Seqrite Data Privacy

SEQRITE

Cookie Consent Enforcement Integration Guide

V1.1

www.seqrite.com

Contents

1. Overview

The Seqrite Cookie Consent Manager lets you display a consent banner on your website and actively enforce your visitors' choices — blocking or activating tracking scripts based on what each visitor actually agrees to.

Built for DPDP Compliance

The enforcement engine is designed to help you meet obligations under India's Digital Personal Data Protection Act (DPDP), 2023. Tracking scripts stay inert until a visitor grants consent for the matching category, consent choices are recorded and stored for later reference, and visitors can revisit and change their choices at any time through the Preference Centre.

Integration Requires Changes to Your Site's Code

Embedding the enforcement snippet on its own does not block or gate anything. The engine has no way of knowing which scripts on your page are tracking scripts unless you tell it — you must still update your site's code so each tracking script or iframe is marked for enforcement, using one of the two approaches covered later in this guide:

- **Markup-based**: change each tracking <script>/<iframe> tag so the engine can intercept it (covered in Step 3, including iframes in 3.4).
- **API-based**: call the enforcement API from your own code to conditionally load scripts based on consent (covered in Step 4), needed for React/Angular/Vue sites or anything loaded dynamically at runtime.

Every tracking script you don't mark or wire up one of these two ways will keep loading unconditionally, regardless of visitor consent. Plan for real developer time on your side, not just a copy-paste snippet.

Key Capabilities

- **Live configuration reflection**: Changes you make in the admin portal (consent banner behaviour, categories, enabling/disabling the Preference Centre) take effect for visitors on their very next page view, without touching the snippet or the markup you've already added.
- **Multi-platform consent signals**: beyond Google Consent Mode v2, the enforcement engine automatically sends live consent signals to Meta (Facebook Pixel), Microsoft Bing (UET), and Microsoft Clarity as visitors accept or reject each category.
- **Preference Centre**: An optional, ready-made "Manage Cookie Preferences" button visitors can use to reopen and change their consent choices at any time after their first visit.

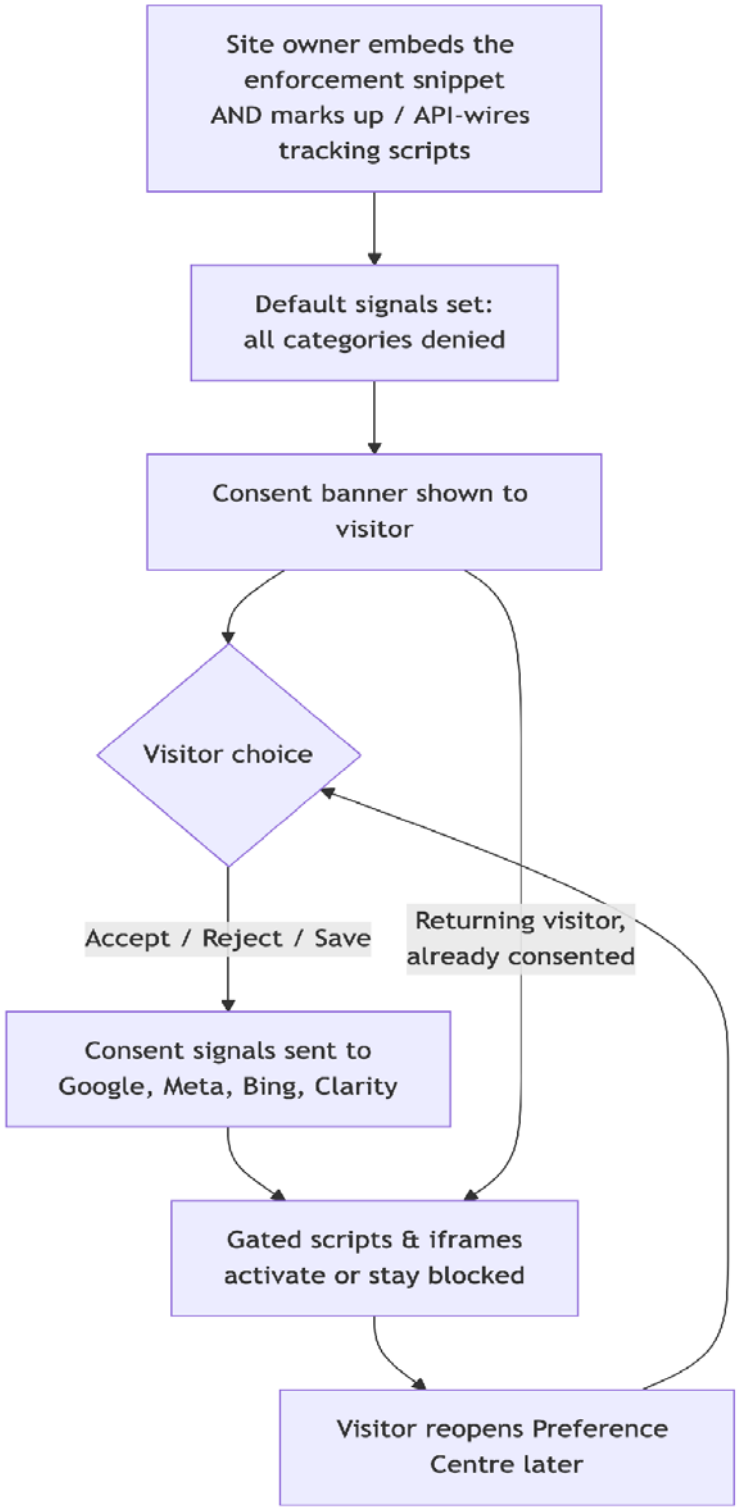
Before You Start

You'll need:

- A published Consent Manager configuration in the Seqrite Data Privacy admin portal.
- Developer access to your site's code — to mark up tracking scripts/iframes or add API-based loading logic.

