You are a senior Python engineer and production-ready bot builder. Build the full Discord bot project from scratch (file-by-file), following the exact technical specification below (this is an authoritative reproduction of the Music Review Bot Overview). Implement every table, command, process, and edge case described here. Additionally implement the three changes requested by the owner:

1. Submissions only accepted in the configured Submissions channel (no passive submissions in other channels).

2. Admin-only force link/unlink commands to attach/detach TikTok handles to Discord users.

3. A full Luxury Coins economy (1 coin per 30 minutes watch time; 2 coins per 100 gifted coins) with user/admin commands and the ability to buy a skip for 1000 coins which moves their submission to the 10 Line.

Project requirements (deliver the actual codebase):

main.py entry point

database.py for schema + connections + migrations

requirements.txt

.env sample

cogs/ directory containing all cogs listed below

Unit test scaffolding and README with setup & deploy steps

PEP8 style, async/await everywhere, parameterized SQL

COMPLETE SPECIFICATION (REPRODUCE THIS EXACTLY + APPLY CHANGES)

> Important: Implement everything in the specification that follows. Do not omit fields, tables, locks, refresh timings, event processing rules, or command behaviors. Where a behavior in the original spec conflicts with the requested changes above, apply the requested change (e.g., submissions channel enforcement) while keeping the remaining original behavior identical.

---

1) Executive summary & purpose

Build a Discord Music Queue Bot that:

Accepts music submissions,

Integrates with TikTok Live to track engagement,

Sorts a priority-based queue,

Allows skip-tier movement via gifts and Luxury Coins,

Displays persistent, auto-updating paginated embeds,

Self-heals after restarts or message deletions,

Logs and reports post-live metrics to a configured metrics channel.

---

2) Submission flow (HYBRID SUBMISSION SYSTEM — reproduce exactly but enforce Submissions channel)

Submission methods (original spec) — implement all three, but enforce channel restriction:

1. Slash Commands

/submit — modal for link submissions (YouTube, Spotify, SoundCloud, Deezer, Ditto). Optional tiktok_handle.

/submitfile — modal/file upload with artist_name, song_title, note (optional), optional tiktok_handle. File validation applies.

2. Passive Submission (original spec allowed passive submissions in any channel) — CHANGE: allow passive/file submissions only in the configured Submissions channel. If a user posts a supported link or uploads audio anywhere else, the bot should:

Delete the message (if configured to keep channels clean) or respond ephemerally with an error.

Ask the user to resubmit in the submissions channel or provide a /submit//submitfile link.

3. File Uploads — accept mp3, m4a, wav, flac up to 25 MB. Auto-upload to Discord CDN if needed.

Platform support (exact):

Supported: soundcloud, spotify, youtube, deezer, ditto

Rejected: apple, itunes (send friendly error)

On successful submission:

If artist missing → set to Discord username

If title missing → set to "Not Known"

If user not linked to a TikTok handle, allow optional manual handle entry

Add to proper queue line (see QUEUES)

DM confirmation; if DMs blocked, fallback to the submission channel

If flagged as a "skip" submission (paid skip), add to Pending Skips and require admin approval for the appropriate tier

Enforcement notes (new):

Provide /set-submission-channel #channel admin command (admin-only) — this persists to bot_config.

Passive detection must only act on messages in that channel. Anywhere else, detection bypassed.

---

3) Queue Priority System (exact)

Queue lines and priority order (exact, implement tiers precisely):

1. 25+ Skip (≥6000 coins) — Highest priority, FIFO

2. 20 Skip (5000–5999)

3. 15 Skip (4000–4999)

4. 10 Skip (2000–3999)

5. 5 Skip (1000–1999)

6. Free — sorted by engagement points (DESC), then submission time (ASC)

7. Pending Skips — admin review

8. Songs Played — archive

9. Removed — admin-deleted

Processing logic (exact):

/next command must atomically (FOR UPDATE) pull the highest priority submission, mark as playing, and move record to Songs Played.

