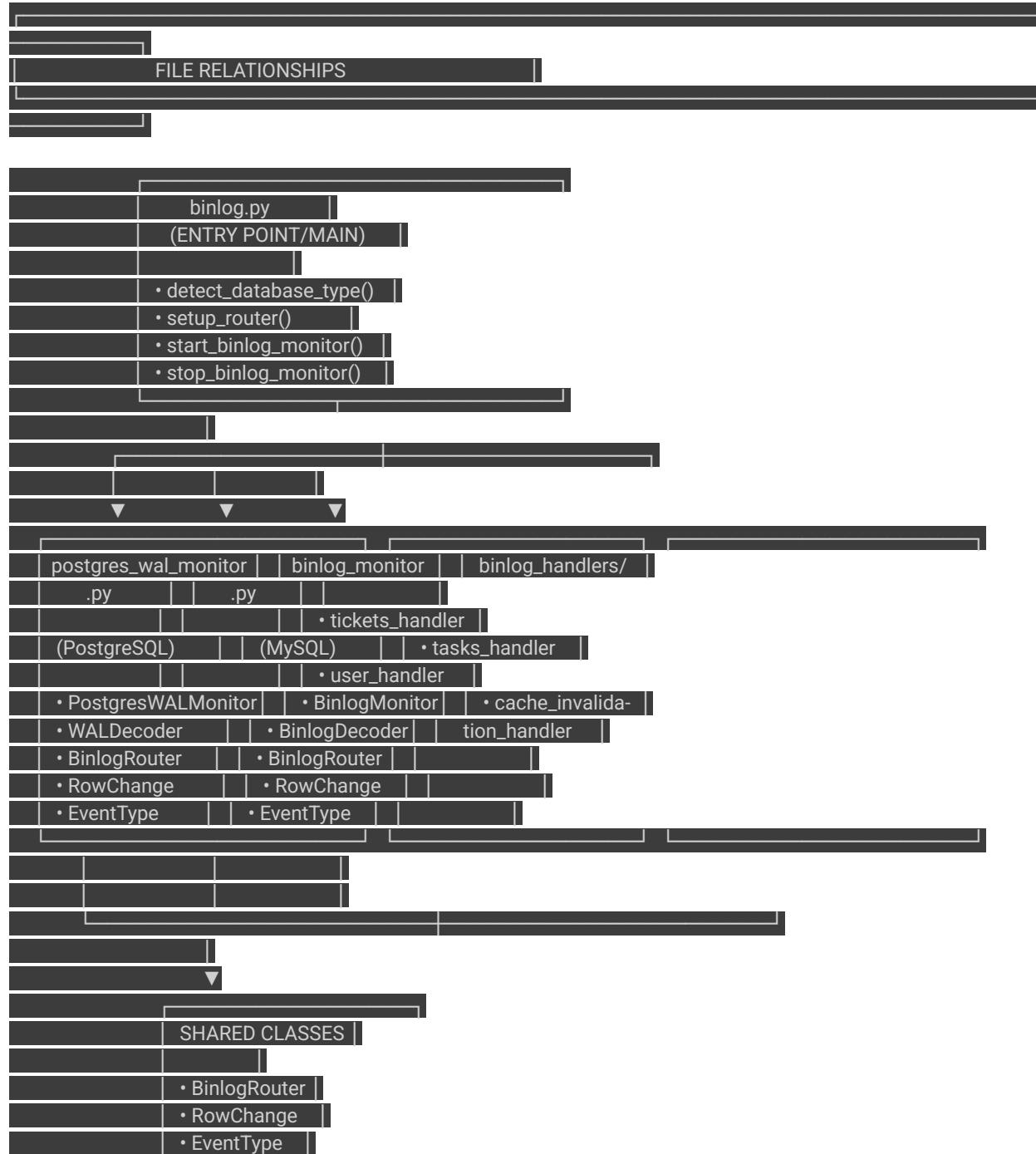
3. REFRESH DATA (cache\_invalidation\_handler)

└→ Re-populate cache with fresh data



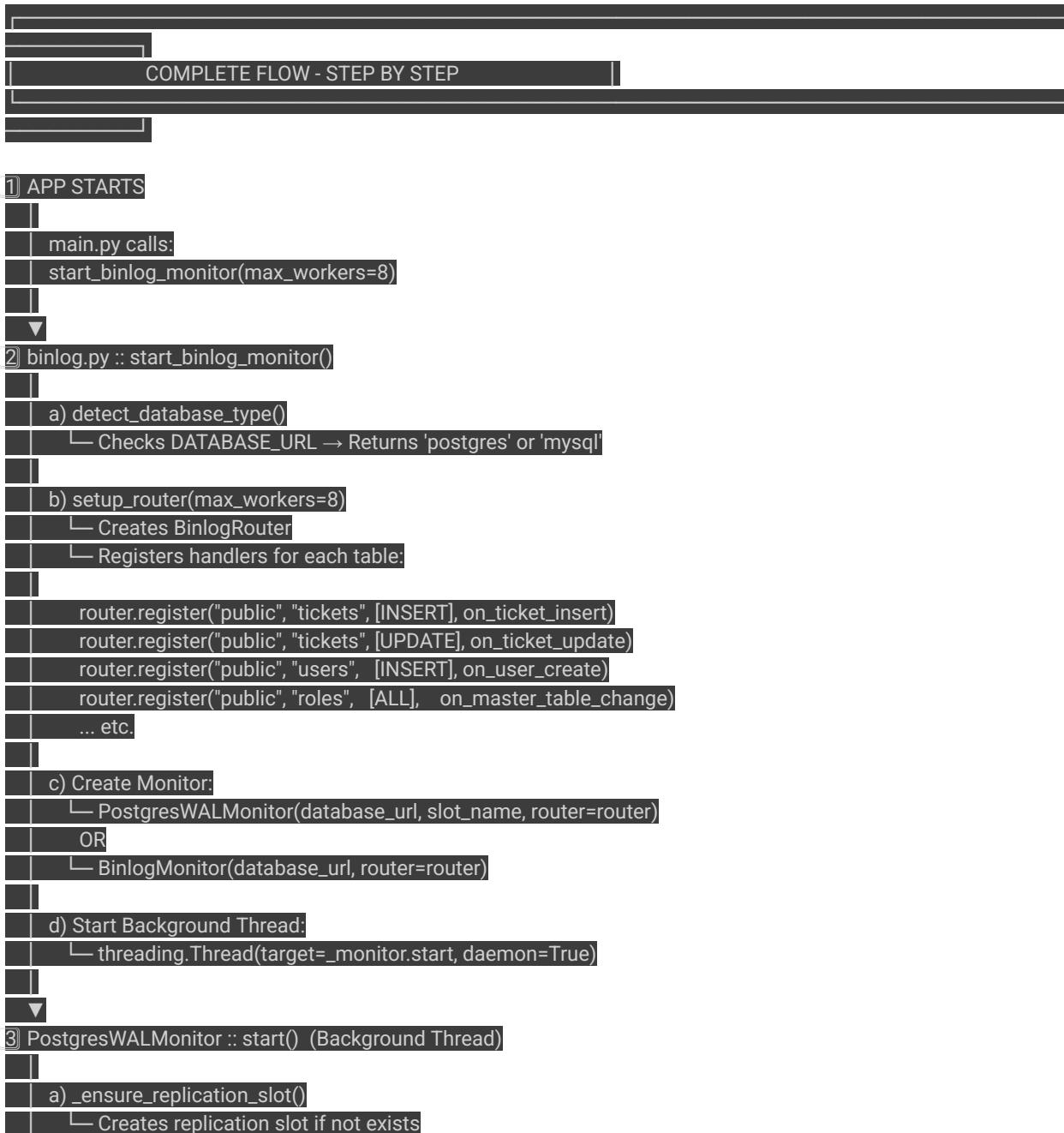
---

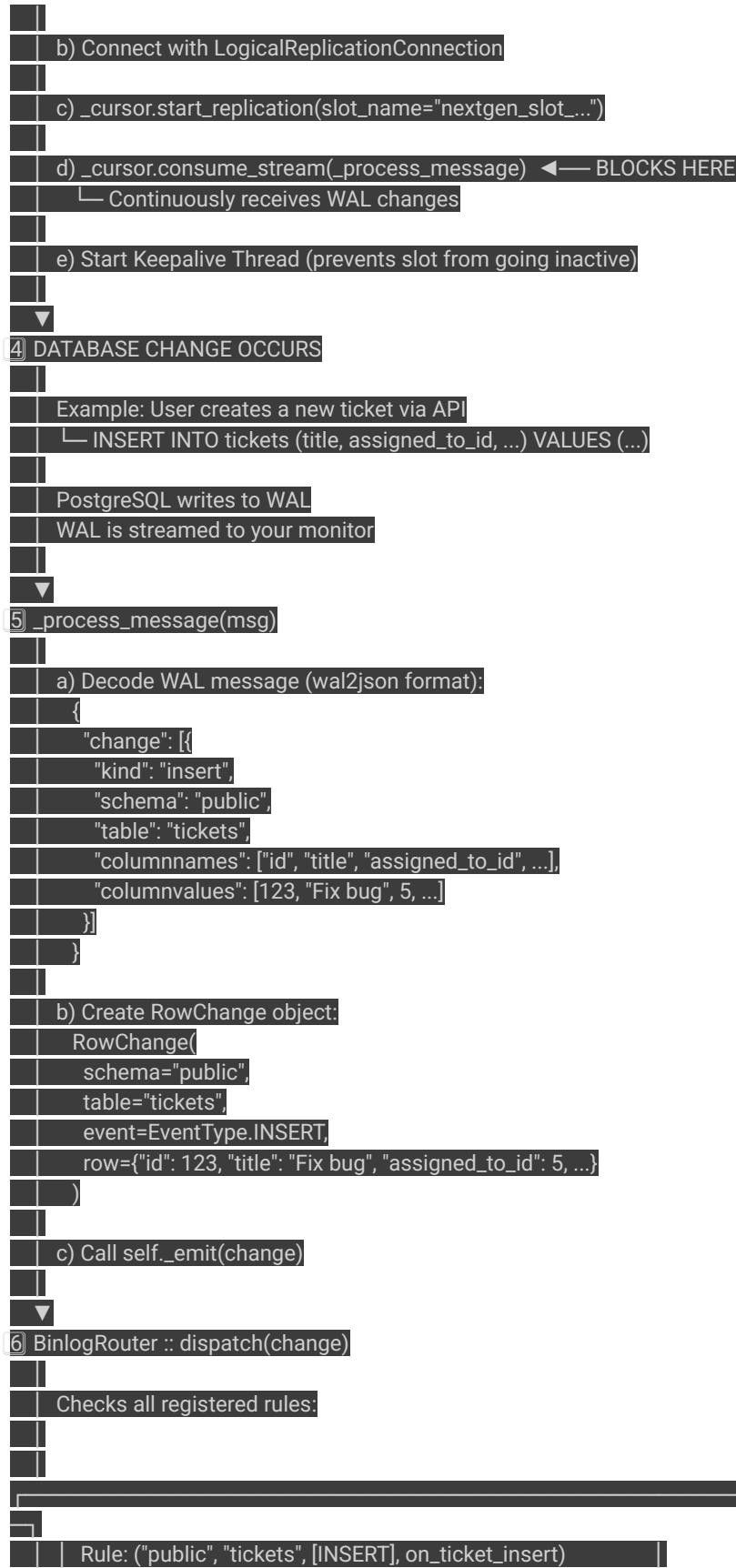
## File Relationships

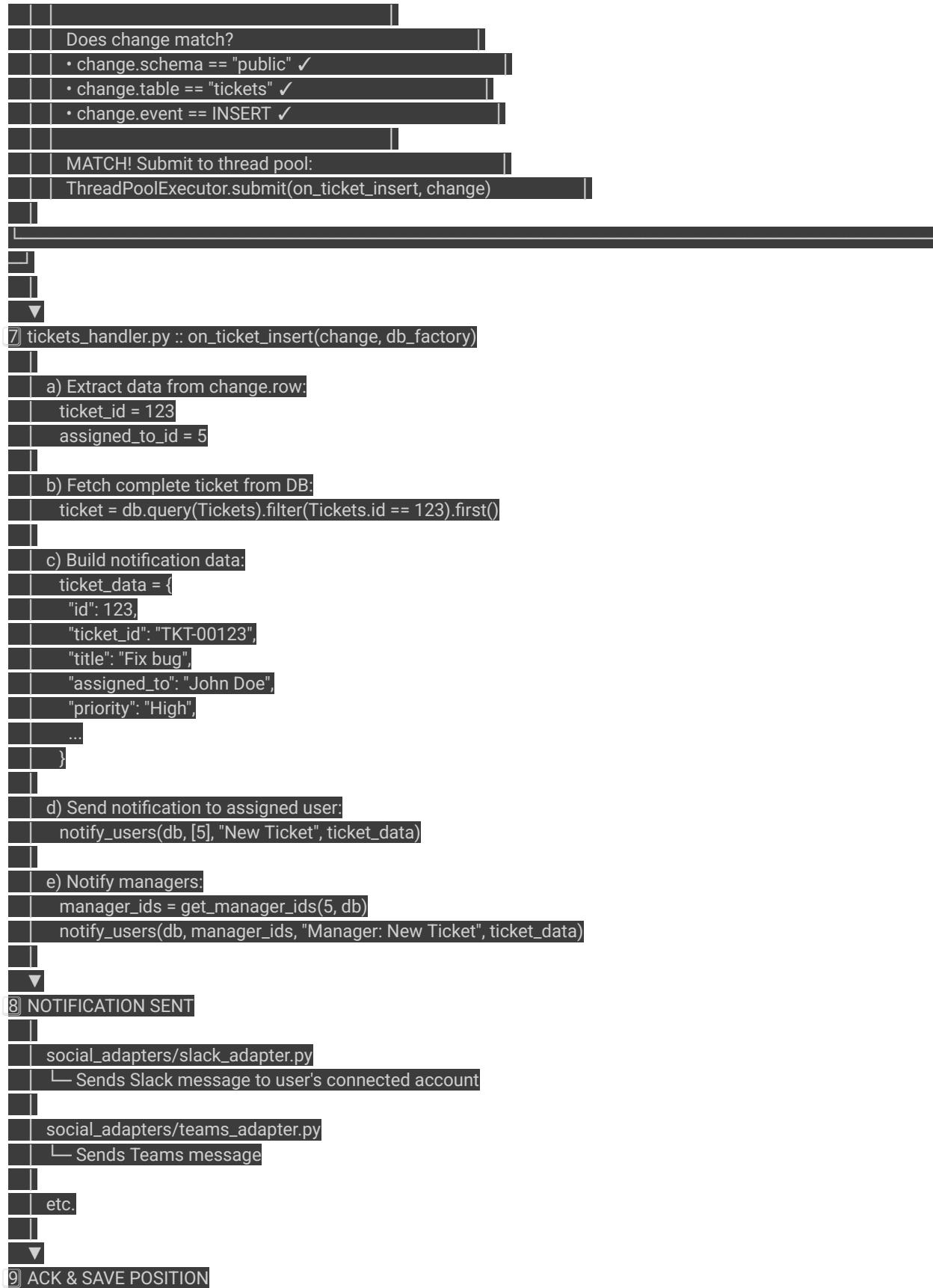


	(Same in both MySQL/Postgres monitors)

## Complete Flow - Step by Step







```

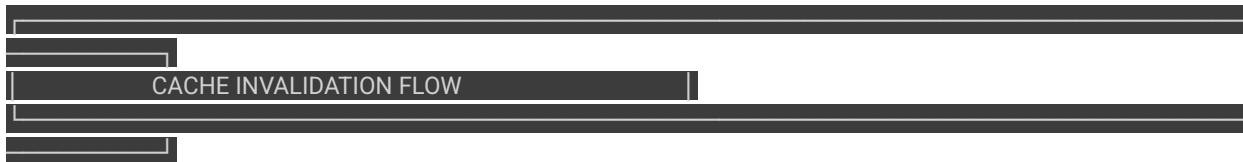
| Back in _process_message():
|   • msg.cursor.send_feedback(flush_lsn=msg.data_start)
|   • _save_offset(msg.data_start)

| This saves the position so if app restarts,
| it continues from where it left off
| ▼
10 LOOP CONTINUES
| _cursor.consume_stream() continues waiting for next change
| → Go back to step 4