How Enforcement Works

The diagram below shows what happens at runtime, once your site has actually been integrated — snippet embedded (Step 1) and your tracking scripts marked up or API-wired using whichever approach Step 2 helped you choose (Step 3 or Step 4).



2. Integration Steps

1. Step 1 — Embed the Dynamic Configuration Snippet

Unlike older integrations that required downloading and hosting files yourself, there's nothing to download. Add these two tags to the `<head>` of every page on your site, using your own **Consent Manager ID**:

```
<!-- Consent banner styling -->
<link rel="stylesheet" href="https://<your-assigned-
host>/dynamic/v1/config/css/{consentManagerId}"/>

<!-- Consent engine bootloader -->
<script src="https://<your-assigned-
host>/dynamic/v1/config/js/{consentManagerId}"></script>
```

Note: Do NOT add `defer` or `async` to the bootloader `<script>` tag, and place it as early as possible in `<head>`, before any of your own tracking scripts. The bootloader's first job is to set every tracking category to "denied" by default *before* anything else on the page has a chance to run. If it loads late or asynchronously, a tracking script placed above it could fire — and set real cookies — before consent has even been considered.

Everything else the bootloader does (loading the banner, wiring up the Preference Centre, fetching enforcement rules) happens automatically after that, without blocking your page's own load.

Replace `{consentManagerId}` and `<your-assigned-host>` with the values from your admin portal.

Easier option: you don't have to build this snippet by hand. Your admin portal's Publish step provides a ready-to-copy version of these same two tags with your Consent Manager ID already filled in — copy it directly from there and paste it into your site to avoid any typos in the ID or host.

2. Step 2 — Choose Your Integration Approach

With the snippet in place, decide how you'll mark your tracking scripts for enforcement. The right choice depends on **how each script actually reaches the browser**, not on what kind of site you have overall — a single site can (and often does) use more than one of these.

- **Static loading** — the `<script>`/`<iframe>` tag is written directly in your HTML source (a template, a CMS page, a server-rendered file) and is already present, as a real tag, in the page's markup by the time the browser parses it. It doesn't matter whether that HTML was generated by hand, a CMS, or server-side logic — what matters is that a real tag exists in the page source. → Use **Approach A (markup-based)**.

- **Runtime loading** — nothing is present as a tag in the page source at all; your own JavaScript creates and inserts the script *after* the page has already loaded, using something like

`document.createElement('script')`, a dynamic `import()`, or a framework's own component lifecycle (common in React/Angular/Vue apps, and in scripts injected by a tag-management library other than GTM). There's no static tag for the engine to find and mark up. → Use **Approach B (API-based)**.

- **Dynamic loading (GTM specifically)** — Google Tag Manager sits in between: its own container script *is* a static tag you can mark up, but everything GTM subsequently injects (GA4, Ads, other vendors) is invisible to the engine, since GTM creates those tags itself at runtime. → Gate the GTM container tag itself with the markup approach (Step 3.2), regardless of which approach you use elsewhere.

Your situation	Use
Tracking scripts are written directly as `<script>`/`<iframe>` tags in your HTML (static loading)	**Approach A — Markup-based** (Step 3)
Your site is a React / Angular / Vue single-page app, or scripts are injected at runtime via `fetch`/`import`/`createElement` (runtime loading)	**Approach B — API-based** (Step 4)
You use Google Tag Manager (dynamic loading)	Follow **Step 3.2** regardless of approach — GTM's container script always needs to be gated at the markup level, since GTM injects everything else itself.
Not sure / mix of both	Use markup-based for anything already present as a static tag, and API-based for anything loaded at runtime. Most real sites end up using both.

3. Step 3 — Approach A: Markup-Based Integration

Below is a guide to the Google-related scripts and embeds most sites need to gate, used here as the worked example — apply the same `type="text/plain"` + `data-*` pattern to any other tracking script or iframe on your site, using whichever category it belongs to in your Consent Manager. For each Google-related script or embed, change `type="text/javascript"` to `type="text/plain"` and add `data-type`/`data-name` attributes so the engine knows to intercept it and which consent category

controls it.

What this actually does, step by step:

1. Browsers only execute `<script>` tags whose `type` is a recognized JavaScript MIME type (or omitted). Setting `type="text/plain"` isn't a special instruction to our engine specifically — it's a standard HTML mechanism that makes the browser itself treat the tag as inert data and refuse to run it, full stop. This is what actually blocks the script; nothing runs by accident even if our engine's own code failed to load for some reason.
2. On every page load, the engine scans the page for elements carrying its `data-*` attributes (as described in this guide) and keeps track of which ones are still pending activation for each category.
3. When a visitor grants consent for a category — either from the initial banner or later via the Preference Centre — the engine activates every pending element whose `data-name` matches that category: for scripts, it takes over execution (running any inline code, or fetching and running the URL from `data-src`); for iframes, it sets the real `src` from `data-src`, which causes the browser to load the embed.
4. If consent for a category is *never* granted, the marked elements simply stay as inert `text/plain` tags for the rest of that visit — the browser never executes or loads them at all.

