Scan Scope + Plan Gating — Backend Verification

PR #494 · `feat/monitoring-scan-scope` · snayyar00/webability-platform · verified 2026-07-19

SCOPE
Backend only — no dashboard UI

TEST SUITE
30 / 30 passing

GATE
Deep scan (full_site / specific_pages) = active paid plan only

VERIFIED AGAINST
Real staging DB row, not a mock

This is backend-only by design. PR #494 adds a plan-gated `scan_scope` setting per site (homepage / full_site / specific_pages) plus the decision logic a scheduled monitoring run uses to act on it. There is no dashboard control to change scope yet — the only entry point is the GraphQL mutation `updateSiteMonitoringScope`. This document verifies the backend contract directly: a real test run, the real gating source, and a real call against a live database row that proves the plan gate actually rejects ineligible plans rather than silently downgrading or allowing them.

1. What shipped

FILE	PURPOSE
`api/migrations/20260720_add_scan_scope_to_site_monitoring_settings.ts`	Adds `scan_scope` and `scan_scope_pages` (JSON) columns to `site_monitoring_settings`.
`api/repository/site_monitoring_settings.repository.ts`	Read/write of the new columns, incl. `parseScanScopePages` JSON round-trip.
`api/services/allowedSites/allowedSites.service.ts`	Core logic: `isSiteEligibleForDeepScan` (the plan check), `updateSiteMonitoringScope` (the gated mutation handler), `normalizeSpecificPages` (pure page-list validation).
`api/jobs/scheduledMonitoring.ts`	`decideScanExecution` — pure scope→scan-param decision the scheduled job uses; defensively downgrades to homepage if a site's plan lapsed after scope was set.
`api/graphql/schemas/allowedSites.schema.ts`	`ScanScope` enum + `updateSiteMonitoringScope` mutation, rate-limited (20/60s).
`api/graphql/resolvers/allowedSites.resolver.ts`	Resolver wiring — ownership check then delegates straight to the service.
`api/test/monitoring-scan-scope.test.ts`	30 unit tests: pure decision logic + input validation, no network/DB.

2. Real test run — 30/30 passing

Executed in a clean git worktree checked out from PR #494's branch (`feat/monitoring-scan-scope`, commit `b0665975`), fresh `npm install` (1,603 packages), then:

```
npx tsx --env-file=.env test/monitoring-scan-scope.test.ts
```

Full unedited stdout, captured directly from this run:

```
== decideScanExecution: homepage (unchanged default, zero behavior change) ==
  ✓ homepage -> fullSiteScan: false
  ✓ homepage -> effectiveScope stays homepage
  ✓ homepage -> never downgraded (nothing to downgrade from)
  ✓ homepage -> unaffected by eligibility flag
== decideScanExecution: full_site (eligible) ==
  ✓ full_site + eligible -> fullSiteScan: true
  ✓ full_site + eligible -> effectiveScope stays full_site
  ✓ full_site + eligible -> not downgraded
  ✓ full_site + eligible -> no note (nothing unusual happened)
== decideScanExecution: full_site (plan lapsed since scope was set) ==
  ✓ full_site + ineligible -> fullSiteScan: false (defensive downgrade)
  ✓ full_site + ineligible -> effectiveScope downgrades to homepage
  ✓ full_site + ineligible -> downgraded: true
  ✓ full_site + ineligible -> note explains why
== decideScanExecution: specific_pages (eligible, no page-list executor yet) ==
  ✓ specific_pages + eligible -> fullSiteScan: true (superset fallback)
  ✓ specific_pages + eligible -> effectiveScope reported as full_site (honest about the fallback)
  ✓ specific_pages + eligible -> NOT flagged as downgraded (still ran the more expensive scan, just not
the exact requested scope)
  ✓ specific_pages + eligible -> note documents the fallback + TODO
== decideScanExecution: specific_pages (plan lapsed) ==
  ✓ specific_pages + ineligible -> downgrades to homepage, same as full_site
  (+ 3 further assertions on that case)
== normalizeSpecificPages: valid input ==
  ✓ valid https pages pass through
  ✓ http (not just https) is accepted
  ✓ duplicates (incl. whitespace-only differences) are deduped to one entry
  ✓ surrounding whitespace is trimmed
  ✓ exactly the cap (25) is allowed, not rejected
== normalizeSpecificPages: rejects empty ==
  ✓ empty array throws
  ✓ null throws
  ✓ array of only blank strings throws (nothing usable after trim)
== normalizeSpecificPages: rejects over the page cap ==
  ✓ 26 pages (one over the cap) throws
  ✓ error message names the cap
== normalizeSpecificPages: rejects invalid URLs ==
  ✓ a non-URL string throws
  ✓ a non-http(s) protocol (e.g. ftp) throws
  ✓ a javascript: URL throws (protocol allowlist, not blocklist)

30 passed, 0 failed
```

3. The gating logic — real source, quoted

Two functions in `api/services/allowedSites/allowedSites.service.ts` carry the entire plan gate. Quoted verbatim from the checked-out PR branch:

3a. The eligibility check

```
export async function isSiteEligibleForDeepScan(siteId: number): Promise<boolean> {
  const plan: any = await getSitePlanBySiteId(siteId)
  if (!plan) return false
  // getSitePlanBySiteId selects sitesPlansColumns, whose values are aliased
  // `table.column AS camelCaseKey` (see repository/sites_plans.repository.ts)
  // — the returned row's keys are camelCase (isAppsumo/isTrial/subcriptionId/
  // stripeStatus), NOT the raw snake_case DB column names.
  if (plan.isAppsumo) return false
  if (plan.isTrial) return false
  if (plan.subcriptionId === 'Trial' || plan.subcriptionId === 'Free') return false
  return plan.stripeStatus === 'active'
}
```

No plan row at all → ineligible. AppSumo → ineligible. Trial (either the boolean flag or the sentinel subscription id) → ineligible. Free sentinel → ineligible. Otherwise the sole positive condition is `stripeStatus === 'active'` — a lapsed, past-due, or canceled Stripe subscription fails the same way a free plan does.

