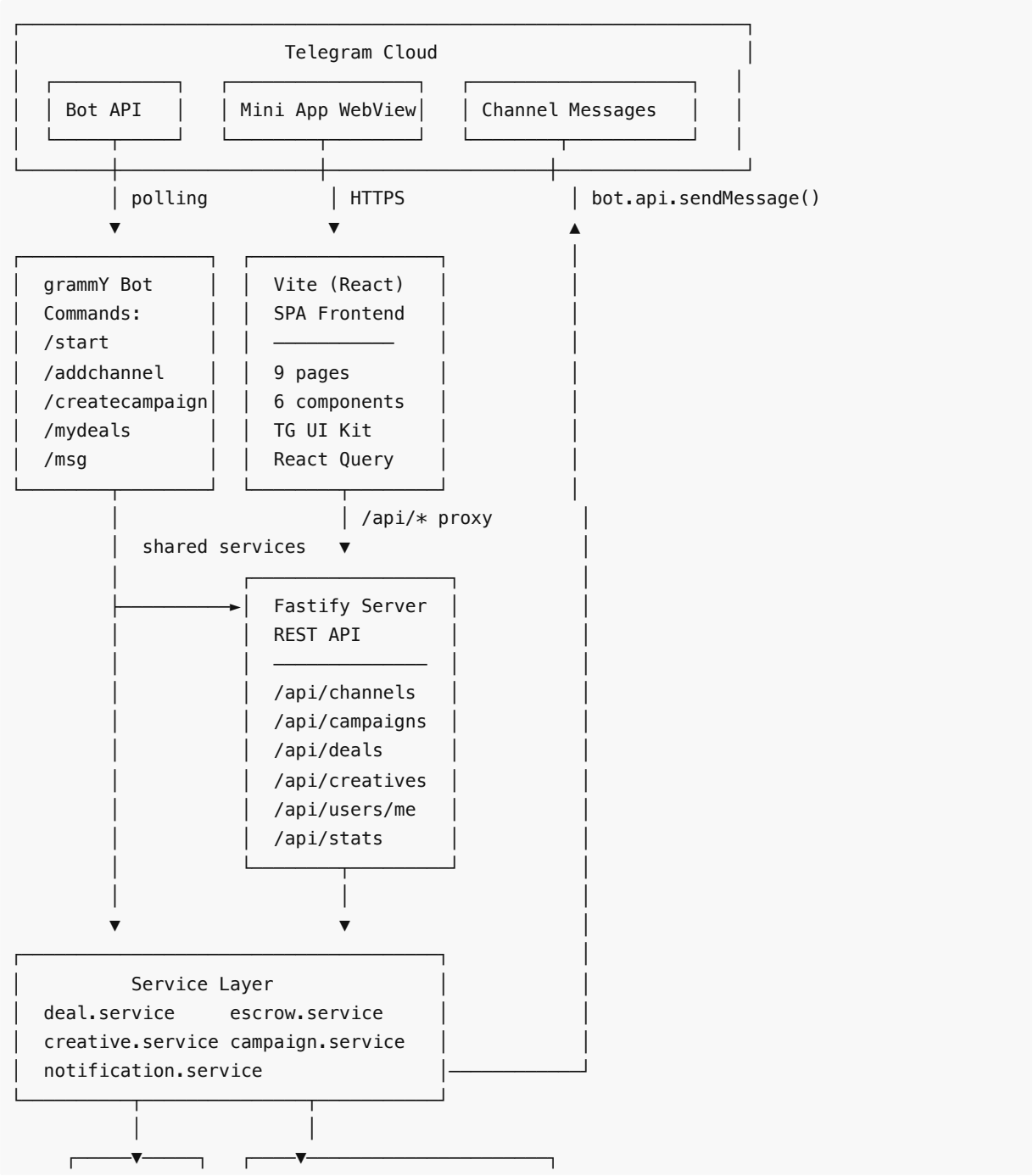
