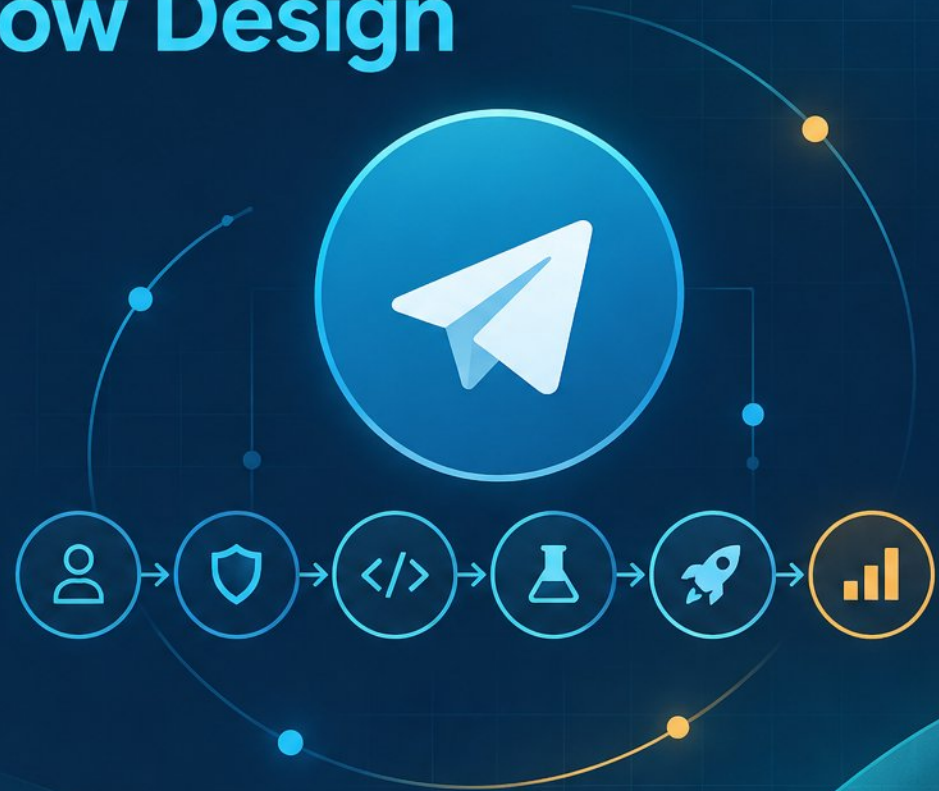


Introduction to My Telegram Org

API Development Tools: Selection and Workflow Design

- Deliberate tool selection prevents expensive rework later
- Lifecycle: registration, authorization, development, testing, deployment, monitoring
- “Authorized” means BotFather tokens, user consent, and policy limits
- Four 2026 API surfaces: Bot API, TDLib, MTProto, Gateway API
- Bot API fits most first authorized applications



The Telegram API Landscape:

Four Surfaces, Four Different Jobs

Same platform. Different entry points. Different responsibilities.



Bot API

HTTPS/JSON, acts as a bot account via BotFather token



TDLib

cross-platform client library for user accounts; api_id, api_hash, phone login



MTProto

binary protocol of official apps; full TL schema, you manage sessions



Gateway API

paid verification codes via Telegram instead of SMS



Order: bot work → Bot API; person sees it → TDLib; raw MTProto last

Bot API vs. MTProto: Capabilities, Limits, and When Each Wins

Bot API

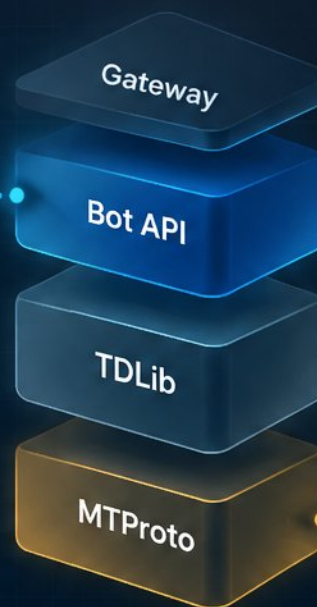
HTTPS JSON (webhook or polling)



