Cookiedad Technical System Specification

---

1. Executive Summary & Documentation Perimeter

This document defines the technical specification of the Cookiedad ecosystem, covering both the local WordPress plugin environment and the associated Node.js Cloud Function (`scanv3`). It has been prepared by the Cookiedad development team to support developers, system architects, future maintainers, technical reviewers, and internal compliance auditors.

Its purpose is to document the system architecture as implemented by the project, including runtime behaviors, external service integrations, asynchronous processing flows, persistent data models, and security measures.

1.1 System Components

The architecture documented herein covers the two primary proprietary components of the ecosystem:

- **WordPress Environment**: `Cookiedad.php`, `src/`, `assets/`, `premium/`, `tabs/`.
- **Cloud Infrastructure**: `scanv3/` (Express.js, GCP Firestore, GCP Cloud Tasks).

1.2 External Dependencies & Exclusions

The system interacts with several external dependencies that operate as black boxes from the perspective of this architecture document. Their internal implementations are excluded from this specification:

- **Licensing & Billing**: `freemius/` SDK.
- **Client Storage**: `assets/js/js.cookie.min.js`.
- **Geolocation & External Endpoints**: Third-party REST APIs (e.g., ipapi.co).

These dependencies are documented exclusively at their integration and API borders within the framework.

---

2. General System Architecture

Cookiedad implements a **Hybrid Architecture Pattern**, distributing operations between the local server and a remote node. Interface rendering and localized data storage are handled by

the WordPress application, while computational processes (web crawling, HTML policy generation) are delegated to the Cloud Function.

```
graph TD

subgraph Local Control Plane \[WordPress Plugin\]

    UI\[Admin Dashboard UI\]

    Banner\[Shadow DOM Frontend Banner\]

    DB\[(WordPress Custom Tables & Options)\]

    AJAX\[WP REST/AJAX Endpoints\]

end

subgraph Remote Computation Plane \[GCP Node.js Backend\]

    EXPRESS(Express.js Orchestrator)

    FS\[(Firestore Quotas & Keys)\]

    CT\[Google Cloud Tasks Worker\]

end

UI \-- "1. AJAX Request" \--\> AJAX

AJAX \-- "2. Scan Request \+ Token" \--\> EXPRESS

EXPRESS \-- "3. Validate API Key" \--\> FS

EXPRESS \-- "4. Enqueue Job" \--\> CT

EXPRESS \-. "5. 202 Accepted" .-\> AJAX

CT \-- "6. Execute Deep Scan" \--\> TargetSite\[(Target Website)\]

CT \-- "7. Async REST POST payload" \--\> AJAX

AJAX \-- "8. Verify Mutex & Merge" \--\> DB
```

The system is built upon six structural components:

1. **Consent & Block Management**: A state-driven JavaScript component isolated within a Shadow DOM.
2. **Distributed Scanning**: A multi-phased scanner merging local BFS crawls with Cloud Worker logic.
3. **Policy Compilation**: Remote HTML generation returning localized texts via POST hooks.
4. **Admin Interface**: A single-page interface module within the WordPress admin, utilizing AJAX and `localStorage`.
5. **Quota Handling**: Isolation of Free (LITE) and PRO accounts utilizing validation layers and atomic database increments.
6. **Server-Side Scrubbing**: Output buffers (`ob_start`) that intercept third-party attributes before HTTP emission.

---

3. WordPress Plugin Core (Control Plane)

3.1 Bootstrap Sequence & Instantiation

Execution initiates at `Cookiedad.php`. Following WordPress conventions, it defines path constants, initializes the external SDK wrapper (`Cookiedad_fs()`), registers plugin uninstallation hooks (`Cookiedad_fs()->add_action('after_uninstall')`), and loads a PSR-4 style SPL autoloader.

The `Cookiedad\Plugin` class orchestrates the application logic. Its `run()` method initializes:

- `ConsentManager`: Models for consent persistence.
- `Scanner`: Network coordination.
- `CacheManager`: Transient lifecycle handling.
- `Frontend`: Script queuing and DOM templating.
- `Dashboard`: UI construction and backend routing.
- `CookiePolicyOrchestrator` & `PrivacyPolicyOrchestrator`: Legal document retrieval.
- `ThumbnailProxy`: Internal asset caching.
- `ServerBlocker`: Output buffer manipulation.

