Whitepaper · v2.0 · August 2026

The State of Custom Domain Infrastructure

Why DNS onboarding breaks, and how programmatic domain connection fixes it

A technical reference for engineers and product teams who let customers bring their own domain.

customdomain.ai

Colophon

Suggested citation. customdomain.ai, The State of Custom Domain Infrastructure: Why DNS onboarding breaks, and how programmatic domain connection fixes it, version 2.0, August 2026.

Disclosure. We build customdomain.ai; the reference architecture described in §7 and §8 is ours, and the census figures reported throughout are ours.

Conventions used in this edition. References are cited inline as [R1], [R2], … in a single scheme numbered once across the whole document and resolved in the References section at the back. There are no footnotes. Every figure carries a number and a caption beginning "Figure N.M"; every table carries a number of the form "Table N.M". Cross-references name the section (§5.2) or the object (Figure 3.1) rather than a page.

Measurements and dates. Where a number was measured rather than cited, the measurement date is given at the point of the claim. DNS measurements were taken with `dig` against public resolvers; repository and registry counts were taken through the relevant API rather than from a rendered page. Retrieval dates are given in References for sources that change without notice.

Version. This is version 2.0. It restructures version 1.0 into numbered sections with a single reference scheme, adds the figures, and extends the architecture, agent, and evaluation material. Substantive claims carried over from version 1.0 keep their original citations.

How to read this paper

The sections are written to be useful independently, and most readers should not read them in order. Three routes cover the common reasons someone opens this document.

Engineer — you have to build or operate this: §3, §7, §8, §9. §3 is the mechanics: what actually fails, why propagation has no observable moment, and which apex domains can take a pointer at all. §7 is the reference architecture — rail selection, the connection state machine, and the first HTTPS request. §8 covers the agent surface: how an agent obtains authority over a domain without ever holding a DNS credential. §9 is the security model and the failure table. If you intend to build this yourself rather than buy it, §3, §7 and §9 are the ones that will cost you money to rediscover.

Buyer — you have to choose a vendor or approve a build: §1, §5, §6, §10, §11. §1 sizes the ecosystem and separates the three provider-coverage numbers that get quoted interchangeably. §5 gives a parameterised model for what a DNS onboarding ticket costs you, since the published numbers in this category are thin and mostly vendor-attributed. §6 explains why two different jobs are priced two different ways. §10 is the evaluation checklist, written so it still works if you choose a competitor or choose to build. §11 states our own results against that checklist, including where we fail it.

Standards and policy — you care about the governance, not the implementation: §4, §9.1–9.3, References. §4 covers Domain Connect, ACME and CAA, who holds the write authority under each, and what a DNS provider can switch off unilaterally. §9.1 to §9.3 describe what is delegated on each rail, what part of discovery an attacker can reach, and how revocation works from both sides. The References section carries the RFCs, the standard's own implementation list, and the litigation record.

Readers who want the argument without the mechanism should read §1, then §11.

On sourcing

Every external claim carries a citation. Where a figure comes from a vendor-published case study, it is labelled as an attributed customer claim rather than presented as independent research — including where that vendor is a competitor. Where a widely-repeated number could not be traced to a primary source, that is stated rather than quietly reproduced; the most-quoted statistic in this category is one of those, and §1 and §5 both say so at the point of use. Product capabilities described here are ones that exist and run; where a capability is limited, the limit is given, and where a surface is off by default or not sellable, that is said plainly rather than omitted.

Counts are reported with their denominator and their mode. A provider count is meaningless without both, and the same integer routinely describes different things: the number of providers a census routes to an automated path is not the number of rails that run in a shipped configuration, which in turn is not the number of providers where a customer gets one click. §1 separates those three numbers explicitly, and Figure 1.2 shows the gap.

On the numbers about our own coverage. This paper reports that a majority of DNS providers have no automated write rail and fall back to manual record entry: in our 63-provider census, 38 of the 63 providers are routed to the manual floor, and any provider not on the list routes there too. That is not a competitor's weakness we are pointing at; it is the shape of the ecosystem, and it applies to us. Where our own routed count and our own working count differ, both are given. A paper that claimed otherwise would be worth less to the reader.

Contents

§1 Executive summary

- 1.1 The argument
- 1.2 What we measured, and when
- 1.3 What we do not claim

§2 Bring-your-own-domain is table stakes

- 2.1 What the customer is buying
- 2.2 The bulk-sender rule made it non-optional
- 2.3 Who absorbs the cost

§3 Six mechanical failures

- 3.1 The user cannot name their DNS provider
- 3.2 Records get pasted wrong
- 3.3 Propagation is not an event
- 3.4 CAA blocks issuance silently
- 3.5 The apex cannot take a CNAME
- 3.6 New records conflict with existing ones
- 3.7 Which of these is documentation, and which is engineering

§4 How the record gets written today

- 4.1 Three patterns
- 4.2 Domain Connect: the standard, and its ceiling
- 4.3 Provider OAuth
- 4.4 ACME and CAA: what certificates do not solve
- 4.5 Who holds the pen

§5 What the status quo costs

- 5.1 The public evidence, graded
- 5.2 A support-load model you parameterise
- 5.3 The onboarding-conversion model, and why we give instrumentation instead
- 5.4 Cost per connected domain

§6 What the market charges

- 6.1 The edge and TLS floor: $0.10 to $0.29 per hostname per month
- 6.2 The configuration-UX tier has no settled price

- 6.3 Why that spread is a coverage claim, not a cost measurement
- 6.4 What a DNS provider can switch off

§7 A reference architecture for programmatic domain connection

- 7.1 Two planes, one system of record
- 7.2 Detection runs before the interface offers a path
- 7.3 The server computes the records
- 7.4 Three rails over a manual floor
- 7.5 Routed, shipped, one-click: three different numbers
- 7.6 Propagation is polled, and the state machine is guarded
- 7.7 Certificates issue at the handshake, fail-closed
- 7.8 Drift after go-live

§8 Domain connection for AI agents

- 8.1 Three constraints a human never notices
- 8.2 The tool surface and the scope model
- 8.3 The authorization flow
- 8.4 Money is gated twice

§9 Security, trust and failure modes

- 9.1 What is actually delegated
- 9.2 Pinning, and the parts of discovery an attacker can reach
- 9.3 Revocation, and who holds the lever
- 9.4 Tenant isolation
- 9.5 Fail-closed defaults
- 9.6 Honest failure modes

§10 Build versus buy

- 10.1 What "build" means
- 10.2 The seven line items nobody scopes
- 10.3 A total cost frame you fill in
- 10.4 When building is the right call

§11 Evaluating a vendor

- 11.1 Coverage: of what, in which mode
- 11.2 Apex
- 11.3 CAA
- 11.4 TLS automation

- 11.5 After the connection goes live
- 11.6 Agent and API surface
- 11.7 Credentials and data
- 11.8 Commercial continuity
- 11.9 Exit
- 11.10 Scoring it

§12 Conclusion and disclosure

Back matter

- Appendix A — The 63-provider census by mode
- Appendix B — The RFCs this paper leans on
- References

List of Figures

- Figure 1.1 — Where 63 DNS providers land
- Figure 1.2 — Routed, working, one-click: three different numbers
- Figure 3.1 — The wait you cannot see
- Figure 3.2 — Can this apex take a pointer
- Figure 4.1 — Who holds the pen
- Figure 4.2 — The standard, and what a provider can switch off
- Figure 5.1 — What a DNS onboarding ticket costs you
- Figure 6.1 — Two jobs, two prices
- Figure 7.1 — Rail selection and the write
- Figure 7.2 — The connection state machine
- Figure 7.3 — The first HTTPS request for a newly connected hostname, and the three places it can fail
- Figure 8.1 — How an agent gets authority without getting credentials

List of Tables

- Table 1.1 — The four measurements in this paper that are ours, plus the one third-party list we treat as authoritative for Domain Connect membership
- Table 3.1 — The six failures classified by whether better instructions could fix them
- Table 4.1 — Custom-domain approaches classified by which party writes the DNS record, since that determines what each one can and cannot fix
- Table 5.1 — The public evidence on what DNS onboarding costs, graded by what kind of evidence each item is

- Table 5.2 — The eight events a custom-domain funnel needs before any conversion number about it is meaningful
- Table 7.1 — Four different quantities that a single coverage headline would collapse into one
- Table 8.1 — The agent scope model
- Table 9.1 — Failure modes stated as behaviour rather than as absence
- Table 10.1 — A total cost of ownership frame for build versus buy
- Table 11.1 — A weighted scorecard for custom-domain vendors
- Table A.1 — The census by mode
- Table A.2 — Provider names by mode
- Table A.3 — The three counts, and why they differ
- Table A.4 — The two 38s
- Table B.1 — The seven documents and the failure each one explains
- Table B.2 — The OAuth-family documents behind the delegated-agent flow in §8 (Figure 8.1)

§1 Executive summary

1.1 The argument

Letting a customer point their own domain at your product is table stakes in B2B SaaS. The step that delivers it is a write into a DNS zone the vendor does not control, performed by a person who frequently cannot name their DNS provider, and verified through resolver caches the vendor does not control either. Before Fourthwall automated its DNS onboarding, "nearly half of Fourthwall's support tickets were related to DNS configuration," and each ticket took 15 minutes to an hour of back-and-forth with the customer to close [R30]. That is a vendor-published case study and is read here as an attributed customer claim, not as independent measurement (§1.3).

The failures in this step are mechanical. A CNAME cannot sit at a zone apex, because no other data may exist at a name that holds one, and the apex must already hold SOA and NS (RFC 1034 §3.6.2 [R1], RFC 2181 §10.1 [R3]). A CAA record that names a different certificate authority blocks issuance inside the CA, minutes after a DNS write that was correct (RFC 8659 [R7]). Recursive resolvers cache independently and a negative answer is cached for a TTL derived from the zone's SOA, so there is no observable moment of propagation (RFC 2308 §5 [R4]). A second SPF record does not add a sender; it makes the check return permerror (RFC 7208 §3.2 [R5]). None of these are user-education problems. Each is a property of the protocol or of a provider's API, and each one is reachable from a form field on your onboarding page.