Files: ~20 MB down / 50 MB up on Bot API



Scope: curated method subset



MTProto

Persistent binary socket with pushed updates



Files: up to ~2 GB on MTProto



Scope: the full TL schema (history, dialogs, participants)



Account model

Bot API is bot-only; MTProto authorizes bots or user accounts



Transport

HTTPS JSON with webhook or polling vs. persistent binary socket with pushed updates



Files

~20 MB down / 50 MB up on Bot API vs. up to ~2 GB on MTProto



Scope

curated method subset vs. the full TL schema (history, dialogs, participants)



Errors

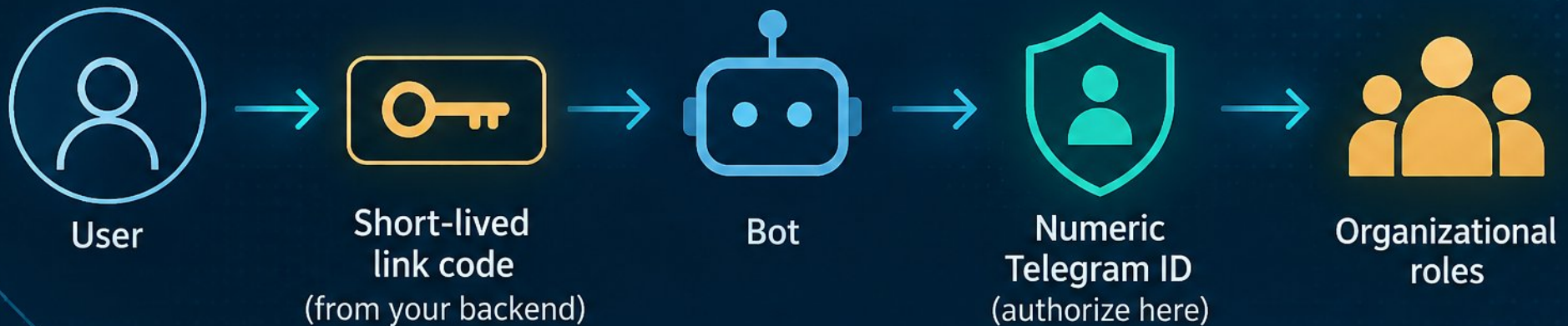
coarse HTTP classes vs. distinct classes like FLOOD_WAIT and PEER_FLOOD



Rule

start with the Bot API; escalate only for userbots, history, or scale

Authorization and Consent Models for Organizational Deployments



- A bot cannot start a chat: the user must /start it first
- Pair a chat with a short-lived, single-use link code from your backend
- Telegram login authenticates identity, not permissions — verify the signed hash
- Authorize on the immutable numeric user ID, never the username
- Privacy Mode filters group messages; DMs always reach the bot, so gate them yourself

Choosing Your Development Toolchain



Python

aiogram

python-telegram-bot

Pyrogram

Python in 2026: aiogram for async-first production bots; python-telegram-bot for docs and stability; Pyrogram for MTProto.



TypeScript/JavaScript

grammY

Telegraf

node-telegram-bot-api

TypeScript/JavaScript: grammY is the modern default; Telegraf suits existing codebases; node-telegram-bot-api is unmaintained.



Go

telebot

auto-generated wrappers

Go: telebot and auto-generated wrappers fit high-throughput, single-binary deployments.