```

---

## Cache Invalidation Flow

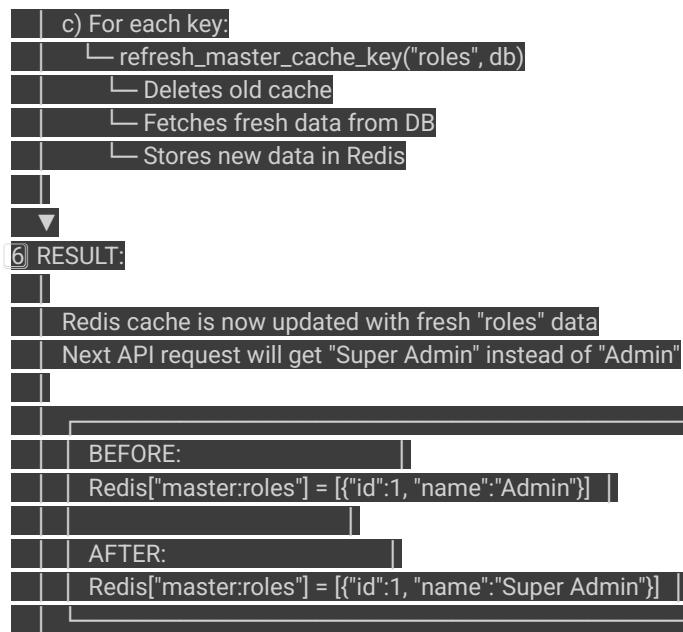


**SCENARIO:** Admin updates a "Role" in the database

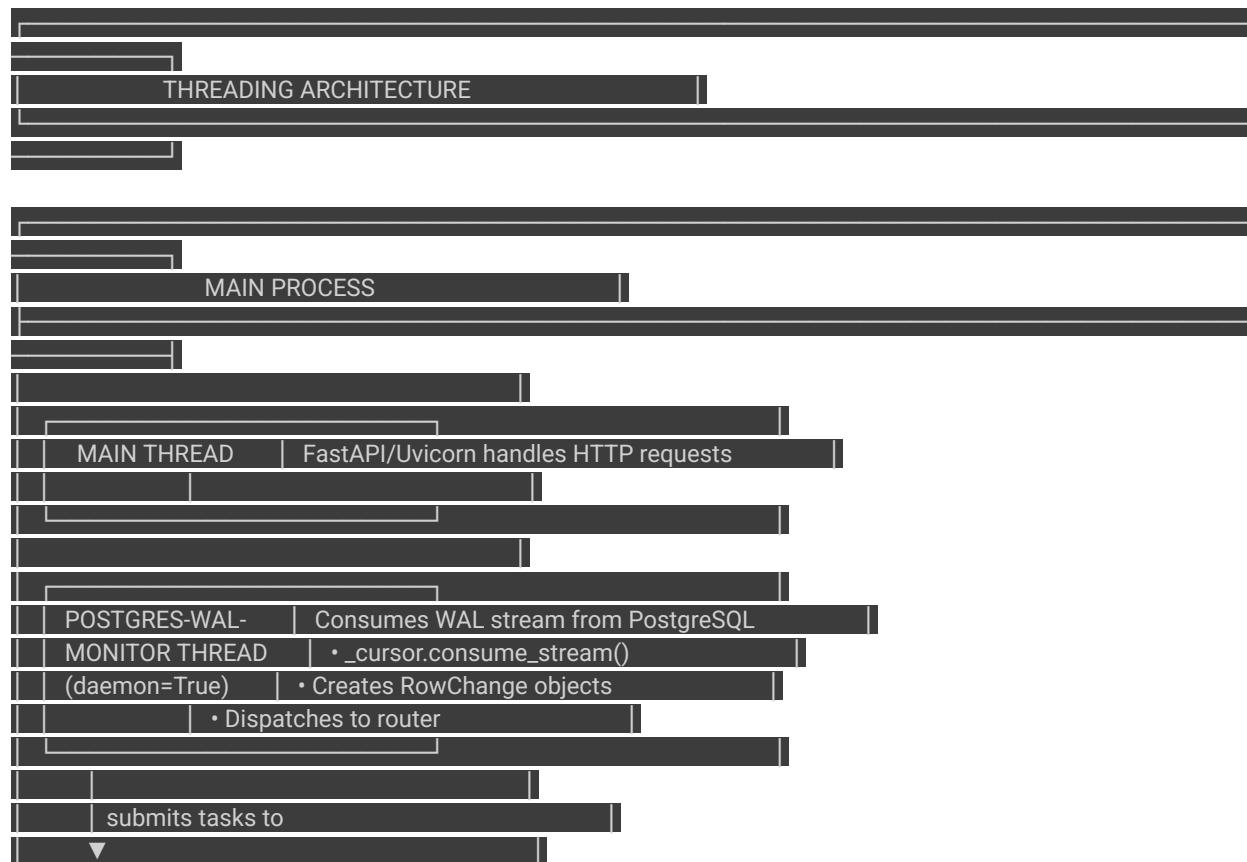
- 1** UPDATE roles SET name = 'Super Admin' WHERE id = 1
- 2** WAL captures the change
- 3** PostgresWALMonitor receives:  

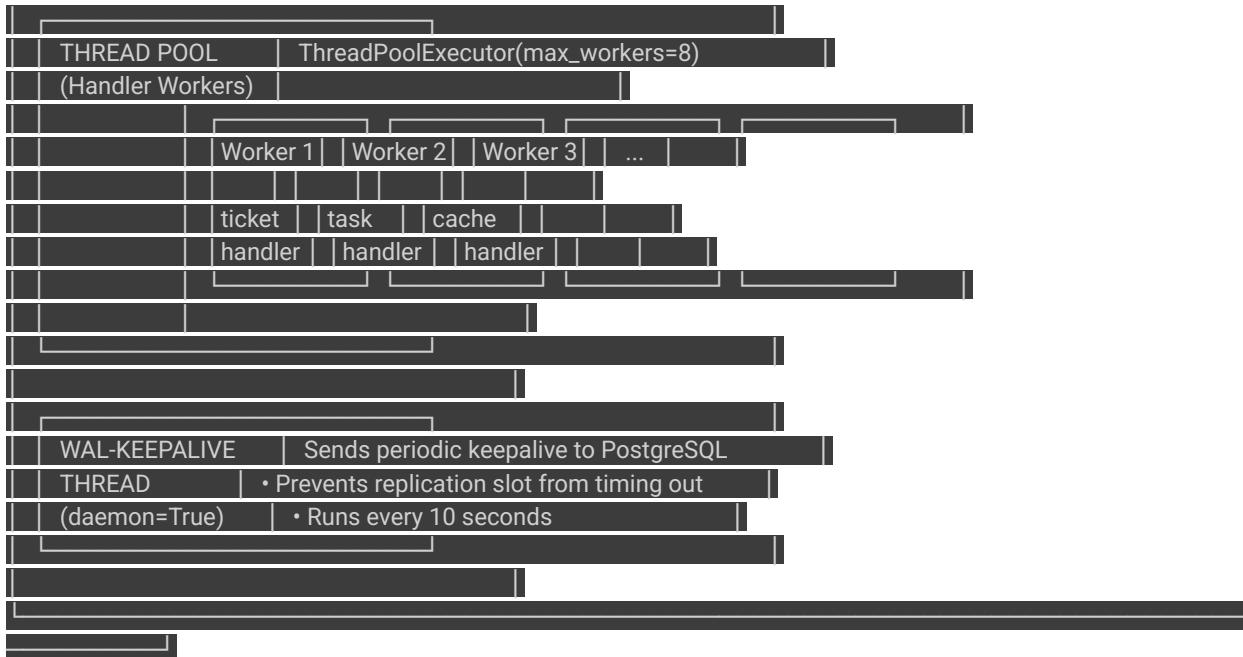
```
RowChange(
    schema="public",
    table="roles",
    event=EventType.UPDATE,
    before={"id": 1, "name": "Admin"},
    after={"id": 1, "name": "Super Admin"}
)
```
- 4** BinlogRouter matches rule:  

```
router.register("public", "roles", [UPDATE], on_master_table_change)
```
- 5** 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)



## Threading Architecture





## 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



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

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

**3** BINLOG ROUTER = Pattern-matching dispatcher  
• You register: (table, event\_type, handler\_function)  
• When change matches, handler is called

**4** HANDLERS = Functions that react to database changes  
• on\_ticket\_insert → Send notification to assignee  
• on\_master\_table\_change → Refresh Redis cache

**5** THREAD POOL = Parallel handler execution  
• max\_workers=8 means 8 handlers can run simultaneously  
• Prevents slow handler from blocking others

**6** 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.

---

Would you like me to explain any specific part in more detail?

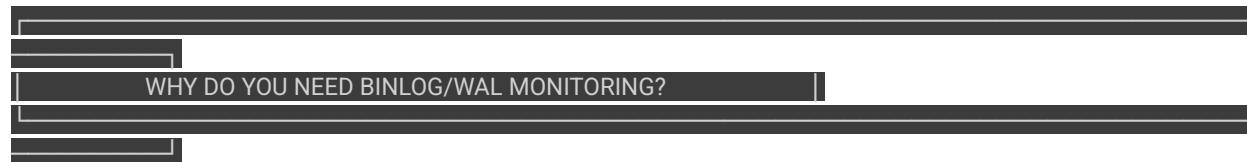
i want to know that if someone has to do this mean binloger functionality then how and from where he will start as this was already implemented so there was no issue but if someone who dont know the binloger from where he will start or how he will structure the code?

Excellent question! Let me give you a complete guide on [how to implement a binlog/WAL monitor from scratch](#) - as if you were starting a new project.

---

## Binlog/WAL Implementation Guide - From Zero to Hero

### Part 1: Understanding the Problem First



PROBLEM:

You want to react to database changes in real-time, but:

- Changes can come from ANYWHERE (API, admin panel, direct SQL, cron jobs)
- You can't add code to every place that modifies data
- You want a SINGLE source of truth for "what changed"

SOLUTION:

Monitor the database's transaction log (WAL/Binlog) - it captures ALL changes regardless of where they came from.

USE CASES:

- ✓ Send notifications when ticket is created/updated
- ✓ Invalidate cache when master data changes
- ✓ Sync data to external systems (Elasticsearch, analytics)
- ✓ Audit logging
- ✓ Real-time dashboards

---

## Part 2: Prerequisites & Setup

### STEP 1: DATABASE SETUP

FOR POSTGRESQL:

1. Enable logical replication in postgresql.conf:

```
wal_level = logical
max_replication_slots = 4
max_wal_senders = 4
```

2. Restart PostgreSQL

3. Install wal2json extension (for readable output):

```
$ sudo apt install postgresql-15-wal2json
```

Or compile from source:

```
$ git clone https://github.com/eulerto/wal2json.git
$ cd wal2json && make && make install
```

4. Create a user with replication privileges:

```
CREATE ROLE replication_user WITH REPLICATION LOGIN PASSWORD 'secret';
GRANT SELECT ON ALL TABLES IN SCHEMA public TO replication_user;
```

FOR MYSQL:

1. Enable binlog in my.cnf:

```
[mysqld]
log-bin = mysql-bin
binlog-format = ROW
binlog-row-image = FULL
server-id = 1
```

2. Restart MySQL

3. Create user with replication privileges:

```
CREATE USER 'replication_user'@'%' IDENTIFIED BY 'secret';
GRANT REPLICATION SLAVE, REPLICATION CLIENT, SELECT ON *.* TO 'replication_user'@'%';
```

---

## Part 3: Project Structure (Start Here!)

STEP 2: CREATE PROJECT STRUCTURE

Start with this folder structure:

```
my_project/
|
├── main.py          # App entry point
├── config/
│   └── settings.py  # Database URL, settings
|
├── database.py      # SQLAlchemy setup
|
├── binlog.py        # ★ ENTRY POINT for binlog monitoring
|
└── services/
    ├── __init__.py
    └── wal_monitor.py # ★ Core WAL/Binlog monitor class
|
└── binlog_handlers/ # ★ Your event handlers
    └── __init__.py
```

```
    |   ticket_handler.py  # Handle ticket changes
    |   user_handler.py  # Handle user changes
    |   cache_handler.py  # Handle cache invalidation
    |
└── requirements.txt
```

## Part 4: Step-by-Step Implementation

## Step 1: Install Dependencies

```
requirements.txt:  
  
# For PostgreSQL  
psycopg2-binary>=2.9.0  
  
# For MySQL (alternative)  
mysql-replication>=0.31  
  
# Common  
sqlalchemy>=2.0.0
```

## Step 2: Create Core Classes

```
# services/wal_monitor.py
=====
Core WAL/Binlog monitoring classes.
This is the FOUNDATION - build this first!
=====
import json
import logging
import threading
```

```
from enum import Enum
from dataclasses import dataclass
from typing import Any, Callable, Dict, List, Optional
from concurrent.futures import ThreadPoolExecutor

logger = logging.getLogger(__name__)
```

```
# =====#
# =====#
# STEP 4.1: Define Event Types
```

```
# =====#
# =====#
# =====#
class EventType(str, Enum):
    """Database operation types."""
    INSERT = "insert"
    UPDATE = "update"
    DELETE = "delete"
```

```
# =====#
# =====#
# =====#
# STEP 4.2: Define RowChange Data Class
```

```
# =====#
# =====#
# =====#
@dataclass
class RowChange:
    """
    Represents a single database row change.
    This is what your handlers will receive.
    """

    schema: str          # Database schema (e.g., "public")
    table: str           # Table name (e.g., "tickets")
    event: EventType     # INSERT, UPDATE, or DELETE
    row: Optional[Dict[str, Any]] = None  # Row data for INSERT/DELETE
    before: Optional[Dict[str, Any]] = None # Old values for UPDATE
    after: Optional[Dict[str, Any]] = None  # New values for UPDATE
```

```
def get_id(self) -> Optional[int]:
    """Helper to get the row ID."""
    data = self.row or self.after or self.before
    return data.get("id") if data else None
```

```
# =====#
# =====#
# =====#
# STEP 4.3: Define Router (Dispatcher)
```

```
#  
=====  
  
class EventRouter:  
    """  
    Routes database change events to appropriate handlers.  
    Think of it as a "switchboard" - directs events to the right handler.  
    """  
  
    def __init__(self, db_session_factory: Callable, max_workers: int = 4):  
        """  
        Args:  
            db_session_factory: Function that creates a new DB session  
            max_workers: Number of parallel handler threads  
        """  
        self._rules: List[tuple] = []  
        self._pool = ThreadPoolExecutor(max_workers=max_workers)  
        self._db_session_factory = db_session_factory  
  
    def register(  
        self,  
        schema: str,  
        table: str,  
        events: List[EventType],  
        handler: Callable[[RowChange, Callable], None]  
    ) -> None:  
        """  
        Register a handler for specific table/event combinations.  
  
        Example:  
            router.register("public", "tickets", [EventType.INSERT], on_ticket_insert)  
        """  
        self._rules.append((schema, table, events, handler))  
        logger.info(f"Registered handler for {schema}.{table} on {events}")  
  
    def dispatch(self, change: RowChange) -> None:  
        """  
        Route a change event to all matching handlers.  
        Runs handlers in thread pool for parallel execution.  
        """  
        for schema, table, events, handler in self._rules:  
            # Check if this change matches the rule  
            if (change.schema == schema and  
                change.table == table and  
                change.event in events):  
                # Submit to thread pool (non-blocking)  
                self._pool.submit(self._safe_call, handler, change)  
  
    def _safe_call(self, handler: Callable, change: RowChange) -> None:  
        """Execute handler with error isolation."""  
        try:  
            handler(change, self._db_session_factory)
```

```

    except Exception as e:
        logger.exception(f"Handler error for {change.table}: {e}")

    def shutdown(self) -> None:
        """Shutdown the thread pool."""
        self._pool.shutdown(wait=True)

# =====#
# STEP 4.4: Define WAL Monitor (PostgreSQL)
# =====#
# class PostgresWALMonitor:
#     """
#         Connects to PostgreSQL logical replication and streams changes.
#         This is the "engine" that reads the database changes.
#     """

def __init__(
    self,
    host: str,
    port: int,
    user: str,
    password: str,
    database: str,
    slot_name: str = "my_app_slot",
    router: Optional[EventRouter] = None
):
    self.conn_params = {
        "host": host,
        "port": port,
        "user": user,
        "password": password,
        "database": database
    }
    self.slot_name = slot_name
    self.router = router
    self.running = False
    self._conn = None
    self._cursor = None

def _ensure_slot_exists(self) -> None:
    """Create replication slot if it doesn't exist."""
    import psycopg2

    conn = psycopg2.connect(**self.conn_params)
    conn.autocommit = True
    cur = conn.cursor()

```

```

# Check if slot exists
cur.execute(
    "SELECT slot_name FROM pg_replication_slots WHERE slot_name = %s",
    (self.slot_name,),
)

if cur.fetchone() is None:
    # Create slot with wal2json plugin
    cur.execute(
        f"SELECT pg_create_logical_replication_slot('{self.slot_name}', 'wal2json')"
    )
    logger.info(f"Created replication slot: {self.slot_name}")

    cur.close()
    conn.close()

def start(self) -> None:
    """Start monitoring WAL stream. Call this in a background thread."""
    import psycopg2
    from psycopg2.extras import LogicalReplicationConnection

    self.running = True
    logger.info(f"Starting WAL monitor with slot: {self.slot_name}")

    # Ensure slot exists
    self._ensure_slot_exists()

    # Connect with replication protocol
    self._conn = psycopg2.connect(
        connection_factory=LogicalReplicationConnection,
        **self.conn_params
    )
    self._cursor = self._conn.cursor()

    # Start consuming the replication stream
    self._cursor.start_replication(
        slot_name=self.slot_name,
        decode=True,
        options={"format-version": "1"}
    )

    # This blocks and calls _process_message for each WAL entry
    self._cursor.consume_stream(self._process_message)

def _process_message(self, msg) -> None:
    """Process a single WAL message."""
    if not self.running:
        raise StopIteration("Monitor stopped")

    try:
        # Parse wal2json message
        payload = json.loads(msg.payload)