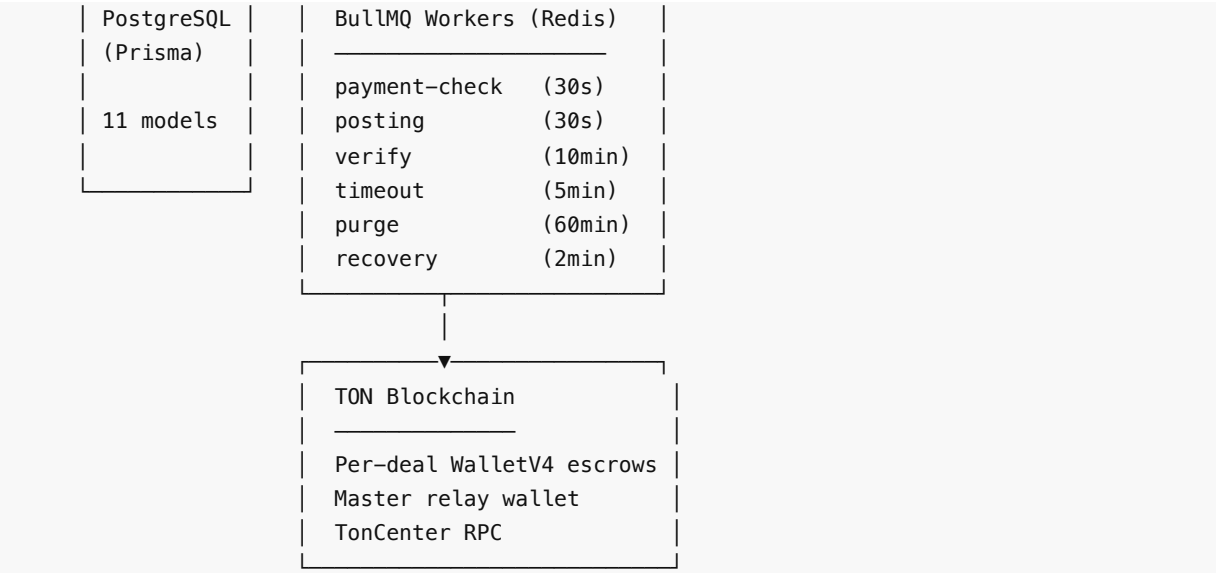
AdVault — Architecture & System Design