Evaluation rubric

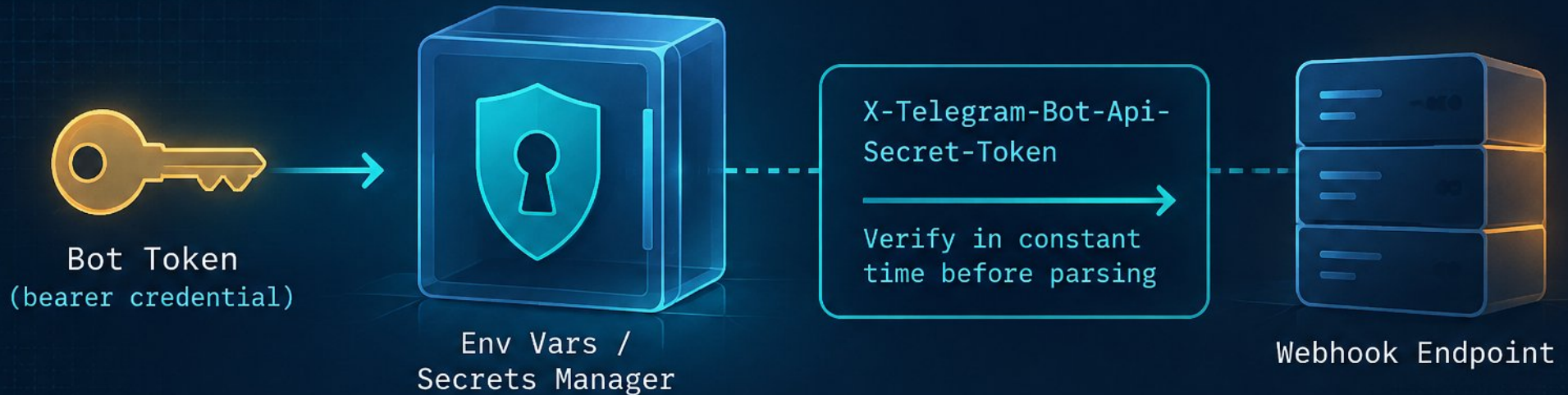
maintenance cadence, Bot API version, type coverage, middleware, community.



Setup hygiene

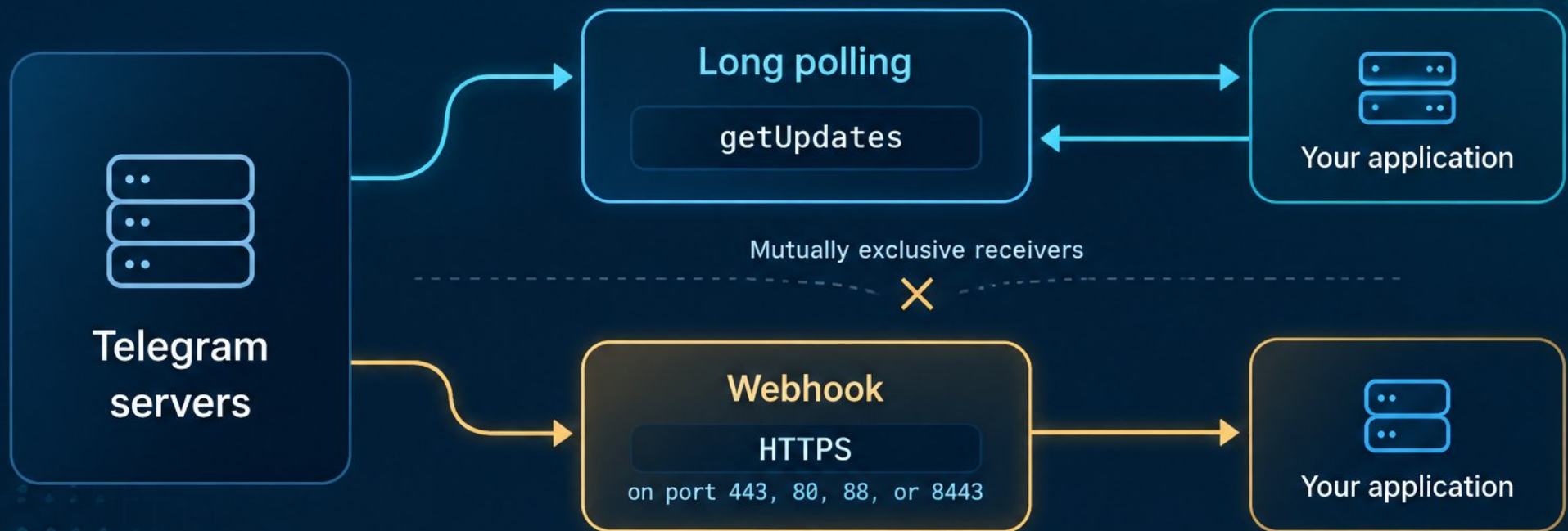
containers, secrets out of source, pinned dependencies, and a rule for raw API calls.