Note: The `data-name` value must match a category name configured in your Consent Manager (For example, "analytics", "marketing", "functional") — confirm the exact names in your admin portal.

3.1 Google Direct Tag (gtag.js / GA4)

```
<!-- BEFORE: loads unconditionally -->
<script async src="https://www.googletagmanager.com/gtag/js?id=G-XXXXXXXXXX"></script>
<script>
  window.dataLayer = window.dataLayer || [];
  function gtag(){dataLayer.push(arguments);}
  gtag("js", new Date());
  gtag("config", "G-XXXXXXXXXX");
</script>

<!-- AFTER: blocked until visitor consents to "analytics" -->
<script type="text/plain"
        data-type="text/javascript"
        data-name="analytics"
        data-src="https://www.googletagmanager.com/gtag/js?id=G-XXXXXXXXXX">
</script>
<script type="text/plain"
        data-type="text/javascript"
        data-name="analytics">
  window.dataLayer = window.dataLayer || [];
  function gtag(){dataLayer.push(arguments);}
  gtag("js", new Date());
  gtag("config", "G-XXXXXXXXXX");
```

```
</script>
```

3.2 Google Tag Manager (GTM)

GTM loads one container script that then injects everything else (GA4, Ads, other vendors' tags) at runtime — those injected tags are never present as static HTML, so the engine can't intercept them directly. Instead, gate the GTM container itself so nothing inside it fires before consent.

Note: Remove the standard GTM snippet entirely — do NOT keep both the old snippet and the new blocked version. Having both will cause double-firing of all tags.

```
<!-- REPLACE the standard GTM <head> snippet with: -->
<script type="text/plain"
        data-type="text/javascript"
        data-name="analytics"
        data-src="https://www.googletagmanager.com/gtm.js?id=GTM-XXXX">
</script>
```

Note: Replace `GTM-XXXX` with your actual GTM Container ID. If GTM was discovered/categorized under something other than "analytics" for your Consent Manager, use that category name instead — confirm in your admin portal.

Blocking the container alone stops GTM from loading at all until consent is granted, but GTM itself still needs to be told which of *its own* tags to fire once that happens — otherwise every tag inside GTM fires together the moment the container loads, regardless of which category the visitor actually accepted. Configure this using GTM's own Custom Event Triggers, listening for the consent events the engine already pushes to `window.dataLayer` automatically:

```
privacyconsent-analytics-accepted
privacyconsent-marketing-accepted
privacyconsent-functional-accepted
```

Note: These follow the pattern `privacyconsent-<category>-accepted`, generated for every category configured in your Consent Manager — not just the three shown above. A matching `-rejected` event is also pushed for each category, but you only need a trigger for it if a tag must actively clean up after itself once already running (For example, stopping an in-progress session-recording tool) — a tag wired only to the "accepted" trigger simply never fires without consent in the first place, so most tags (GA4, Facebook Pixel, standard analytics/marketing pixels) don't need a "rejected" trigger at all.

Create a trigger for each category you use:

1. In your GTM workspace, go to **Tags > Triggers > New**.
2. Choose Trigger Type: **Custom Event**.
3. `privacyconsent-analytics-accepted`.
4. Set "This trigger fires on": **All Custom Events**.

5. Name the trigger “Seqrite: Analytics Accepted” and click **Save**.
6. Repeat for each category you use — For example, `privacyconsent-marketing-accepted` → “Seqrite: Marketing Accepted”, `privacyconsent-functional-accepted` → “Seqrite: Functional Accepted”.

Assign triggers to your tags:

1. Open your GA4 Configuration tag.
2. Under Triggering, remove the existing “Page View” trigger.
3. Add trigger: “Seqrite: Analytics Accepted”. Save.
4. Open your Facebook Pixel tag and assign trigger: “Seqrite: Marketing Accepted”.
5. Open any other tags (Hotjar, Bing UET, etc.) and assign the relevant category’s trigger.
6. Publish your GTM container.

Test in GTM Preview Mode:

1. Open GTM workspace > **Preview**. Enter your website URL and click Connect.
2. On your website, do NOT consent yet. Verify the GA4 tag does **NOT** appear in the GTM debug panel.
3. Accept Analytics consent. Verify `privacyconsent-analytics-accepted` appears in the debug panel and the GA4 tag fires.
4. Publish your GTM workspace changes once testing is complete.

3.3 Google Consent Mode v2 — Automatic, No Code Needed

Unlike everything else in this section, **you do not need to add any Consent Mode v2 code yourself.** The bootloader snippet from Step 1 already sets Google’s default signals (`analytics_storage`, `ad_storage`, `ad_user_data`, `ad_personalization`) to `denied` the moment it loads, and updates them to `granted` automatically the instant a visitor accepts the matching category — before your GA4/Ads tags (gated per 3.1/3.2/3.5) ever fire. This is shown here only so you know what’s happening under the hood if you check it directly in GA4’s own DebugView or Google Tag Assistant.