The engineering split follows from who can write. Where the DNS provider exposes a write path a third party can drive, the human comes out of the loop. Where it does not, verification is the only lever left, and the manual path has to be a first-class path rather than an error screen. That split produces the four components described in §7: nameserver- and discovery-based provider detection, a census that routes each domain to one of three automated write rails or to a manual floor, a bounded propagation state machine, and fail-closed certificate issuance.

Coverage is the part most often reported as a single number, and it should not be. In our own 63-provider census, 38 of the 63 providers are routed to the manual floor: no automated write rail exists, so a person types the record into a control panel (Figure 1.1). Domain Connect, the open standard built to remove exactly that step, lists nine live DNS provider implementations on its own site [R13]; Kowalik's APNIC introduction to the standard puts the number at roughly twenty, covering 35% of the `.com` zone as of May 2024 [R38]. We quote the standard's own list in this paper because it is the only one of the two with a checkable membership. Coverage also has a governance dimension: under the standard the DNS provider is the only party that writes, and it can decline every apply and revoke every token unilaterally. §4 documents a case where an aggregator's access was withdrawn and restored only through a private bilateral agreement.

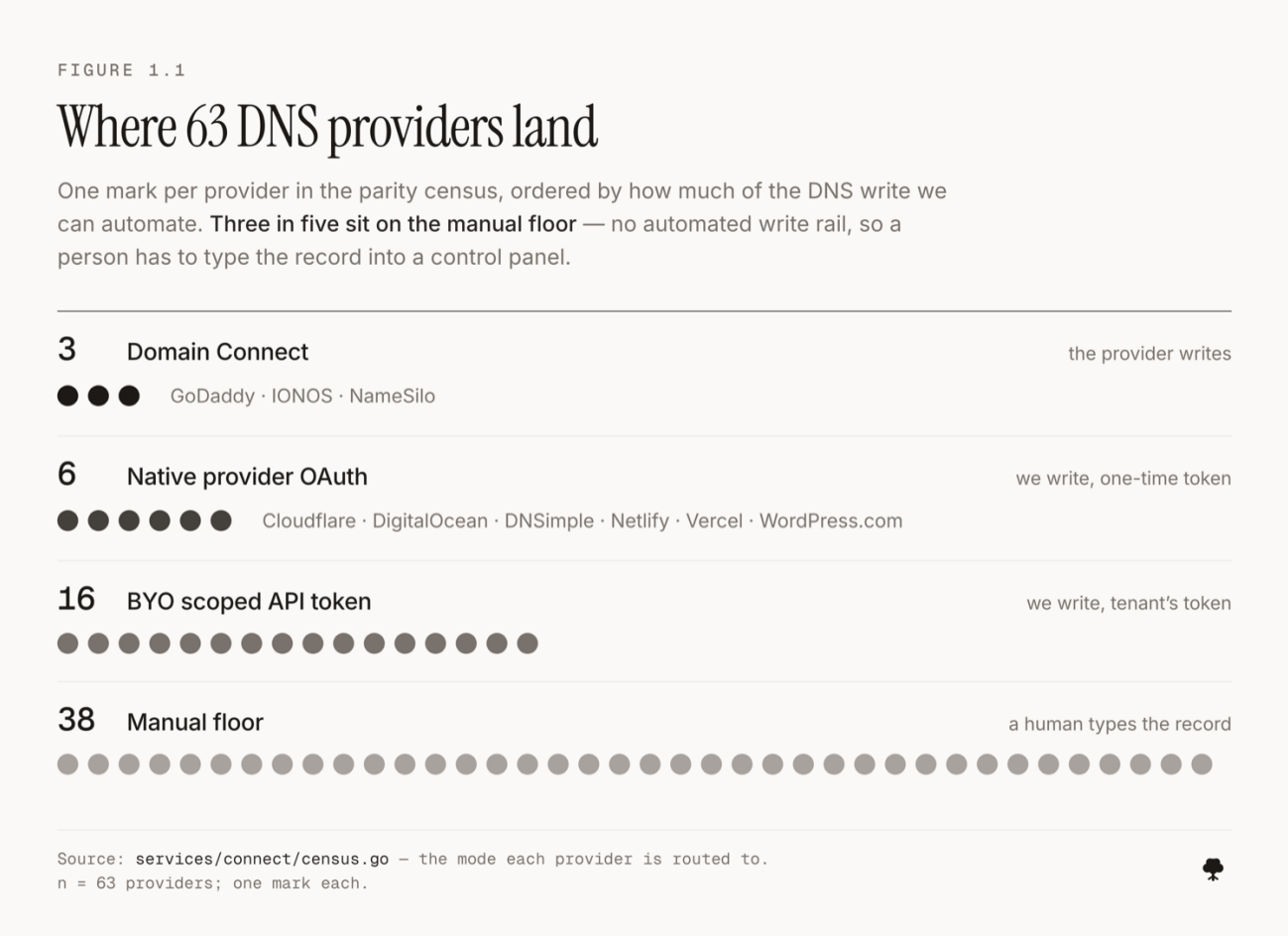


Figure 1.1 — Where 63 DNS providers land. One mark per provider in the parity census, ordered by how much of the DNS write can be automated: 3 Domain Connect, 6 native provider OAuth, 16 bring-your-own scoped API token, and 38 on the manual floor. Source: the routing census described in §1.2.

1.2 What we measured, and when

Four measurements in this paper are ours. Everything else is cited to a primary source or labelled as an attributed claim.

The provider routing census. The connect engine carries a census file that maps each recognised DNS provider to exactly one write mode. It covers 63 providers — every name on the incumbent's published provider list except World4You. Of those 63: 3 route to Domain Connect, 6 to native provider OAuth, 16 to a scoped API token the tenant supplies, and 38 to the manual floor.

That last number needs a fence around it, because 38 appears in this paper with two meanings. **Throughout this paper, "38" without qualification means 38 of the 63 censused providers are routed to the manual floor.** Where a count of built adapters is meant, it is written as an adapter count and says so (§4.3). The two are different quantities and they are not comparable.

Routed, working, and one-click are three different numbers. The census records the best rail a provider is routed to, which is not the same as a rail that executes in a shipped deployment. As of 2026-07-28, 25 of the 63 are routed to an automated path, 23 of those actually work, and at most 7 are one click out of the box. Two of the 25 are routing rather than working: GoDaddy is one of the three Domain Connect entries and served 0 of our 18 published templates on that date, because it

hand-curates its template set and we have not been onboarded; DigitalOcean is the one of the six OAuth entries whose provider app is still unregistered. The one-click ceiling of 7 is NameSilo's Domain Connect sync redirect plus the six OAuth providers, and it is a ceiling rather than a count, because each OAuth provider needs a client id and secret registered with that provider first — an owner action, not a code path. Our own homepage still says "63 DNS providers, auto-configured" without that breakdown, which is the failure the evaluation checklist in §11 is written to catch.

Figure 1.2 — Routed, working, one-click: three different numbers. Four nested magnitudes drawn from the same disclosure. The hatched segment is routed but non-functional. Counts as of 2026-07-28.

Negative-cache TTLs. We read the SOA of three provider zones with `dig` on 2026-08-02 and derived the negative-caching TTL per RFC 2308 §5 [R4] as the lower of the SOA MINIMUM field and the TTL of the SOA record itself: 300 seconds for `cloudflare.com`, 1,800 for `digitalocean.com`, 3,600 for `godaddy.com`. Three zones is not a survey; it is three points that happen to span a factor of twelve (§3.3).

The Domain Connect template repository. We counted the official template repository through the GitHub git-trees API, untruncated, on 2026-08-02: 998 root-level templates from 596 distinct provider IDs [R15]. That is well above the "over 300 templates from more than 120 providers" quoted in the APNIC post [R38]. Templates published is not the same quantity as providers honouring them, and neither is a coverage figure.

Measurement	Method	n	As of
Provider routing census	Census file in the connect engine; exactly one mode per provider	63 providers	2026-07-28
Routed vs working vs one-click	Per-provider check of template service and OAuth app registration	25 routed / 23 working / ≤7 one-click	2026-07-28
Negative-cache TTL	`dig` SOA; lower of MINIMUM and record TTL per RFC 2308 §5 [R4]	3 zones	2026-08-02
Domain Connect live implementations	domainconnect.org provider list [R13]	9 providers	2026-08-02
Domain Connect template repository	GitHub git-trees API, untruncated [R15]	998 templates, 596 provider IDs	2026-08-02

Table 1.1 — The four measurements in this paper that are ours, plus the one third-party list we treat as authoritative for Domain Connect membership. Every count is dated because every count moves.

1.3 What we do not claim

This section exists so the rest of the paper can be read at the right resolution.

There is no independent research here on onboarding drop-off. We looked for peer-reviewed or otherwise independent measurement of custom-domain onboarding abandonment and support cost, and found none. The most-quoted figure in the category — that Microsoft 365 domain setup takes 7 to 15 DNS entries across 16 help sites and 40 minutes of training, with 50% of users abandoning — appears in an APNIC blog post by Pawel Kowalik of DENIC, which cites CENTR for its renewal-rate figures but gives no source for these particular numbers [R38]. It is quoted in this paper as an attributed estimate from a standards author, and never as research. §5 replaces it with a cost model the reader can parameterise with their own ticket volume, which is a model and is labelled as one.

Vendor case-study figures are attributed customer claims. The Fourthwall and Crisp numbers [R30] [R31] are published by a vendor, including where that vendor is a competitor. They are specific, they name people, and they are consistent with the shape of the problem. They are not independent evidence and are not presented as such.