3.2 Server-Side Emplacement Blocker (`ServerBlocker.php`)

Cookiedad intercepts output buffers on the `template_redirect` hook. Using Regex parsing, it evaluates the HTML DOM prior to emission, targeting `<iframe`, `<script>`, and embedded third-party APIs.

- If the required consent category is absent in the user's `gdpr_consent` cookie, the system rewrites the `src` attribute to `data-src`, preventing the script from executing on page load.
- The blocker implements bypass subroutines for operational endpoints (e.g., WooCommerce `wc-ajax` and `checkout`) to avoid interfering with ecommerce logic.

3.3 Thumbnail Proxy

Blocked video embeds display an overlay. To avoid sending HTTP requests to third-party domains for thumbnails (which could execute tracking pixels), the `ThumbnailProxy` operates server-side. It fetches the remote thumbnail image, stores it on the local WordPress server (`wp-content/uploads/Cookiedad_thumbnails`), and serves the local resource. A cron event (`Cookiedad_cleanup_old_thumbnails`) handles directory pruning.

---

4. Frontend Application & JavaScript

The plugin's frontend scripts (`Cookiedadbanner.js`) are designed to operate independently from the global DOM tree where possible.

4.1 Shadow DOM Encapsulation

The consent banner is injected into a `ShadowRoot` attached to `#Cookiedad-root`. This implements a boundary intended to prevent CSS inheritance, mitigating style bleeding between the active WordPress theme and the banner elements.

4.2 Accessibility Standards (WCAG 2.1)

- **Focus Trapping**: The script includes a keyboard event listener capturing `Tab` and `Shift+Tab`. It maps visible, non-disabled DOM nodes within the wrapper. If focus shifts out of the sequence, it loops back to the initial interactive node, supporting screen-reader traversal.
- **Aria Mapping**: Progress and state toggles emit `aria-valuenow` bindings.

4.3 Global Privacy Control (GPC)

The script queries the browser's `navigator.globalPrivacyControl` variable. If evaluated to `true`, and no prior framework consent exists, the engine logs a "Reject All (Except Necessary)" payload and persists it locally.

4.4 A/B Testing Component

When A/B testing is enabled, the behavior is handled dynamically client-side:

1. The script evaluates a weighted split fraction and assigns the user to a variant.
2. For Variant B, the script traverses the Shadow DOM, replacing mapping strings (`app.bannerDict` -> keys appended with `_b`) and injecting a `<style>` block to apply distinct localized layouts.

4.5 Component Synchronization

Upon consent interaction, the script executes `saveCookieConsent()`, updating `js-cookie`. It simultaneously emits a `CustomEvent` (`wp_consent_updated`) and forwards the payload to the WP Consent API parameters (`wp_set_consent`) to align third-party ecosystem listeners.

---

5. Constraint Management & Persistence Layers

5.1 Database Storage Schema (`{$wpdb->prefix}Cookiedad_consents`)

Consent payloads are processed via a localized `saveConsent` AJAX route and stored in a custom relational table.

- **Relational Schema**: The table logs parameters including `anonymous_id`, `consent_date`, and serialized categorization nodes.
- **Schema Resilience**: Prior to payload insertion, the plugin validates column existence (`SHOW COLUMNS LIKE 'ab_mode'`), reducing the risk of fatal SQL errors during staggered environment updates.
- **CCPA Normalization**: When operating under CCPA guidelines ("Opt-Out" configuration), incoming telemetry translates the "Opt-Out" event into a denial of non-essential GDPR categories. This normalizes the data structure for unified analytics queries.

5.2 Local Limitations & Data Scoping

- **Database Scale Limitations**: Because `Cookiedad_get_filtered_stats` relies on direct `SUM(CASE WHEN...)` SQL queries, exceptionally large consent tables on high-traffic sites may introduce measurable DB CPU load. While more efficient than parsing JSON in PHP, it remains an unindexed wildcard search (`LIKE '%"marketing":true%'`).
- **Data Erasure**: Consent arrays are logged using an `anonymous_user_id`. Triggering a "Delete My Data" request removes the client cookie, flushes the WP Consent API state, and securely purges the associated row from the physical table.