A trustless Telegram ads marketplace where advertisers fund per-deal TON escrow wallets, channel owners submit creatives for approval, and a bot auto-posts, verifies delivery after 24h, then releases funds — no middleman, no trust required.

System Architecture





Key Architectural Decisions

Monolith with logical separation. A single Node.js process runs the API server, bot, and workers. Services are cleanly separated but share the same Prisma client and process — simple to deploy, no inter-service latency.

Per-deal escrow wallets. Each deal gets its own TON WalletContractV4 with an AES-256-GCM encrypted mnemonic. Funds are isolated per-deal — one compromised wallet doesn't affect others.

Two-hop privacy relay. Payouts go `escrow → master → recipient`. On-chain observers only see the master wallet paying out, breaking the link between advertiser deposits and owner payouts.

Encryption at rest. Creative content (ad text, media URLs) and wallet mnemonics are encrypted with AES-256-GCM using per-encryption random IVs. Data is purged after 30 days, leaving only a SHA-256 receipt as proof.

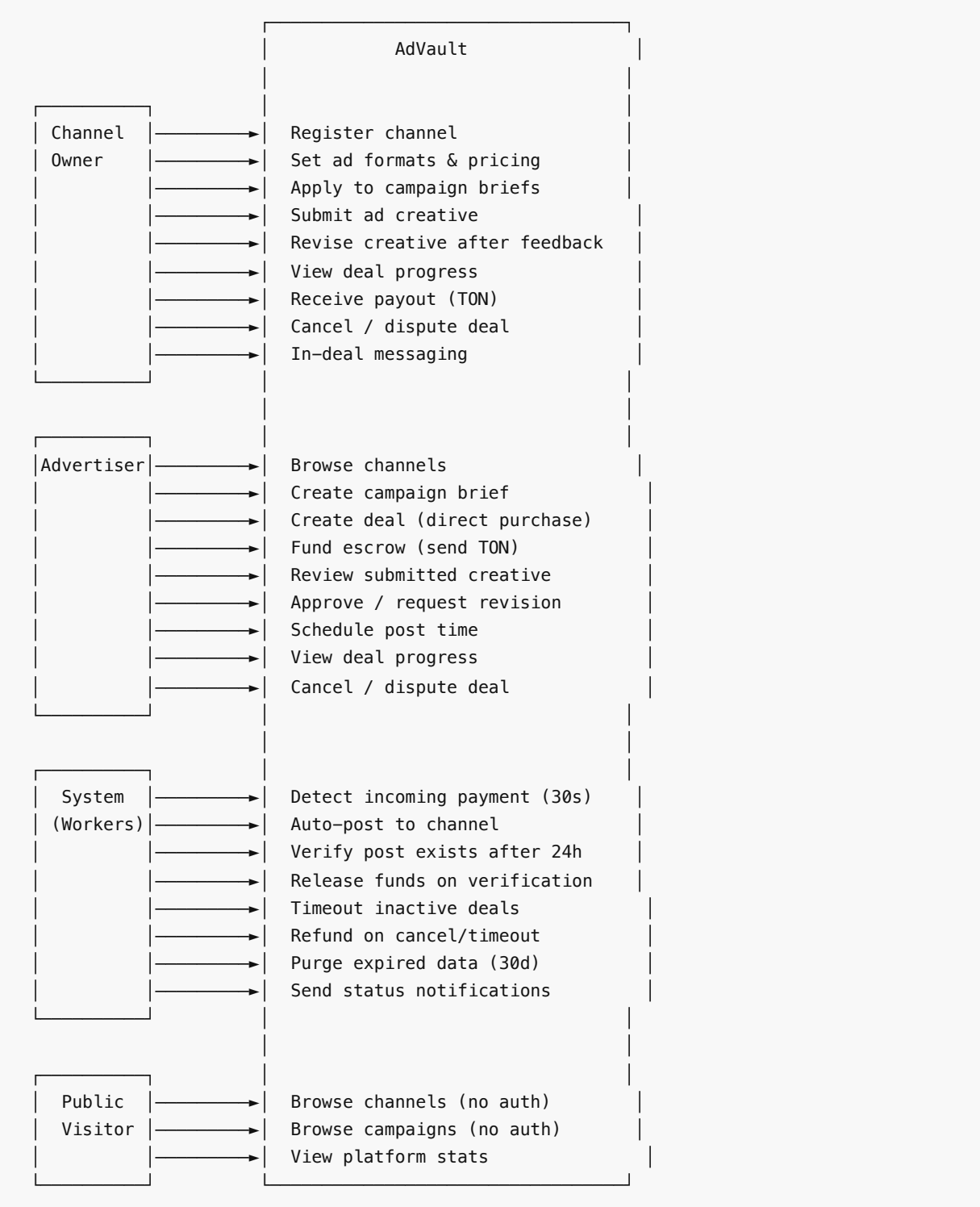
BullMQ worker queues. Six independent workers handle async operations (payment detection, auto-posting, verification, timeouts, data purge, transfer recovery). Each runs on its own Redis queue with independent polling intervals.

Tech Stack

Layer	Technology
HTTP server	Fastify 5
Telegram bot	grammY 1.35 + conversations plugin
Frontend	React 19 + Vite 6 + React Router 7 + TanStack Query 5
UI components	@telegram-apps/telegram-ui (official TG Mini App kit)
Database	PostgreSQL + Prisma 6 ORM
Job queues	BullMQ 5 + Redis (ioredis)
Blockchain	@ton/ton + @ton/crypto + @ton/core (TonCenter RPC)
Validation	Zod 3
Auth	Telegram initData HMAC-SHA256 verification

