

Runtime TelegramBot Titan Edition:

Architecture, Resilience, and Security in a Multi-Channel Broadcast Automation System

Simone Pizzi (Runtime Radio)

Document Revision: 1.0 (Software Reference v1.7.x “OmniSync”)
March 2026

Abstract

The automated distribution of digital content through instant messaging platforms raises complex engineering challenges, primarily related to concurrency management, compliance with network constraints (rate-limiting), and the security of sensitive data. This document analyzes the research and development path that led to the creation of *Runtime TelegramBot Titan Edition*, a desktop platform designed for the asynchronous orchestration of RSS feeds, Podcasts, and YouTube channels toward the Telegram ecosystem. We examine the limitations of legacy automation systems based on procedural scripting and illustrate the architectural solutions adopted to resolve them: the implementation of a *Producer-Consumer* design pattern for the separation of I/O loads, the development of heuristic scraping techniques for data extraction from closed platforms, and the introduction of the proprietary *OmniSync* (.rtb) protocol to ensure configuration portability by cryptographically binding API tokens to the host hardware.

Contents

1	Chapter 1: Preface and Operational Context	3
1.1	1.1 The State of the Art of Editorial Automation	3
1.2	1.2 The “Legacy” Problem and the Need for Evolution	3
2	Chapter 2: The Paradigmatic Leap (From Procedural Script to Desktop Ecosystem)	4
2.1	2.1 The Choice of Technology Stack	4
2.2	2.2 Data Layer Transition: From Flat Files to Relational SQLite	4
3	Chapter 3: Engine Architecture (Producer-Consumer and Rate-Limit Management)	5
3.1	3.1 Synchrony Limits and UI Freeze	5
3.2	3.2 Job Queue Implementation (Decoupled Asynchrony)	5
3.3	3.3 Dynamic Network Error Management (Telegram Retry Logic)	5
4	Chapter 4: YouTube Integration (Reverse Engineering and Anti-Spam Heuristics)	7
4.1	4.1 The Failure of the XML Standard and the “Trap” of v3 APIs	7
4.2	4.2 “Zero-Config” Scraping via InnerTube	7

4.3	4.3 Advanced Heuristics: Anti-Premiere Filter	7
4.4	4.4 Graceful Degradation and System Alerting (v1.7.2)	8
5	Chapter 5: Security and Portability (Hardware-Bound Encryption and OmniSync Protocol)	9
5.1	5.1 The Vulnerability Vector: Token Persistence	9
5.2	5.2 Hardware-Bound Encryption with safeStorage	9
5.3	5.3 The Portability Paradox: The Invention of the OmniSync (.rtb) Protocol . .	9
6	Chapter 6: Environmental Resilience (VPS Management, Anti-Ghosting, and Maintenance)	11
6.1	6.1 Challenges of Headless and Virtualized Environments	11
6.2	6.2 Active Crash Prevention: Anti-Ghosting Protocol	11
6.3	6.3 Repository Hygiene and Operational De-Crufting	11
7	Chapter 7: Conclusions and Architectural Vision	13

1 Chapter 1: Preface and Operational Context

1.1 1.1 The State of the Art of Editorial Automation

In today's digital information landscape, the timeliness and capillarity of content distribution represent critical success factors for editorial networks, independent creators, and media companies. Platforms like Telegram have taken on a central role in the disintermediation of communication, allowing the delivery of direct push notifications (broadcasts) to vast audience segments.

However, the automation of this process often encounters significant technical limitations. Traditional RSS-to-Telegram forwarding systems (whether third-party Cloud services or homebrew scripts) tend to exhibit three recurring vulnerabilities:

1. **Latency and Synchrony:** Sequential feed processing leads to bottlenecks, especially when the system must wait for responses from slow servers or manage strict blocks imposed by target API anti-spam systems.
2. **Dependency on Third-Party APIs:** Metadata extraction from complex proprietary ecosystems (such as YouTube) usually requires the use of official APIs that impose restrictive usage quotas, maintenance costs, and complex authentication procedures (OAuth/API Keys), significantly raising the barrier to entry for the final operator.
3. **Data Management and Security:** Storing access tokens with administrative privileges in plain text within configuration files exposes the entire infrastructure to critical risks in the event of exfiltration or unauthorized access to the file system.