Our census is a routing table, not market share. It says how our engine treats 63 providers. It says nothing about how many domains sit at each provider, so no share-of-market claim can be derived from it. The manual-floor count is our number about our own system, not a competitor's weakness: 38 of 63 is the shape of the ecosystem and it applies to us.

Every count is a snapshot. Provider APIs, template catalogues and OAuth programmes change without notice. Each figure and table in this paper carries the date on which it was taken.

Capabilities described are capabilities that run. Where a surface is off by default, that is stated with the shipped behaviour beside it. Two surfaces in our own product — Secure (`/ssl`) and Power (`/power`) — along with Premium/Enterprise provisioning, SSO and SCIM, are not sellable today. They are not described anywhere in this paper as working, and where their absence changes runtime behaviour, §7 and §9 give the behaviour of the default deployment instead.

---

§2 Bring-your-own-domain is table stakes

2.1 What the customer is buying

Customers want `app.theircompany.com` , not `theircompany.yourproduct.io` . The reasons are functional at least as often as they are cosmetic.

Cookie and session scope is the first. Cookies are scoped by registrable domain, so a session established on a vendor-owned hostname is a separate cookie jar from the customer's own applications, and the browser treats it as a different site for the purposes of third-party cookie policy. Anything the customer wants to share across their own properties — session continuity, a single sign-out, a consent state — depends on the hostname sitting inside their registrable domain rather than yours.

Link equity is the second. Pages published on a vendor hostname accumulate whatever authority they earn against the vendor's domain. Moving them later means changing every URL that was ever shared, and the customer knows it, which is why the request usually arrives before the first page is published rather than after.

Security review is the third. Vendor questionnaires and internal review checklists tend to classify a vendor-owned hostname as third-party data handling, with the paperwork that implies. A customer-owned hostname keeps the asset inside the customer's own inventory. This one is not a technical argument and it does not need to be; it decides deals.

2.2 The bulk-sender rule made it non-optional

Preference became requirement on 2024-02-01. From that date, Google requires senders of more than 5,000 messages per day to personal Gmail accounts to publish SPF and DKIM on the sending domain and to have a DMARC policy, which may be `p=none` [R20].

The consequence for a product that sends mail on a customer's behalf is structural rather than promotional. SPF and DKIM must be published in the customer's zone, because the sending domain is the customer's. DMARC must exist there too. There is no version of the feature that skips DNS, and no volume threshold a growing customer stays under indefinitely. A product that never intended to be in the DNS business is now writing records on domains it does not own, and the failure modes in §3 arrive with the feature.

2.3 Who absorbs the cost

The cost of the DNS step lands on the vendor, in support.

Crisp reported going from "five or six DNS tickets per day" to "maybe one" after automating the flow, an 83% reduction [R31]. Fourthwall reported that nearly half its support tickets concerned DNS configuration, that each took 15 minutes to an hour to close, and that it had hired part-time contractors specifically to absorb the load [R30]. Both are vendor-published and both are read here as attributed customer claims (§1.3).

Two claims of that shape, from two companies, are enough to establish direction and not enough to establish magnitude for your own product. §5 therefore gives a parameterised model instead of a market figure: ticket rate, minutes per ticket, loaded cost per support hour, and the share of tickets that DNS accounts for, with the arithmetic exposed so the inputs can be replaced with your own. It is a model, and it is labelled as one throughout.

---

§3 Six mechanical failures

The six failures below account for the great majority of what a custom-domain onboarding queue actually contains. They are listed in the order a connection meets them.

3.1 The user cannot name their DNS provider

Registrar and DNS operator are separate roles, and customers routinely conflate them. The registrar holds the registration and submits the delegation to the registry, which publishes NS records in the parent zone; whoever operates those nameservers answers queries for the zone. A domain registered at GoDaddy but delegated to `ns1.digitalocean.com` is managed at DigitalOcean, and the GoDaddy DNS panel is inert — records typed there are served to nobody.

Onboarding asks the customer to recall a delegation decision that a contractor may have made three years ago. They answer with the name on the invoice, which is the registrar. Detection has to come from the zone rather than from the user: read the NS RRset, and separately attempt Domain Connect discovery, before the interface offers any path at all (§7.2).

3.2 Records get pasted wrong

Panels disagree about what a record's name field means. GoDaddy and Namecheap take a host relative to the zone origin and append the domain for you. Paste `_acme-challenge.example.com` into a panel rooted at `example.com` and you have created `_acme-challenge.example.com.example.com`, which resolves for nobody and looks correct in a screenshot. Trailing dots, the `@` convention for the apex, and quoting rules all vary by provider.

Length is the other trap. A TXT record's RDATA is one or more character-strings, each capped at 255 bytes (RFC 1035 §3.3.14 [R2]), so a value longer than that must be split into multiple character-strings — and providers disagree about whether they perform the split for you, reject the write, or truncate. DKIM keys and some verification tokens cross that boundary routinely.

The vendor observes none of this, because the write happens in a UI the vendor has never rendered. The only signal is that verification does not pass, which arrives after the customer has already concluded the product is broken.

3.3 Propagation is not an event

There is no propagation event to wait for. Authoritative servers hold the record the instant the write commits. Every recursive resolver learns it independently, whenever its own cached entry for that name expires.

The expensive case is a query issued before the record existed. The resolver caches the negative answer for the zone's negative TTL, which RFC 2308 §5 [R4] takes as the lower of the SOA MINIMUM field and the TTL of the SOA record itself. We measured 300 seconds for `cloudflare.com`, 1,800 for `digitalocean.com` and 3,600 for `godaddy.com` with `dig` on 2026-08-02 — a twelvefold spread across three registrars, none of it visible to the customer. Checking too early makes the wait longer, which inverts the instinct of both the customer and the support agent helping them. Meanwhile the customer refreshes, sees nothing, and opens a ticket.

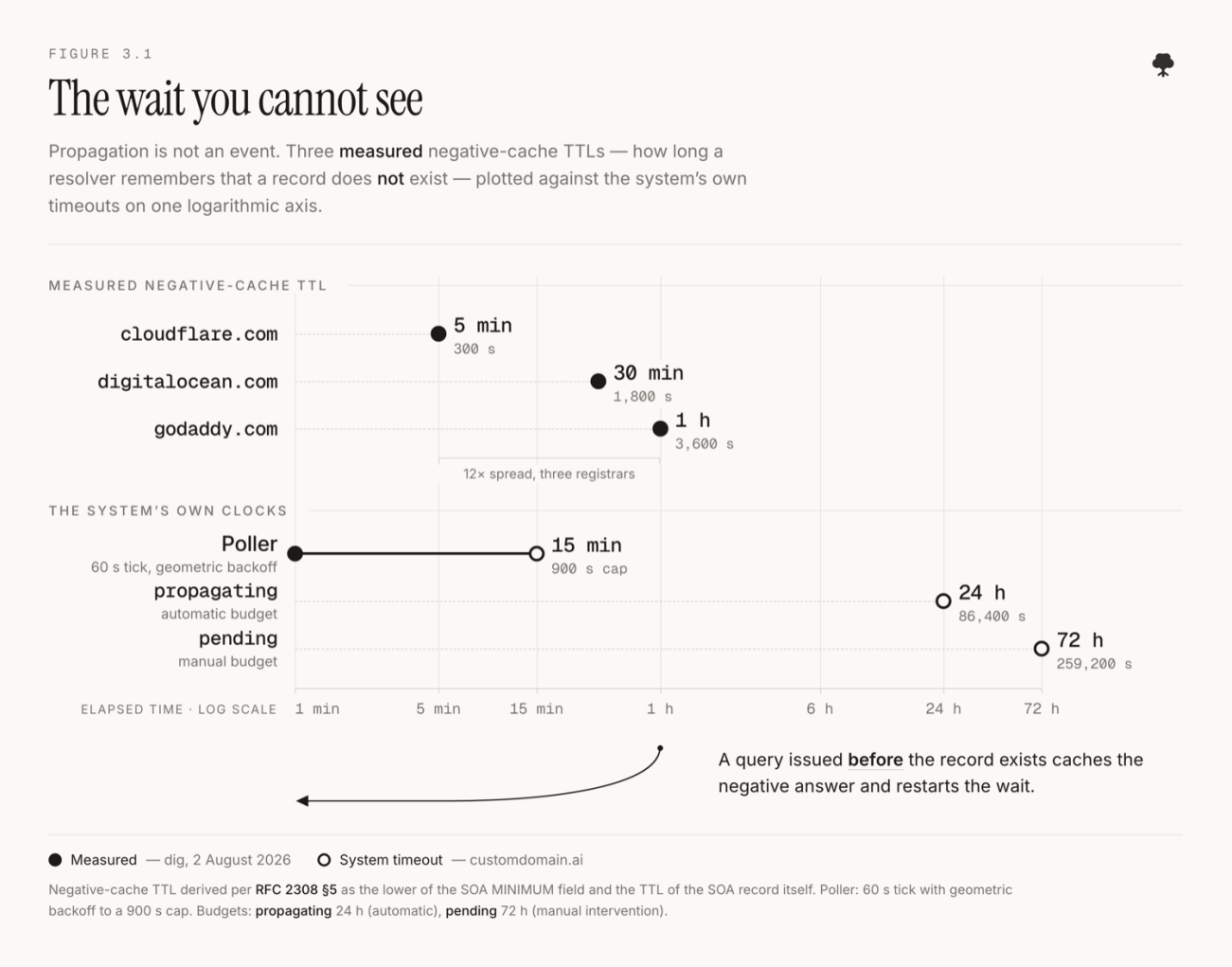


Figure 3.1 — The wait you cannot see. Three measured negative-cache TTLs (`dig`, 2026-08-02) plotted against the system's own timeouts on one logarithmic axis: a 60-second poll tick with geometric backoff to a 900-second cap, a 24-hour budget for `propagating`, and a 72-hour budget for `pending` on the manual path. A query issued before the record exists restarts the wait.