Skip tiers are processed before the Free line. Within a tier, FIFO.

When a Free-line song plays, reset submitter's points to zero (and reset linked TikTok handle points).

Ensure atomic transactions and lock the queue during selection to avoid race conditions.

---

4) TikTok handle linking (exact + admin force link/unlink change)

Original spec:

/link-tiktok @handle — autocompletes from handles seen on live; links to user if present.

Users can link multiple TikTok handles.

Captured handles during livestreams are stored whether or not linked.

Autocomplete: only show 2+ characters; use prefix search; both linked and unlinked autocompletes present.

Add (exact requested change):

Admin commands (admin-only):

/admin-link @discord_user handle — force-link a tiktok handle to a discord user (bypass checking if handle seen, but prefer to validate if present in tiktok_accounts).

/admin-unlink @discord_user handle — force-unlink.

Log admin actions to debug channel.

Maintain existing validation and safety checks where possible.

Temporary bypass note (preserve original behavior):

The original had a temporary bypass allowing linking any handle; preserve an option in configuration ALLOW_ANY_HANDLE_LINKING (default false). Admins can enable/disable.

---

5) TikTok Live Integration architecture (exact)

Connection system (exact):

Commands:

/tiktok connect @username [persistent=true] — connect to a live stream (persistent retry by default)

/tiktok status

/tiktok disconnect

Keep asynchronous background connection task(s) that auto-retries until the user goes live (persistent true).

Automatic reconnection and robust state management: distinguish user vs unexpected disconnects.

Use TikTokLive library (or adapter) in a guarded wrapper to handle schema differences.

Event tracking (exact list + values): Track all events: JoinEvent, LikeEvent, CommentEvent, ShareEvent, FollowEvent, SubscribeEvent, GiftEvent, RoomUserSeqEvent, PollEvent, LinkMicBattleEvent, ConnectEvent, DisconnectEvent, LiveEndEvent.

Gift processing logic (exact):

Points calculation:

If diamond_count < 1000: points = diamond_count * 2 (2 points per coin)

Else: points = diamond_count (1 point per coin)

Skip tier determination (based on aggregated gift coins):

coins >= 6000 → 25+ Skip

>= 5000 → 20 Skip

>= 4000 → 15 Skip

>= 2000 → 10 Skip

>= 1000 → 5 Skip

Move most recent Free or Pending Skips submission into the proper skip tier upon confirmed gift threshold.

Streakable gift handling (exact):

Detect streaking attribute; only process the final gift of a streak to avoid duplicates. If a gift event has streaking=True, wait for the final event where streaking=False before processing.

RoomUserSeq & Viewer snapshots (exact):

Periodically log viewer_count snapshots to viewer_count_snapshots table.

---

6) Database schema (exact)

Create and initialize all tables below. If missing, create them automatically on first run.