1.2 1.2 The “Legacy” Problem and the Need for Evolution

The *Runtime TelegramBot* project was originally born as an internal response to these needs, taking shape in a first prototype developed in the Python language. This legacy software, while functional for elementary use cases, operated entirely through a command-line interface (CLI) and was designed according to a *single-thread* paradigm.

This setup generated non-negligible operational constraints. Configuring a new channel or adding a new RSS source required manual manipulation of JSON files, an operation prone to human error. Furthermore, the synchronous architecture meant that a network interruption during the reading of one feed would cascade and block the update of all subsequent channels. The software lacked a true relational database, relying instead on flat data structures that made unique delivery tracking difficult in scenarios where multiple bots shared the same sources.

As the volume of processed data and the number of simultaneously managed channels increased, the need emerged to move beyond the concept of an “automation script” toward a true “orchestration ecosystem,” leading to the conception of the *Titan Edition*.

2 Chapter 2: The Paradigmatic Leap (From Procedural Script to Desktop Ecosystem)

2.1 2.1 The Choice of Technology Stack

The design of the *Titan Edition* required the definitive abandonment of the Python execution environment in favor of an architecture based on **Node.js** and **Electron**, orchestrated through **TypeScript** and presented to the end user via **React** and **Vite**.

This transition addressed three fundamental engineering requirements:

- **Native Concurrency:** The non-blocking, event-driven I/O model of Node.js (based on the Reactor pattern and the *libuv* library) is intrinsically superior for applications that spend most of their clock cycles waiting for network responses (HTTP requests for feed fetching, API requests to Telegram).
- **Interface Abstraction (GUI):** Electron allows packaging the bot's Node.js instance within a "Main" process, while the user interaction level (UI) is delegated to an isolated "Renderer" process (Chromium). This ensures that background data processing does not interfere with the graphical interface's frame rate.
- **Cross-Platform Distribution:** By relying on `electron-builder`, the source code can be packaged into isolated native executables (`.exe` for Windows, `.dmg` for macOS, `.AppImage` for Linux), freeing the end user from the need to install or configure runtimes, dependency libraries, or virtual environments (e.g., `venv` or `pip`) on the host machine.

2.2 2.2 Data Layer Transition: From Flat Files to Relational SQLite

A critical limit of the *legacy* version was publication history management. In the old paradigm, delivery status was stored within text files. If two separate bot instances (e.g., "News Bot" and "Deep-Dive Bot") shared the same RSS source, the MD5 hash of the first article's link would be written to the log, leading the second instance to ignore it even though it was destined for a different Telegram channel.

To solve this "shared feed collision" problem, the *Titan Edition* implements an embedded relational database via the `better-sqlite3` library. History management is handled by a `history` table that adopts a **Composite Primary Key** consisting of (`id`, `bot_id`).

- `id`: The MD5 cryptographic hash generated deterministically from the news URL.
- `bot_id`: The unique identifier of the specific bot instance within the `bots` table.

This normalization scheme ensures the principle of *Multi-Tenant Scalability* within the local database: each bot operates within its own logical *silos*, allowing configurations where infinite instances can draw from the same RSS source without interference, duplication, or data loss.

3 Chapter 3: Engine Architecture (Producer-Consumer and Rate-Limit Management)

3.1 3.1 Synchrony Limits and UI Freeze

In its first conceptual implementation within the Electron framework (alpha and v1.0.x versions), the control iteration (the polling loop) operated in a sequential blocking manner. The engine queried a feed, formatted new items, and for each of them, executed the HTTP POST call to the Telegram APIs.

To prevent spam blocks, Telegram imposes strict limitations (*Rate Limiting*), known as 429 Too Many Requests errors (commonly managed through *FloodWait* parameters). Consequently, sequentially sending multiple messages required the insertion of an artificial *delay* of 2-3 seconds between calls. In scenarios with dozens of high-traffic feeds, these delays accumulated, generating execution cycles lasting several minutes. This blocked the *Main Process* event loop, leading to the desynchronization of general execution timers and, in the most severe cases, the freezing of the graphical interface in the *Renderer* process.