The design response is to treat the wait as a state with a budget rather than as a spinner: poll public DNS on a fixed tick with backoff, and let the state machine expire the connection on a stated deadline (§7.4). The budgets in Figure 3.1 are the shipped ones.

3.4 CAA blocks issuance silently

CAA checking is mandatory for every publicly trusted CA under the CA/Browser Forum Baseline Requirements. Before issuing, the CA queries CAA at the FQDN and, finding no RRset, climbs the parent labels until it finds one (RFC 8659 [R7]).

The common phrasing of this trap is too broad, and the imprecision costs debugging time. "A pre-existing CAA record blocks issuance" is wrong. What blocks issuance is a relevant CAA RRset that contains an `issue` or `issuewild` property naming some other CA, or the empty-value form that prohibits issuance altogether. An RRset holding only `iodef`, or holding an `issue` property that names the CA you are actually using, restricts nothing. So the pre-flight check worth building is not

"does a CAA record exist" but "does the relevant RRset contain an issuance property, and does that set omit my issuer" — and it should warn only in the second case, because warning on the first trains people to ignore it.

A leftover `example.com. CAA 0 issue "digicert.com"` from a previous vendor relationship forces Let's Encrypt to refuse. Every record the customer added is correct and resolving. The failure occurs inside the CA, minutes later, and reaches end users as a TLS handshake error with nothing in the product's own logs pointing at DNS.

There is a second-order version that is worse, because the customer holds no CAA record at all. Let's Encrypt notes that a DNS provider "does not need to specifically support CAA records; it only needs to reply with a NOERROR response for unknown query types" [R22]. Providers that answer SERVFAIL for unknown types break issuance while holding no CAA record for anyone to find.

3.5 The apex cannot take a CNAME

RFC 1034 §3.6.2 [R1] says no other data should be present at a name that holds a CNAME, and RFC 2181 §10.1 [R3] turns that into a hard prohibition. A zone apex already holds SOA and NS. So `example.com. CNAME edge.vendor.net.` is illegal, and any provider that appears to offer one is doing something else underneath.

Vendors want a CNAME at the apex precisely because their edge addresses change, and an A record pinned to today's address is a future outage. Providers therefore ship non-standard substitutes: Cloudflare and others flatten the CNAME at query time and answer with address records; a second group exposes an `ALIAS` pseudo-type; a third spells the same mechanism `ANAME`. Twelve provider mappings in our engine ship one of those spellings — five flattened, five ALIAS, two ANAME — and the spelling is a property of the provider's API rather than of DNS (Figure 3.2).

Two named providers ship none of them. Route 53 is the instructive case: it has no ALIAS record type at all, only an `<AliasTarget>` element inside its change API, and that element cannot point at an arbitrary external hostname. Azure DNS has a closed record-type switch with no apex spelling in it. For both, the correct product behaviour is to report the capability as false at check time rather than to accept the connection and fail at apply time.

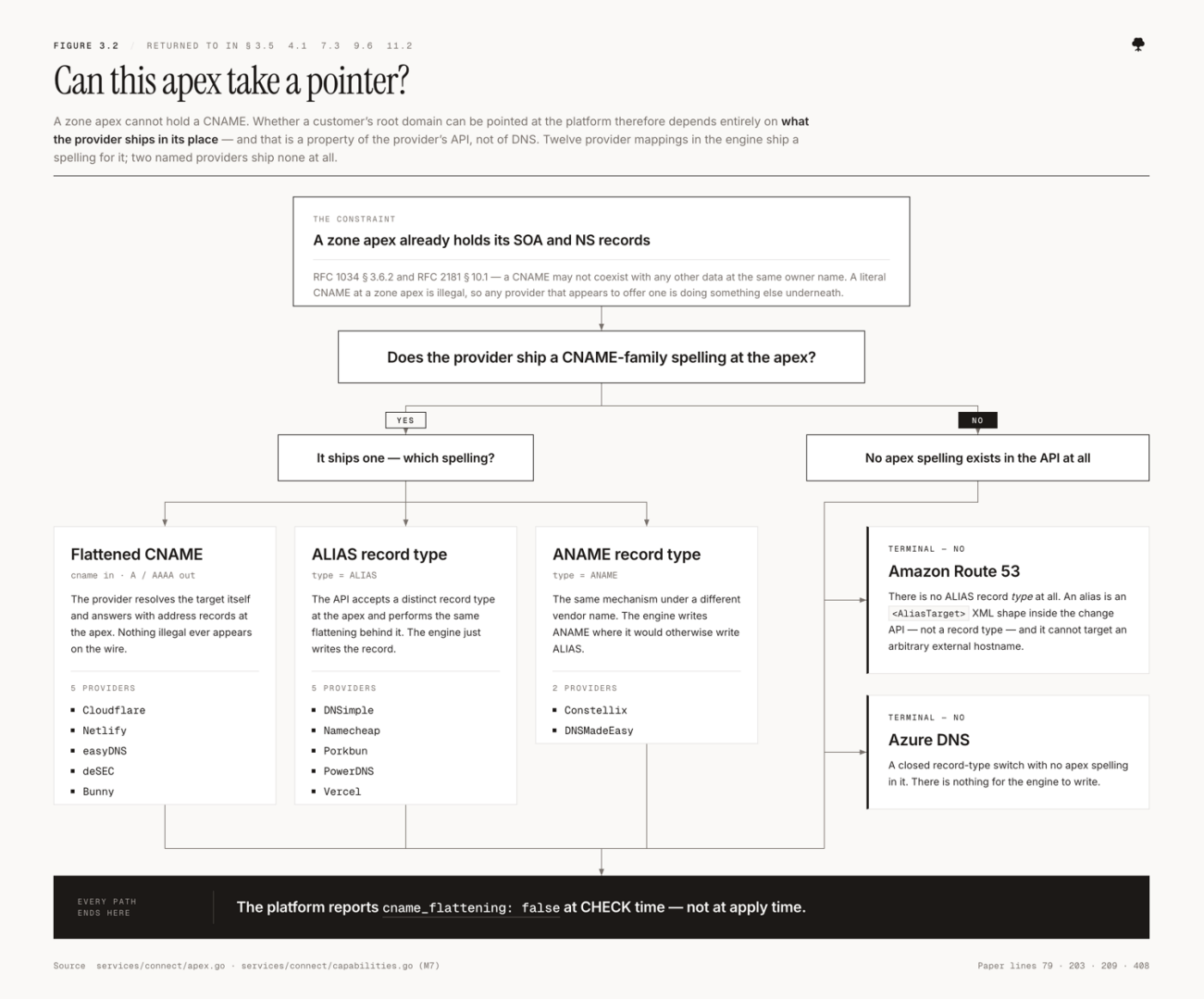


Figure 3.2 — Can this apex take a pointer? The decision runs on what the provider ships in place of a CNAME, not on what DNS permits. Twelve provider mappings ship a spelling; Route 53 and Azure DNS ship none, and every path ends at the same rule — report `cname_flattening: false` at check time, not at apply time.

Apex is returned to three more times, because it is the constraint that shapes the most code: as a vendor-evaluation question in §4.1, as a record-type design decision in §7.3, and as a failure mode in §9.6.

3.6 New records conflict with existing ones

The zone is not empty. Onboarding adds records to a live configuration that somebody else is depending on.

SPF is the sharp edge. A domain publishes exactly one SPF policy record (RFC 7208 §3.2 [R5]); a second one makes the check return permerror rather than adding a sender. Under a DMARC policy of `p=quarantine` or `p=reject` (RFC 7489 [R18]) that permerror costs delivery for mail that authenticated correctly the day before, and the customer's first symptom is their own invoices landing in spam. The correct operation is not a write but a merge: insert an `include:` mechanism

into the existing string, and stay inside the 10-lookup processing limit of RFC 7208 §4.6.4 [R5], which several vendors' `include:` chains are already consuming. A product that appends an SPF record instead of merging one has shipped a mail outage on a timer.

The same class shows up without mail. An existing A record at `www` cannot coexist with a new CNAME at `www`, by the same RFC 1034 §3.6.2 [R1] rule as the apex. Some panels reject that write with an error the customer does not understand. Others accept it, replace the A record, and quietly take down the customer's marketing site — which is a support ticket that does not mention DNS anywhere in its subject line.

3.7 Which of these is documentation, and which is engineering

Showing the correct records is a documentation problem. Getting them into a stranger's zone and confirming they arrived is an engineering problem, and it is the one that generates the tickets. The split is not evenly distributed across the six failures.

#	Failure	Where it surfaces	Visible to the vendor at the time?	Documentation or engineering
3.1	Cannot name the DNS provider	Onboarding form, before any write	No — the vendor sees an unanswered question	Engineering: detect from the NS RRset and Domain Connect discovery (§7.2)
3.2	Records pasted wrong	The provider's control panel	No — the UI is one the vendor never renders	Mixed: generated values are documentation, an automated write removes the step (§7.1)
3.3	Propagation is not an event	Resolver caches worldwide	Partly — only by polling public DNS	Engineering: a state machine with budgets, not a spinner (§7.4)
3.4	CAA blocks issuance	Inside the CA, minutes after a correct write	No — until the ACME order fails	Engineering: pre-flight the relevant RRset for a missing issuer (§9.4)
3.5	Apex cannot take a CNAME	Apply time, unless capability is checked first	Yes, if capability is modelled per provider	Engineering: an apex record type rewritten per provider at write time (§7.3)
3.6	Conflicts with existing records	The customer's mail flow or marketing site	No — and often not attributed to DNS at all	Engineering: read-modify-write with a merge, not an append (§7.3)

Table 3.1 — The six failures classified by whether better instructions could fix them. Only 3.2 has a documentation half, and only for the values themselves; the write and the confirmation stay on the vendor's side of the line in all six rows.

That is the argument for building or buying a mechanism rather than writing a better help-center article. §4 surveys what the mechanisms are and which party holds the pen in each.