Tables (implement exactly as specified):

```
-- live_sessions
CREATE TABLE live_sessions (
  id SERIAL PRIMARY KEY,
  tiktok_username TEXT NOT NULL,
  start_time TIMESTAMPTZ DEFAULT NOW(),
  end_time TIMESTAMPTZ
);

-- tiktok_accounts
CREATE TABLE tiktok_accounts (
  handle_id SERIAL PRIMARY KEY,
  handle_name TEXT UNIQUE NOT NULL,
  first_seen TIMESTAMPTZ DEFAULT NOW(),
  last_seen TIMESTAMPTZ DEFAULT NOW(),
  linked_discord_id BIGINT,
  points INTEGER DEFAULT 0 NOT NULL,
  last_known_level INTEGER DEFAULT 0
);

-- tiktok_interactions
CREATE TABLE tiktok_interactions (
  id SERIAL PRIMARY KEY,
  session_id INTEGER REFERENCES live_sessions(id) ON DELETE CASCADE,
  tiktok_account_id INTEGER REFERENCES tiktok_accounts(handle_id) ON DELETE SET
NULL,
  interaction_type TEXT NOT NULL,
  value TEXT,
  coin_value INTEGER,
  user_level INTEGER,
  timestamp TIMESTAMPTZ DEFAULT NOW()
);

-- viewer_count_snapshots
CREATE TABLE viewer_count_snapshots (
  id SERIAL PRIMARY KEY,
  session_id INTEGER REFERENCES live_sessions(id) ON DELETE CASCADE,
  viewer_count INTEGER NOT NULL,
```

```
  timestamp TIMESTAMPTZ DEFAULT NOW()
);

-- submissions
CREATE TABLE submissions (
  id SERIAL PRIMARY KEY,
  public_id TEXT UNIQUE NOT NULL,
  user_id BIGINT NOT NULL,
  username TEXT NOT NULL,
  artist_name TEXT NOT NULL,
  song_name TEXT NOT NULL,
  link_or_file TEXT,
  queue_line TEXT,
  submission_time TIMESTAMPTZ DEFAULT NOW(),
  played_time TIMESTAMPTZ,
  note TEXT,
  tiktok_username TEXT,
  total_score REAL DEFAULT 0
);

-- user_points
CREATE TABLE user_points (
  user_id BIGINT PRIMARY KEY,
  points INTEGER DEFAULT 0 NOT NULL
);

-- bot_config
CREATE TABLE bot_config (
  key TEXT PRIMARY KEY,
  value TEXT,
  channel_id BIGINT,
  message_id BIGINT
);

-- persistent_embeds
CREATE TABLE persistent_embeds (
  id SERIAL PRIMARY KEY,
  embed_type TEXT NOT NULL,
  channel_id BIGINT NOT NULL,
  message_id BIGINT NOT NULL,
  current_page INTEGER DEFAULT 0,
  last_content_hash TEXT,
  last_updated TIMESTAMPTZ DEFAULT NOW(),
  is_active BOOLEAN DEFAULT TRUE,
  UNIQUE (embed_type, channel_id)
);

-- queue_config
```

```
CREATE TABLE queue_config (
  queue_line TEXT PRIMARY KEY,
  channel_id BIGINT,
  pinned_message_id BIGINT
);

-- luxury_coins (NEW)
CREATE TABLE luxury_coins (
  user_id BIGINT PRIMARY KEY,
  balance INTEGER DEFAULT 0 NOT NULL
);
```

Indexes (exact):

idx_submissions_user_id ON submissions(user_id)

idx_submissions_queue_line ON submissions(queue_line)

idx_submissions_played_time ON submissions(played_time)

idx_submissions_submission_time ON submissions(submission_time)

idx_submissions_free_queue ON submissions(queue_line, total_score DESC, submission_time ASC) WHERE queue_line = 'Free'

idx_tiktok_interactions_session_id ON tiktok_interactions(session_id)

idx_tiktok_interactions_tiktok_account_id ON tiktok_interactions(tiktok_account_id)

idx_tiktok_accounts_linked_discord_id ON tiktok_accounts(linked_discord_id)

idx_tiktok_handles_search ON tiktok_accounts(handle_name text_pattern_ops)

idx_tiktok_handles_unlinked ON tiktok_accounts(linked_discord_id) WHERE linked_discord_id IS NULL

Extra table for watch time accounting (to implement Luxury Coins precisely):

```
CREATE TABLE tiktok_watch_time (
  id SERIAL PRIMARY KEY,
  session_id INTEGER REFERENCES live_sessions(id) ON DELETE CASCADE,
  tiktok_account_id INTEGER REFERENCES tiktok_accounts(handle_id) ON DELETE SET NULL,
  linked_discord_id BIGINT,
  watch_seconds INTEGER DEFAULT 0 NOT NULL,
  last_updated TIMESTAMPTZ DEFAULT NOW()
);
```

---

7) Points & engagement system (exact)

Point sources (exact):

Points stored in user_points table for Discord users.