3.2 3.2 Job Queue Implementation (Decoupled Asynchrony)

Starting from version 1.3.4, the logical architecture of the *BotEngine* instance was completely refactored following the **Producer-Consumer** design pattern. This architectural choice drastically separated the data extraction phase from the distribution phase:

1. **Producer Phase (Fetching):** The primary cycle (*checkLoop*) queries RSS sources. It analyzes new entries, verifies time exclusion rules (cutoff date), and compares MD5s with the *history* table in SQLite. If an entry is valid for publication, it is no longer sent to Telegram immediately, but is instead **injected into a RAM queue** (*PublishJob Queue*). This operation is instantaneous and allows the Producer to scan the entire feed set in fractions of a second.
2. **Consumer Phase (Sending):** A separate asynchronous function (*processPublishQueue*) acts as an extractor daemon. If it detects elements within the queue, it extracts them one by one for processing. It applies the user-configured text template and calls the Telegram API. Upon success, it records the operation in the database and asynchronously waits (*await new Promise...*) for 3000 milliseconds before processing the next element.

3.3 3.3 Dynamic Network Error Management (Telegram Retry Logic)

The Consumer implementation is equipped with a logical wrapper encapsulated in the *TelegramClient* class, designed for **Network Resilience**. The client autonomously manages up to 5 retry attempts for each message, applying differentiated reaction strategies based on the error topology returned by the APIs:

- **Transient Errors (Network/Timeout):** Managed with a fixed 10-second suspension followed by a *retry*.
- **FloodWait Errors (Rate Limiting):** The system extracts the *retry_after* parameter natively provided by Telegram and suspends processing for the exact number of seconds required by the server before making a new attempt.
- **Fatal Errors (e.g., Bot Blocked, Non-existent Chat):** Generate the immediate discard of the message (*abort*) to prevent infinite loops or reputational penalties for the token.

- **Strategic Fallback:** Should a non-fatal error persist until the fifth attempt, the engine deduces that the anomaly may derive from a failure to process the multimedia file (e.g., timeout in Link Preview generation on the Telegram server) and attempts a delivery in “Safe” mode, forcibly disabling the message’s URL preview.

This separation of tasks has made the engine immune to locking, transforming the *Titan Edition* into an asynchronous orchestrator capable of accumulating theoretically infinite amounts of data and then disposing of them in full compliance with destination network constraints.

4 Chapter 4: YouTube Integration (Reverse Engineering and Anti-Spam Heuristics)

4.1 4.1 The Failure of the XML Standard and the “Trap” of v3 APIs

Integrating YouTube into a broadcast automation ecosystem represents, historically, a complex technical challenge. The orthodox approach involves using the Atom (XML) feeds publicly exposed by channels. However, empirical experience gained during the development of early releases (v1.0.x) demonstrated the inadequacy of this solution: Google began to silently depower and fragment the delivery of these feeds, which increasingly returned 404 Not Found errors for various channels, making the entire layer unreliable.

The natural alternative necessitated a transition to the *YouTube Data API v3*. This solution, while technically stable, was **scientifically rejected** for reasons related to product engineering and User Experience (UX). Using the v3 APIs requires the end user to:

1. Possess a Google Cloud Platform account.
2. Have the skills to configure and generate a dedicated API Key.
3. Undergo a daily call quota regime (upstream rate-limiting), with a potential requirement to link a credit card.

This barrier to entry would have compromised the “Plug & Play” nature of the platform.

4.2 4.2 “Zero-Config” Scraping via InnerTube

To ensure fluid and native integration, the *Titan Edition* (starting from v1.3.0) implemented the `youtubei.js` module. This library reverse-engineers the undocumented **InnerTube** interface, the same communication layer used internally by official YouTube web and mobile clients. This approach, technically definable as intelligent scraping of the internal JSON payload, allows the engine to retrieve metadata (ID, Title, Thumbnail, Publication Date) by simulating normal web traffic. The architectural advantages were immediate:

- **Total Absence of Configuration:** No API Key or Cloud configuration required.
- **Quota Bypass:** The system is not subject to the strict restrictions imposed on public APIs.
- **Dynamic Identifier Resolution:** The user can provide the bot with a channel’s generic Handle (e.g., @ChannelName) instead of complex and archaic Channel IDs (UC...), drastically improving UX.

4.3 4.3 Advanced Heuristics: Anti-Premiere Filter

Direct access to the InnerTube payload raised an editorial issue: so-called “Ghost Events.” The internal module returns as top results not only currently viewable videos, but also “Premieres” and “Scheduled” videos, which possess valid IDs and titles but are not yet publicly viewable. Sending such content to a Telegram channel generated false expectations and spam.