```

```

for change_data in payload.get("change", []):
    # Convert to RowChange object
    change = self._parse_change(change_data)
    if change and self.router:
        self.router.dispatch(change)

    # Acknowledge the message
    msg.cursor.send_feedback(flush_lsn=msg.data_start)

except Exception as e:
    logger.exception(f"Error processing WAL message: {e}")

def _parse_change(self, data: Dict) -> Optional[RowChange]:
    """Convert wal2json data to RowChange object"""
    kind = data.get("kind")
    schema = data.get("schema")
    table = data.get("table")

    if not all([kind, schema, table]):
        return None

    # Build row data from columns
    columns = data.get("columnnames", [])
    values = data.get("columnvalues", [])
    row_data = dict(zip(columns, values))

    if kind == "insert":
        return RowChange(
            schema=schema,
            table=table,
            event=EventType.INSERT,
            row=row_data
        )
    elif kind == "delete":
        return RowChange(
            schema=schema,
            table=table,
            event=EventType.DELETE,
            row=row_data
        )
    elif kind == "update":
        # For UPDATE, get old values too
        old_cols = data.get("oldkeys", {}).get("keynames", [])
        old_vals = data.get("oldkeys", {}).get("keyvalues", [])
        before_data = dict(zip(old_cols, old_vals))

        return RowChange(
            schema=schema,
            table=table,
            event=EventType.UPDATE,
            before=before_data,

```

```

        after=row_data
    )

    return None

def stop(self) -> None:
    """Stop the monitor."""
    self.running = False
    if self._cursor:
        self._cursor.close()
    if self._conn:
        self._conn.close()
    if self.router:
        self.router.shutdown()
    logger.info("WAL monitor stopped")

```

### Step 3: Create Entry Point



```

# binlog.py
"""
Entry point for database change monitoring.

This file:
1. Creates the router
2. Registers all handlers
3. Starts the monitor in a background thread
"""

import threading
import logging
from typing import Optional

from config.settings import settings
from database import SessionLocal
from services.wal_monitor import PostgresWALMonitor, EventRouter, EventType

# Import your handlers
from services.binlog_handlers.ticket_handler import on_ticket_insert, on_ticket_update
from services.binlog_handlers.user_handler import on_user_create
from services.binlog_handlers.cache_handler import on_cache_table_change

logger = logging.getLogger(__name__)

```

```
# Global state (singleton pattern)
_monitor: Optional[PostgresWALMonitor] = None
_thread: Optional[threading.Thread] = None

def db_session_factory():
    """Creates a new database session for handlers."""
    return SessionLocal()

def setup_router() -> EventRouter:
    """
    Setup the event router with all handlers.

    THIS IS WHERE YOU REGISTER YOUR HANDLERS!
    Add new handlers here when you want to react to new tables.
    """

    router = EventRouter(db_session_factory, max_workers=4)

    #
    # TICKET HANDLERS
    #

    router.register("public", "tickets", [EventType.INSERT], on_ticket_insert)
    router.register("public", "tickets", [EventType.UPDATE], on_ticket_update)

    #
    # USER HANDLERS
    #

    router.register("public", "users", [EventType.INSERT], on_user_create)

    #
    # CACHE INVALIDATION HANDLERS
    # React to changes in "master" tables and refresh cache
    #

    cache_tables = ["roles", "companies", "departments", "priorities"]
    for table in cache_tables:
        router.register(
            "public",
            table,
            [EventType.INSERT, EventType.UPDATE, EventType.DELETE],
```

```

        on_cache_table_change
    )

    return router

def start_binlog_monitor() -> None:
    """
    Start the database change monitor in a background thread.
    Call this once when your app starts.
    """
    global _monitor, _thread

    if _monitor is not None:
        logger.info("Monitor already running, skipping...")
        return

    # Setup router with handlers
    router = setup_router()

    # Create monitor
    _monitor = PostgresWALMonitor(
        host=settings.DB_HOST,
        port=settings.DB_PORT,
        user=settings.DB_USER,
        password=settings.DB_PASSWORD,
        database=settings.DB_DATABASE,
        slot_name="my_app_slot",
        router=router
    )

    # Start in background thread (daemon=True so it dies with main app)
    _thread = threading.Thread(
        target=_monitor.start,
        daemon=True,
        name="wal-monitor"
    )
    _thread.start()

    logger.info("Database change monitor started in background")

def stop_binlog_monitor() -> None:
    """
    Stop the monitor (for graceful shutdown).
    """
    global _monitor, _thread

    if _monitor:
        _monitor.stop()
        _monitor = None
        _thread = None
    logger.info("Database change monitor stopped")

```

## Step 4: Create Handlers

```
|  
|  
| STEP 6: CREATE HANDLERS |  
| File: services/binlog_handlers/ticket_handler.py |  
|  
|
```

```
# services/binlog_handlers/ticket_handler.py  
=====  
Handlers for ticket table changes.  
Each handler receives:  
- change: RowChange object with the change data  
- db_factory: Function to create a DB session  
=====
```

```
import logging  
from typing import Callable  
from sqlalchemy.orm import Session  
  
from services.wal_monitor import RowChange
```

```
logger = logging.getLogger(__name__)  
  
def on_ticket_insert(change: RowChange, db_factory: Callable[[], Session]) -> None:  
    """  
    Handle new ticket creation.
```

```
    Example: Send notification to assigned user  
    =====  
    db = db_factory()  
    try:  
        # Get data from the change  
        row = change.row or {}  
        ticket_id = row.get("id")  
        assigned_to_id = row.get("assigned_to_id")  
        title = row.get("title")  
  
        if not ticket_id or not assigned_to_id:  
            return
```

```
        logger.info(f"New ticket created: #{ticket_id} - {title}")
```

```
    # =====  
    # YOUR LOGIC HERE!  
    # Examples:  
    # - Send email notification
```

```

# - Send Slack message
# - Push to websocket
# - Update analytics
# =====

# Example: Send notification (pseudo-code)
# send_notification(
#   user_id=assigned_to_id,
#   title="New Ticket Assigned",
#   message=f"Ticket #{ticket_id}: {title}"
# )

except Exception as e:
    logger.error(f"Error in on_ticket_insert: {e}")
finally:
    db.close()

def on_ticket_update(change: RowChange, db_factory: Callable[[], Session]) -> None:
    """
    Handle ticket updates.

    Example: Notify if status or assignee changed
    """
    db = db_factory()
    try:
        before = change.before or {}
        after = change.after or {}
        ticket_id = after.get("id")

        if not ticket_id:
            return

        # Check what changed
        old_status = before.get("status_id")
        new_status = after.get("status_id")

        old_assignee = before.get("assigned_to_id")
        new_assignee = after.get("assigned_to_id")

        # Status changed?
        if old_status != new_status:
            logger.info(f"Ticket #{ticket_id} status changed: {old_status} → {new_status}")
            # Send status change notification...

        # Assignee changed?
        if old_assignee != new_assignee:
            logger.info(f"Ticket #{ticket_id} reassigned: {old_assignee} → {new_assignee}")
            # Notify old and new assignee...

    except Exception as e:
        logger.error(f"Error in on_ticket_update: {e}")

```

```
    finally:  
        db.close()
```

```
# services/binlog_handlers/cache_handler.py  
=====  
Handler for cache invalidation.  
When master tables change, invalidate/refresh the cache.  
=====
```

```
import logging  
from typing import Callable  
from sqlalchemy.orm import Session  
  
from services.wal_monitor import RowChange  
  
logger = logging.getLogger(__name__)
```

```
def on_cache_table_change(change: RowChange, db_factory: Callable[[], Session]) -> None:  
    """  
    Handle changes to tables that are cached.  
    Invalidate/refresh the cache when these tables change.  
    """  
    db = db_factory()  
    try:  
        table = change.table  
        event = change.event.value  
  
        logger.info(f"Cache table changed: {table} ({event})")
```

```
# ======  
# YOUR CACHE LOGIC HERE!  
# Examples:  
# - Delete Redis key  
# - Refresh cache with new data  
# ======
```

```
# Example (pseudo-code):  
# cache_key = f"master:{table}"  
# redis_client.delete(cache_key)  
#  
# Or refresh:  
# fresh_data = db.query(Model).all()  
# redis_client.set(cache_key, serialize(fresh_data))
```

```
except Exception as e:  
    logger.error(f"Error in on_cache_table_change: {e}")  
finally:  
    db.close()
```

## Step 5: Start on App Launch

STEP 7: START ON APP LAUNCH

File: main.py

```
# main.py
from fastapi import FastAPI
from contextlib import asynccontextmanager

from binlog import start_binlog_monitor, stop_binlog_monitor
```

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    """
    App lifespan - runs on startup and shutdown.
    #

```

# STARTUP

#

```
    print("Starting binlog monitor...")
    start_binlog_monitor()
```

```
    yield # App is running
```

#

# SHUTDOWN

#

```
    print("Stopping binlog monitor...")
    stop_binlog_monitor()
```

```
app = FastAPI(lifespan=lifespan)
```

```
@app.get("/")
def root():
    return {"status": "ok"}
```

---

## Part 5: Complete Roadmap Summary



### PHASE 1: PREPARATION

- 
- A horizontal bar divided into four segments. The first two segments are filled black, while the third and fourth are white.
- Step 1: Configure database for logical replication
  - Step 2: Install wal2json extension (PostgreSQL) or enable binlog (MySQL)
  - Step 3: Create replication user with proper privileges

### PHASE 2: CORE INFRASTRUCTURE

- 
- A horizontal bar divided into five segments. The first two segments are filled black, while the remaining three are white.
- Step 4: Create project structure
  - Step 5: Define core classes:
    - EventType (enum)
    - RowChange (data class)
    - EventRouter (dispatcher)
    - PostgresWALMonitor (or MySQLBinlogMonitor)

### PHASE 3: ENTRY POINT

- 
- A horizontal bar divided into four segments. The first two segments are filled black, while the last two are white.
- Step 6: Create binlog.py:
    - setup\_router() - register all handlers
    - start\_binlog\_monitor() - start background thread
    - stop\_binlog\_monitor() - graceful shutdown

### PHASE 4: HANDLERS

- 
- A horizontal bar divided into four segments. The first two segments are filled black, while the last two are white.
- Step 7: Create handler files for each domain:
    - ticket\_handler.py
    - user\_handler.py
    - cache\_handler.py
    - (add more as needed)

### PHASE 5: INTEGRATION

- 
- A horizontal bar divided into four segments. The first two segments are filled black, while the last two are white.
- Step 8: Start monitor in main.py (app lifespan)
  - Step 9: Test with real database changes
  - Step 10: Add error handling, logging, monitoring

## PHASE 6: PRODUCTION READINESS

---

- Step 11: Add offset saving (resume from last position)
  - Step 12: Add reconnection logic (handle connection drops)
  - Step 13: Add keepalive thread (prevent slot timeout)
  - Step 14: Add metrics/monitoring
- 

## Part 6: How to Add a New Handler (Day-to-Day Usage)

---



SCENARIO: You want to send notifications when a new "Task" is created

### STEP 1: Create handler function

---

File: services/binlog\_handlers/task\_handler.py

```
def on_task_create(change: RowChange, db_factory: Callable) -> None:  
    db = db_factory()  
    try:  
        task_id = change.row.get("id")  
        assigned_to = change.row.get("assigned_to_id")  
  
        # Your notification logic here  
        send_notification(assigned_to, f"New task #{task_id}")  
    finally:  
        db.close()
```

### STEP 2: Register handler in binlog.py

---

File: binlog.py

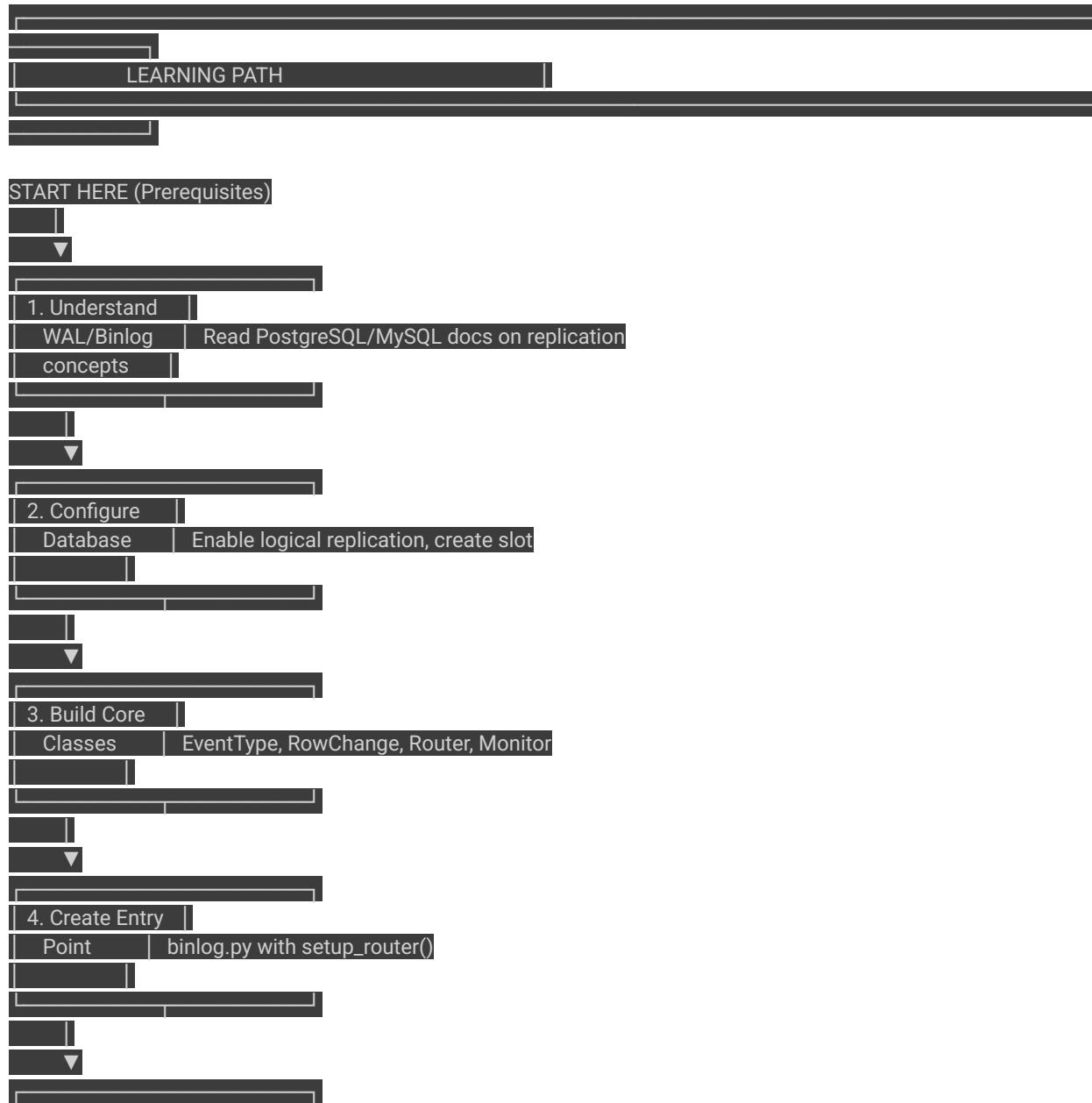
```
from services.binlog_handlers.task_handler import on_task_create  
  
def setup_router() -> EventRouter:  
    router = EventRouter(...)  
  