Points also stored in tiktok_accounts.points per TikTok handle.

Points from TikTok events sync to linked Discord users.

Lifecycle (exact):

On TikTok event, award points to handle.

If handle linked to a Discord user, add to user_points.

sync_submission_scores() runs every 30 seconds to update submissions.total_score for Free queue ordering.

When Free-line song plays:

Reset submitter’s points to 0.

Reset linked TikTok handles' points to 0.

Backups (exact):

Hourly JSON backups of user_points and tiktok_accounts with timestamps.

---

8) Persistent auto-updating embed system (exact)

Persistent embeds table — already in DB schema above.

Auto-Refresh Loop (exact timing and behavior):

Run every 10 seconds (do not go below 10 s).

Introduce a 1-second delay between updating each embed to avoid rate limits.

Use a content_hash to skip updates when unchanged.

Defer interactions immediately (await interaction.response.defer()).

Event-driven updates:

Bot dispatches queue_update events on changes; views listen and update immediately.

Embed types (exact):

Public Live Queue: /setup-live-queue #channel — shows all active queues, pagination, emoji indicators.

Reviewer Main Queue: /setup-reviewer-channel #channel — shows reviewer view with Approve/Remove buttons.

Reviewer Pending Skips: same reviewer channel; lists submissions awaiting approval.

Self-healing (exact):

Every 5 minutes:

Verify all persistent embed configs.

If channel missing, create or disable.

If message missing, create new pinned message and re-register view.

Clean up messages without views.

Manual trigger: /selfheal (admin only).

---

9) Command reference (exact + new commands)

User commands (implement exactly):

/help — list commands

/submit — link modal; optional tiktok_handle

/submitfile — file upload; artist_name, song_title, note, optional tiktok_handle

/mysubmissions — list user's submissions

/link-tiktok handle — link tiktok handle (autocomplete)

/unlink-tiktok handle

/my-tiktok-handles — display linked handles

/balance — display Luxury Coins balance (NEW)

/buy-skip — spend 1000 Luxury Coins to move latest submission to 10 Skip (NEW)

Admin commands (exact + additions):

Queue management:

/next — get next submission to play

/move submission_id target_line

/remove submission_id

/clear-free

/open-submissions

/close-submissions

Channel setup:

/setup-live-queue #channel

/setup-reviewer-channel #channel

/set-submission-channel #channel — ENFORCE submissions only in this channel (NEW)

/setbookmarkchannel #channel

/setnowplayingchannel #channel

/setup-post-live-metrics #channel

/setdebugchannel #channel

TikTok integration:

/tiktok connect unique_id persistent=true

/tiktok status

/tiktok disconnect

System:

/show-settings — must include “Submissions Channel” and “Economy System Enabled” reporting

/selfheal — trigger healing manually

/cleandebugchannel

Admin link/unlink (NEW)

/admin-link @user handle — force-link TikTok handle to discord user

/admin-unlink @user handle — force-unlink

Economy admin (NEW)

/coins @user view

/coins @user add amount

/coins @user remove amount

Autocompletes (exact behavior):

tiktok_handle_autocomplete uses prefix search; 2+ characters minimum

unlinked_handle_autocomplete shows handles not linked

linked_handle_autocomplete shows handles linked to current user

Permissions: Admin commands require Administrator permission. All slash commands should validate permissions and return clear, ephemeral responses on permission failure.

---

10) Economy — Luxury Coins (exact requested behavior + integration)

Rules (implement exactly):

Earnings

Watch time: +1 Luxury Coin per 30 minutes (1800 seconds) of watch time on TikTok Live.

Track per tiktok_watch_time row (per session per handle / linked discord user). Award coins in increments: when watch_seconds >= 1800, increment coins by floor(watch_seconds / 1800), decrement watch_seconds by the awarded seconds (or reset to remainder).