3.4 Google-Hosted iframes (Maps, and other Google embeds)

```
<!-- BEFORE: loads immediately -->
<iframe src="https://www.google.com/maps/embed?pb=..."
        width="600" height="450" style="border:0;" allowfullscreen>
</iframe>

<!-- AFTER: blocked until visitor consents to "functional" -->
<iframe data-src="https://www.google.com/maps/embed?pb=..."
        data-name="functional"
        width="600" height="450" style="border:0;" allowfullscreen>
</iframe>
```

Note: A contextual overlay is shown where the iframe would appear, informing visitors that they

must consent to view the embedded content. The overlay disappears automatically once consent is granted.

3.5 Google Ads (Conversion Tracking / Remarketing)

Google Ads conversion and remarketing tags are separate from GA4 — most sites running both need to gate each independently.

```
<!-- BEFORE: loads unconditionally -->
<script async src="https://www.googletagmanager.com/gtag/js?id=AW-XXXXXXXXX"></script>
<script>
  window.dataLayer = window.dataLayer || [];
  function gtag(){dataLayer.push(arguments);}
  gtag("js", new Date());
  gtag("config", "AW-XXXXXXXXX");
</script>

<!-- AFTER: blocked until visitor consents to "marketing" -->
<script type="text/plain"
        data-type="text/javascript"
        data-name="marketing"
        data-src="https://www.googletagmanager.com/gtag/js?id=AW-XXXXXXXXX">
</script>
<script type="text/plain"
        data-type="text/javascript"
        data-name="marketing">
  window.dataLayer = window.dataLayer || [];
  function gtag(){dataLayer.push(arguments);}
  gtag("js", new Date());
  gtag("config", "AW-XXXXXXXXX");
</script>
```

3.6 Other Google-Owned Embeds (reCAPTCHA, Fonts, YouTube)

These are still Google properties, and are gated the same markup way — pick whichever category fits your site’s classification of them (typically “functional”).

```
<!-- Google reCAPTCHA -->
<script type="text/plain"
        data-type="text/javascript"
        data-name="functional"
        data-src="https://www.google.com/recaptcha/api.js">
</script>

<!-- Google Fonts -->
<link type="text/plain"
      data-type="text/css"
      data-name="functional"
      data-href="https://fonts.googleapis.com/css2?family=Roboto"/>
```

```
<!-- YouTube embed -->
<iframe data-src="https://www.youtube.com/embed/VIDEO_ID"
        data-name="functional"
        width="560" height="315" frameborder="0" allowfullscreen>
</iframe>
```

3.7 Facebook / Meta Pixel (Direct Tag) — Only Gate the Init/Track Calls

Unlike the Google examples above, do **not** copy the standard Meta Pixel base code — the `!function(f,b,e,v,n,t,s){...}` loader block — onto your page at all, gated or otherwise.

Note: Do not paste the Pixel loader/stub snippet yourself, even inside a gated block. The Step 1 bootloader already defines `window.fbq` and loads `fbevents.js` automatically, synchronously, before anything else on the page runs — the same mechanism it uses to arm Google Consent Mode v2's default-denied state (3.3) — with Meta's own consent defaulted to `revoke`. By the time any of your own gated scripts activate, `window.fbq` already exists, so the standard snippet's own `if(f.fbq)return;` guard makes your copy do nothing — it's not just redundant, it silently no-ops. Only the tracking calls below actually need to run on your page, and only those need gating.

```
<!-- BEFORE: fires PageView unconditionally -->
<script>
fbq('init', '1234567890123456');
fbq('track', 'PageView');
</script>

<!-- AFTER: blocked until visitor consents to "marketing" -->
<script type="text/plain"
        data-type="text/javascript"
        data-name="marketing">
fbq('init', '1234567890123456');
fbq('track', 'PageView');
</script>
```

Note: Gate any other Pixel event calls (`fbq('track', 'Lead')`, `fbq('track', 'Purchase')`, etc.) the same way, under `data-name="marketing"`. The loader itself never needs to appear in your markup — it's already handled for you.

3.8 Meta Pixel via Google Tag Manager

If your Meta Pixel runs as a tag inside GTM rather than as a separate direct tag, no extra markup is needed for it specifically — gating the GTM container itself (3.2) already keeps it from firing until consent is granted. Assign it the same "Seqrite: Marketing Accepted" Custom Event Trigger you created in 3.2, in place of GTM's default Page View / Initialization trigger.

Note: If you run Meta Pixel both via GTM and as a direct tag on the same page (a common "belt-and-suspenders" setup, so tracking still works if GTM itself is blocked by an ad blocker), gate both

independently as shown in 3.2 and 3.7 — don't rely on Meta's built-in event deduplication as a substitute for consent gating. Deduplication only prevents double-counting once both copies are already allowed to fire; it does nothing to stop either one from firing before consent.