    # ... existing handlers ...  
  
    # ADD THIS LINE:  
    router.register("public", "tasks", [EventType.INSERT], on_task_create)
```

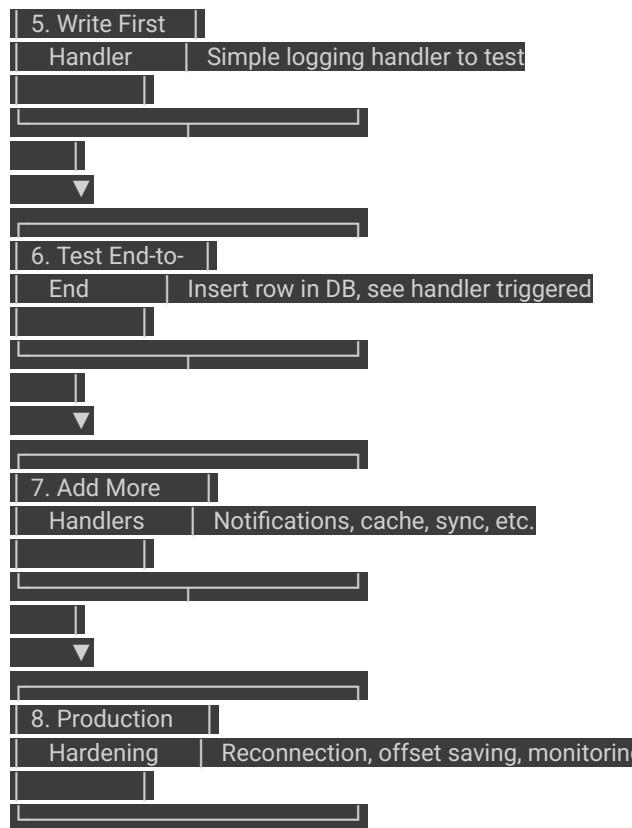
```
return router
```

STEP 3: Done! Restart app and test

Any INSERT to "tasks" table will now trigger your handler.

## Part 7: Visual Learning Path





## Quick Reference Card



### FILE PURPOSES:

<code>binlog.py</code>	→ Entry point, registers handlers, starts monitor
<code>services/wal_monitor.py</code>	→ Core classes (Monitor, Router, RowChange)
<code>services/binlog_handlers/*</code>	→ Your business logic (handlers)

### KEY CLASSES:

<code>EventType</code>	→ INSERT, UPDATE, DELETE
<code>RowChange</code>	→ Contains change data (table, before, after)
<code>EventRouter</code>	→ Matches changes to handlers
<code>PostgresWALMonitor</code>	→ Reads WAL stream from PostgreSQL

### HANDLER SIGNATURE:

```
=====
```

```
def my_handler(change: RowChange, db_factory: Callable[[], Session]) -> None:
    db = db_factory()
    try:
        # Your logic here
        # change.table - table name
        # change.event - INSERT/UPDATE/DELETE
        # change.row - row data (INSERT/DELETE)
        # change.before/after - for UPDATE
    finally:
        db.close()
```

REGISTER HANDLER:

```
=====
```

```
router.register(
    "public",          # schema name
    "tickets",         # table name
    [EventType.INSERT], # event types to listen for
    on_ticket_insert   # handler function
)
```

---

Would you like me to explain any specific part in more detail, or show you how to implement something specific like MySQL binlog monitoring or error handling?

i want to know the track of my binlog code as it is started from binlog.py file where our binlogrouter are set and handlers ,and these are defned in servces folder where those again calling other functions or classes or dtypes ,actually my still same issue is existing that i dont know fully or always confused that how i should defined a class or a function by keeping in mind what?

This is a **very important question** - you're asking about **software design thinking**. Let me break down your existing binlog code and teach you **HOW to think** when designing classes and functions.

---

## Understanding Code Design Through Your Binlog Example

### Part 1: Let's Trace Your Binlog Code First

```
=====
|           YOUR BINLOG CODE TRACE           |
=====
```

START: binlog.py

```
=====
```



## Part 2: The Design Questions You Should Ask

When you're confused about whether to create a **class** or **function**, ask these questions:



## THE 5 KEY QUESTIONS

? QUESTION 1: Does it need to REMEMBER something?

YES → Use a CLASS (has state/data to store)

NO → Use a FUNCTION (just does something and returns)

EXAMPLES FROM YOUR CODE:

PostgresWALMonitor → CLASS

WHY? Needs to remember:

- self.\_conn (connection)
- self.\_cursor (cursor)
- self.running (state flag)
- self.router (reference to router)
- self.slot\_name (configuration)

detect\_database\_type() → FUNCTION

WHY? Just checks URL and returns 'mysql' or 'postgres'

Doesn't need to remember anything

? QUESTION 2: Does it have a LIFECYCLE (start/stop, open/close)?

YES → Use a CLASS

NO → Use a FUNCTION

EXAMPLES:

PostgresWALMonitor → CLASS

WHY? Has lifecycle:

- start() - begins monitoring
- stop() - ends monitoring
- \_cleanup() - releases resources

on\_ticket\_insert() → FUNCTION

WHY? Just runs once when called, no lifecycle

**?** QUESTION 3: Will there be MULTIPLE INSTANCES with different data?

YES → Use a CLASS

NO → Could be either (depends on other factors)

EXAMPLES:

RowChange → CLASS (dataclass)  
WHY? Each database change creates a NEW RowChange with different:  
• table name  
• event type  
• row data  
You might have 100 RowChange objects at any time, each different

**?** QUESTION 4: Does it GROUP related operations together?

YES → Use a CLASS

NO → Use separate FUNCTIONS

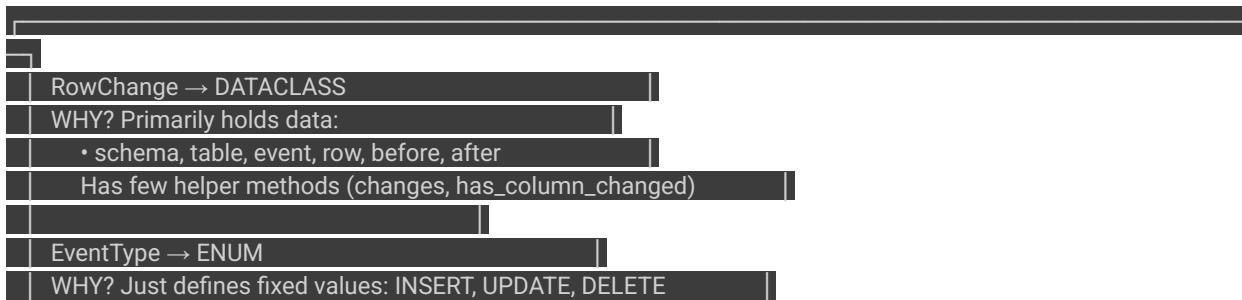
EXAMPLES:

BinlogRouter → CLASS  
WHY? Groups related operations:  
• register() - add a handler  
• dispatch() - route to handlers  
• shutdown() - cleanup  
All operate on same data (\_rules, \_pool)  
  
WALDecoder → CLASS  
WHY? Groups decoding operations:  
• decode\_wal2json()  
• decode\_test\_decoding()  
• \_decode\_v1(), \_decode\_v2()  
All share schema\_cache

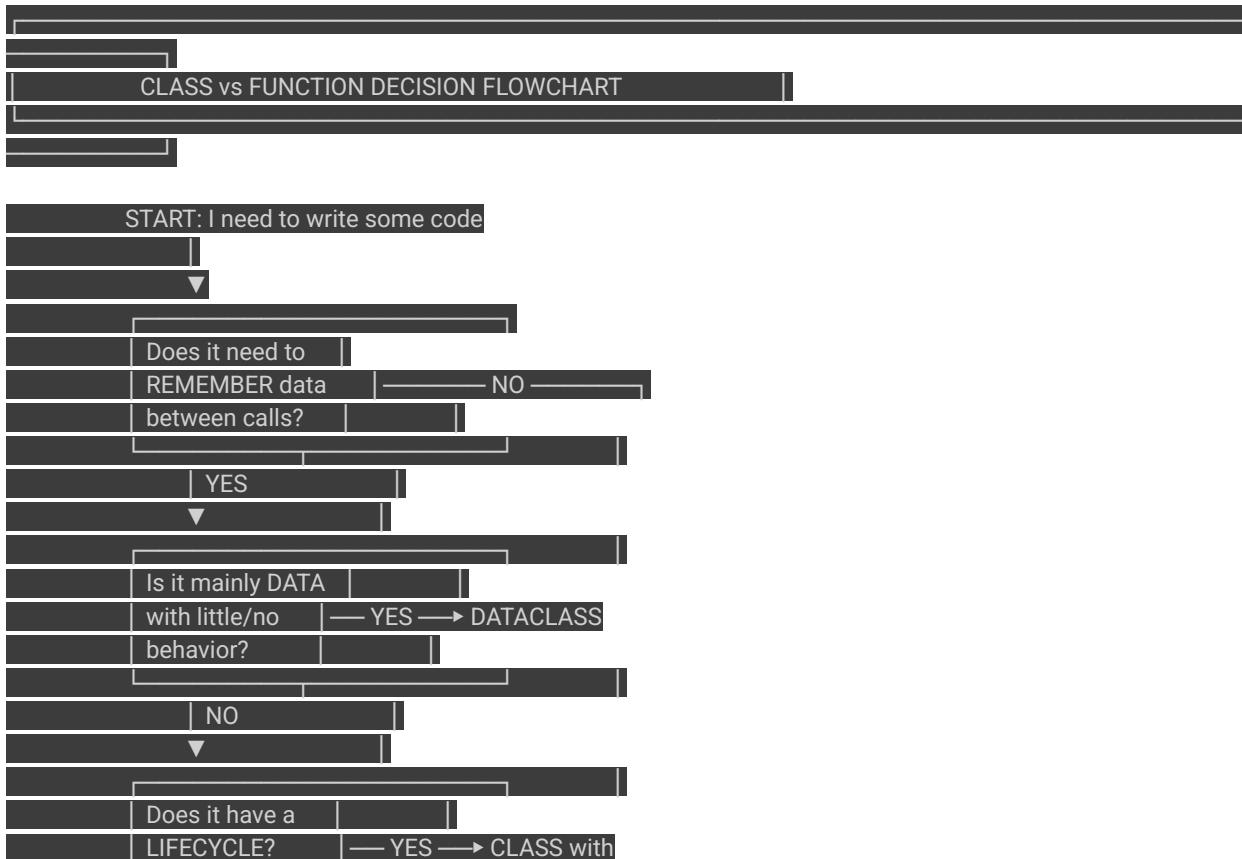
**?** QUESTION 5: Is it just DATA with no behavior?

YES → Use a DATACLASS or simple CLASS  
NO → Use a regular CLASS with methods

EXAMPLES:



## Part 3: Decision Flowchart





## Part 4: Your Binlog Code - Why Each Thing Exists

Let me explain **WHY** each class/function in your binlog code was designed that way:



### ① EventType (Enum)

```

class EventType(str, Enum):
    INSERT = "insert"
    UPDATE = "update"
    DELETE = "delete"
  
```

#### WHY ENUM?

- Fixed set of values that never changes
- Provides type safety (can't accidentally use "insrt")
- Self-documenting code

#### ALTERNATIVE (BAD):

- Using strings directly: "insert", "update", "delete"
- Problem: Easy to make typos, no autocomplete

### ② RowChange (Dataclass)

---

---

```
@dataclass
class RowChange:
    schema: str
    table: str
    event: EventType
    row: Optional[Dict] = None
    before: Optional[Dict] = None
    after: Optional[Dict] = None
```

#### WHY DATACLASS?

- Primarily holds DATA (schema, table, row data)
- Automatically gets `__init__`, `__repr__`, `__eq__`
- Clean, readable structure
- Has some helper methods (`changes`, `has_column_changed`)

#### WHY NOT DICT?

- Dict has no structure: `data["schmea"]` typo won't be caught
- No autocomplete in IDE
- Hard to understand what fields exist

### ③ BinlogRouter (Class)

---

---

```
class BinlogRouter:
    def __init__(self, db_session_factory, max_workers):
        self._rules = [] # REMEMBERS rules
        self._pool = ThreadPoolExecutor(max_workers) # REMEMBERS pool
        self._db_session_factory = db_session_factory # REMEMBERS factory

    def register(self, schema, table, events, handler): ...
    def dispatch(self, change): ...
    def shutdown(self): ...
```

#### WHY CLASS?

- ✓ REMEMBERS data: `_rules`, `_pool`, `_db_session_factory`
- ✓ GROUPS related operations: `register`, `dispatch`, `shutdown`
- ✓ Has LIFECYCLE: created → used → shutdown

#### ALTERNATIVE (BAD):

- Global variables + functions
- Problem: Hard to test, can't have multiple routers

#### 4 WALDecoder (Class)

```
class WALDecoder:  
    def __init__(self, schema_cache):  
        self.schema_cache = schema_cache # REMEMBERS cache  
  
    def decode_wal2json(self, payload): ...  
    def _decode_v1(self, payload): ...  
    def _decode_v2(self, payload): ...  
    def decode_test_decoding(self, message): ...
```

##### WHY CLASS?

- ✓ REMEMBERS: schema\_cache (shared across all decode calls)
- ✓ GROUPS related operations: all decode methods

##### WHY NOT FUNCTIONS?

- All decode methods need access to schema\_cache
- Would have to pass schema\_cache to every function call

#### 5 PostgresWALMonitor (Class)

```
class PostgresWALMonitor:  
    def __init__(self, database_url, slot_name, router, ...):  
        self.running = False # REMEMBERS state  
        self._conn = None # REMEMBERS connection  
        self._cursor = None # REMEMBERS cursor  
        self.router = router # REMEMBERS router  
        self.slot_name = slot_name # REMEMBERS config  
        self.decoder = WALDecoder(...) # REMEMBERS decoder  
  
    def start(self): ... # LIFECYCLE  
    def stop(self): ... # LIFECYCLE  
    def _cleanup(self): ... # LIFECYCLE  
    def _process_message(self, msg): ...
```

##### WHY CLASS?

- ✓ REMEMBERS lots of state: connection, cursor, running flag
- ✓ Has LIFECYCLE: start → running → stop
- ✓ GROUPS related operations: start, stop, process, cleanup

##### THIS IS THE MOST COMPLEX CLASS because:

- Manages a long-running connection
- Has multiple states (stopped, running, reconnecting)
- Coordinates multiple components (decoder, router)

## 6] on\_ticket\_insert (Function)

---

---

```
def on_ticket_insert(change: RowChange, db_factory: Callable) -> None:
    db = db_factory()
    try:
        # Process the change
        ticket_id = change.row.get("id")
        # Send notification...
    finally:
        db.close()
```

### WHY FUNCTION?

- ✗ Doesn't REMEMBER anything between calls
- ✗ No LIFECYCLE (just runs once)
- ✗ Single operation (not a group)
- ✓ One-time transformation: change → side effect (notification)

## 7] detect\_database\_type (Function)

---

---

```
def detect_database_type() -> str:
    db_url = DATABASE_URL.lower()
    if 'postgres' in db_url:
        return 'postgres'
    elif 'mysql' in db_url:
        return 'mysql'
    return 'postgres'
```

### WHY FUNCTION?

- ✗ Doesn't REMEMBER anything
- ✗ No LIFECYCLE
- ✓ Simple input → output transformation
- ✓ Stateless operation

## 8] start\_binlog\_monitor (Function)

---

---

```
_monitor = None # Global state
_thread = None # Global state

def start_binlog_monitor():
    global _monitor, _thread
    router = setup_router()
    _monitor = PostgresWALMonitor(...)
    _thread = threading.Thread(target=_monitor.start, daemon=True)
    _thread.start()
```

#### WHY FUNCTION (not class)?

- This is an "orchestrator" function
- Uses global variables for singleton pattern
- COULD be a class, but function is simpler for this use case

#### WHEN WOULD IT BE A CLASS?

- If you needed multiple monitors running
  - If you needed more complex lifecycle management
- 

## Part 5: Design Patterns Used in Your Code

### DESIGN PATTERNS IN YOUR BINLOG CODE

#### 1] SINGLETON PATTERN (binlog.py)

```
_monitor = None  
_thread = None  
  
def start_binlog_monitor():  
    global _monitor, _thread  
    if _monitor is not None:  
        return # Already running  
    _monitor = PostgresWALMonitor(...)
```

PURPOSE: Only ONE monitor should run at a time

HOW: Global variable + check before creating

#### 2] FACTORY PATTERN (db\_session\_factory)

```
def db_session_factory():  
    return SessionLocal()  
  
# Used by handlers:  
def on_ticket_insert(change, db_factory):  
    db = db_factory() # Creates new session  
    try:  
        # Use db  
    finally:  
        db.close()
```

PURPOSE: Handlers create their own DB sessions

HOW: Pass a function that creates sessions

### ③ OBSERVER/PUBLISH-SUBSCRIBE PATTERN (Router)

---

---

```
# Register (subscribe)
router.register("public", "tickets", [INSERT], on_ticket_insert)
```

```
# Dispatch (publish)
router.dispatch(change) # Notifies all matching handlers
```

PURPOSE: Decouple change detection from handling

HOW: Register handlers, dispatch calls all matching ones

### ④ STRATEGY PATTERN (Database Type)

---

---

```
if db_type == 'mysql':
    from services.binlog_monitor import BinlogMonitor
    _monitor = BinlogMonitor(...)
else:
    from services.postgres_wal_monitor import PostgresWALMonitor
    _monitor = PostgresWALMonitor(...)
```

PURPOSE: Same interface, different implementations

HOW: Both monitors have start(), stop(), same behavior

### ⑤ TEMPLATE METHOD PATTERN (Handlers)

---

---

```
# All handlers follow same template:
def on_something(change, db_factory):
    db = db_factory()
    try:
        # 1. Extract data from change
        # 2. Do business logic
        # 3. Side effects (notify, cache, etc.)
    except Exception as e:
        logger.error(...)
    finally:
        db.close()
```

PURPOSE: Consistent structure across handlers

HOW: Same signature, same try/finally pattern

---

## Part 6: How to Think When Designing NEW Code



Let's say you need to add: "Email notification system for database changes"

### STEP 1: IDENTIFY THE RESPONSIBILITIES

What does this system need to do?

- Connect to email server (SMTP)
- Format email content
- Send emails
- Track sent emails (optional)
- Handle failures/retries

Write them down as a list!

### STEP 2: GROUP RELATED RESPONSIBILITIES

Group 1: Email Server Connection

- Connect to SMTP
- Disconnect
- Check connection status

Group 2: Email Formatting

- Format ticket notification email
- Format user notification email
- Generate HTML body

Group 3: Email Sending

- Send single email
- Send batch emails

Group 4: Each group might become a CLASS

### STEP 3: ASK THE 5 QUESTIONS FOR EACH GROUP

Group 1: Email Server Connection

- Q1: Remember something? YES (connection, config)  
Q2: Lifecycle? YES (connect/disconnect)  
→ DECISION: CLASS (EmailConnection or SMTPClient)

Group 2: Email Formatting