§4 How the record gets written today

§3 described the mechanical failures. This section describes the mechanisms products actually ship against them, in the order a buyer meets them: three patterns, then the one open standard built to remove the human, then the provider-native alternative, then the certificate layer that is often mistaken for a solution to the DNS layer.

The organising question is not which approach is best. It is which party performs the write, because that determines both what an approach can fix and who can take it away. §4.5 makes that grouping explicit.

4.1 Three patterns

Three patterns cover how products ask a customer to write a DNS record.

A support document. The product displays a target hostname, links to a help-centre article, and the customer leaves to find their registrar. Whoever wrote the article now owns a matrix problem: it has to be correct for GoDaddy's panel, Squarespace's panel, Namecheap's two different panels, and whatever the customer's agency set up in 2019. Screenshots rot faster than the text around them.

A record table rendered inside the app. Type, host, value, TTL, a copy button, a verify poll. This is better, because the values are generated rather than transcribed, but the customer still has to log in somewhere else, find the right zone, and interpret what `host: @` means in a panel that calls it something else. Apex records are where the pattern breaks: a root domain cannot hold a CNAME (§3.5), so the correct instruction depends on whether the provider offers flattening, ALIAS, or ANAME, and the product usually does not know which. Figure 3.2 is that decision as a tree.

An integration vendor. A JavaScript widget that detects the provider and writes the record on the customer's behalf, over Domain Connect (§4.2) or a provider OAuth flow (§4.3), falling back to a rendered record table when it recognises neither.

A fourth thing is often filed alongside these and does not belong there. An edge/TLS service — Cloudflare for SaaS, Approximated, SaaS Custom Domains — changes

what the record points at

, collapsing several records into one pointer at a shared edge. It does not change who writes it. The customer still visits the panel unaided. §6.1 prices that job separately for exactly this reason.

4.2 Domain Connect: the standard, and its ceiling

Domain Connect is the direct attack on the problem. It is MIT-licensed, originated at GoDaddy in 2016, and is now on the IETF Standards Track as `draft-ietf-dconn-domainconnect-03` in the `dconn` working group, co-authored by Pawel Kowalik (DENIC), A. Blinn, Jody Kolker (GoDaddy) and Sami Kerola (Cloudflare), with a milestone targeting IESG submission in December 2026 [R14]. The shape is simple: a service provider publishes a template, and the DNS provider hosts the consent screen and applies the records. There are two rails — a signed browser redirect (sync) and an OAuth-based API flow (async).

Coverage is the first limit, and the public numbers do not agree.

The standard's own site lists **nine** DNS providers under "Implementation is Live": IONOS, Cloudflare, Domain Chief, Glauca Digital, GoDaddy, NameSilo, Plesk, Vercel and WordPress.com, retrieved 2026-08-02 [R13]. Kowalik's APNIC introduction to the standard puts the number at roughly twenty implementations covering 35% of the `.com` zone as of May 2024 [R38]. The two are not reconcilable from public sources. The site lists only implementations it has been told about and carries no last-updated date; the APNIC post gives a number with no list behind it. We use the site's list throughout this paper because it is the only one of the two with a checkable membership.

Template volume is a third number, and it measures something else again. We counted the official template repository on 2026-08-02 with the GitHub git-trees API and found 998 root-level templates from 596 distinct provider IDs [R15], well above the "over 300 templates from more than 120 providers" quoted in the APNIC post [R38]. Templates published is not the same as providers honouring them. A template in the repository is a service provider asking; a provider under "Implementation is Live" is a DNS operator answering.

The deeper limit is structural rather than numeric. Under the specification the DNS provider is the only party that writes records, runs discovery, and issues or revokes tokens. A service provider can request [R12]. Everything an aggregator sells on top of Domain Connect sits on a permission the counterparty grants and can withdraw. That has already been litigated once, and §6.4 walks the timeline; §9.3 covers what it means for revocation design.

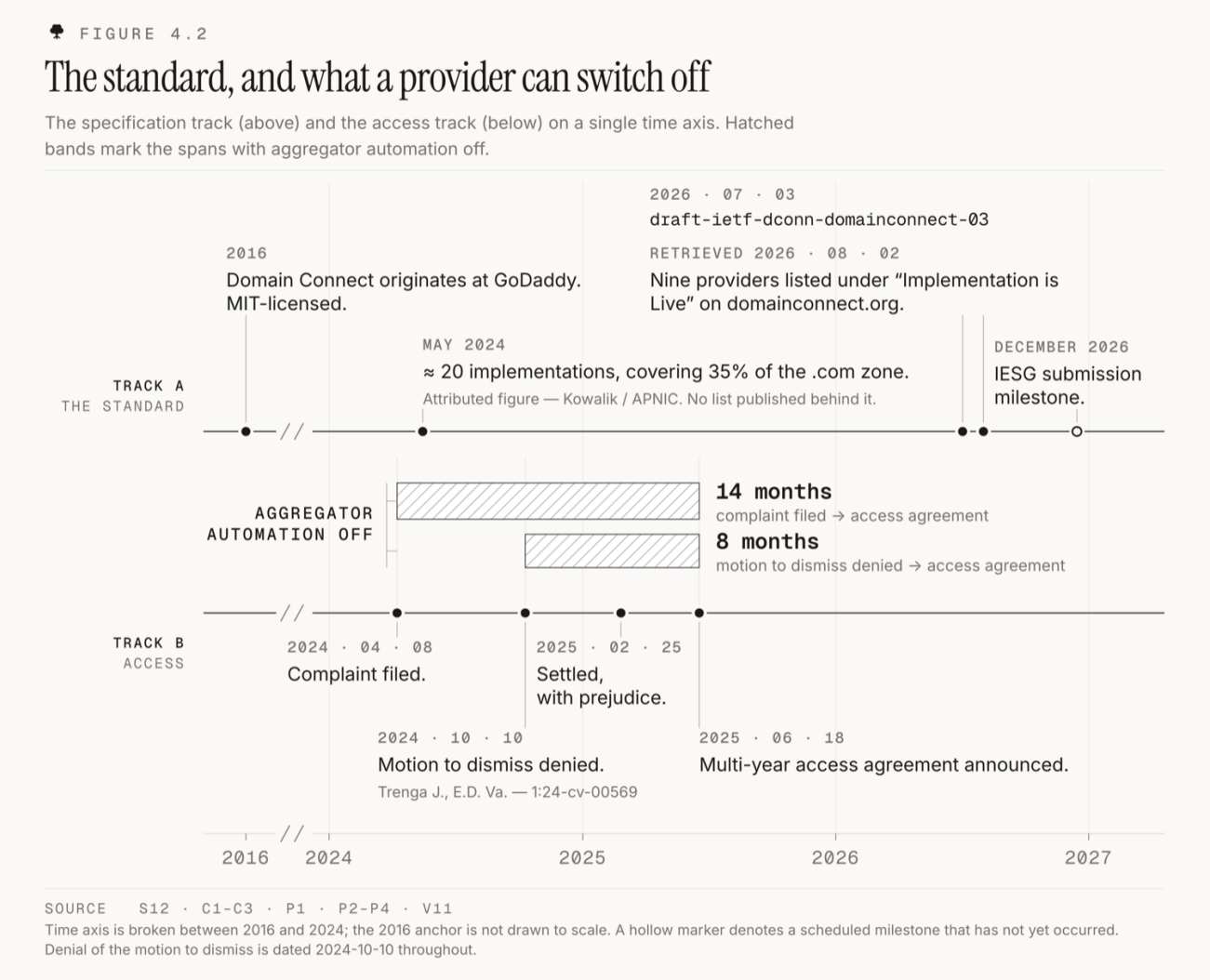


Figure 4.2 — The standard, and what a provider can switch off. The specification track and the access track on one time axis. The hatched bands are the spans with aggregator automation off: fourteen months from complaint to access agreement, eight months from the 2024-10-10 denial of the motion to dismiss to the same agreement. Dates and sources in §6.4.

4.3 Provider OAuth

The other native rail is a provider's own OAuth flow. Where one exists, the customer authorises once inside the widget and the platform writes the record with no value copied by hand. It is the cleanest of the automated paths, and it is rare.

Six providers expose a provider-hosted OAuth flow usable for third-party DNS writes: Cloudflare, DigitalOcean, DNSimple, Netlify, Vercel and WordPress.com. GoDaddy is not among them — its one-click path is Domain Connect async, not OAuth. Each of the six also requires a client id and secret registered with that provider before the rail runs at all, which is an owner action rather than a code path.

Most of the rest offer nothing better than a scoped API token the customer mints by hand, which is a DNS panel visit with extra steps. Namecheap is routed to manual records for a specific reason: its API replaces the entire zone in a single call, so a write cannot be made additive against records the platform did not author.

> **On the two 38s.** This paper uses the number 38 for two unrelated quantities, and neither is ever left bare. **38 of the 63 providers in the census are routed to the manual floor** (§1, Figure 1.1). Separately, **38 is the number of providers for which an adapter exists** in the connect engine. The six OAuth providers above are six of the 63 in the census and six of the 38 with an adapter. Where a count appears below, the denominator is stated.

4.4 ACME and CAA: what certificates do not solve

ACME (RFC 8555, March 2019) issues certificates [R6]. It does not configure the records that route traffic, and treating it as the custom-domain solution is a common category error. Its HTTP-01 and TLS-ALPN-01 challenges

assume

the customer's DNS already points at you. Its DNS-01 challenge drops that assumption and instead requires a TXT record on the customer's domain, which is the DNS-writing problem again under a different name.

The constraints that bite at volume are the rate limits rather than the protocol. Let's Encrypt allows 300 new orders per account per 3 hours, 5 certificates per identical identifier set per 7 days, 5 authorisation failures per account per hostname per hour, 50 certificates per registered domain per 7 days, and 100 identifiers per certificate [R21]. For a custom-domain platform the 300-orders-per-3-hours account limit binds first, because every new customer domain is a separate registered domain but shares one ACME account. The authorisation-failure limit is scoped per account per hostname, so it does not aggregate across the fleet: it turns one misconfigured customer domain into a one-hour retry lockout on that domain alone. §7 describes what a control plane has to do to stay inside those numbers; §9 describes what happens when it does not.