3.9 Meta Consent Signal — Automatic, No Code Needed

Like Google Consent Mode v2 (3.3), you do not need to write any consent-signalling code for Meta yourself. The Step 1 bootloader defines `window.fbq` and loads `fbevents.js` immediately on every page (see 3.7), with Meta's own consent state defaulted to `revoke` before anything else runs — then automatically calls `fbq('consent', 'grant')` or `fbq('consent', 'revoke')` the instant the visitor's "marketing" choice changes, whether that's on the initial banner or later via the Preference Centre. This happens regardless of whether the Pixel has actually tracked anything yet — what still gates the tracking itself is gating `fbq('init', ...)` / `fbq('track', ...)` as shown in 3.7, since Meta's consent API controls Limited Data Use mode, not whether the Pixel's own tracking calls execute at all.

3.10 Facebook Page Plugin / Like Box (iframe)

```
<!-- BEFORE: loads immediately -->
<iframe
src="https://www.facebook.com/plugins/page.php?href=https%3A%2F%2Fwww.facebook.com%2Fyourpage&tabs=&width=400&height=130&small_header=true"
        width="400" height="130" style="border:none;overflow:hidden" scrolling="no"
frameborder="0" allowfullscreen="true"
        allow="autoplay; clipboard-write; encrypted-media; picture-in-picture; web-share"></iframe>

<!-- AFTER: blocked until visitor consents to "marketing" -->
<iframe data-
src="https://www.facebook.com/plugins/page.php?href=https%3A%2F%2Fwww.facebook.com%2Fyourpage&tabs=&width=400&height=130&small_header=true"
        data-name="marketing"
        width="400" height="130" style="border:none;overflow:hidden" scrolling="no"
frameborder="0" allowfullscreen="true"
        allow="autoplay; clipboard-write; encrypted-media; picture-in-picture; web-share"></iframe>
```

Note: Same contextual-overlay behavior as Google-hosted iframes (3.4) — a placeholder is shown asking visitors to consent until "marketing" is granted, then the plugin loads automatically.

3.11 Microsoft Bing UET (Direct Tag)

Unlike Meta (3.7), the Step 1 bootloader does **not** load the real Bing UET script (`bat.js`) for you — it only seeds an empty `window.uetq` queue array and pushes a default-denied consent entry onto it. The actual UET loader still has to be present in your own markup, gated in full, exactly as it normally appears:

```
<!-- BEFORE: loads and fires pageLoad unconditionally -->
<script>
(function(w,d,t,r,u){var f,n,i;w[u]=w[u]||[];
f=function(){var o={ti:'YOUR_UET_TAG_ID',enableAutoSpaTracking:true};
o.q=w[u];w[u]=new UET(o);w[u].push('pageLoad')};
n=d.createElement(t);n.src=r;n.async=1;
n.onload=n.onreadystatechange=function(){
  var s=this.readyState;
  s&&s!=='loaded'&&s!=='complete'||(f(),n.onload=n.onreadystatechange=null)
};i=d.getElementsByTagName(t)[0];i.parentNode.insertBefore(n,i)
})(window,document,'script','//bat.bing.net/bat.js','uetq');
</script>

<!-- AFTER: blocked until visitor consents to "marketing" -->
<script type="text/plain"
        data-type="text/javascript"
        data-name="marketing">
(function(w,d,t,r,u){var f,n,i;w[u]=w[u]||[];
f=function(){var o={ti:'YOUR_UET_TAG_ID',enableAutoSpaTracking:true};
o.q=w[u];w[u]=new UET(o);w[u].push('pageLoad')};
n=d.createElement(t);n.src=r;n.async=1;
n.onload=n.onreadystatechange=function(){
  var s=this.readyState;
  s&&s!=='loaded'&&s!=='complete'||(f(),n.onload=n.onreadystatechange=null)
};i=d.getElementsByTagName(t)[0];i.parentNode.insertBefore(n,i)
})(window,document,'script','//bat.bing.net/bat.js','uetq');
</script>
```

Note: The `w[u]=w[u]||[]` line means this snippet safely picks up the queue array the bootloader already created, instead of overwriting it — no changes needed to the snippet itself beyond the usual `type="text/plain"` gating.

3.12 Microsoft Clarity (Direct Tag)

Clarity behaves the same way as Bing UET here: the bootloader only defines a queueing stub for `window.clarity` and defaults its consent state to denied — it does not load the real `clarity.ms` script. Gate the full loader snippet yourself:

Note: Before gating the snippet, you must also turn off Clarity's own default cookie behaviour in the Clarity project console itself. By default, Clarity sets its cookies (`_clck`, `_clsk`) as soon as its script loads, regardless of any markup-level gating on your site — gating only controls *when the script loads*, not what it does once it has. In the Clarity dashboard, go to your project's **Settings > Setup > Advanced settings > Cookies** and turn this toggle **Off**. This puts Clarity into Consent Mode, where it will not set any cookies until it receives an explicit consent signal — which is exactly what the automatic relay in 3.13 (`clarity('consentv2', {...})`) then provides once the visitor consents. Skipping this console-side step means Clarity will still drop cookies the moment its script executes, even though you've gated the script tag and even though the engine is sending the correct consent

signal.

```
<!-- BEFORE: loads unconditionally -->
<script>
(function(c,l,a,r,i,t,y){c[a]=c[a]||function(){(c[a].q=c[a].q||[]).push(arguments)};
t=l.createElement(r);t.async=1;t.src='https://www.clarity.ms/tag/'+i;
y=l.getElementsByTagName(r)[0];y.parentNode.insertBefore(t,y)}
)(window,document,'clarity','script','YOUR_CLARITY_ID');
</script>

<!-- AFTER: blocked until visitor consents to "analytics" -->
<script type="text/plain"
        data-type="text/javascript"
        data-name="analytics">
(function(c,l,a,r,i,t,y){c[a]=c[a]||function(){(c[a].q=c[a].q||[]).push(arguments)};
t=l.createElement(r);t.async=1;t.src='https://www.clarity.ms/tag/'+i;
y=l.getElementsByTagName(r)[0];y.parentNode.insertBefore(t,y)}
)(window,document,'clarity','script','YOUR_CLARITY_ID');
</script>
```

Note: Clarity is typically classified “analytics” rather than “marketing” — confirm which category Clarity was discovered/assigned under in your admin portal before picking `data-name`.