```
Q1: Remember something? NO (just transforms data)
Q2: Lifecycle? NO
→ DECISION: FUNCTIONS (format_ticket_email, format_user_email)
→ OR: CLASS if they share templates/config
```

### Group 3: Email Sending

```
Q1: Remember something? YES (connection, queue)
Q2: Lifecycle? YES (start/stop for background sending)
→ DECISION: CLASS (EmailSender)
```

## STEP 4: DESIGN THE INTERFACES

---

---

Before writing code, define WHAT each class/function does:

```
class EmailSender:
    def __init__(self, smtp_host, smtp_port, ...): ...
    def connect(self) -> None: ...
    def disconnect(self) -> None: ...
    def send(self, to: str, subject: str, body: str) -> bool: ...
    def send_async(self, to: str, subject: str, body: str) -> None: ...

    def format_ticket_email(ticket_data: dict) -> tuple[str, str]:
        """Returns (subject, body)"""
        ...
        ...

    def format_user_email(user_data: dict) -> tuple[str, str]:
        """Returns (subject, body)"""
        ...
        ...
```

## STEP 5: IMPLEMENT

---

---

Now write the actual code!

---

---

## Part 7: Practical Exercise

Let me show you how to think through a real example:

```
|_
|_
|_ EXERCISE: Design a Cache System |_
|_
|_
```

REQUIREMENT: Cache database query results in Redis

## STEP 1: What does it need to do?

- Connect to Redis
- Store data with key
- Retrieve data by key
- Delete data by key
- Set expiration time
- Handle connection failures

## STEP 2: Ask the questions

"Connect to Redis"

- Q1: Remember something? YES (connection)
  - Q2: Lifecycle? YES (connect/disconnect)
- CLASS: RedisClient

"Store/Retrieve/Delete data"

- Q1: Remember something? YES (uses RedisClient)
  - Q2: Lifecycle? Not really
  - Q3: Group related operations? YES
- CLASS: CacheService (uses RedisClient)

"Set expiration"

- Part of CacheService.set() method

## STEP 3: Design interface

```
class RedisClient:  
    def __init__(self, host, port, db): ...  
    def connect(self) -> None: ...  
    def disconnect(self) -> None: ...  
    def get(self, key: str) -> Optional[str]: ...  
    def set(self, key: str, value: str, ttl: int = None) -> None: ...  
    def delete(self, key: str) -> None: ...
```

```
class CacheService:  
    def __init__(self, redis_client: RedisClient):  
        self.redis = redis_client  
  
    def get_cached(self, key: str) -> Optional[Any]:  
        data = self.redis.get(key)  
        return json.loads(data) if data else None
```

```
    def cache(self, key: str, data: Any, ttl: int = 3600) -> None:  
        self.redis.set(key, json.dumps(data), ttl)
```

```
    def invalidate(self, key: str) -> None:  
        self.redis.delete(key)
```

## STEP 4: Usage

```
=====
```

```
# Setup
redis = RedisClient(host="localhost", port=6379, db=0)
redis.connect()
cache = CacheService(redis)

# Use
data = cache.get_cached("users:all")
if not data:
    data = db.query(User).all()
    cache.cache("users:all", data, ttl=3600)
```

---

## Part 8: Quick Reference Card



### USE A CLASS WHEN:

- =====
- ✓ It needs to remember data between method calls
- ✓ It has a lifecycle (start/stop, open/close, connect/disconnect)
- ✓ It groups related operations that share data
- ✓ You need multiple instances with different data
- ✓ It represents a "thing" (Monitor, Router, Connection, Service)

### USE A FUNCTION WHEN:

- =====
- ✓ It's a one-time operation
- ✓ It transforms input → output
- ✓ It doesn't need to remember anything
- ✓ It's stateless (same input always gives same output)
- ✓ It represents an "action" (validate, format, calculate, detect)

### USE A DATACLASS WHEN:

- =====
- ✓ It's primarily data with little/no behavior
- ✓ It represents a "record" or "message" (RowChange, UserDTO, Config)
- ✓ You want automatic \_\_init\_\_, \_\_repr\_\_, \_\_eq\_\_

### USE AN ENUM WHEN:

- =====
- ✓ It's a fixed set of choices
- ✓ Values don't change at runtime
- ✓ You want type safety and autocomplete

## NAMING CONVENTIONS:

---

Classes: PascalCase → PostgresWALMonitor, BinlogRouter, RowChange  
Functions: snake\_case → detect\_database\_type, on\_ticket\_insert  
Constants: UPPER\_CASE → EVENT\_TYPE, MAX\_WORKERS  
Variables: snake\_case → db\_session, router, change  
Private: \_prefix → \_rules, \_pool, \_process\_message

---

## Summary

The key insight is: Design is about asking the right questions

1. **What does it need to DO?** (list responsibilities)
2. **Does it need to REMEMBER?** (state → class)
3. **Does it have a LIFECYCLE?** (start/stop → class)
4. **Does it GROUP related things?** (multiple methods → class)
5. **Is it just DATA?** (dataclass)
6. **Is it a one-time ACTION?** (function)

Would you like me to walk through designing any specific feature using this thinking process?

I think we should first understand the flow of the task and then Think about When Designing NEW Code, so now tell me this thing about my binlogger code

You're absolutely right! Let me trace your binlog code flow **step by step** first, then explain the design thinking behind each part.

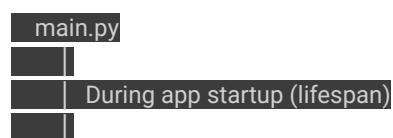
---

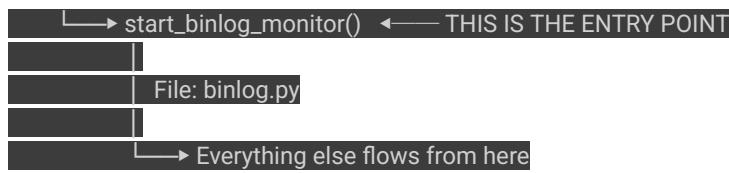
## Binlog Code Flow - Complete Understanding

### Step 1: The Entry Point (Where It All Starts)



Your app starts in main.py:





## Complete Binlog Flow with Design Thinking

### The Big Picture First



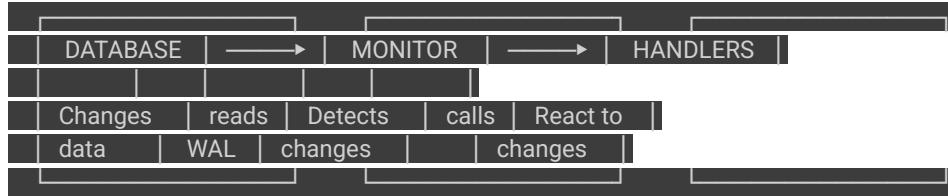
#### WHAT IS THE GOAL?

"When database changes, do something automatically"

#### Examples:

- Ticket created → Send notification to assignee
- User created → Send welcome email
- Role updated → Invalidate cache

#### HOW DOES IT WORK?



### Step-by-Step Flow

#### FLOW STEP 1: App Starts → Calls start\_binlog\_monitor()



## FLOW STEP 1: APP STARTUP

File: main.py → binlog.py

main.py (line 36)

```
@asynccontextmanager
async def app_lifespan(app: FastAPI):
    # Startup
    start_binlog_monitor() ←— ENTRY POINT
    ...
```

binlog.py :: start\_binlog\_monitor()

```
def start_binlog_monitor(max_workers: int = 8) -> None:
    global _monitor, _thread

    # Step 1.1: Check if already running (Singleton)
    if _monitor is not None:
        return # Already running, skip

    # Step 1.2: Detect database type
    db_type = detect_database_type() # Returns 'postgres' or 'mysql'

    # Step 1.3: Setup router with handlers
    router, EventType = setup_router(max_workers)

    # Step 1.4: Create monitor
    if db_type == 'mysql':
        _monitor = BinlogMonitor(...)
    else:
        _monitor = PostgresWALMonitor(...)

    # Step 1.5: Start background thread
    _thread = threading.Thread(target=_monitor.start, daemon=True)
    _thread.start()
```

## DESIGN THINKING:

Q: Why is start\_binlog\_monitor() a FUNCTION, not a class?

A: It's an "orchestrator" - it coordinates other components but doesn't need to remember much itself. Uses global variables for singleton pattern.

Q: Why global variables (\_monitor, \_thread)?

A: Singleton pattern - only ONE monitor should run at a time.

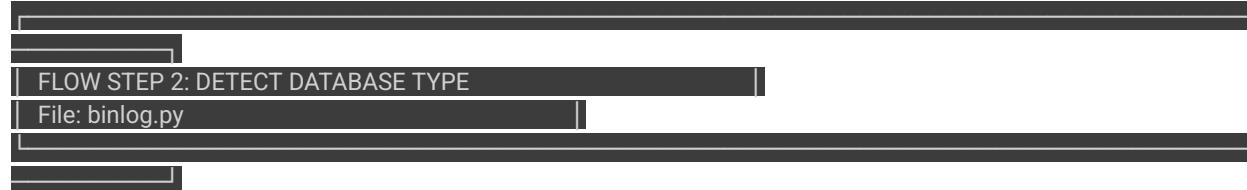
If we call start\_binlog\_monitor() twice, second call does nothing.

Q: Why daemon=True for the thread?

A: Daemon thread dies when main app dies. Don't want orphan threads.

---

## FLOW STEP 2: detect\_database\_type()



```
def detect_database_type() -> str:  
    db_url = DATABASE_URL.lower()  
    if 'postgres' in db_url:  
        return 'postgres'  
    elif 'mysql' in db_url:  
        return 'mysql'  
    else:  
        return 'postgres' # Default
```

### DESIGN THINKING:

Q: Why is this a FUNCTION?

A:

- Doesn't remember anything
- No lifecycle
- Simple input → output (URL → 'postgres' or 'mysql')
- Stateless (same input = same output)

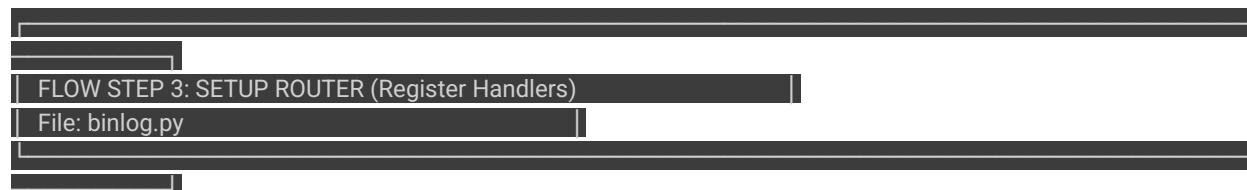
Q: Why is it a SEPARATE function?

A: Single Responsibility - one function does one thing.

Makes code readable and testable.

---

## FLOW STEP 3: setup\_router() - The Most Important Part



```
def setup_router(max_workers: int = 8):  
    db_type = detect_database_type()
```

```

# Step 3.1: Import correct classes based on DB type
if db_type == 'mysql':
    from services.binlog_monitor import BinlogRouter, EventType
else:
    from services.postgres_wal_monitor import BinlogRouter, EventType

schema_name = "public" # For PostgreSQL

# Step 3.2: Create the router
router = BinlogRouter(db_session_factory, max_workers=max_workers)

# Step 3.3: Register handlers for each table
#
# FORMAT: router.register(schema, table, [events], handler_function)
#
# This says: "When THIS table has THESE events, call THIS function"

# Ticket handlers
router.register(schema_name, "tickets", [EventType.INSERT], on_ticket_insert)
router.register(schema_name, "tickets", [EventType.UPDATE], on_ticket_update)
router.register(schema_name, "ticket_replies", [EventType.INSERT], on_ticket_reply_insert)

# Task handlers
router.register(schema_name, "tasks", [EventType.INSERT], on_task_create)
router.register(schema_name, "tasks", [EventType.UPDATE], on_task_update)

# Cache invalidation handlers (for many tables)
master_tables = ["roles", "companies", "departments", ...]
for table in master_tables:
    router.register(
        schema_name,
        table,
        [EventType.INSERT, EventType.UPDATE, EventType.DELETE],
        on_master_table_change
    )

return router, EventType

```

WHAT DOES THIS CREATE?

---



---

```

router._rules = [
    ("public", "tickets",      [INSERT],          on_ticket_insert),
    ("public", "tickets",      [UPDATE],         on_ticket_update),
    ("public", "ticket_replies", [INSERT],       on_ticket_reply_insert),
    ("public", "tasks",        [INSERT],          on_task_create),
    ("public", "tasks",        [UPDATE],         on_task_update),
    ("public", "roles",        [INSERT, UPDATE, DELETE], on_master_table_change),
    ("public", "companies",   [INSERT, UPDATE, DELETE], on_master_table_change),
    ... more rules ...
]

```

#### DESIGN THINKING:

---

---

Q: Why is setup\_router() a FUNCTION, not a class?

A: It just creates and configures a router, then returns it.

No state to remember. One-time setup operation.

Q: Why is BinlogRouter a CLASS?

A:

- ✓ REMEMBERS: \_rules (list of registered handlers)
- ✓ REMEMBERS: \_pool (ThreadPoolExecutor)
- ✓ REMEMBERS: \_db\_session\_factory
- ✓ GROUPS related operations: register(), dispatch(), shutdown()

Q: Why separate handlers into different files?

A: Organization by domain:

- tickets\_handler.py - all ticket-related handlers
- tasks\_handler.py - all task-related handlers
- cache\_invalidation\_handler.py - all cache handlers

---

## FLOW STEP 4: Create Monitor



```
# In start_binlog_monitor():

_monitor = PostgresWALMonitor(
    database_url=DATABASE_URL,
    slot_name="nextgen_slot_mydb",
    plugin="wal2json",
    only_schemas=["public"],
    router=router,           ← Router passed here
    resume_stream=True
)
```

#### WHAT DOES THIS CREATE?

---

---

```
PostgresWALMonitor instance with:
    self.running = False
    self.database_url = "postgresql://..."
    self.slot_name = "nextgen_slot_mydb"
```

```
    self.plugin = "wal2json"
    self.only_schemas = ["public"]
    self.router = router      ← Reference to router
    self.resume_stream = True
    self._conn = None          (will be set later)
    self._cursor = None        (will be set later)
    self.schema_cache = SchemaCache(conn_settings)
    self.decoder = WALDecoder(schema_cache)
```

#### DESIGN THINKING:

Q: Why is PostgresWALMonitor a CLASS?

A:

- ✓ REMEMBERS lots of state: connection, cursor, running flag, config
- ✓ Has LIFECYCLE: start() → running → stop()
- ✓ GROUPS related operations: start(), stop(), \_process\_message(), \_cleanup()
- ✓ Manages resources: database connection, replication slot

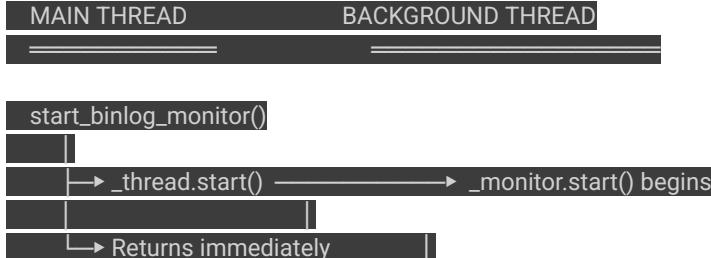
---

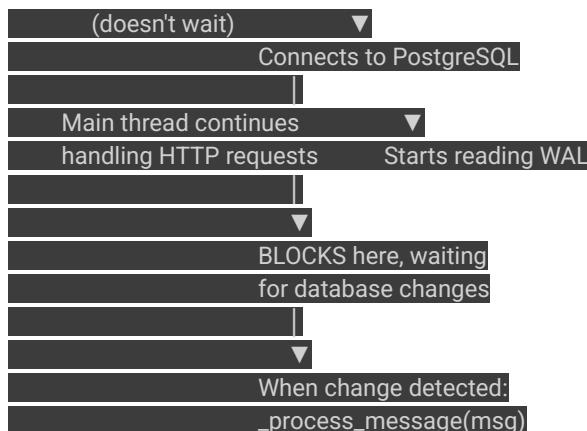
#### FLOW STEP 5: Start Background Thread

```
| FLOW STEP 5: START BACKGROUND THREAD |
| File: binlog.py |
|  
# In start_binlog_monitor():