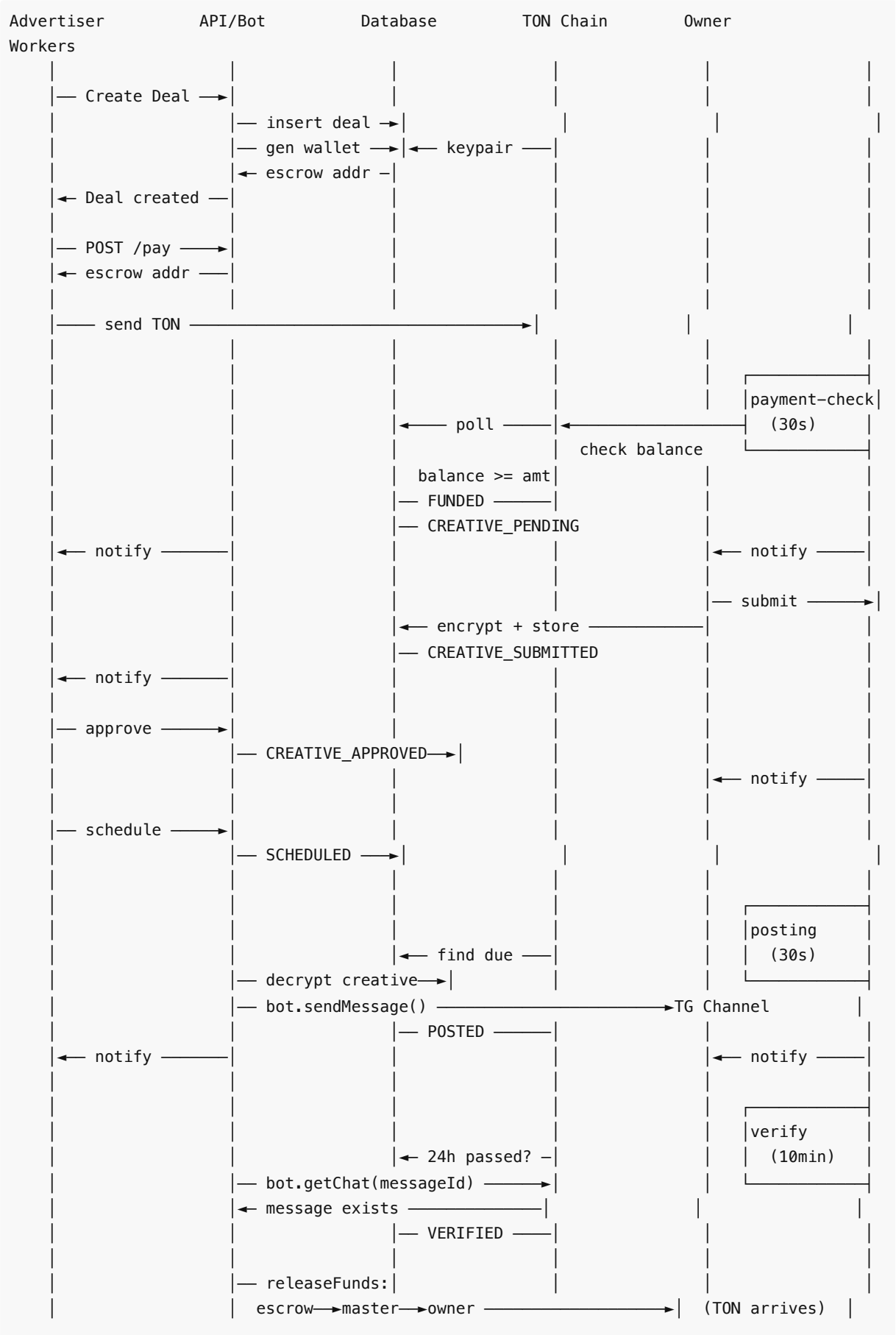
Language	TypeScript 5 throughout
Tests	Vitest 4 (647+ tests)

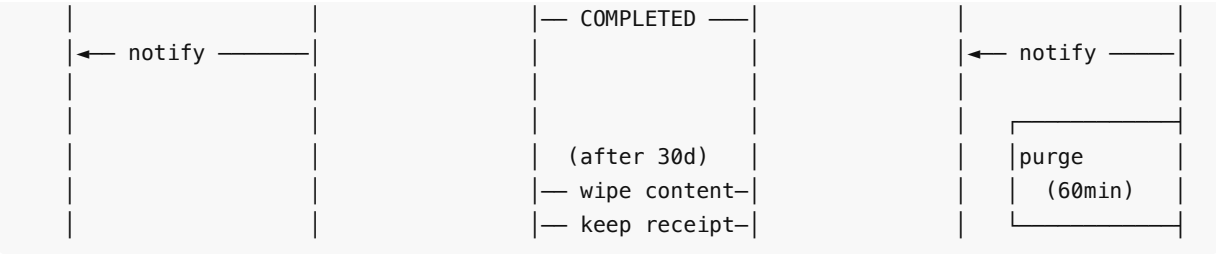
Use-Case Diagram



Three actor types: Channel Owner, Advertiser, and System (automated workers). A user can set their role to "Both" and act as either in different deals.