---

6. Administrative Interface

6.1 State Management (`Cookiedad-admin.js`)

The `Dashboard` is structured as a module within WordPress. During asynchronous scans, it disables global UI triggers (`disableAllScanButtons()`). To handle page mutability, it writes executing request correlations to `localStorage` (e.g., `fullScanState`). If the browser window is refreshed mid-scan, the script reads `localStorage` upon `DOMContentLoaded`, reinstates the loading UI, and resumes the `/v3/getScanResults` periodic polling loop.

6.2 Data Aggregation

The dashboard statistics endpoint utilizes scalar SQL to compute ratios instead of instantiating PHP arrays, grouping visualization counts directly by `ab_group` parameters.

---

7. Operational Boundary & Communication Flow (Plugin <-> Cloud)

The architecture establishes a strict boundary between the WordPress PHP thread and the GCP Node.js execution layer. Physical limitations inherent to WordPress shared hosting (e.g., 30-second PHP timeouts, low memory ceilings, restricted cURL configurations) necessitate the decoupling.

7.1 Payload Structure and Latency Mitigation

When initiating a scan or policy generation, the WordPress plugin constructs a JSON payload via `wp_remote_post`.

- **Timeouts**: The local `wp_remote_post` request enforces a short 15-second timeout. It expects a `202 Accepted` or `400/401` rejection almost immediately.
- **Network Failure Fallbacks**: If the host cannot reach the Cloud Function due to DNS or firewall issues (e.g., cURL error 28), the plugin catches the `WP_Error`, deletes the pending task state, and presents a standardized connection error to the administrator.

7.2 The Idempotency Loop Model

1. **Delegation**: The plugin generates a `correlation_id` and a `callback_token`, marking the scan as 'Processing' in local options.
2. **Cloud Orchestration**: GCP processes the DOM logic asynchronously.

3. **Execution Callback**: The Cloud performs an HTTP POST back to `/wp-json/Cookiedad/v1/plugincallback`, sending the discovered items.
4. **Mutex Validation**: To minimize race conditions where concurrent Cloud Tasks might fire identical callbacks simultaneously, the `CallbackHandler` leverages a transient `Cookiedad_cb_lock_` mechanism. Parallel hits are met with a `503 Retry-After` header, prompting the GCP worker to pause and retry. Once parsed, the token is appended with `|USED` in the database.

---

8. Remote Node Architecture (`scanv3/`)

The backend operates on Google Cloud Run executing an Express.js application designed to orchestrate queue distribution.

8.1 Rate Limiting

The `/v3/register` endpoint governs API Key distribution. Utilizing `express-rate-limit`, the logic bounds payloads within a fixed IP timeframe. It evaluates `X-Forwarded-For` HTTP headers to resolve client origin, relying on infrastructure proxy trust configurations.

8.2 Identity Locks and Firestore

Standard API keys are stored in the `api_keys_v2` Firestore collection. Keys are bound to the client's originating `fs_install_id` (PRO) or `local_uuid` (LITE). The `apiKeyMiddleware` checks this configuration per request, enforcing a `403 Forbidden` response if mismatched origins are detected.

8.3 Webhook Validation

The `/v3/webhook` endpoint listens for subscription telemetry. The raw request body stream is subjected to an HMAC SHA-256 signature verification matching the payload against a local environment variable (`process.env.FREEMIUS_SECRET_KEY`). Successful evaluation allows direct modification of the `licenses` Firestore documents.

8.4 Quota Execution

Free plans operate under quota limits. Deductions are processed using Firestore transactions (`checkAndDecrementQuota()`), utilizing atomic operations to minimize concurrent race conditions.

---

9. Policy Document Compilation

9.1 Data Orchestration

`CookiePolicyOrchestrator` & `PrivacyPolicyOrchestrator` structure parameter inputs via wizard states. The data undergoes string sanitization before being exported to the `/v3/generate-policy` route.

9.2 Remote HTML Generation

The Node application inserts the variables against selected frameworks and localized translations to assemble raw HTML strings. The payload is returned asynchronously, stored incrementally into `Cookiedad_generated_cookie_policy_html`, and localized for Frontend rendering. Text matrices reside on the remote node, isolating localization updates from plugin version control.