_thread = threading.Thread(
    target=_monitor.start,      ← This function will run in thread
    daemon=True,               ← Dies with main app
    name="postgres-wal-monitor"
)
_thread.start()           ← Thread starts running
```

#### WHAT HAPPENS?





#### DESIGN THINKING:

Q: Why a separate thread?

A: WAL monitoring BLOCKS - it waits for changes.

Can't block the main thread (would freeze HTTP server).

Q: Why daemon=True?

A: Daemon threads are automatically killed when main app exits.

Don't want orphan threads running after app stops.

---

#### FLOW STEP 6: Monitor Starts Running

```

class PostgresWALMonitor:

    def start(self):
        self.running = True

    # Step 6.1: Create replication slot if needed
    self._ensure_replication_slot()

    # Step 6.2: Connect to PostgreSQL with replication protocol
    self._conn = psycopg2.connect(
        connection_factory=LogicalReplicationConnection,
        **self.conn_settings
    )
    self._cursor = self._conn.cursor()

```

```

# Step 6.3: Start consuming the WAL stream
self._cursor.start_replication(
    slot_name=self.slot_name,
    decode=True,
    options={"format-version": "1"}
)

# Step 6.4: Start keepalive thread
self._start_keepalive_thread()

# Step 6.5: BLOCK and process messages
#     This line BLOCKS forever until stop() is called
self._cursor.consume_stream(self._process_message)
#     ▲
#     |
#     For each WAL message, call _process_message()

```

#### DESIGN THINKING:

---

Q: Why `_ensure_replication_slot()` separate method?

A: Single responsibility - one method does one thing.  
Also might fail, want clear error messages.

Q: Why `consume_stream()` with callback?

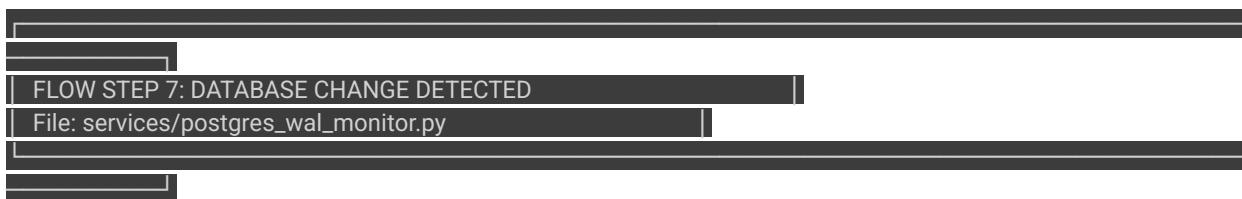
A: PostgreSQL library design - it calls your function for each message.  
Alternative would be polling loop (less efficient).

Q: Why keepalive thread?

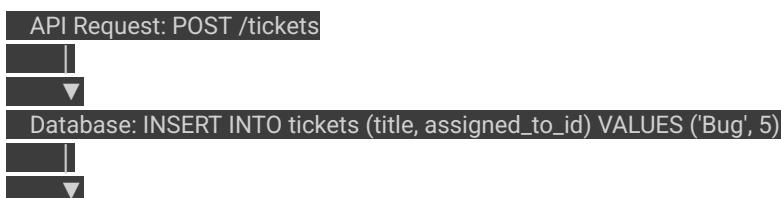
A: PostgreSQL closes inactive connections. Need to periodically  
say "I'm still here" to keep connection alive.

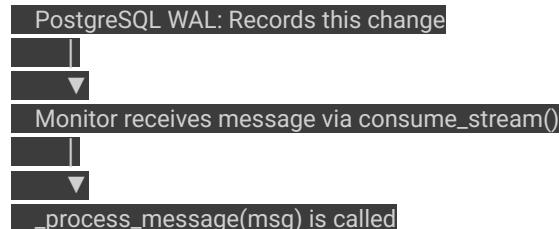
---

## FLOW STEP 7: Database Change Detected



SCENARIO: Someone inserts a new ticket via API





```

def _process_message(self, msg):
    # Step 7.1: Parse the WAL message (JSON)
    payload = json.loads(msg.payload)

    # payload looks like:
    # {
    #   "change": [
    #     "kind": "insert",
    #     "schema": "public",
    #     "table": "tickets",
    #     "columnnames": ["id", "title", "assigned_to_id", ...],
    #     "columnvalues": [123, "Bug", 5, ...]
    #   ]
    # }

    # Step 7.2: Decode into RowChange objects
    changes = self.decoder.decode_wal2json(payload)

    # changes = [
    #   RowChange(
    #     schema="public",
    #     table="tickets",
    #     event=EventType.INSERT,
    #     row={"id": 123, "title": "Bug", "assigned_to_id": 5, ...}
    #   )
    # ]

    # Step 7.3: Dispatch to router
    for change in changes:
        self.router.dispatch(change)

    # Step 7.4: Acknowledge message
    msg.cursor.send_feedback(flush_lsn=msg.data_start)
    _save_offset(msg.data_start)

```

#### DESIGN THINKING:

---



---

Q: Why WALDecoder separate class?

A:

- ✓ REMEMBERS: schema\_cache
- ✓ GROUPS related operations: decode\_wal2json(), decode\_test\_decoding()
- ✓ Single responsibility: only decodes WAL messages

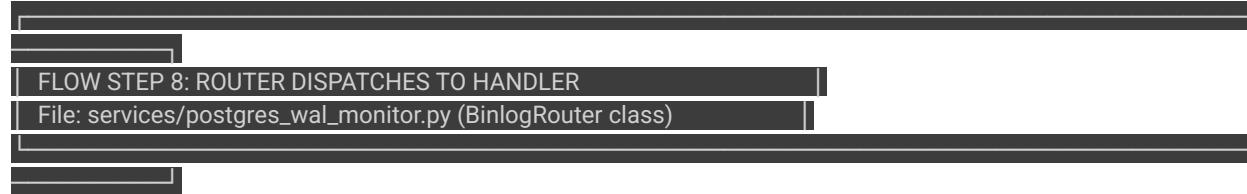
Q: Why RowChange as dataclass?

A: It's just DATA - schema, table, event, row data.

Dataclass gives automatic `__init__`, `__repr__`, `__eq__`.

---

## FLOW STEP 8: Router Dispatches to Handler



```
class BinlogRouter:

    def dispatch(self, change: RowChange) -> None:
        # Check all registered rules
        for schema_pat, table_pat, events, handler in self._rules:

            # Does this change match this rule?
            if (change.event in events and
                fnmatch.fnmatch(change.schema, schema_pat) and
                fnmatch.fnmatch(change.table, table_pat)):

                # YES! Submit to thread pool
                self._pool.submit(self._safe_call, handler, change)

    def _safe_call(self, handler, change):
        try:
            handler(change, self._db_session_factory)
        except Exception as e:
            logger.exception(f"Handler failed: {e}")
```

### EXAMPLE EXECUTION:

```
change = RowChange(schema="public", table="tickets", event=INSERT, ...)
```

Checking rules:

Rule	Match?	Action
("public", "tickets", [INSERT], on_ticket_insert)	YES	SUBMIT
("public", "tickets", [UPDATE], on_ticket_update)	NO	skip

("public", "users", [INSERT], on_user_create)	NO	skip	
("public", "tickets", [ALL], on_master_change)	YES	SUBMIT	



Two handlers will be called (in parallel via thread pool):

- on\_ticket\_insert(change, db\_factory)
- on\_master\_table\_change(change, db\_factory)

#### DESIGN THINKING:

Q: Why thread pool (ThreadPoolExecutor)?

A: Handlers run in parallel - one slow handler doesn't block others.

Also isolates failures - one handler crashing doesn't crash others.

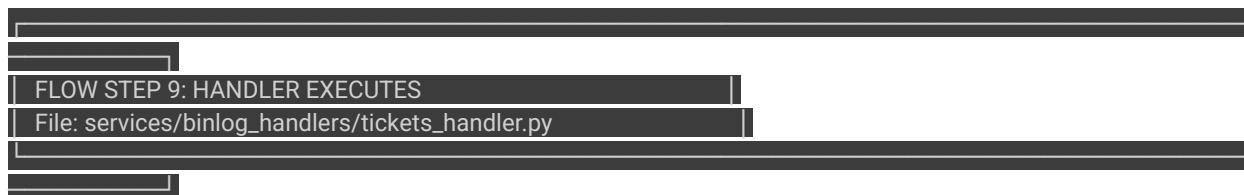
Q: Why \_safe\_call wrapper?

A: Error isolation. If handler throws exception, log it but don't crash.

Q: Why fnmatch (pattern matching)?

A: Allows wildcards in future: router.register("public", "ticket\*", ...)

## FLOW STEP 9: Handler Executes



```
def on_ticket_insert(change, db_factory: Callable[[], Session]):
    """Handle new ticket creation."""
    db = db_factory() # Create new DB session
    try:
        # Step 9.1: Extract data from change
        row = change.row or {}
        ticket_id = row.get("id")
        assigned_to_id = row.get("assigned_to_id")
```

```
        if not ticket_id or not assigned_to_id:
            return
```

```
        # Step 9.2: Fetch complete ticket from DB (if needed)
        ticket = db.query(Tickets).filter(Tickets.id == ticket_id).first()
        if not ticket:
            return
```

```

# Step 9.3: Build notification data
ticket_data = extract_ticket_data(ticket, db)

# Step 9.4: Send notification to assigned user
notify_users(
    db,
    [assigned_to_id],
    "New Ticket",
    ticket_data,
    "ticket",
    log=True
)

# Step 9.5: Notify managers
manager_ids = get_manager_ids(assigned_to_id, db)
if manager_ids:
    notify_users(db, manager_ids, "Manager: New Ticket", ticket_data)

except Exception as e:
    logger.error(f"Error in on_ticket_insert: {e}")
finally:
    db.close() # ALWAYS close DB session

```

**DESIGN THINKING:**

---



---

Q: Why handlers are FUNCTIONS, not classes?

A:

- ✗ Don't remember anything between calls
- ✗ No lifecycle
- ✓ One-time operation: receive change → do something
- ✓ Each call is independent

Q: Why db\_factory instead of passing db directly?

A: Each handler needs its OWN session.

If we passed same session, handlers would conflict (running in parallel).

Q: Why try/finally with db.close()?

A: DB sessions are resources that MUST be closed.

finally ensures close happens even if exception occurs.

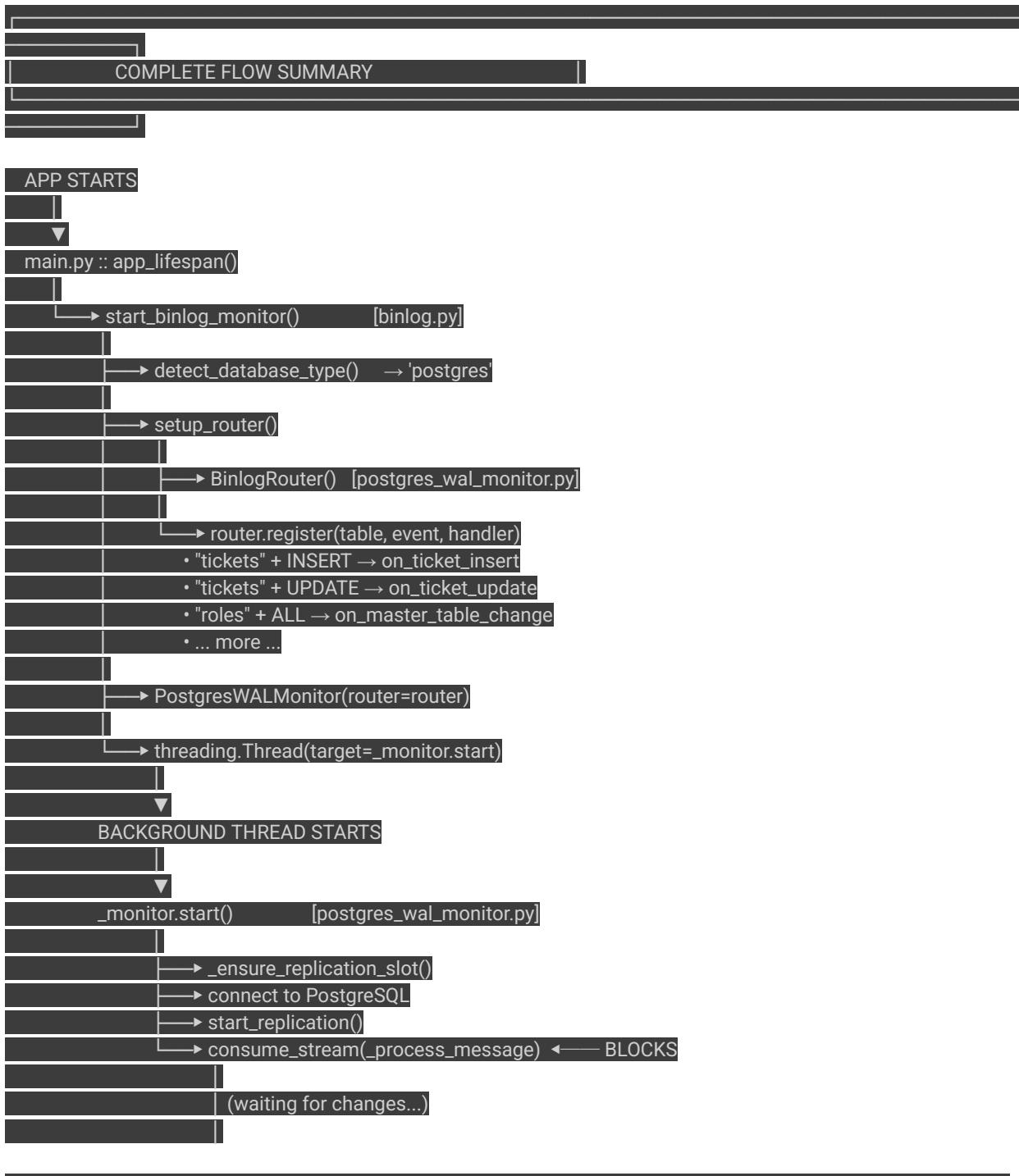
Q: Why separate handler files?

A: Organization by domain:

- tickets\_handler.py - knows about tickets
- tasks\_handler.py - knows about tasks
- cache\_invalidation\_handler.py - knows about cache

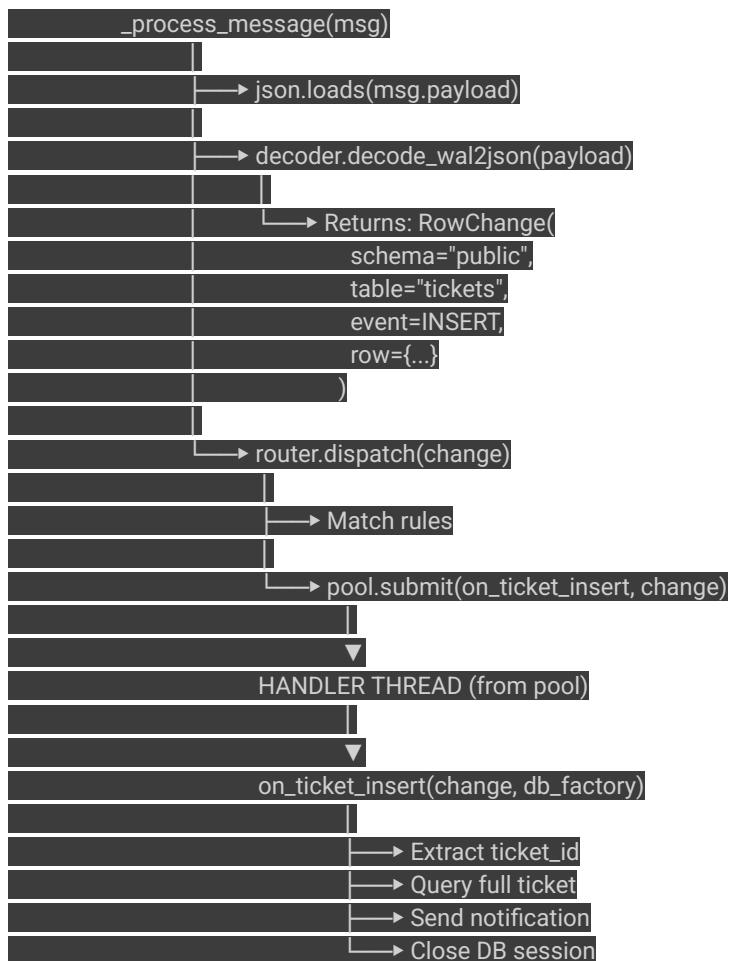
---

## Complete Flow Summary



LATER: Database change occurs

INSERT INTO tickets (title) VALUES ('Bug')



## Design Decisions Summary

WHY EACH THING IS DESIGNED THIS WAY		
Component	Type	Why This Design?
start_binlog_monitor()	FUNCTION	Orchestrator, uses global singleton
detect_database_type()	FUNCTION	Simple input→output, no state
setup_router()	FUNCTION	One-time setup, returns router

EventType	ENUM	Fixed values: INSERT, UPDATE, DELETE
RowChange	DATACLASS	Mainly data, few helper methods
<hr/>		
BinlogRouter	CLASS	Remembers rules, has dispatch/shutdown
WALDecoder	CLASS	Remembers schema_cache, groups decode
SchemaCache	CLASS	Remembers column info, caches queries
PostgresWALMonitor	CLASS	Lifecycle, lots of state, manages conn
<hr/>		
on_ticket_insert()	FUNCTION	One-time action, no state to remember
on_ticket_update()	FUNCTION	One-time action, no state to remember
on_master_table_change()	FUNCTION	One-time action, no state to remember
<hr/>		
_monitor (global)	VARIABLE	Singleton - only one monitor runs
_thread (global)	VARIABLE	Reference to background thread
<hr/>		

---

Now you understand the complete flow! The key insight is:

1. First understand **WHAT** the code does (the flow)
2. Then understand **WHY** it's designed that way (the thinking)

Would you like me to trace any specific part in more detail, or explain how to add a new feature to this system?

why we use thread in between and for what purpose ,how we come to know that we shuld use the thread and what things we have to keep in mind to implement or to use the thread ,also in how many ways or types of threads can be implemented as i am confused that why we are using it?

Excellent question! Let me explain threading from the ground up - why, when, and how to use it.

---

## Threading - Complete Understanding

### Part 1: Why Do We Need Threads?

THE PROBLEM: BLOCKING OPERATIONS
<hr/>

SCEENARIO: Your app needs to do TWO things:

1. Handle HTTP requests (FastAPI server)

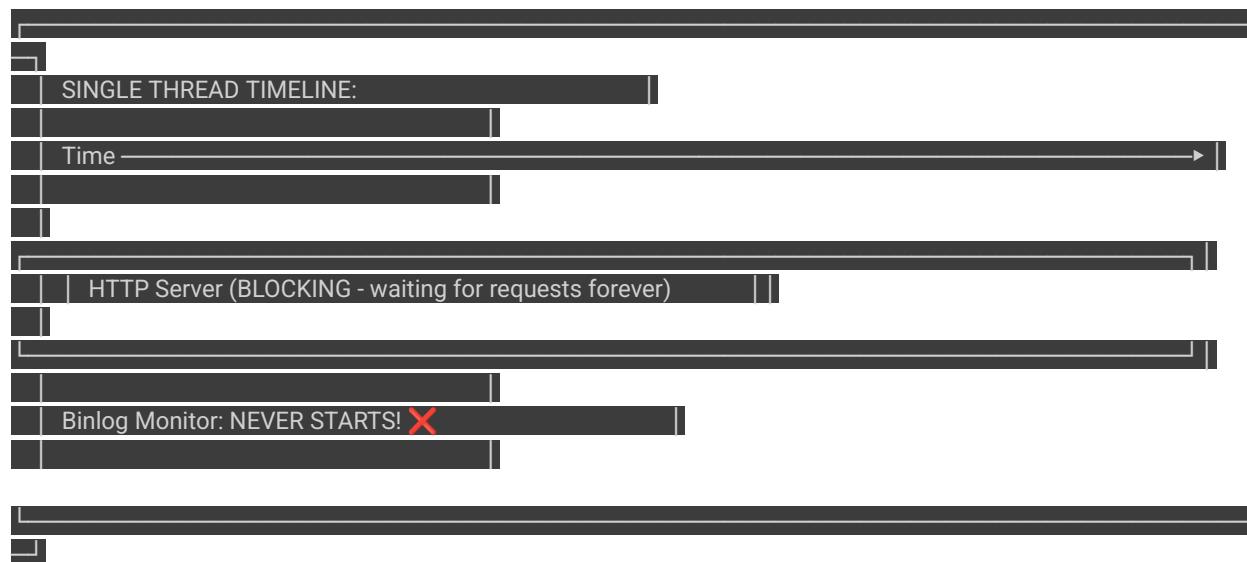
## 2. Monitor database changes (Binlog monitor)

WITHOUT THREADS (Single Thread):