Gifts: +2 Luxury Coins per 100 gifted coins.

For each tiktok_interactions gift event with coin_value, compute coins_awarded = floor(coin_value / 100) * 2 and add to luxury_coins.balance.

Spending

/buy-skip deducts 1000 Luxury Coins and moves user's most recent (eligible) submission into the 10 Skip queue (ties and validation follow original queue rules).

If user lacks coins, deny with friendly message and current /balance.

DB (repeat):

```
CREATE TABLE luxury_coins (
  user_id BIGINT PRIMARY KEY,
  balance INTEGER DEFAULT 0 NOT NULL
);
```

Concurrency & safety:

Use asyncio.Lock (or row-level transactions with FOR UPDATE) around all coin modifications to prevent race conditions.

All coin ops must be logged to the debug channel with user_id, operation, amount, reason, before_balance, after_balance.

Admin control:

/coins @user add amount → add

/coins @user remove amount → remove (floor at 0)

/coins @user view → view

Integration with gift processing:

When gift events processed to calculate skip-tier movement and points, also update luxury coins per rules (2 coins per 100 gift coins).

---

11) Persistent embed & display features (exact)

Public queue displays should show:

Tiers and top N items per page

Current playing item

Emojis indicating state (playing/pending/removed)

Pagination controls

Reviewer views should have Approve/Remove buttons and per-submission actions (Approve to a specific tier)

Views must use immediate defer() for interactions to avoid Unknown interaction errors.

---

12) Self-healing & monitoring (exact)

Automatic healing every 5 minutes (verify channels, messages, views; re-create missing messages; pin replacements if original deleted).

Manual /selfheal command for admins returns a detailed report of actions taken.

Metrics posted post-live in configured metrics channel: total session duration, participating handles, counts (likes/comments/shares/follows/subscribes/gifts), total coins received, engagement points, viewer min/max/avg. Format as ASCII table; truncate if > 1024 chars.

---

13) File structure & cogs (exact)

```
project/
├── main.py
├── database.py
├── requirements.txt
├── .env.example
├── README.md
└── cogs/
    ├── admin_cog.py
    ├── submission_cog.py
    ├── passive_submission_cog.py    # will only operate in Submissions channel
    ├── reviewer_cog.py
    ├── queue_cog.py
    ├── tiktok_cog.py
    ├── economy_cog.py               # Luxury Coins
    ├── embed_refresh_cog.py
    ├── self_healing_cog.py
    ├── metrics_cog.py
    └── checks.py
```

All cogs must be importable, registered in main.py and follow a consistent pattern with an async setup function for bot.load_extension.

---

14) Technical stack & dependencies (exact)

Python 3.12+

discord.py >= 2.3.2

TikTokLive >= 2.2.1

asyncpg >= 0.27.0 (production), aiosqlite (local dev fallback)

python-dotenv

psycopg2-binary (if needed for sync paths, but prefer asyncpg)

Additional optional: alembic for migrations (nice-to-have)

Intents & permissions required (exact):

Intents: guilds, messages, message_content

Bot perms: Send Messages, Embed Links, Attach Files, Read Message History, Add Reactions, Manage Messages, Manage Channels, View Channels

---

15) Critical implementation notes (exact)

Rate limiting

Embed refresh: 10-second interval overall; 1-second between updating each embed.

Avoid 429 errors using content-hash to skip redundant updates.

Button interactions

Defer interactions immediately: await interaction.response.defer().

Guarantee follow-up messages within 3 seconds.

TikTok API considerations

Be robust to schema changes (e.g., nickName vs nick_name).

Log full event payloads to debug logs when unexpected fields appear.

Database performance

Implement the composite Free queue index above.

Use FOR UPDATE locks on queue ops.

Score synchronization

sync_submission_scores() every 30 seconds.

Passive submission edge cases

If file has wrong content_type, reject.

Max file size: 25 MB.

Delete original messages in submissions channel after processing (configurable).