3.13 Microsoft Consent Signal — Automatic, No Code Needed

As with Google (3.3) and Meta (3.9), you don’t write any consent-relay code for Bing or Clarity yourself — the engine automatically pushes the right signal as the visitor’s choice changes:

- **Bing UET**: `window.uetq.push('consent', 'update', { 'ad_storage': ... })`, tied to the “marketing” category.
- **Clarity**: `window.clarity('consentv2', { ad_Storage: ..., analytics_Storage: ... })`, tied to both “marketing” and “analytics”.

Unlike Meta, this signal only has anything to act on once the real `bat.js` / `clarity.ms` script has actually loaded — which, per 3.11/3.12, only happens once you’ve gated and the visitor has consented. Until then, the automatic default-denied stub is all that exists; gating your own script tags is still what prevents tracking before consent.

3.14 Bing Maps (iframe)

```
<!-- BEFORE: loads immediately -->
<iframe src="https://www.bing.com/maps/embed?h=300&w=560&cp=51.5074~-
0.1278&lvl=12&typ=d&sty=r&src=SHELL&FORM=MBEDV8"
        width="560" height="300" style="border:none;display:block;max-width:100%"
        scrolling="no" frameborder="0" allowfullscreen="true"></iframe>

<!-- AFTER: blocked until visitor consents to "marketing" -->
<iframe data-src="https://www.bing.com/maps/embed?h=300&w=560&cp=51.5074~-
```

```
0.1278&lvl=12&typ=d&sty=r&src=SHELL&FORM=MBEDV8"
        data-name="marketing"
        width="560" height="300" style="border:none;display:block;max-width:100%"
        scrolling="no" frameborder="0" allowfullscreen="true"></iframe>
```

Note: Same contextual-overlay behavior as other gated iframes (3.4, 3.10). Bing Maps sets `MUID/MR/SRCHUSR/SRCHD/_SS/MSFPC` under the bing.com domain — confirm the category assigned to it in your admin portal, since some sites classify mapping embeds as "functional" instead.

3.15 Any Other Script (Generic Pattern)

Everything above is really the same four-line pattern applied to specific vendors. For any other tracking script or iframe your site uses — a support-chat widget, a self-hosted analytics tool, a survey tool, an A/B testing script, anything — apply it the same way:

```
<!-- BEFORE: loads unconditionally -->
<script src="https://example-vendor.com/tracker.js"></script>
<script>
  ExampleVendor.init('YOUR-ID');
</script>

<!-- AFTER: blocked until visitor consents to the matching category -->
<script type="text/plain"
        data-type="text/javascript"
        data-name="analytics"
        data-src="https://example-vendor.com/tracker.js">
</script>
<script type="text/plain"
        data-type="text/javascript"
        data-name="analytics">
  ExampleVendor.init('YOUR-ID');
</script>
```

The same rule applies to any `<iframe>` embed: move its URL from `src` to `data-src` and add `data-name="<category>"`.

Note: There is nothing Google/Meta/Microsoft-specific about the mechanism itself (3's intro walks through exactly what it does) — every example in this guide is the same `type="text/plain"` + `data-*` pattern, just with a different URL and category each time. If a script isn't covered by name anywhere in this guide, this is the pattern to use for it.

4. Step 4 — Approach B: API-Based Integration

Use this approach for anything that isn't a static tag in your page source — React/Angular/Vue components, scripts injected via `fetch`/`import()`/`createElement`, or any other runtime-loaded logic the engine can't find and mark up by scanning the page (see Step 2). Instead of gating a tag, your own code asks the engine directly, via the global `PrivacyConsent` object the bootloader attaches to

`window.`

4.1 Check Consent Before Loading

```
// Returns true only if ALL services in the category are consented:
if (PrivacyConsent.getConsentByCategory('analytics')) {
    initGoogleAnalytics();
}

// Or check a specific service directly, if your Consent Manager
// defines services individually rather than one-per-category:
if (PrivacyConsent.getConsent('analytics')) {
    initGoogleAnalytics();
}
```

Call this wherever your code would otherwise unconditionally load a script — a component's mount/init logic, a route guard, a module's top-level setup.