Sequence Diagram — Full Deal Lifecycle





Deal State Machine

14 States

```
PENDING_PAYMENT ──► FUNDED ──► CREATIVE_PENDING ──► CREATIVE_SUBMITTED
                                                          │        │
                                                          ▼        ▼
                                                    CREATIVE_   CREATIVE_
                                                    REVISION    APPROVED
                                                      │            │
                                                      └──► (back)  ▼
                                                                SCHEDULED ──► POSTED ──►
VERIFIED ──► COMPLETED

Any active state ──► CANCELLED
Any funded state ──► REFUNDED (auto-refund on cancel)
Any active state ──► TIMED_OUT ──► REFUNDED
Post deleted     ──► DISPUTED  ──► REFUNDED or COMPLETED (admin decision)
```

Timeout Durations

Status	Auto-timeout
PENDING_PAYMENT	24 hours
FUNDED	72 hours
CREATIVE_PENDING	72 hours
CREATIVE_SUBMITTED	96 hours
CREATIVE_REVISION	72 hours

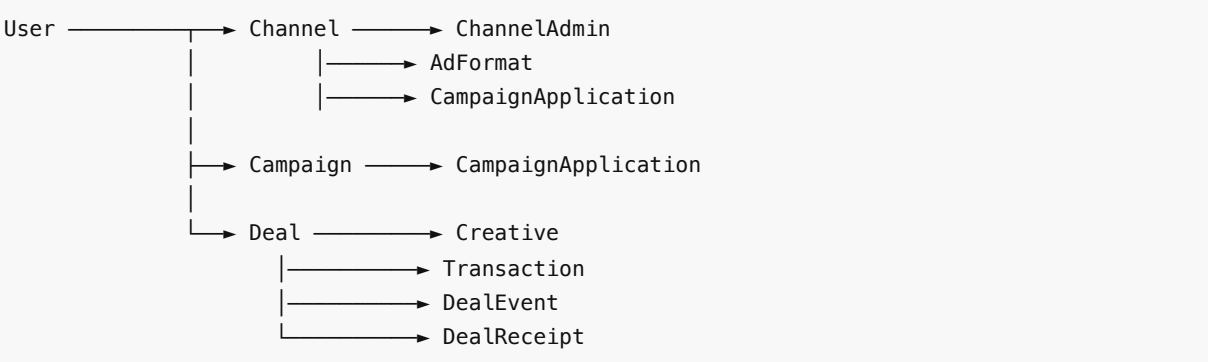
After timeout, the deal transitions to `TIMED_OUT` and any funded amount is auto-refunded.

Transition Rules

```
PENDING_PAYMENT    → [FUNDED, CANCELLED, TIMED_OUT]
FUNDED             → [CREATIVE_PENDING, CANCELLED, REFUNDED, DISPUTED, TIMED_OUT]
CREATIVE_PENDING   → [CREATIVE_SUBMITTED, CANCELLED, REFUNDED, DISPUTED, TIMED_OUT]
CREATIVE_SUBMITTED → [CREATIVE_APPROVED, CREATIVE_REVISION, CANCELLED, REFUNDED, DISPUTED,
TIMED_OUT]
CREATIVE_REVISION  → [CREATIVE_SUBMITTED, CANCELLED, REFUNDED, DISPUTED, TIMED_OUT]
CREATIVE_APPROVED  → [SCHEDULED, CANCELLED, REFUNDED, DISPUTED, TIMED_OUT]
SCHEDULED          → [POSTED, CANCELLED, REFUNDED, DISPUTED, TIMED_OUT]
POSTED             → [VERIFIED, DISPUTED, TIMED_OUT]
VERIFIED           → [COMPLETED]
```

```
COMPLETED          → [] (terminal)
CANCELLED          → [] (terminal)
REFUNDED           → [] (terminal)
DISPUTED           → [REFUNDED, COMPLETED]
TIMED_OUT          → [REFUNDED]
```

Data Model (11 Tables)