---

16) Testing & deployment (exact)

Testing strategy

Unit tests for DB functions: queue priority, gift tier mapping, handle linking, point calculations.

Integration tests simulate: submit → gift → move to skip, /next ordering, economy transactions.

Local dev

Use SQLite (aiosqlite) for local dev and asyncpg for production PostgreSQL.

Hosting

Recommend Replit, Railway, Render, DigitalOcean, or AWS EC2.

Minimum server: 512MB RAM, 1 CPU, 1GB disk.

Environment variables (exact list)

DISCORD_BOT_TOKEN

DATABASE_URL (postgres URI or sqlite+aiosqlite:///music_queue.db)

GUILD_ID (optional)

DEBUG_CHANNEL_ID (optional)

ALLOW_ANY_HANDLE_LINKING (default false)

EMBED_REFRESH_INTERVAL (default 10)

SELF_HEAL_INTERVAL (default 300)

---

17) Deliverables (what Replit agent must produce)

1. A complete runnable repository implementing every item above.

2. main.py that loads cogs, sets up DB, registers commands, and starts the bot.

3. database.py that creates all tables and indexes if absent and exposes async helper functions.

4. Full cogs implementing the exact commands and behaviors (submission_cog, passive_submission_cog constrained to Submissions channel, queue_cog, tiktok_cog, economy_cog, embed_refresh_cog, reviewer_cog, admin_cog, self_healing_cog).

5. Detailed README.md with setup, .env example, and deployment tips.

6. Unit test scaffolding covering core flows (queue selection, gift handling, coin awarding, /buy-skip, admin coin operations).

7. Logging to console + file; debug channel logging for coin ops and important events.

8. A migrations/ or database.py system to ensure DB schemas are created/updated on startup.

9. All slash commands registered on startup (fast sync if GUILD_ID present).

10. Code comments explaining key algorithms, locks, and tradeoffs.

---

18) Implementation hints & developer notes (for agent)

Use a cog-based pattern; make each cog independent and unit-testable.

Use asyncio.Lock() or transaction-level locking on critical sections: queue selection, coin modifications, skip movement.

When awarding watch-time coins: accumulate seconds in tiktok_watch_time, increment luxury_coins for each 1800-second chunk and persist remainder.

When processing gifts: award both points (per original spec) and Luxury Coins (2 per 100 coins).

Provide a DEBUG mode to simulate TikTok events during tests.

Keep all user-facing messages concise and developer-friendly (ephemeral for command confirmations where appropriate).

Keep a bot_config key for submission_channel_id, metrics_channel_id, debug_channel_id, embed_configs, allow_any_handle_linking.

---

19) Acceptance criteria (how to verify the agent succeeded)

All database tables and indexes (above) exist and are created automatically on first run.

Submissions are only accepted in the configured Submissions channel; other channels cannot create submissions.

Admin /admin-link and /admin-unlink commands function and produce audit logs in debug channel.

Luxury Coins implemented exactly: watch-time awarding (1 coin/30 min) and gift-based awarding (2 coins/100 gifted coins). Admin coin commands work and log operations.

/buy-skip deducts exactly 1000 coins and moves user’s most recent submission to the 10 Skip tier.

sync_submission_scores() runs every 30s; embed refresh runs every 10s; self-heal runs every 5 minutes.

Gift processing logic matches the diamond_count mapping and skip-tier mapping exactly; streak handling avoids duplicates.

Persistent embeds survive a bot restart and self-heal can rebuild them.

Unit tests provided and run locally.

Closing (agent instructions)

Implement the full codebase exactly as described above. Do not rely on external files — this prompt is the authoritative spec. Produce the repository files ready to run on Replit, including a run command and README. Ensure all commands, DB schemas, periodic tasks, and behaviors are reproducible and testable.

Paste this prompt into Replit’s AI agent. It must produce the full project file tree and code implementing the exact bot described above, integrating your requested changes.