To mitigate this effect, a **Proactive Heuristic Filter** was implemented. Before a YouTube element is considered valid for MD5 hashing, the algorithm analyzes a series of combined parameters:

1. **Boolean Analysis:** Querying native payload flags (`is_upcoming`, `is_premiere`).

2. **Lexical Date Analysis:** Scanning publication-related strings via pattern-matching, intercepting and discarding terms indicative of a future state (e.g., "Premiere", "Scheduled", "Waiting").

This system ensures that the publication queue is populated exclusively by material immediately consumable by the audience.

4.4 4.4 Graceful Degradation and System Alerting (v1.7.2)

Relying on a scraping library implies accepting a physiological structural vulnerability: any minor variation in the DOM or JSON payload by Google can generate exceptions (*Cannot read properties of undefined*). To prevent the failure of a third-party interface from causing the crash of the entire Node.js ecosystem, a **Graceful Degradation** mechanism was integrated. The engine isolates errors deriving specifically from the `youtube` feed type and sends an *IPC (Inter-Process Communication)* signal to the Renderer. The graphical interface intercepts this signal and generates a native notification (Toast) informing the operator of the Google-side failure, suggesting temporary deactivation of video feeds without ever interrupting the fluid processing of audio and text RSS streams.

5 Chapter 5: Security and Portability (Hardware-Bound Encryption and OmniSync Protocol)

5.1 5.1 The Vulnerability Vector: Token Persistence

In Telegram automation, the *Bot Token* (generated via @BotFather) represents the cryptographic keystone: possessing it is equivalent to having absolute write control over the bot and administered channels. In legacy applications, common practice involves saving these credentials in plain text in databases or configuration files (`.env`, `.json`). If the host machine were compromised or the database exfiltrated, the attacker would obtain the keys.

To inhibit this vulnerability at the root, *Runtime TelegramBot Titan Edition* implements hardware-level protection of the Data Layer.

5.2 5.2 Hardware-Bound Encryption with safeStorage

All tokens are saved within SQLite (`titan.db`) exclusively in encrypted format. This operation does not use a symmetric key hardcoded in the source code but leverages Electron's `safeStorage` APIs. This module interacts directly with the host operating system's native credential managers (such as **DPAPI** on Windows or the **Keychain** on macOS). The result is *Hardware-Bound* encryption: the token is encoded using a key that exists only and exclusively within the physical environment (or certified virtual environment) on which the application is installed. If the `titan.db` file is copied and opened on a different computer, the tokens appear as totally unintelligible sequences of bytes. Communication between the *Main Process* and the graphical interface (*Renderer*) occurs through a rigid, isolated `contextBridge` that temporarily decrypts the token in RAM only when the operator requests to view it in the settings panel, preventing any possibility of exfiltration via malicious frontend scripting.

5.3 5.3 The Portability Paradox: The Invention of the OmniSync (.rtb) Protocol

Cryptographic rigor, however, generated an operational paradox: how can a company or a team of developers migrate or share a bot's configuration from one machine to another if the database is unreadable outside the original hardware?

To overcome this obstacle, the team engineered the **OmniSync** ecosystem (v1.7.0), introducing a certified proprietary file standard: the `.rtb` (**Runtime Telegram Bot**) format. The OmniSync architectural flow is structured into four secure transition phases:

1. **Controlled Extraction:** The operator requests the export of a single profile. The engine isolates the data (name, configuration, feed URLs).
2. **Decryption in Volatile Memory:** Only in this instance, the token is decrypted within RAM (via the hardware API) and exposed in plain text within the JSON cartridge encoded inside the `.rtb` file.
3. **Transitivity:** The file is physically moved to the new machine (via USB stick, secure network, etc.).
4. **Dynamic Re-Encryption:** During the import routine, the new instance's database absorbs the configuration and identifies the token. Before writing it to SQLite, the system intercepts it and **re-encrypts it using the new receiving machine's hardware key**.

The `.rtb` format thus establishes itself as the professional standard for automation profile exchange, combining extreme operational portability with persistent and inviolable cryptographic blindness.