User — `telegramId`, `role` (OWNER | ADVERTISER | BOTH), `tonWalletAddress`

Channel — `telegramChatId`, `subscribers`, `avgViews`, `avgReach`, `language`, `category`, `botIsAdmin`, `isVerified`

AdFormat — `formatType` (POST | FORWARD | STORY | CUSTOM), `priceTon`, `label`, `description`

Campaign — `budgetTon`, `targetSubscribersMin/Max`, `targetLanguage`, `targetCategory`, `status`

Deal — `amountTon`, `status` (14 states), `escrowAddress`, `escrowMnemonicEncrypted`, `scheduledPostAt`, `postedMessageId`, `ownerAlias`, `advertiserAlias`, `timeoutAt`

Creative — `contentText` (encrypted), `mediaUrl` (encrypted), `version`, `status`, `reviewerNotes`

Transaction — `type` (DEPOSIT | RELEASE | REFUND), `amountTon`, `txHash`, `fromAddress`, `toAddress`

DealEvent — `eventType`, `oldStatus`, `newStatus`, `actorId`, `metadata` (JSON audit trail)

DealReceipt — `dataHash` (SHA-256), survives data purge as proof of deal completion

---

Background Workers

Worker	Interval	What it does
payment-check	30s	Polls TON escrow wallet balances for `PENDING_PAYMENT` deals. Transitions to `FUNDED` → `CREATIVE_PENDING` when deposit detected.
posting	30s	Finds `SCHEDULED` deals past their `scheduledPostAt` time. Decrypts creative, posts to channel via bot API, saves `postedMessageId`.
verify	10min	Checks `POSTED` deals after 24h hold. Verifies message still exists in channel. If intact → `VERIFIED` → releases funds. If deleted → `DISPUTED`.
timeout	5min	Finds deals past their `timeoutAt`. Transitions to `TIMED_OUT`, triggers refund if funded.
purge	60min	Wipes sensitive data (creative text, media URLs, mnemonics, tx hashes) from terminal deals older than 30 days. Preserves `DealReceipt` with SHA-256 hash.

		Processes max 50 per run.
recovery	2min	Retries failed two-hop transfers. If hop 1 (escrow → master) succeeded but hop 2 (master → recipient) failed, recovery picks up where it left off. Prevents funds from getting stuck in the master wallet.

Security Model

Authentication: Telegram `initData` validated via HMAC-SHA256 using `BOT_TOKEN`. Auth date checked within 24h window. Development mode supports `x-dev-user-id` header bypass.

Identity masking: Each deal generates random aliases (`Seller-a7x3`, `Buyer-m9p2`). The API strips real identity information based on the requester's role — neither party learns the other's Telegram identity during the deal.

Encryption at rest: AES-256-GCM with random 12-byte IVs for creative content and wallet mnemonics. Same plaintext encrypts differently each time, preventing pattern analysis.

Fund privacy: Two-hop relay (`escrow → master → recipient`) ensures on-chain payout transactions all originate from the same master wallet. Observers cannot correlate deposits to payouts.

Data minimization: The purge worker deletes sensitive data 30 days after deal completion. Only a `DealReceipt` with a SHA-256 hash remains as cryptographic proof.

API Endpoints

Channels

```
GET    /api/channels                    — Browse (filters: language, category, subscribers,
price, page)
GET    /api/channels/:id                — Detail with admins
POST   /api/channels                    — Register (auth)
PUT    /api/channels/:id                — Update (auth, owner)
POST   /api/channels/:id/refresh-stats  — Refresh stats (auth)
POST   /api/channels/:id/formats        — Add ad format (auth, owner)
GET    /api/channels/:id/admins         — List admins (auth)
POST   /api/channels/:id/admins         — Add admin (auth, owner)
GET    /api/channels/:id/stats          — Channel stats (public)
```

Campaigns

```
GET    /api/campaigns                   — Browse (filters: budget, language, category,
status, page)
GET    /api/campaigns/:id               — Detail
POST   /api/campaigns                   — Create (auth)
PUT    /api/campaigns/:id               — Update (auth, advertiser)
POST   /api/campaigns/:id/apply         — Apply (auth, channel owner)
```