Secrets, Tokens, and Webhook Authentication



- A bot token is a bearer credential: no second factor, full control
- Store tokens in env vars or a secrets manager; never log full request URLs
- Use separate development and production tokens; a leak stays contained
- Set a secret_token and verify the header in constant time before parsing
- Revocation is instant — inventory every consumer before you rotate

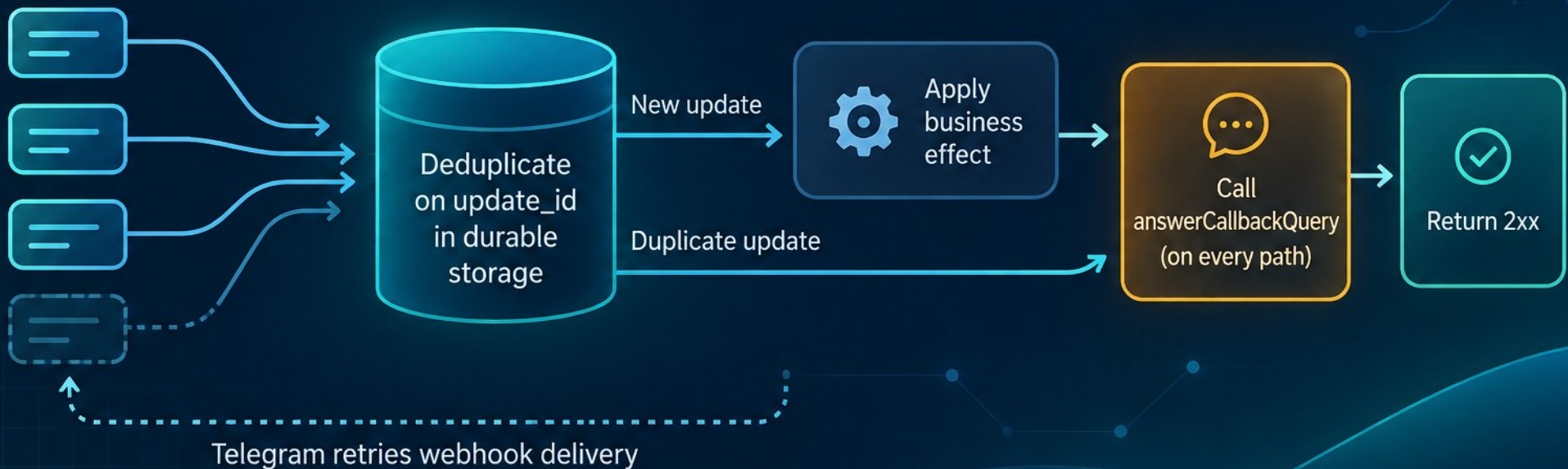
Inbound Architecture: Webhook or Long Polling



- Mutually exclusive receivers: with a webhook set, `getUpdates` returns 409 conflict
- Long polling: no domain, certificate, or inbound port – but one polling process per token
- Webhook: HTTPS on port 443, 80, 88, or 8443; bot becomes a request handler
- Acknowledge with 200 fast, then process asynchronously; slow endpoints get redelivered
- Updates live 24 hours max; `getWebhookInfo` is your first diagnostic call