```
def main():
    start_http_server() # This BLOCKS forever (waiting for requests)
    start_binlog_monitor() # X NEVER RUNS! (stuck on line above)
```

OR

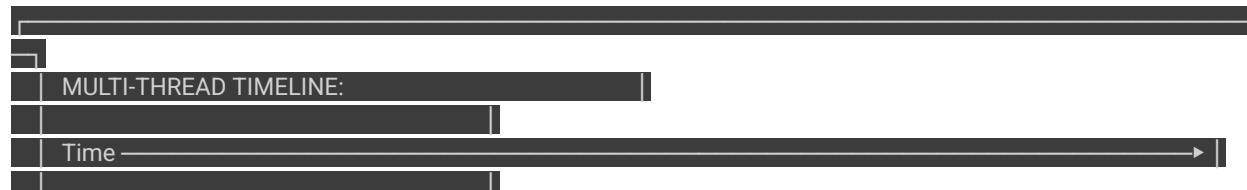
```
def main():
    start_binlog_monitor() # This BLOCKS forever (waiting for DB changes)
    start_http_server() # X NEVER RUNS! (stuck on line above)
```

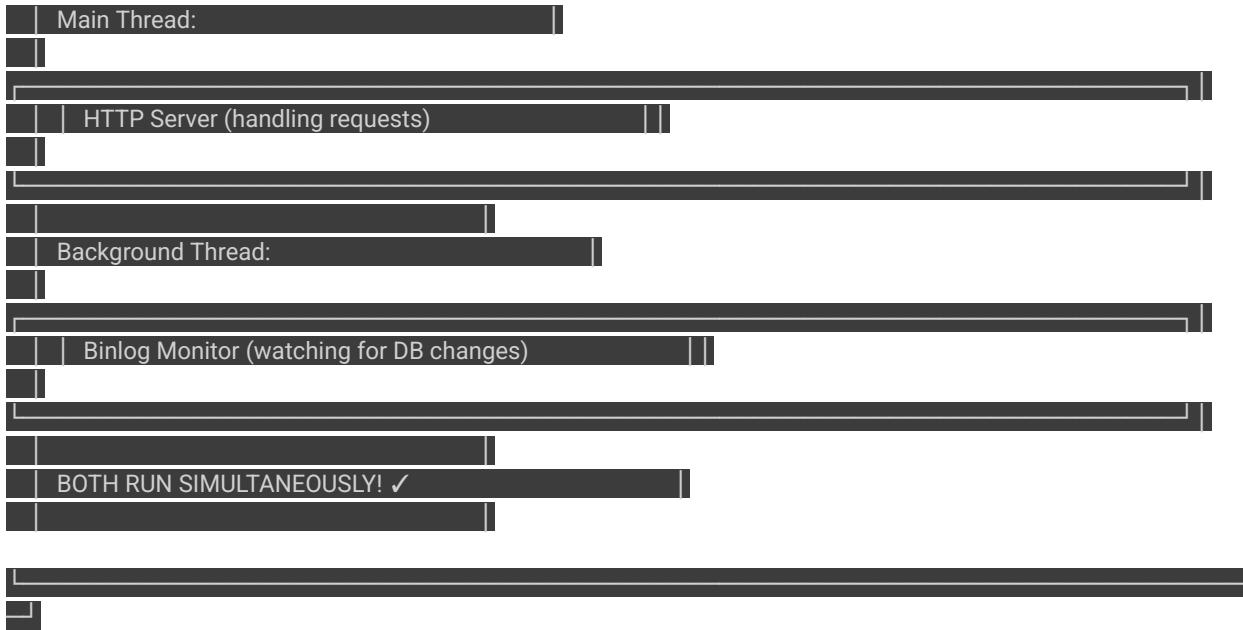


WITH THREADS (Multiple Threads):

```
def main():
    # Start binlog in separate thread
    thread = threading.Thread(target=start_binlog_monitor)
    thread.start() # Starts running in PARALLEL

    start_http_server() # Main thread handles HTTP
```





## Part 2: What is "Blocking"?



BLOCKING = Code that WAITS for something and doesn't return

EXAMPLES OF BLOCKING OPERATIONS:

### 1. WAITING FOR USER INPUT

```
name = input("Enter name: ") # BLOCKS until user types
print(f"Hello {name}")      # Only runs AFTER user types
```

### 2. WAITING FOR NETWORK RESPONSE

```
response = requests.get("https://api.example.com") # BLOCKS until response
print(response.json())    # Only runs AFTER response arrives
```

### 3. WAITING FOR DATABASE CHANGES (YOUR BINLOG)

```
cursor.consume_stream(handler) # BLOCKS forever, waiting for changes
```

```
print("Done")      # NEVER RUNS!
```

#### 4. WAITING FOR FILE DOWNLOAD

```
download_large_file() # BLOCKS until download complete (maybe 10 minutes)  
process_file()      # Only runs AFTER download
```

#### 5. SLEEP

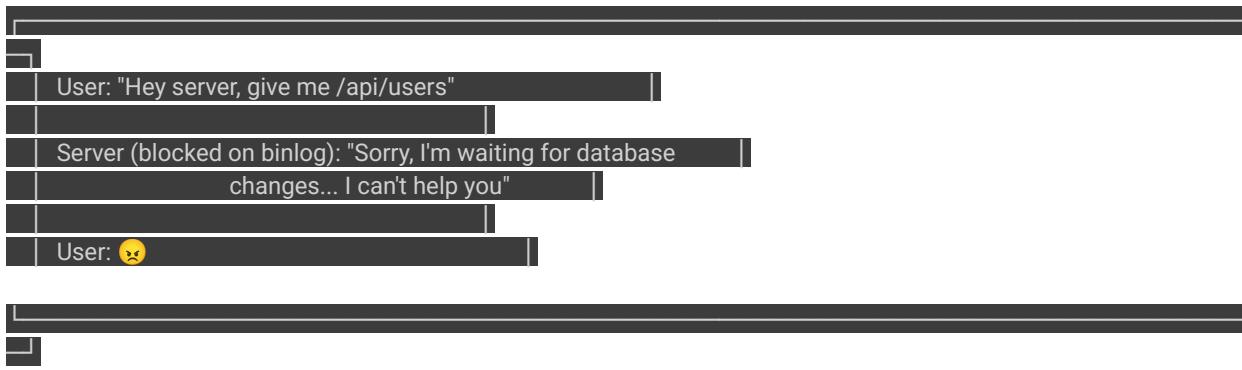
```
time.sleep(60)    # BLOCKS for 60 seconds  
print("Woke up") # Runs after 60 seconds
```

#### WHY IS BLOCKING A PROBLEM?

---

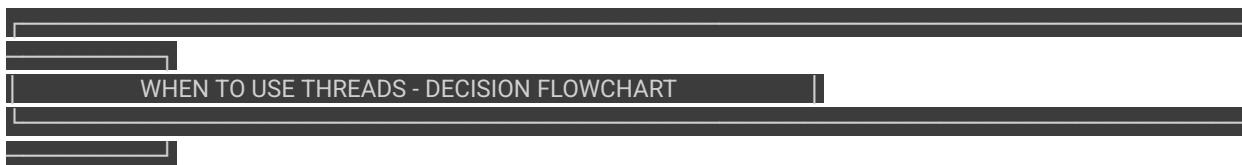
---

While code is BLOCKED, nothing else can run in that thread.

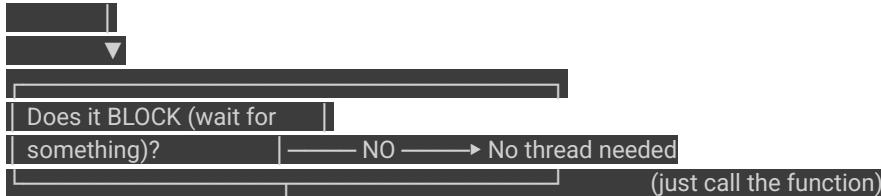


---

## Part 3: When to Use Threads (Decision Guide)



START: I have code that needs to run





#### EXAMPLES FROM YOUR CODEBASE:

Use Case	Why Thread?
Binlog Monitor (postgres_wal_monitor)	BLOCKS forever waiting for DB changes. Main thread needs to handle HTTP requests. → Thread with daemon=True
RabbitMQ Consumer (grandstream_consumer)	BLOCKS forever waiting for messages. Main thread needs to handle HTTP requests. → Thread with daemon=True
SFTP Sync on Startup	Takes time (network operation). Don't want to delay app startup. → Thread with daemon=True
Missed Call Timer	Wait 5 seconds then process. Don't want to block call handling. → Timer thread (special thread type)
Handler Execution	Multiple handlers might be slow.



## Part 4: Types of Threading in Python



### TYPE 1: BASIC THREAD (threading.Thread)

Use when: You have ONE task to run in background

```

import threading

def my_task():
    print("Running in background")
    time.sleep(10)
    print("Done!")

# Create and start thread
thread = threading.Thread(target=my_task)
thread.start()

print("Main continues immediately")

```

Your code example (binlog.py):

```

_thread = threading.Thread(
    target=_monitor.start,  # Function to run
    daemon=True,          # Die with main app
    name="postgres-wal-monitor"
)
_thread.start()

```

### TYPE 2: DAEMON THREAD (daemon=True)

Use when: Thread should die when main app dies

```
# DAEMON THREAD
```

```

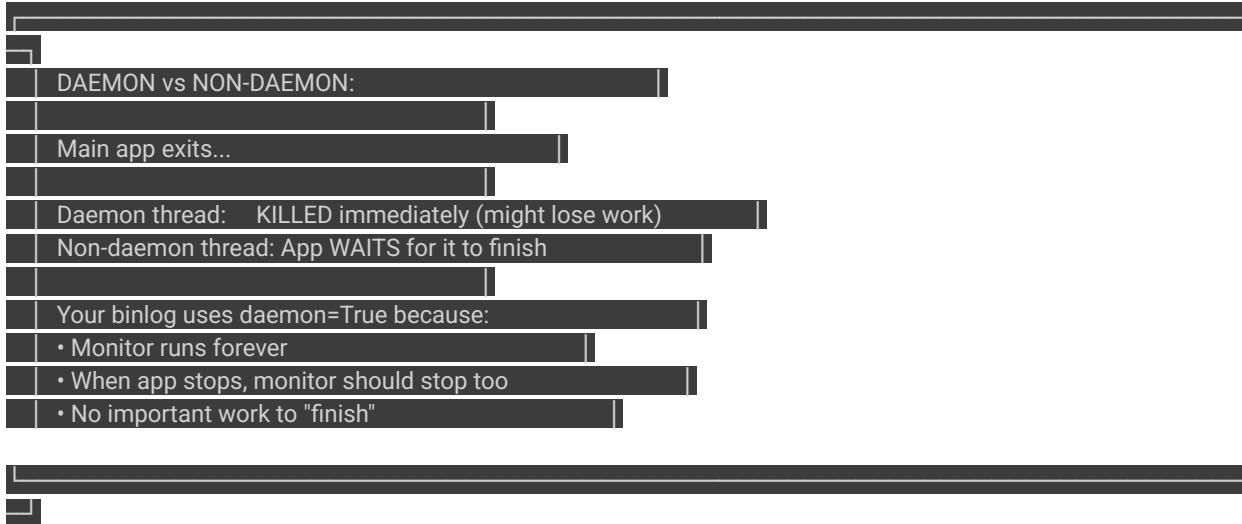
thread = threading.Thread(target=my_task, daemon=True)
thread.start()

# If main app exits, this thread is KILLED automatically
# Good for: background monitors, cleanup tasks

# NON-DAEMON THREAD (default)
thread = threading.Thread(target=my_task, daemon=False)
thread.start()

# Main app will WAIT for this thread to finish before exiting
# Good for: important tasks that MUST complete

```



### TYPE 3: TIMER THREAD (threading.Timer)

Use when: Run something AFTER a delay

```

import threading

def delayed_task():
    print("This runs after 5 seconds!")