CAA (RFC 8659) is the quiet one [R7]. CAA checking is mandatory for every publicly trusted CA under the CA/Browser Forum Baseline Requirements, so a customer's pre-existing CAA record that omits your issuer blocks a certificate for a hostname whose DNS is otherwise correct. The trap underneath the trap: a DNS provider does not need to support the CAA record type at all — it only needs to return NOERROR for unknown query types [R22]. Providers that answer SERVFAIL for unknown types break issuance while holding no CAA record whatsoever. The failure surfaces as a certificate that never appears, with no cause visible in the product's own logs and nothing wrong in the customer's zone.

4.5 Who holds the pen

Grouping the approaches by the party that performs the write puts the trade-offs in one place. Table 4.1 is the compact form.

Approach	Who writes the DNS record	Solves	Leaves open
Support doc	The customer, in their panel	Nothing; documents the work	Full support load, apex ambiguity
In-app record table	The customer, from generated values	Transcription errors	Panel navigation, provider variance
Domain Connect	The DNS provider	One-click on supported providers	Coverage; the provider can revoke
Provider OAuth	The platform, with a user-scoped token	Clean writes where offered	6 of the 63 census providers, after per-provider app registration
Edge/TLS service	The customer, unaided, pointing one record at your edge	TLS, routing, certificate lifecycle	The DNS entry itself
Config aggregator	The aggregator, via Domain Connect or OAuth	The onboarding UX	Price floor, standards dependency

Table 4.1 — Custom-domain approaches classified by which party writes the DNS record, since that determines what each one can and cannot fix.

Figure 4.1 expands the same grouping to eight approaches and adds the two columns that decide the security posture: whether the provider can revoke the arrangement unilaterally, and whether the platform is left holding a credential afterwards. Those columns are drawn from §9.1 and §9.3.

FIGURE 4.1

Who holds the pen

Eight ways a DNS record gets written on a domain the platform does not own, grouped by the party that performs the write — which decides what an approach fixes and **who can switch it off**.

APPROACH		WHO PERFORMS THE WRITE	CUSTOMER VISITS A DNS PANEL?	APEX-SAFE?	PROVIDER CAN REVOKE IT UNILATERALLY?	PLATFORM HOLDS A CREDENTIAL AFTERWARDS?	WHAT IT LEAVES OPEN
THE CUSTOMER WRITES	**Support document** a static instructions page	The customer, in their own panel	● Yes — unaided	○ No — apex ambiguity is left behind	○ No — nothing to revoke	○ None	Full support load; apex ambiguity
	In-app record table generated values, copied by hand	The customer, from generated values	● Yes — with the values supplied	◐ Only with flattening, ALIAS or ANAME support	○ No — nothing to revoke	○ None	Panel navigation; provider variance
	Edge / TLS service one record pointed at your edge	The customer, unaided, pointing one record	● Yes — one record, still their panel	◐ Dedicated IP (A record) yes; CNAME edge no	○ No — nothing to revoke	○ None — no DNS credential in the flow	The DNS entry itself
THE DNS PROVIDER WRITES	**Domain Connect sync** signed browser redirect	The DNS provider, after a signed redirect	○ No — a consent screen, not the editor	◐ Only within the template the provider honors	● Yes — it can decline every apply	○ None — it only signs the request	Coverage — on only for verified providers
	Domain Connect async OAuth grant, applied server-side	The DNS provider, from a stored grant	○ No — one consent, then server-side	◐ Only within the template the provider honors	● Yes — decline the apply or revoke the grant	● Yes — the only stored grant, encrypted at rest	No partner credential ships; use-once without a key
THE PLATFORM WRITES	**Provider OAuth** user authorizes inside the widget	The platform, with a user-scoped token	○ No — authorized inside the widget	● APEXCNAME rewritten to native spelling	● Yes — it registers the client id	○ No — one SetRecords call, not persisted	6 of 63 providers, after app registration
	BYO scoped API token customer mints it at the provider	The platform, with a token the customer minted	● Yes — minting the token is a panel visit	● APEXCNAME rewritten to native spelling	◐ Not the token, but its terms govern the API	○ No — passed per apply call, not persisted	Account-wide width it cannot narrow; 16 of 63
AN AGGREGATOR WRITES	**Configuration aggregator** resold Domain Connect or OAuth	The aggregator, via Domain Connect or OAuth	◐ Not on covered providers; the tail copies	◐ Inherits the rail it runs on	● Yes — GoDaddy's terms cut access for eight months	◐ Varies — ask where the token lives afterwards	Price floor; standards dependency

● Yes ◐ Partly, or conditional ○ No

Each mark answers the question in its own column, not a verdict on the approach: a filled mark is good news under *apex-safe* and bad news under *revoke*. The three customer-written rows are the floor — nobody else can switch them off.

Rebuilt and expanded from Table 4.1. The revocation and credential columns are drawn from §9.1 *What is actually delegated* and §9.3 *Revocation, and who holds the lever*. Provider counts are the 63-provider census.

Figure 4.1 — Who holds the pen. Eight ways a DNS record gets written on a domain the platform does not own, grouped by the party performing the write. A filled mark is good news under "apex-safe" and bad news under "provider can revoke". The three customer-written rows at the top are the floor: nobody else can switch them off.

Two readings fall out of the grouping. First, every approach that removes the panel visit also introduces a party who can decline — which is why §7 treats manual not as an error screen but as a rail with the same computed records, the same state machine and the same verification. Second, the approaches differ in what they leave behind: a signed redirect leaves the platform holding nothing, a stored Domain Connect grant leaves it holding a credential bounded by a template, and a customer-supplied API token leaves it holding account-wide authority it cannot narrow.

---

§5 What the status quo costs

The honest answer is that nobody has measured it independently. What follows is therefore in two parts, kept apart on purpose: the published evidence, graded (§5.1), and models the reader parameterises from their own systems (§5.2 to §5.4). We supply no benchmark, because we have no benchmark to supply.

5.1 The public evidence, graded

We looked for independent, peer-reviewed measurement of custom-domain onboarding drop-off or support cost and found none. Everything in circulation traces back to vendor case studies or to a single uncited figure. Table 5.1 grades each item rather than reproducing it.

#	Claim	Source	Grade	How to use it
1	Nearly half of Fourthwall's support tickets were DNS configuration; 15 minutes to an hour of back-and-forth each; part-time contractors hired to absorb the load	Entri case study [R30]	Vendor-published; attributed customer claim; no methodology stated	Directional. The denominator is all support tickets, not new onboards, so it is not a value for t in §5.2
2	Fourthwall reduction of "as much as 97%" after automating	Entri case study [R30]	Vendor-published outcome claim, hedged in the source	No term in the model corresponds to it. Not an input
3	Crisp went from five or six DNS tickets a day to about one, an 83% cut	Entri case study [R31]; CEO Baptiste Jamin quoted directly	Vendor-published; named speaker; the arithmetic is consistent with the quoted counts	Same as row 2: an outcome, not an input
4	Microsoft 365 domain setup: a six-step process needing 7 to 15 DNS entries, 16 help sites and 40 minutes of training, with 50% of users abandoning	Kowalik, APNIC [R38]	**No traceable primary source.** The post cites CENTR for its renewal-rate figures and gives no source for these	Do not use as a benchmark. If cited at all, cite it as an attributed estimate by a standards author
5	Nine DNS providers under "Implementation is Live"	domainconnect.org [R13]	Checkable membership list; self-reported and undated	The only public coverage figure with names behind it
6	Roughly twenty Domain Connect implementations, covering 35% of the `.com` zone as of May 2024	Kowalik, APNIC [R38]	Attributed; no list published behind the number	Not reconcilable with row 5 from public sources
7	998 root-level templates from 596 distinct provider IDs, 2026-08-02	Our count via the GitHub git-trees API, untruncated [R15]	First-party; method stated; reproducible	Measures templates published, not providers honouring them
8	63-provider census: 38 routed to manual, 16 to a scoped API token, 6 to native OAuth, 3 to Domain Connect	First-party (§1)	First-party routing table; the gap between routed, working and one-click is published with it	Read against Figure 1.2. Routed is not working

Table 5.1 — The public evidence on what DNS onboarding costs, graded by what kind of evidence each item is. Rows 1 to 3 are the only quantified cost claims in the category and all three are published by the same vendor. Row 4 is the most-repeated number in the category and has no traceable primary source.

Rows 1 to 3 are consistent with each other and with the shape of the problem. They are also not independent evidence, they share a publisher, and two of the three are outcome claims measured by the party selling the outcome. They belong in a paper as attributed customer claims and nowhere else.

Row 4 deserves its own warning because it circulates without one. The 50%-abandonment figure is quoted in decks and blog posts as though it were research. It appears in an APNIC blog post by a DENIC engineer who is also a co-author of the standard, with no source given for those specific numbers. Repeating it as research is how an unsourced figure becomes a fact.

5.2 A support-load model you parameterise

Because no defensible published number exists, this paper supplies a model instead of a figure. Four inputs, all of which the reader already has:

- **D** — new custom-domain onboards per month. Read it off the provisioning log: domains connected across one full month.
- **t** — the share of those raising a DNS ticket. Read it off the helpdesk: tickets tagged DNS or domain, from that same cohort.
- **h** — handle minutes per ticket. Median first touch to close, DNS tag only.
- **c** — loaded support cost per hour. Salary plus benefits plus overhead, divided by productive hours.

The arithmetic:

```
monthly cost  =  D × t × (h ÷ 60) × c
annual cost   =  monthly cost × 12
```

That is the whole model. It computes nothing on its own and ships with no default values, because a default value here is an invented statistic. Four blanks, one meeting with whoever owns the helpdesk, no vendor required.