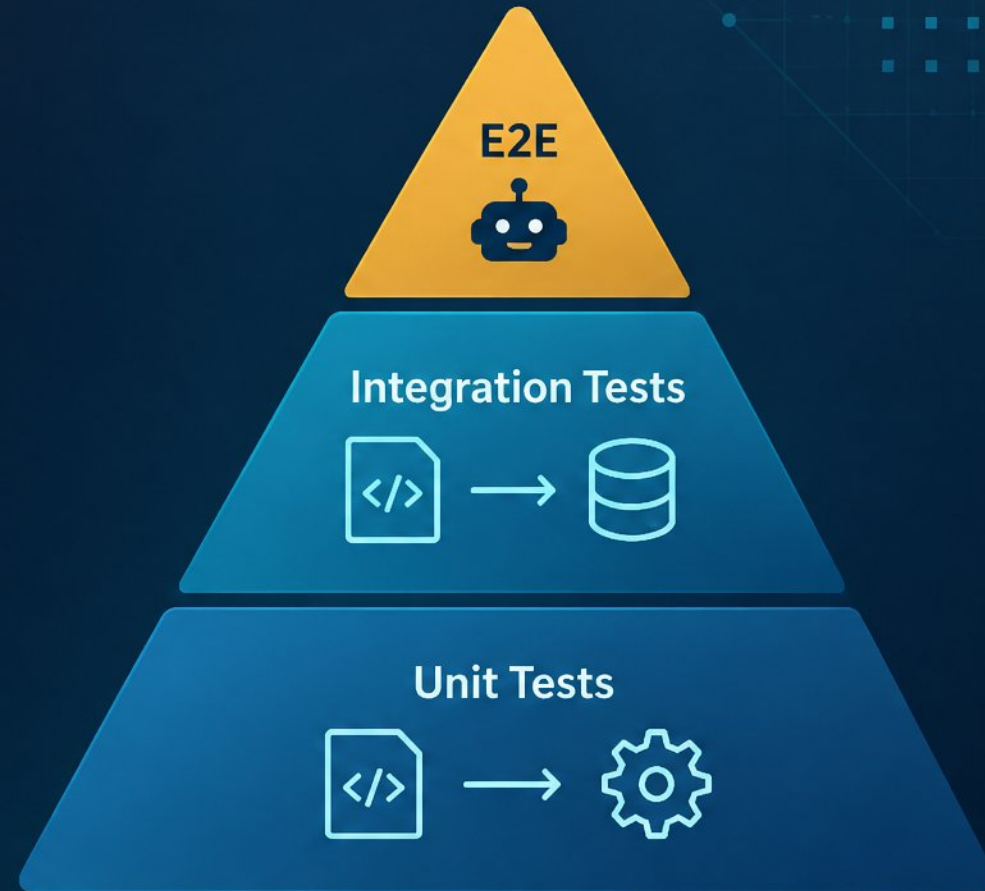
Designing Reliable Update Handling: Idempotency and State

- Telegram retries webhook delivery: treat every update as at-least-once
- Deduplicate on update_id in durable storage before applying any business effect
- Return 2xx for an already-completed duplicate without resending the outcome
- Call answerCallbackQuery on every path, including rejections, so clients don't spin
- Keep state in a database; Telegram stores no update or message history for you



Development Workflow and Testing Strategy

- Test pyramid: many mocked unit tests on handlers, fewer integration tests through the dispatcher, a small E2E tier
- Integration tests pair a mock Bot API with a real state store; run them through the dispatcher
- Mock `api.telegram.org` locally: removes the token, network flakiness, and rate limits from CI
- Use webhook mode or `handleUpdate` so tests never race the polling loop for the same updates
- Keep PR pipelines under ~10 minutes; push slow E2E checks to a nightly job