6 Chapter 6: Environmental Resilience (VPS Management, Anti-Ghosting, and Maintenance)

6.1 6.1 Challenges of Headless and Virtualized Environments

An automation system like the *Titan Edition* is intrinsically conceived for continuous operation, 24/7. The most common deployment scenario for applications of this type is not a standard user's personal computer, but rather a **Virtual Private Server (VPS)** or a remote Virtual Machine accessible via Remote Desktop (RDP).

These environments introduce critical environmental variables that test the stability of software based on complex frameworks like Electron:

- **Absence of Dedicated Hardware Acceleration:** Budget VPSs do not have physical GPUs or updated graphics drivers (often using standard virtualized display adapters like the *Microsoft Basic Display Adapter*).
- **Latency and Background Computations:** Virtualized CPU limitations can slow down the triggering of native application lifecycle events.

Failing to contain these factors generated, in early iterations (pre-v1.5.2), the phenomenon known as **"Ghost Processes"**: the application would start within the Node.js event loop, allocating RAM and executing tasks, but the graphical interface would stall or become invisible, making it effectively impossible for the user to interact with the system or stop automations without brutally killing the task via Task Manager.

6.2 6.2 Active Crash Prevention: Anti-Ghosting Protocol

To ensure an industrial-grade uptime level (*Zero Downtime*), three sequential lines of defense were integrated into the `Main Process` to counter ghosting and environmental crashes:

1. **Disabling Hardware Acceleration** (`app.disableHardwareAcceleration()`): Inserted in the preliminary boot phases, this instruction forces Chromium to render the UI entirely via software, bypassing the search for hardware graphics APIs. This instantly eliminates the vast majority of *blank screens* generated by incompatible drivers typical of server environments.
2. **UI Lifecycle Safety Timeout:** It is common for the native Electron `ready-to-show` event to fail to trigger within standard times on overloaded systems. The *Titan Edition* applies an asynchronous "safety parachute": if the event is not registered within 10,000 milliseconds, the `Main Process` forcibly triggers the `mainWindow.show()` method, preventing the application from hanging in a silent state.
3. **Global Native Fallback Handler:** In case of fatal V8 Engine errors or network calls that cannot be resolved asynchronously, the integration of low-level listeners (`process.on('uncaughtException')` and `unhandledRejection`) prevents the system from silently collapsing. The error is intercepted, injected into a native OS Dialog, and subsequently, the Node.js process is safely suppressed (`app.exit(1)`), ensuring the restoration of system integrity.

6.3 6.3 Repository Hygiene and Operational De-Crufting

The long-term reliability of the system lies not only in the compiled code but also in workspace maintenance during development (DevOps and CI/CD). Excessive weight from

obsolete binary artifacts (Binary Bloat) and degraded caches compromises the efficiency of Vite's file watchers and the `electron-builder` assembler. To this end, a strict corporate protocol for *De-Crufting* (Deep Cleaning) and archiving was established.

The ecosystem has been structurally isolated by permanently separating historical instances (located within `ARCHIVIO_LEGACY/`) from the current development tree (`TITAN_DESKTOP/`). This operation protects the *shell paths* necessary for the correct local recompilation of the `better-sqlite3` native library, whose integrity is compromised by proximity to virtual environments (Python `venv`) or unvalidated residues (misaligned `node_modules`). Such rigorous *Repository Hygiene* allows for multi-station allocation and development while ensuring absolute environmental reproducibility.

7 Chapter 7: Conclusions and Architectural Vision

The transformation of *Runtime TelegramBot* from its embryonic nature into *Titan Edition v1.7.x* represents an exemplary case study of how software architecture determines a product's scalability. Overcoming synchronous and manual logics through the adoption of the Node.js/Electron stack, combined with the development of high-value-added custom solutions — such as the Producer-Consumer pattern for Rate-Limit bypassing, the implementation of “Zero-Config” parsing to overcome YouTube restrictions, and the introduction of the `.rtb` standard for dynamic token migration — today positions the software no longer as a personal tool, but as an *Enterprise*-grade infrastructure for broadcast automation and orchestration.

The acquired stability, certified by advanced Anti-Ghosting and Graceful Degradation systems in case of external API failures (i18n), serves as a consolidated technological *Plateau*. The architecture is now sufficiently mature, isolated, and modular to support future integrations of external ecosystems (social networks or data providers) without requiring alterations to the core engines presiding over asynchronous computation, storage, and sensitive data security, ensuring absolute reliability in the era of information overload.