FIGURE 5.1

What a DNS onboarding ticket costs you

No one has measured this independently, so this paper supplies no number. It supplies the **model** — four blanks you fill from your own helpdesk — and, held separately from it, the only two published anchors we can attribute.

THE MODEL

Every input is yours. Nothing here is computed for you.

	INPUT	WHERE TO READ IT OFF	YOUR NUMBER	
D	**New custom-domain onboards**	**Provisioning log** — domains connected across one full month		/ month
t	**Share of those raising a DNS ticket**	**Helpdesk** — tickets tagged DNS or domain, from that same cohort		%
h	**Handle minutes per ticket**	**Helpdesk** — median first touch to close, DNS tag only		min
c	**Loaded support cost per hour**	**Payroll** — salary, benefits and overhead ÷ productive hours		$ / hr

Monthly cost $D \times t \times (h \div 60) \times c$ = $ _____ per month

Multiply by **12** for the annual figure. Four blanks, one meeting, no vendor required.

REFERENCE MARKS – OUTSIDE THE MODEL

Drawn in grey. Never multiplied into the total above.

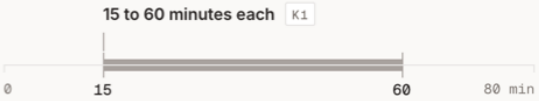
Two published figures, drawn as marks on their own axes so you can see where they sit. **Neither is a value for D, t, h or c in your business.** Both come from vendor-published pages and each is graded in the key below. Hollow marks are approximations in the source; solid marks are figures as published.

Adjacent to t — share of support tickets that are DNS-related

Fourthwall. Its denominator is *all* support tickets, not new onboards — so the mark sits near t, not on it.

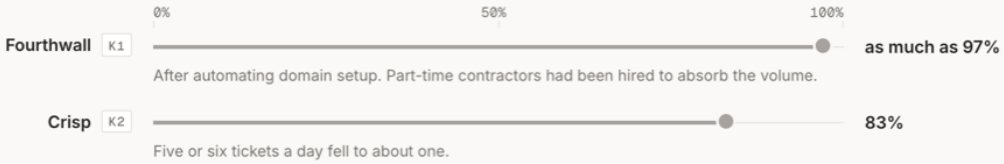
On the h axis — handle minutes per ticket

Fourthwall. Published as a range, so it is drawn as a range.



Not in the model — reported ticket reduction after automating

Outcome claims. There is no term for them in D × t × (h ÷ 60) × c.



K1 **Fourthwall** — vendor-published; attributed customer claim. Nearly half of support tickets DNS-related, 15 to 60 minutes each, part-time contractors hired, a reduction of as much as 97% after automating.

K2 **Crisp** — vendor-published; CEO quoted directly. Five or six tickets per day down to about one, an 83% cut.

Neither claim is independently verified, and neither is a benchmark. Use your own four numbers.

customdomain.ai

Grey marks are reference only, graded K1 / K2 as keyed.
This figure computes nothing. The model ships with no default values.

Figure 5.1 — What a DNS onboarding ticket costs you. The four inputs and where to read each one off, with the two published anchors drawn separately in grey — adjacent to t and on the h axis, never multiplied into the total. The reduction claims are drawn last, because no term in D × t × (h ÷ 60) × c corresponds to them.

Two cautions about the model's edges. First, the published anchors in Figure 5.1 sit near the model rather than in it: Fourthwall's "nearly half" is a share of all support tickets, not of new onboards, so it is adjacent to t and not a value for it. Second, the model measures the visible half. It counts a ticket, not the engineer pulled into an escalation, not the certificate that never issued because of a CAA record (§4.4), and not the customer who never opened a ticket at all because they stopped trying. §5.3 is about that last group.

5.3 The onboarding-conversion model, and why we give instrumentation instead

We would like to tell you what share of customers abandon a custom-domain setup. We cannot, and neither can anyone else who has published on it: the only figure in circulation is row 4 of Table 5.1, and it has no source. Publishing a conversion number we cannot trace would put a fifth unsourced statistic into the world. So this section gives the instrumentation instead, and the two ratios worth computing from it.

Event	Recorded when	What it lets you compute
`domain.submitted`	The customer enters a domain	The funnel denominator
`provider.detected`	The detection cascade returns a provider or classifies manual	The rail mix of your population, not the market's
`rail.offered`	The interface renders a path	How often detection produced something better than manual
`write.attempted`	The customer starts the apply, or is shown the records to copy	Drop-off at the consent step, which is where automated rails lose people
`write.succeeded`	The provider or adapter confirms the write	Per-rail write success, segmented by provider
`connection.propagating`	First successful public-DNS read of the expected rrset	Write-to-visible latency, per provider
`connection.live`	Verification passes	Rail-conditional completion — the ratio that matters
`connection.failed`	The pending or propagating budget expires	Abandonment, separated from cache latency

Table 5.2 — The eight events a custom-domain funnel needs before any conversion number about it is meaningful. Segment every one by detected provider and by rail.

Two ratios come out of that, and both are only interpretable inside one population:

Rail-conditional completion. `connection.live ÷ domain.submitted`, segmented by the rail detection chose. This is the number a vendor's coverage claim should move. Compare your manual segment against your automated segments; do not compare either against somebody else's published figure, because their provider mix is not yours.

Time to live, by provider. The distribution — not the mean — of submitted to live. The tail is the interesting part, and the tail is set by the negative-TTL behaviour described in §3.3 rather than by anything the product does.

One measurement trap is worth naming because it manufactures abandonment that is not there. A customer who writes the record, sees nothing, closes the tab and returns ninety minutes later has not abandoned; they have waited out a negative cache entry (Figure 3.1). A funnel keyed on sessions will score that as a loss. Key it on the connection.

5.4 Cost per connected domain

The three cost terms belong in one number, and only one of them is published by anybody:

```
cost per connected domain  =  (S + E + C) ÷ N

  S  support cost for the period          from §5.2 — yours, unpublishable by us
  E  edge / TLS cost for the period       from §6.1 — published, $0.10-$0.29 per hostname
per month
  C  configuration-tier cost, if bought   from §6.2 — published, but not per-unit
  N  connections that reached live        from Table 5.2 — yours
```

Three properties of that expression are worth stating plainly.

E is the only term with a market price.

Terminating TLS for an arbitrary hostname is metered, commoditised, and published to the cent by three vendors (§6.1). It is also, for most teams, the smallest of the three.

S is usually the largest and is nowhere on a price list.

That asymmetry is the whole commercial argument in this category, and it is why the vendor evidence in Table 5.1 is about ticket counts rather than about prices.

N must be connections that reached live.

If the denominator is domains submitted, the number flatters a product that fails quietly. A vendor quote priced per domain and an internal cost priced per submission are not comparable, and the difference is exactly the population §5.3 tells you to instrument.

---

§6 What the market charges

Two different jobs are priced in this category, and conflating them is how buyers end up comparing a $0.10 line item against a $249 invoice. Terminating TLS for an arbitrary hostname is one job. Getting a non-technical customer's record written correctly, on the first attempt, in a panel neither party controls, is the other. Figure 6.1 draws both.

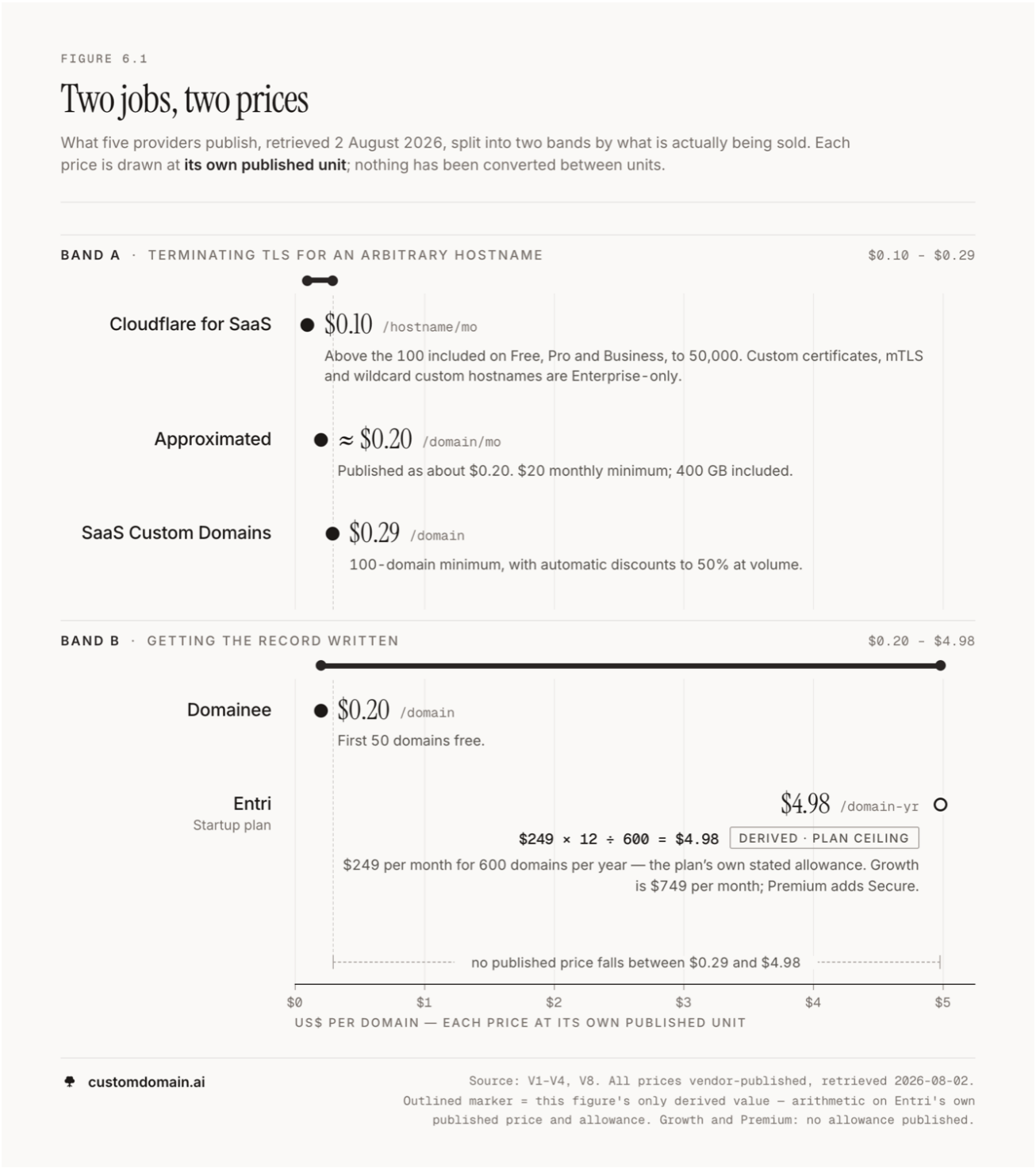


Figure 6.1 — Two jobs, two prices. Five published prices retrieved 2026-08-02, split into the band that terminates TLS and the band that gets the record written. Each price is drawn at its own published unit; nothing has been converted between units. The single outlined marker is the figure's only derived value.