Keep PR pipelines under ~10 minutes;
push slow E2E checks to a nightly job

Deployment, Rate Limits, and Sending at Scale

- Match deployment to receiver: polling runs as a worker, webhooks suit HTTP or serverless
- Design around ~1 msg/s per chat, ~20/min per group, ~30/s overall
- On 429, honor parameters.retry_after exactly, then back off with jitter
- Cap total retry budget so one bad hour does not become a cascade
- Queue slow work with pacing; keep handlers fast and updates flowing
- Deploy safely: verify getMe and setWebhook, shift traffic, retire old process

Send Rate Limits



~1 msg/s
per chat



~20/min
per group



~30/s
overall

Rate Limit Handling (429)

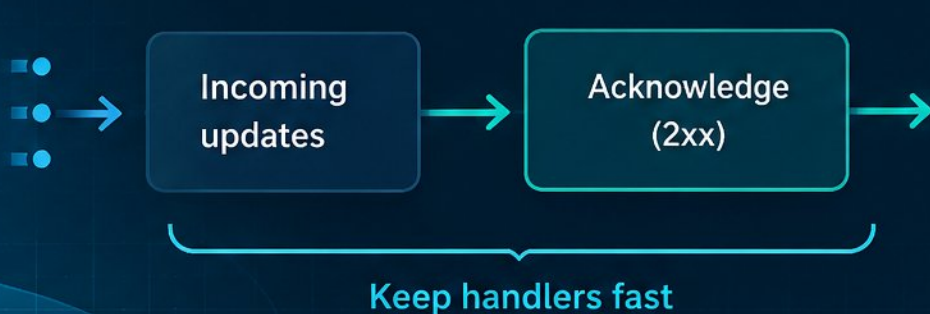
429



honor
parameters.retry_after
exactly



then back off
with jitter



Monitoring Signals and Incident Response

- Update arrival is your best signal: alert on silence during active hours
- Poll `getWebhookInfo` for `pending_update_count` and `last_error_date`
- Compare `last_error_date` to now — it is not cleared on recovery
- Canary account plus TLS expiry check three weeks out
- Run monitors and alerts outside the failure domain
- Write the runbook before you need it



Workflow Design Patterns You Can Reuse



Notification bot: one-way outbound alerts — the Bot API is usually enough



Moderation assistant: reports queue up; confident actions auto-run, ambiguous cases escalate with approve/deny buttons



Content pipeline: generation writes a row as pending and stops; only explicit approval lets a publisher pick it up



Classroom bot: structured intake instead of raw DMs, visible queue, per-class data minimization



Adaptation signals: rising queue depth, repeated reviewer corrections, or a new compliance rule



Common Risks, Compliance, and Troubleshooting Playbook



Top failures: leaked tokens, logged request URLs, missing secret-token checks



Never authorize on usernames; compare immutable numeric user IDs



SSL error = TLS chain or hostname; 4xx = path mismatch or secret rejection



Read timeout = slow handler; 409 conflict = forgotten webhook or two pollers



Collect minimal data, delete on request, rate-limit per user, cap input



Fix the integration shape before patching around it



Capstone: Your Tool Selection and Workflow Plan

PLAN // BUILD // REVIEW / RELEASE



Plan



Build



Peer Review



Pre-release
Checklist



First Production
Release



- **Decision worksheet:** API surface, language/SDK, receiver model, state store, deployment target — record every reason



- **Draft a 30-day plan:** BotFather registration to first production release; schedule auth and secrets first



- **Pre-release checklist:** numeric-ID authorization, token and webhook-secret rotation, update_id dedup, 429 handling, monitoring, runbook



- **Peer review:** test a classmate's plan against the checklist and find the assumption most likely to break



- **Keep learning:** official Bot API reference and changelog, Papercraft Book, framework docs and chats

PersonWise.

PersonWise • AI digital-human course platform



Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/ce7acf52/public