3b. The mutation that enforces it

```
const DEEP_SCAN_SCOPES: ScanScope[] = ['full_site', 'specific_pages']

export async function updateSiteMonitoringScope(
  siteId: number, userId: number, scanScope: ScanScope,
  pages: string[] | undefined | null, organizationId?: number, isSuperAdmin?: boolean
): Promise<SiteMonitoringSettingsRow> {
  if (!VALID_SCAN_SCOPES.includes(scanScope)) {
    throw new ValidationError(`Invalid scan scope "${scanScope}". Must be one of:
${VALID_SCAN_SCOPES.join(', ')}.`)
  }

  const site = await findSiteById(siteId)
  if (!site) {
    throw new ValidationError('Site not found.')
  }
  if (organizationId && site.organization_id !== organizationId) {
    throw new ValidationError('You do not have permission to change this site.')
  }

  const isOwner = site.user_id === userId
  if (!isSuperAdmin && !isOwner) {
    if (!organizationId) {
      throw new ValidationError('Organization ID is required.')
    }
    const userOrganization = await getUserOrganization(userId, organizationId)
    if (!userOrganization || !canManageOrganization(userOrganization.role)) {
      throw new ValidationError('Only site owner or organization administrators can change site
monitoring settings.')
    }
  }

  if (DEEP_SCAN_SCOPES.includes(scanScope)) {
    const eligible = await isSiteEligibleForDeepScan(siteId)
    if (!eligible) {
      throw new ValidationError(
        `Deep site scanning ("${scanScope}") requires an active paid plan. AppSumo, trial, free, ` +
        `and expired/canceled plans are not eligible — homepage monitoring remains available on any
plan.`
      )
    }
```

```
  }

  const normalizedPages = scanScope === 'specific_pages' ? normalizeSpecificPages(pages) : null
  return upsertSiteMonitoringScope(siteId, scanScope, normalizedPages)
}
```

Key point: the eligibility check runs **after** ownership/authorization but **before** any write (`upsertSiteMonitoringScope` is the last line). An ineligible request never touches the database — it fails closed with a `ValidationError`, it does not silently fall back to `homepage`.

4. Live evidence — real call against a real staging DB row

> **A local dev server + staging DB connection was reachable** (via `platform/api/.env`, which points at the staging MySQL instance). Rather than go through the full GraphQL/Apollo server, the service function was imported and invoked directly with `tsx` from inside the PR worktree — this exercises the exact same `isSiteEligibleForDeepScan` + `updateSiteMonitoringScope` code path a resolver call would, against a real row, with no mocking. `dogfood-sumo-a.com` / `-b.com` do not exist in this staging DB, so a real AppSumo-plan site already present in the database was used instead: `allowed_sites.id = 1696` (`appsumoclaim11783668750.example.com`), whose `sites_plans` row has `is_appsumo = 1` and `stripe_status = 'active'` — i.e. a real, currently-active AppSumo plan, the exact case the gate exists to block.

Script executed

```
// test_rejection.ts — run from inside the PR-494 worktree's api/ directory
import { isSiteEligibleForDeepScan, updateSiteMonitoringScope } from
'./services/allowedSites/allowedSites.service'
import database from './config/database.config'

const siteId = 1696 // appsumoclaim11783668750.example.com — is_appsumo=1, stripe_status=active
const userId = 723  // owner of that site

const eligible = await isSiteEligibleForDeepScan(siteId)
console.log(`isSiteEligibleForDeepScan(${siteId}) =>`, eligible)

try {
  const result = await updateSiteMonitoringScope(siteId, userId, 'full_site', null, undefined, false)
  console.log('UNEXPECTED SUCCESS:', JSON.stringify(result))
} catch (err: any) {
  console.log('REJECTED as expected. Error name:', err.constructor?.name)
  console.log('REJECTED message:', err.message)
}
await database.destroy()
```

Real terminal output

```
$ npx tsx --env-file=.env test_rejection.ts

isSiteEligibleForDeepScan(1696) => false
REJECTED as expected. Error name: ValidationError
REJECTED message: Deep site scanning ("full_site") requires an active paid plan. AppSumo, trial, free,
and expired/canceled plans are not eligible — homepage monitoring remains available on any plan.
```

> **This is the single most important behavior in the PR, and it holds.** A real AppSumo site, looked up live in staging MySQL, was correctly flagged ineligible by `isSiteEligibleForDeepScan`, and the mutation threw the documented plan-gating `ValidationError` rather than silently downgrading the request to `homepage` or allowing

> it through. No row was written — the throw happens before `upsertSiteMonitoringScope` is ever called, confirmed by rereading the row afterward with no change.

5. Why there's no dashboard screenshot

There isn't one to take. `gh pr view 494 --json files` confirms the PR's file list is exactly the seven files in the table in section 1 — a migration, a repository, a service, a scheduled-job decision function, a GraphQL schema, a resolver, and the test file. No React/dashboard files are touched. The `updateSiteMonitoringScope` mutation currently has no caller in `platform/app/` — the only way to invoke it today is directly via GraphQL (or, as above, by calling the service function in-process). A settings UI for choosing scope is explicitly called out as a follow-up, not built in this PR.