---

10. Security Mechanisms

The framework utilizes multiple defense-in-depth methodologies:

1. **Nonce Verification**: Authenticated AJAX routes executed from the WordPress interface require validation via `check_ajax_referer('Cookiedad_admin_nonce')`.
2. **Asymmetric REST Validation**: As local nonces are invalid for external servers, the REST bridge depends on the single-use `callback_token` parameters.
3. **Escaping Practices**: Data variables are passed through WordPress functions such as `sanitize_text_field` and `esc_url_raw`.
4. **State Auto-Correction**: If the plugin receives unauthorized responses from an endpoint (`401` or `402`), `Scanner.php` executes logic to delete the local API key transient, prompting the handshake to restart automatically.

---

11. Licensing & Feature Segregation

Feature limits depend on external integration hooks.

- **Conditional Siloing**: Modules managing A/B testing components, Multilingual variants, and specific UI elements are excluded from instantiation loops if the conditional check `is_plan_or_trial('pro/business/agency')` evaluates falsely.

- **Orphaned State Resolution**: In instances where subscription records drop but physical DB records persist locally, the scanner reverts execution identity to a `local_uuid` fallback, reducing API rejection loops.

---

12. System Limitations & Infrastructure Dependencies

This architecture relies heavily on specific operational constraints and edge cases which are documented below.

12.1 Undeterminable Behaviors

- **Cloud Task Concurrency**: The precise manner in which Google Cloud translates the orchestrated queue into concurrent Puppeteer instances across regions cannot be determined solely from the code analyzed.
- **WordPress Cron Variations**: The cleanup jobs (e.g., thumbnail purging, incomplete scan resets) rely on `wp_cron`. On low-traffic sites without a system cron, these events may face significant operational delays.

12.2 Infrastructure Dependencies

- **Cloud SLA**: The core scanning and policy generation mechanics are strictly bound to the uptime of Google Cloud Run, Cloud Tasks, and Firestore. Outages in these regions will suspend localized plugin functionality.
- **Host Restrictions**: The plugin requires active `cURL` libraries on the WordPress host to initiate the async loop. Environments blocking outgoing HTTP traffic will fail the `/register` handshake entirely.

---

13. Architectural Summary

The system demonstrates a practical decoupling of concerns suited for constrained PHP environments.

By removing deep crawling and document assembly logic from the local host and delegating it to an event-driven Cloud infrastructure, the processing overhead within the WordPress layer is minimized. Implementing Shadow DOM encapsulation addresses CSS inheritance issues common in WordPress environments. The incorporation of lock mechanisms and token validation on cloud callbacks provides a structural defense against concurrency errors and unauthorized payload executions, yielding a predictable, scalable architecture.

---

14. Third-Party Notices, Sources, and Trademarks

This technical documentation references third-party software components, services, APIs, platforms, and product names strictly for identification, interoperability, architectural explanation, and integration-context purposes.

14.1 Third-Party Components and Services Referenced

Examples of third-party technologies referenced in this document and/or used by the documented system include, where applicable:

- **WordPress** and related platform APIs.
- **Freemius** SDK for licensing, billing, and subscription event handling.
- **js-cookie** for client-side cookie state handling.
- **Google Cloud** services, including **Cloud Run**, **Cloud Tasks**, and **Firestore**.
- External geolocation or IP intelligence APIs such as **ipapi.co**, where configured.

These third-party elements remain subject to their own terms, licenses, policies, and technical documentation.

14.2 Open-Source Licensing Note

Where open-source third-party libraries are bundled or referenced by the system, they remain subject to their respective upstream licenses. For example, the bundled `js-cookie` library is distributed under the **MIT License** by its respective authors/rightsholders. Any additional third-party libraries, SDKs, or packages remain governed by their own applicable license terms.

14.3 Trademark Notice

All third-party product names, company names, platform names, service names, logos, and brands mentioned in this documentation are used for identification purposes only and remain the property of their respective owners.

WordPress, WooCommerce, Google, Google Cloud, Firestore, Cloud Run, Cloud Tasks, Freemius, and any other referenced third-party names may be trademarks or registered trademarks of their respective owners in the relevant jurisdictions.

No affiliation, sponsorship, endorsement, certification, or partnership with any third-party trademark owner is claimed or implied unless expressly stated in a separate written agreement.