4.2 React to Consent Changes

A visitor can grant or withdraw consent at any time after your code has already run once — via the initial banner, or later through the Preference Centre (Step 5). `PrivacyConsent.watch()` calls your function on every change, not just once at load:

```
PrivacyConsent.watch(function(consents) {
    if (consents['analytics'] === true) {
        initGoogleAnalytics();
    }
    if (consents['marketing'] === false) {
        // Clean up cookies already set before the visitor withdrew consent —
        // gating only stops NEW activity, it does not retroactively delete
        // cookies a script already wrote while consent was previously granted.
        document.cookie = '_fbp=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/';
        document.cookie = '_gcl_au=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/';
    }
});

// Stop listening if your component unmounts:
PrivacyConsent.unwatch(myCallback);
```

Note: Consent can be withdrawn, not just granted. A common integration mistake is only handling the "consent granted → load script" direction and forgetting the reverse. If your tracking tool has any client-side state (cookies, an active session, a running recorder) that should stop when a visitor revokes consent, handle that inside the same `watch()` callback — don't assume gating on load is enough for the whole visit.

4.3 Framework Example (React)

```
useEffect(() => {
  if (window.PrivacyConsent && window.PrivacyConsent.getConsentByCategory('analytics'))
{
    initGoogleAnalytics();
  }

  const onConsentChange = (consents) => {
    if (consents['analytics'] === true) initGoogleAnalytics();
  };
  window.PrivacyConsent && window.PrivacyConsent.watch(onConsentChange);

  return () => {
    window.PrivacyConsent && window.PrivacyConsent.unwatch(onConsentChange);
  };
}, []);
```

Note: Guard every call with `window.PrivacyConsent &&` — the bootloader from Step 1 attaches this object asynchronously, so it may not exist yet the instant your component's first render runs, particularly on a slow connection.

4.4 Consent-Mode Signals — Automatic, Manual Trigger Available

Google/Meta/Microsoft consent-mode signals (3.3, 3.9, 3.13) are sent automatically on every change picked up by `watch()` internally — you don't need to call these yourself in normal use. They're exposed for the rare case where your own runtime-loaded script needs the signal sent again on demand (For example, after re-initializing a vendor SDK your own code tore down and rebuilt):

```
PrivacyConsent.updateGoogleConsentMode({ analytics: true, marketing: false });
PrivacyConsent.updateMetaConsentMode({ marketing: false });
PrivacyConsent.updateMicrosoftConsentMode({ marketing: false });
PrivacyConsent.updateClarityConsentMode({ analytics: true, marketing: false });
```

4.5 Banner Controls

```
PrivacyConsent.show();       // Show the banner
PrivacyConsent.showModal();  // Show the full consent-manager modal directly
PrivacyConsent.acceptAll();  // Programmatically accept every category
PrivacyConsent.rejectAll();  // Programmatically reject every category
PrivacyConsent.resetAll();   // Clear stored consent (e.g. for a "forget my choice"
link)
```

Note: Use `acceptAll()` / `rejectAll()` only behind an explicit visitor action (a button they clicked) — calling either automatically on page load defeats the purpose of asking for consent at all, and would not hold up as valid consent under DPDP.

5. Step – 5 - Preference Centre Integration

If you've enabled the Preference Centre for your Consent Manager in the admin portal, add a way for visitors to reopen it and change their choices after their first visit.

5.1 Embed the Launcher

Easier option: your admin portal's Publish step provides a ready-to-copy launcher snippet, the same way it does for Step 1's configuration tags — copy it directly from there.

To build it yourself instead:

```
<!-- Cookie Preference Centre Launcher -->
<button type="button" class="pc-preference-btn" style="display:none;"
onclick="PrivacyConsent.openPreferenceCentre()">
  Manage Cookie Preferences
</button>
```

Note: The `style="display:none"` is intentional and required — leave it as-is. The engine reveals every element with class `pc-preference-btn` on the page automatically, but only once the visitor has completed their initial Accept/Reject/Save. A returning visitor sees it immediately on load; a first-time visitor sees it appear right after they respond to the initial banner. This is deliberate: showing a "manage your preferences" control before a visitor has made any choice at all is confusing and is blocked by design — don't remove the inline style to "fix" it.

You can use `class="pc-preference-btn"` on more than one element if you want the control available in multiple places on a page (For example, a footer link and a settings-page button) — all of them are revealed and wired up the same way.

5.2 Custom Trigger Element (Floating Icon Example)

The button markup in 5.1 is just a convenience default — any element works as a trigger, as long as it keeps the class `pc-preference-btn`, keeps `style="display:none;"` so the reveal logic still controls it, and calls `PrivacyConsent.openPreferenceCentre()`. A common real-world variant is a floating circular icon fixed to a corner of the screen instead of an inline button, so it's reachable from every page without taking up layout space:

```
<style>
  .pc-preference-btn {
    position: fixed;
    bottom: 20px;
    right: 20px;
    width: 56px;
    height: 56px;
    border-radius: 50%;
    background-color: #1b4f72;
    color: white;
```

```
    border: none;
    cursor: pointer;
    box-shadow: 0 4px 12px rgba(0, 0, 0, 0.3);
    font-size: 24px;
    display: flex;
    align-items: center;
    justify-content: center;
    z-index: 999;
    transition: background-color 0.3s, box-shadow 0.3s;
  }
  .pc-preference-btn:hover {
    background-color: #0f3849;
    box-shadow: 0 6px 16px rgba(0, 0, 0, 0.4);
  }
</style>

<button type="button" class="pc-preference-btn" style="display:none;"
        onclick="PrivacyConsent.openPreferenceCentre()"
        title="Manage Cookie Preferences">⚙</button>
```

Note: Only `display:none` in the inline `style` attribute is meaningful to the reveal logic — everything else in the CSS block (position, size, color, the ⚙ icon) is purely cosmetic and yours to change freely. The engine still finds this element by its `pc-preference-btn` class and un-hides it the same way as the plain-button version in 5.1, once the visitor has completed their initial Accept/Reject/Save — it doesn't care what the element looks like or where it's positioned.