# Start timer - will call delayed_task after 5 seconds
timer = threading.Timer(5.0, delayed_task)
timer.start()

print("Main continues immediately")
# ... 5 seconds later ...
# "This runs after 5 seconds!" prints

```

Your code example (grandstream\_consumer.py):

```

# Wait 5 seconds before processing missed call
# (in case call gets answered on another device)

missed_call_timer = threading.Timer(
    delay_seconds + 0.5,      # Wait this long
    process_missed_queue_timer # Then call this function
)
missed_call_timer.start()

```

#### TYPE 4: THREAD POOL (ThreadPoolExecutor)

---



---

Use when: Many small tasks that can run in parallel

```

from concurrent.futures import ThreadPoolExecutor

def process_item(item):
    print(f"Processing {item}")
    time.sleep(1)
    return f"Done: {item}"

# Create pool with 4 worker threads
with ThreadPoolExecutor(max_workers=4) as pool:
    # Submit 10 tasks - only 4 run at a time
    futures = [pool.submit(process_item, i) for i in range(10)]

    # Get results
    for future in futures:
        print(future.result())

```

Your code example (BinlogRouter):

---



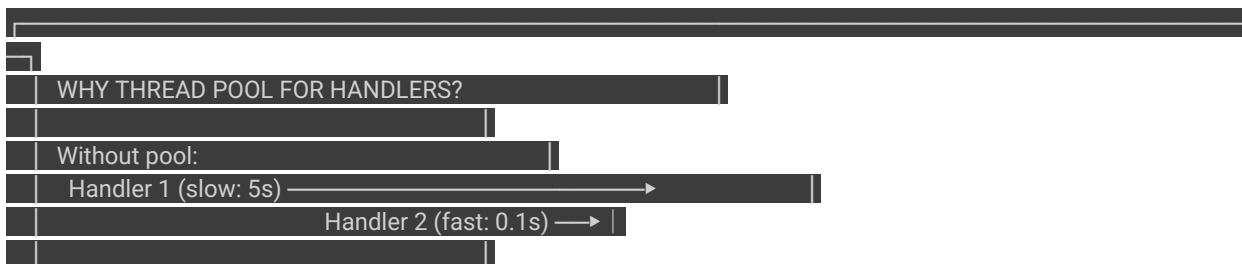
---

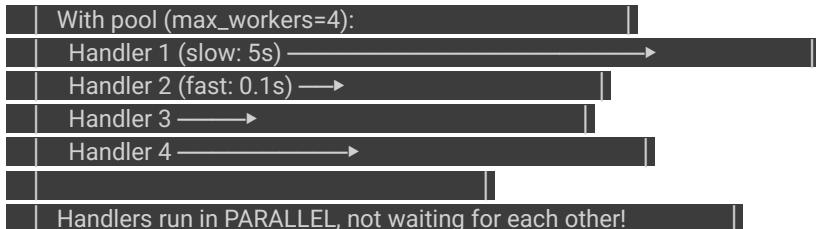
```

class BinlogRouter:
    def __init__(self, db_session_factory, max_workers=8):
        self._pool = ThreadPoolExecutor(max_workers=max_workers)

    def dispatch(self, change):
        for schema, table, events, handler in self._rules:
            if matches(change):
                # Submit to pool - runs in background
                self._pool.submit(self._safe_call, handler, change)

```





#### TYPE 5: THREAD WITH EVENT (threading.Event)

Use when: Need to signal thread to stop

```
import threading

stop_event = threading.Event()

def worker():
    while not stop_event.is_set(): # Keep running until told to stop
        print("Working...")
        time.sleep(1)
    print("Stopped!")

thread = threading.Thread(target=worker)
thread.start()

time.sleep(5)      # Let it run for 5 seconds
stop_event.set()  # Signal to stop
thread.join()     # Wait for thread to finish
```

#### Your code example (availability\_cleanup.py):

```
_cleanup_stop_event = threading.Event()

def cleanup_worker():
    while not _cleanup_stop_event.is_set():
        # Do cleanup work
        cleanup_stale_sessions()
        # Wait 60 seconds OR until stop signal
        _cleanup_stop_event.wait(timeout=60)

def stop_cleanup():
    _cleanup_stop_event.set() # Signal worker to stop
```

#### TYPE 6: THREAD WITH LOCK (threading.Lock)

Use when: Multiple threads access SAME data

```
import threading

counter = 0
lock = threading.Lock()

def increment():
    global counter
    with lock: # Only ONE thread can be here at a time
        temp = counter
        time.sleep(0.001) # Simulate work
        counter = temp + 1

threads = [threading.Thread(target=increment) for _ in range(100)]
for t in threads: t.start()
for t in threads: t.join()

print(f"Counter: {counter}") # Always 100 with lock
```

Your code example (rabbitmq.py):

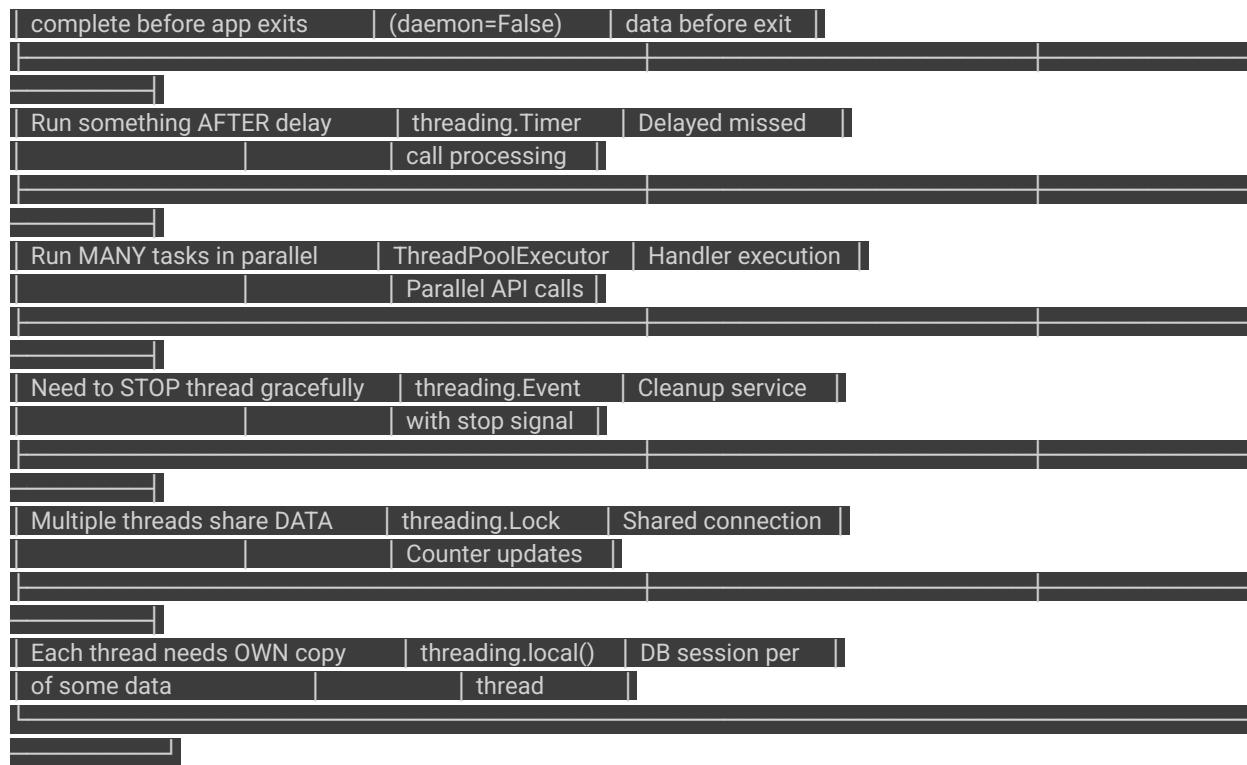
```
_connection_lock = threading.RLock()

def get_connection():
    with _connection_lock: # Only one thread can get/create connection
        if _connection is None:
            _connection = create_connection()
    return _connection
```

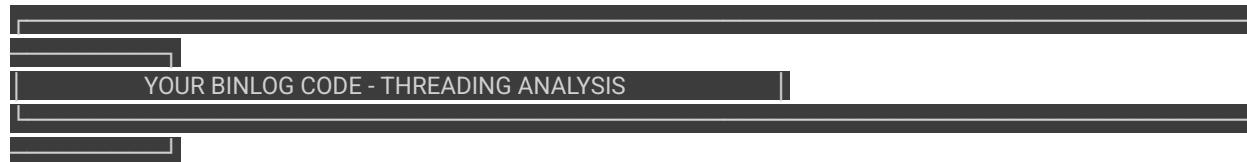
---

## Part 5: Thread Types Summary

WHICH THREAD TYPE TO USE?		
Situation	Thread Type	Example
Run ONE task in background that runs FOREVER	threading.Thread (daemon=True)	Binlog monitor RabbitMQ consumer
Run ONE task that MUST	threading.Thread	Save important



## Part 6: Your Binlog Code - Why Thread is Used



QUESTION: Why does binlog need a thread?

```
# In PostgresWALMonitor.start()
self._cursor.consume_stream(self._process_message)
# ▲
# | THIS LINE BLOCKS FOREVER!
#       It waits for database changes and NEVER returns.
```

PROBLEM WITHOUT THREAD:

```
def main():
    # If we call this directly...
    _monitor.start() # BLOCKS FOREVER
```

```
# This never runs!
start_http_server() # ❌ Users can't access your app!
```

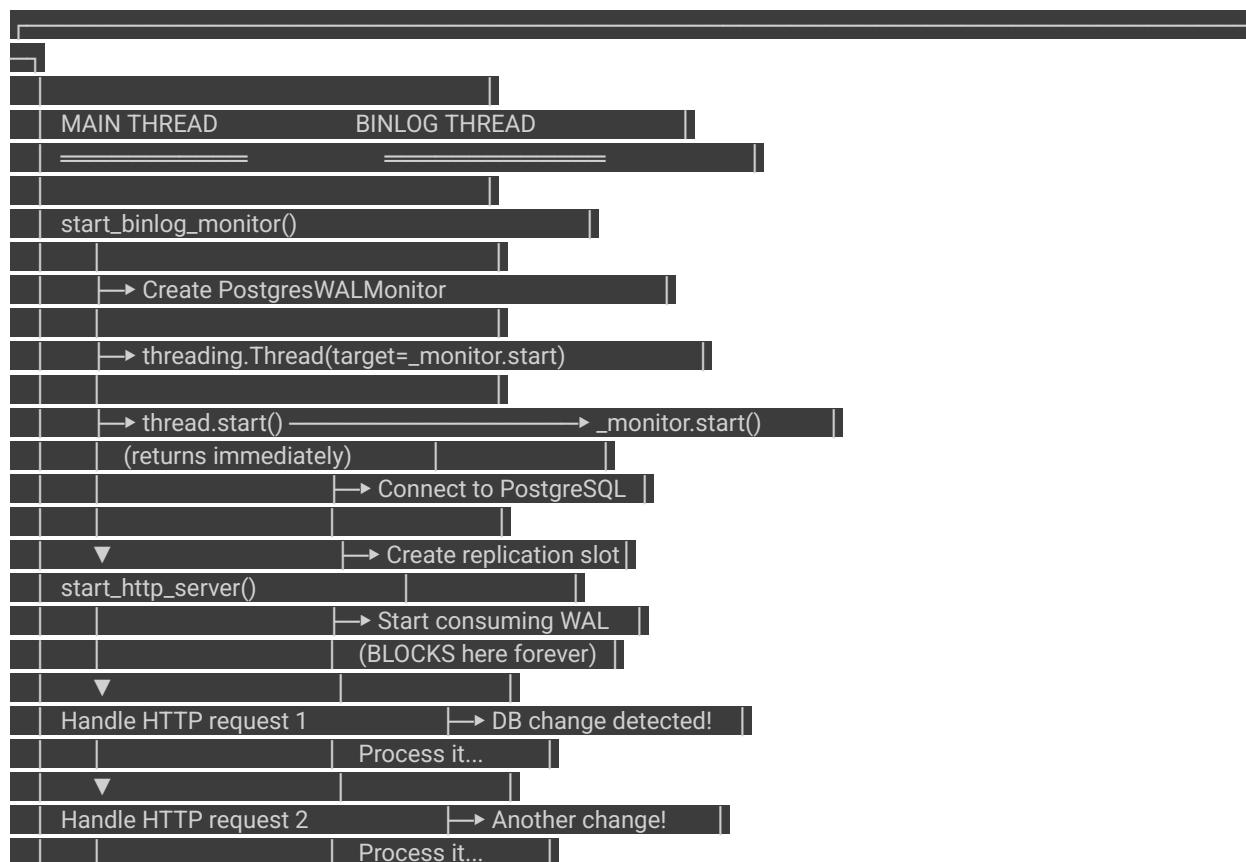
## SOLUTION WITH THREAD:

```
def start_binlog_monitor():
    _monitor = PostgresWALMonitor(...)

    # Start in SEPARATE thread
    _thread = threading.Thread(
        target=_monitor.start, # This function will block
        daemon=True           # But in its own thread!
    )
    _thread.start()
```

```
# This returns IMMEDIATELY  
# Main thread continues to start HTTP server
```

## VISUAL EXPLANATION:





## Part 7: Things to Keep in Mind When Using Threads

### RULE 1: DAEMON THREADS FOR "FOREVER" TASKS

```
# GOOD: Daemon thread for monitor
thread = threading.Thread(target=monitor, daemon=True)
```

```
# When app exits, thread dies automatically
# No orphan threads left running
```

```
# BAD: Non-daemon for forever task
thread = threading.Thread(target=monitor, daemon=False)
```

```
# App tries to exit, but waits for thread...
# Thread runs forever...
# App NEVER EXITS! 😱
```

### RULE 2: EACH THREAD NEEDS ITS OWN DB SESSION

```
# BAD: Sharing session across threads
db = SessionLocal()
```

```
def handler(change):
    db.query(...) # ❌ Multiple threads use same session = CRASH
```

```
# GOOD: Each thread creates own session
def handler(change, db_factory):
    db = db_factory() # New session for THIS thread
    try:
        db.query(...)
    finally:
        db.close()
```

#### RULE 3: USE LOCKS FOR SHARED DATA

---

---

```
# BAD: Multiple threads modify same variable
counter = 0

def increment():
    global counter
    counter += 1 # ❌ Race condition! Results will be wrong

# GOOD: Use lock
counter = 0
lock = threading.Lock()

def increment():
    global counter
    with lock:
        counter += 1 # ✓ Only one thread at a time
```

#### RULE 4: HANDLE EXCEPTIONS IN THREADS

---

---

```
# BAD: Exception kills thread silently
def worker():
    raise Exception("Oops!") # Thread dies, no one knows

# GOOD: Catch and log exceptions
def worker():
    try:
        # ... work ...
    except Exception as e:
        logger.exception(f"Thread error: {e}")

# Your code does this:
def _safe_call(self, handler, change):
    try:
        handler(change, self._db_session_factory)
    except Exception as e:
        logger.exception(f"Handler failed: {e}")
```

#### RULE 5: CLEAN UP RESOURCES

```
=====
```

```
# BAD: Thread creates resources, never closes
def worker():
    conn = connect_to_db()
    while True:
        process(conn)
    # Connection never closed!
```

```
# GOOD: Clean up on exit
def worker():
    conn = connect_to_db()
    try:
        while running:
            process(conn)
    finally:
        conn.close() # Always clean up
```

#### RULE 6: DON'T START TOO MANY THREADS

```
=====
```

```
# BAD: New thread for each task
for item in million_items:
    thread = threading.Thread(target=process, args=(item,))
    thread.start() # 1 million threads! 😱 System crashes
```

```
# GOOD: Use thread pool
with ThreadPoolExecutor(max_workers=10) as pool:
    for item in million_items:
        pool.submit(process, item) # Only 10 threads, tasks queue up
```

---

## Part 8: Complete Threading Pattern in Your Binlog

```
=====
```

```
|          ALL THREADING IN YOUR BINLOG CODE          |
```

```
THREAD 1: Main Monitor Thread
```

```
=====
```

```
File: binlog.py
```

```
_thread = threading.Thread(
    target=_monitor.start,
    daemon=True,
    name="postgres-wal-monitor"
```

```
)  
_thread.start()
```

Purpose: Run WAL monitor in background (blocks forever)

#### THREAD 2: Keepalive Thread

File: services/postgres\_wal\_monitor.py

```
self._keepalive_thread = threading.Thread(  
    target=keepalive_worker,  
    daemon=True,  
    name="wal-keepalive"  
)  
self._keepalive_thread.start()
```

Purpose: Send periodic heartbeat to PostgreSQL (prevents timeout)

#### THREAD 3-10: Handler Pool (ThreadPoolExecutor)

File: services/postgres\_wal\_monitor.py (BinlogRouter)

```
self._pool = ThreadPoolExecutor(max_workers=max_workers)  
  
# When dispatching:  
self._pool.submit(self._safe_call, handler, change)
```

Purpose: Run multiple handlers in parallel (max 8 at a time)

#### THREAD LOCK: LSN Update Lock

File: services/postgres\_wal\_monitor.py

```
self._lsn_lock = threading.Lock()  
  
# When updating LSN:  
with self._lsn_lock:  
    self._last_lsn = msg.data_start  
    self._last_feedback_time = time.time()
```

Purpose: Prevent race condition when main thread and keepalive thread  
both try to update/read LSN

#### COMPLETE PICTURE:



## Part 9: Quick Decision Guide



1. Does my code BLOCK (wait for something)?

- Network request?
- Database query that takes long?
- Waiting for messages (RabbitMQ, WebSocket)?
- Monitoring something forever?
- Sleep/delay?

2. Do I need to do OTHER things while waiting?

- Handle HTTP requests?
- Process other events?
- Keep UI responsive?

3. If YES to both → USE A THREAD

WHICH THREAD TYPE?

---

---

"Run forever in background"	→ threading.Thread(daemon=True)
"Run once after delay"	→ threading.Timer
"Run many tasks in parallel"	→ ThreadPoolExecutor
"Need to stop gracefully"	→ threading.Event + while loop
"Threads share data"	→ threading.Lock
"Each thread needs own copy"	→ threading.local()

---

---

## Summary

The key insight is: **Thread = Way to do multiple things at the same time** Your binlog uses threads because:

1. **Monitor BLOCKS forever** (waiting for DB changes)
2. **Main thread needs to handle HTTP requests** (can't be blocked)
3. **Solution: Monitor runs in separate thread**