Deals

```
GET    /api/deals                       — User's deals (filter: role=owner|advertiser)
GET    /api/deals/:id                   — Detail with identity masking (auth)
POST   /api/deals                       — Create (auth)
POST   /api/deals/:id/pay               — Get escrow address (auth)
```

```
POST   /api/deals/:id/cancel            — Cancel (auth)
POST   /api/deals/:id/dispute           — Dispute with reason (auth)
POST   /api/deals/:id/creative/schedule — Schedule post time (auth)
GET    /api/deals/:id/receipt           — Receipt after purge (auth)
```

Creatives

```
POST   /api/deals/:id/creative          — Submit creative (auth, owner)
POST   /api/deals/:id/creative/approve  — Approve (auth, advertiser)
POST   /api/deals/:id/creative/revision — Request revision with notes (auth, advertiser)
GET    /api/deals/:id/creatives         — All versions (auth)
```

Users & Stats

```
GET    /api/users/me                    — Profile (auth)
PUT    /api/users/me                    — Update profile (auth)
GET    /api/users/me/channels           — My channels (auth)
GET    /api/users/me/campaigns          — My campaigns (auth)
GET    /api/stats                       — Platform stats (public)
GET    /api/health                      — Health check (public)
```

Scalability & Extensibility

What scales well today

Database layer. Prisma + PostgreSQL with proper indexing. Adding new models or fields is a migration away.

Worker architecture. BullMQ on Redis is horizontally scalable. Each worker type runs on its own queue with independent intervals. You can run multiple worker instances across machines (BullMQ handles job locking) and add new workers (analytics, fraud detection) by adding a queue name and processor function.

State machine. The `VALID_TRANSITIONS` map and `transitionDeal()` function make it trivial to add new states. Adding a `MODERATOR_REVIEW` state means adding one enum value, one transition rule, and one timeout — the rest of the system (notifications, workers, UI) follows naturally.

Service layer separation. Each service (`deal` , `escrow` , `creative` , `notification` , `campaign`) owns its domain. Adding a feature like milestone-based payments means a new service + route, not modifying existing ones.

TON integration. The wallet layer abstracts behind `generateWallet` , `getEscrowBalance` , `transferFunds` . Swapping chains or adding multi-chain support means implementing a new adapter behind the same interface.

What needs work at scale

Concern	Current	Change needed
Process model	Single process runs API + bot + workers	Split into 3 deploys: API, bot, workers. BullMQ already supports this — workers just need their own process entry point.
Payment detection	Poll every 30s for all pending deals	Replace with TON webhook/streaming (TonCenter v3 websocket). Polling works fine under ~1000 concurrent deals.
Bot mode	Long polling via `bot.start()`	Switch to webhook mode (`bot.api.setWebhook()`). grammY supports both — one config change.

Auth	Telegram initData HMAC validation per request	Add JWT session tokens after initial validation to reduce per-request crypto.
Data purge	50 deals per hourly run	Sufficient for thousands. At 100k+, move to database-level TTL or partitioned tables.
Media storage	Creative media URLs point to external hosts	Add S3/R2 upload service for creative assets.
Search	SQL filtering with basic WHERE clauses	Add PostgreSQL full-text search or Elasticsearch for channel/campaign discovery.
Rate limiting	None	Add Fastify rate-limit plugin before production.

Built-in extensibility points

- **Role system** (`OWNER | ADVERTISER | BOTH`) — extend to `MODERATOR` , `ADMIN`
- **Ad format types** (`POST | FORWARD | STORY | CUSTOM`) — new formats are just enum values
- **Campaign applications** — accept/reject flow is ready for auto-matching algorithms
- **DealEvent audit trail** — every state change logged with actor + metadata, ready for analytics dashboards
- **Deal receipts** — SHA-256 hashed proof survives data purge, useful for disputes or compliance
- **Channel admins** — `canManageDeals` and `canManagePricing` permissions modeled, supporting team accounts
- **Config-driven tuning** — `PLATFORM_FEE_PERCENT` , `PURGE_AFTER_DAYS` , per-status timeouts — all adjustable without code changes

Verdict

The architecture is a well-structured monolith with clear service boundaries. It handles a contest demo and early traction without changes. For production scale (10k+ concurrent deals), the main moves are: split the process into 3 deploys (API, bot, workers), switch bot to webhook mode, and add rate limiting. The data model, state machine, and queue system don't need rearchitecting — they're already designed for horizontal scaling.