`openPreferenceCentre()` takes no required argument — whether it opens the banner first or jumps straight to the full modal is decided automatically from your Consent Manager's current Preference Centre behaviour setting, re-read on every page load. If you change that setting in the admin portal later, this snippet does not need to be re-copied or touched.

5.3 Live Enable/Disable — No Site Changes Needed

Enabling or disabling the Preference Centre in the admin portal takes effect on visitors on their very next-page view. If you disable it, `openPreferenceCentre()` becomes a no-op and the launcher button/link stays hidden — you don't need to remove the snippet from your site first and re-enabling it later doesn't require re-adding anything either.

6. Step 6 - Frequently Asked Questions

Q: The consent banner isn't showing up on my site at all — what's wrong?

Check, in order: the bootloader `<script>` from Step 1 is actually present in `<head>` and loading (check your browser's Network tab for a failed/404 request), the Consent Manager configuration is actually **published** in the admin portal (an unpublished config won't serve a working banner), and your browser console for JavaScript errors — an error thrown by another script on the page before the bootloader runs can stop it from initializing.

Q: I gated a script correctly but it's still firing before consent — why?

The two most common causes: a typo in `type="text/plain"` (it must be exactly that string, not `"text/javascript"` left in by mistake), or a `data-name` value that doesn't exactly match a category name configured in your Consent Manager (case-sensitive — confirm the exact spelling in your admin portal). Also double-check you gated **every** related `<script>` tag — several examples in Section 3 (3.1, 3.5) use two separate tags (the loader and an inline config call), and it's easy to gate only the first one.

Q: I'm using GTM, and now every tag inside it fires together the instant the container loads — what did I miss?

Gating the GTM container tag itself (3.2) only blocks the container from loading at all until consent — it does nothing to control which tags *inside* GTM fire once it has loaded. You still need to create Custom Event Triggers in GTM itself and assign them to each tag, as described in 3.2.

Q: An iframe's "consent required" overlay never goes away, even after I accept.

Confirm the `data-name` on the iframe matches the category the visitor consented to, and that `data-src` (not `src`) holds the real embedded URL. If both are correct, try a hard refresh — some browsers cache the pre-consent DOM state briefly.

Q: Do cookies I've classified as "necessary" need to be gated too?

No. Cookies in the "necessary" category are excluded from denial by design and load regardless of visitor consent — the `type="text/plain"` treatment in Section 3 and the API checks in Section 4 are only for categories a visitor can accept or reject (analytics, marketing, functional, etc.).

Q: A visitor withdrew consent for a category after cookies in it were already set — are those cookies automatically deleted?

No. Gating and the automatic consent-mode signals (3.3, 3.9, 3.13) only control *future* activity — a script that already ran and set cookies while consent was granted isn't retroactively cleaned up. If you need that, delete the cookies yourself inside a `PrivacyConsent.watch()` callback (4.2). Some cookies genuinely cannot be deleted client-side at all (For example, once a third party sets server-side, like Meta's `fr` cookie via `facebook.com/tr`) those simply expire on their own schedule.

Q: Should I use markup-based (Approach A) or API-based (Approach B)?

It depends on how each script reaches the browser, not on what kind of site you have — see the decision table in Step 2. Most real sites end up using both: markup-based for anything already written as a static `<script>`/`<iframe>` tag, API-based for anything injected at runtime (React/Angular/Vue components, dynamically constructed scripts).

Q: My component calls `PrivacyConsent.something()` on mount and throws "PrivacyConsent is not defined" — why?

The bootloader from Step 1 attaches `window.PrivacyConsent` asynchronously, so it may not exist yet the instant your code first runs — particularly on a slow connection. Guard every call with `window.PrivacyConsent &&`, as shown in 4.3.

Q: How do I test that my integration is working before going live?

Open your site in a private/incognito window and check the Network tab: no analytics/marketing cookies or requests should be fired until you interact with the banner. For GTM-managed tags, use GTM's own Preview Mode (3.2). For Google Consent Mode v2, check GA4 DebugView or Google Tag Assistant (3.3). Finally, reopen the Preference Centre (Section 5), withdraw the consent you previously granted, and confirm no further activity for that category occurs afterward.

Q: Does the consent banner reappear on every page visit?

No — once a visitor has made a choice (Accept/Reject/Save), that decision is stored and the banner won't reappear on later visits or page views. It shows again only for a new visitor who hasn't decided yet, or if their stored consent is cleared (For example, via `PrivacyConsent.resetAll()`, 4.5, or if they clear their browser storage).

Q: I renamed a category (or added a new one) in the admin portal — do I need to update my site?

For markup-based tags, yes: every `data-src`/`data-name` block's `data-name` must exactly match the current category name in your Consent Manager. Renaming a category without updating the matching `data-name` values means those elements stop being recognized and never activate again. API-based checks (`getConsentByCategory('oldName')`, Section 4) need the same update.