6.1 The edge and TLS floor: $0.10 to $0.29 per hostname per month

Three vendors publish a per-hostname price for the same job — terminate TLS for an arbitrary customer hostname and reverse-proxy it to an origin.

Cloudflare for SaaS includes 100 custom hostnames on Free, Pro and Business, then charges **$0.10 per hostname per month** up to 50,000. Custom certificates, mTLS and wildcard custom hostnames are reserved for Enterprise [R23].

Approximated charges **about $0.20 per domain per month** with a $20 monthly minimum and 400 GB of bandwidth included, and describes its setup as one A record pointed at your cluster's dedicated IP [R24].

SaaS Custom Domains prices at **$0.29 per domain** with a 100-domain minimum and automatic discounts up to 50% as volume scales [R25].

All three prices are vendor-published and were retrieved on 2026-08-02. The spread across them is nineteen cents, which is what a commoditised, metered job looks like when three vendors price it independently. None of the three touches the DNS configuration step: the customer still visits the panel unaided, and the one record they are asked to write is still subject to every apex constraint in §3.5.

6.2 The configuration-UX tier has no settled price

The band above prices bundles rather than units, and the two published points in it are more than an order of magnitude apart per domain.

Entri prices on plans: Startup at **249permonth * *forastatedallowanceof600domainsperyear, Growthat * *749 per month**, with everything above Startup demo-gated [R26]. Reduced to the same unit as the band below — and this is arithmetic on the vendor's own published price and its own stated allowance, not a rate Entri publishes — Startup is 249 × 12 ÷ 600 = **4.98 per domain-year**. Figure 6.1 draws it as an outlined marker for that reason. Growth and Premium publish no domain allowance, so no equivalent figure can be derived for them.

Domainee, a newer entrant, competes directly on the price axis: **50 domains free, then $0.20 per domain**, while conceding in its own comparison post that Entri's widget is one of the best onboarding components in the market [R32].

Among the five published prices in Figure 6.1, nothing falls between $0.29 and $4.98.

Three further public data points sit around the incumbent, and each is worth exactly what its source is worth. The `entrijs` npm package recorded 236,511 downloads in the week ending 2026-08-01, from the registry's own downloads API [R33] — a distribution signal, not a revenue one. IONOS took a minority stake of under 20% in August 2025 [R34], booked in its FY2025 consolidated statements as €5,028k of additions at FVOCI, Level 3 [R35], which is the only audited financial number in Entri's public record. And Entri is the standard's largest template publisher, with 77 templates in the Domain-Connect repository against GoDaddy's 27 across its `godaddy` and `secureserver` provider IDs [R15].

Coverage, the thing the premium is actually charged for, is published inconsistently by the incumbent itself: the Connect product page markets 50+ DNS providers and "75% of the market", the developer overview says 60+ providers with direct API login, and the developer provider list enumerates 64 named providers, all retrieved 2026-08-02 [R27] [R28] [R29]. Three numbers, one product, one day.

6.3 Why that spread is a coverage claim, not a cost measurement

The two bands in Figure 6.1 differ in kind, not only in magnitude.

In Band A, the unit of sale is a hostname and the underlying resource is metered: a certificate, a TLS termination, some bandwidth. Three vendors converged within nineteen cents because they are pricing something measurable that costs them something measurable.

In Band B, the unit of sale is a successful DNS write, and nothing about it is metered. The marginal cost of one more Domain Connect apply, or one more OAuth `SetRecords` call, is not what separates $0 for the first 50 domains from $4.98 per domain-year. Both vendors ship a widget over the same standard [R14] and the same provider APIs. What separates them is the claim about how many providers can be reached one-click, and behind that claim, which bilateral agreements exist.

So the spread should be read as a coverage assertion, and priced the way you would price any assertion: by asking for it broken out. §11.1 gives the question in its buyer form — coverage by

mode

, as a file rather than a slide, because "supported" spans a provider writing the record itself, a user authorising through the provider's OAuth, the platform writing with a customer-supplied token, and the customer copying and pasting. Those are four different products sold under one number.

This applies to our own numbers, and §1 states the gap rather than burying it: the census routes 63 providers, 38 of them to the manual floor; of the 25 routed to an automated rail, 23 work today; and at most 7 are one-click, and only after per-provider app registration. Routed, working and one-click are three different numbers (Figure 1.2). A coverage claim that does not distinguish them is not a coverage claim.

6.4 What a DNS provider can switch off

A price built on one-click coverage is a price built on permissions, and permissions get revised. Under the Domain Connect specification the DNS provider is the only party that writes records, runs discovery, and issues or revokes tokens; a service provider can only ask [R12]. Any DNS provider can sever any aggregator unilaterally, by declining applies or revoking grants.

That is not a hypothetical. GoDaddy revised its terms of use in a way that cut off aggregator DNS access, and the resulting case is the only public record of what the dependency is worth in time. In

Entri LLC v. GoDaddy.com LLC

, No. 1:24-cv-00569 (E.D. Va.) [R37]:

- **2024-04-08** — complaint filed.
- **2024-10-10** — Judge Anthony J. Trenga denies GoDaddy's motion to dismiss, on a negative-tying theory under Sherman Act §1. The opinion is dated and signed 2024-10-10; the quoted language that the alleged tie "cuts off all aggregator services … from competing" reaches the public through reporting the following day, 2024-10-11 [R39], with further contemporaneous coverage on 2024-10-21 [R42]. **The ruling date is 2024-10-10; 2024-10-11 is a reporting date.**
- **2025-02-25** — the parties settle, with prejudice [R40].
- **2025-06-18** — a multi-year agreement restoring access is announced [R41].

Figure 4.2 draws the two spans that matter to a buyer: fourteen months from complaint to access agreement, and eight months from the denial of the motion to dismiss to the same agreement, with the aggregator's automation off in between. Access came back through a private bilateral agreement between two companies, not through the standard and not through the court.

Two consequences follow, one commercial and one architectural.

Commercially, the premium in Band B is rented. It is contingent on agreements the buyer is not party to and cannot inspect, with a counterparty whose own DNS business competes for the same customer. That does not make the premium unreasonable; it makes it a different kind of purchase from the metered one in Band A, and it should be diligenced differently (§11.1).

Architecturally, the conclusion is blunter: every automated rail has to degrade to something nobody else can switch off. That floor is manual copy-paste records — the same computed rrset, the same state machine, the same verification, and a distinct terminal status because a manual success is a success. §7 describes how that floor is built, and §9.3 covers what remains revocable above it.

§7 A reference architecture for programmatic domain connection

A domain connection is three operations wearing one button: a write to a zone the platform does not own, a wait for that write to become globally visible, and a proof that the result is what was asked for. The common implementation collapses all three into a static instructions page that lists the records and trusts the user to copy them. Everything in §3 is a consequence of that collapse — the transcription errors, the invisible propagation window, the apex that will not take a CNAME, the CAA record nobody remembered.

The architecture below separates the three operations, gives each one an owner, and treats the failure of any one as a state rather than an error page. It is described concretely because a vendor-neutral version would be too vague to argue with. Where the shipped system is narrower than the design, §7.5 says so with numbers.

7.1 Two planes, one system of record

Split the system before writing any DNS code.

A **control plane** holds tenant state, computes records, drives the write, and watches propagation. A **data plane** — the edge — terminates TLS and reverse-proxies live custom-domain traffic to each tenant's origin. In the implementation described here they are separate services, and only one of them is stateful.

The control plane is the single system of record. It owns the Postgres instance that holds connections, records, baselines, grants and tenant configuration. The edge holds no persistent state beyond its certificate cache, and the API gateway is never in the proxied request path. At request time the edge calls back to exactly one control-plane endpoint, `POST /internal/ask`, and asks a question it cannot answer locally: is this host approved, and where does its traffic go.

The reason for the split is blast radius. A bad deploy of a REST API should degrade onboarding — new connections stop advancing, the dashboard errors — and should not drop every customer's production traffic. Keeping the edge free of its own durable state also means an edge instance is disposable: it can be replaced, scaled out, or moved without a migration, because the only thing it would lose is a cache it knows how to refill.

The cost of one system of record is that it is one system of record. Everything downstream of `POST /internal/ask` inherits the control plane's availability. §7.7 describes what the edge does when that call fails, and the answer is deliberately unhelpful to attackers and to customers alike: deny.

7.2 Detection runs before the interface offers a path

The interface cannot offer a path until the system knows who operates the zone. Detection runs first, and it is a cascade with two probes and a floor.

Domain Connect discovery goes first, per the specification [R12]. A TXT lookup of `_domainconnect.<domain>` yields a host; a `GET /v2/{domain}/settings` against that host returns `providerId`, `urlSyncUX`, `urlAsyncUX` and `urlAPI`. If the TXT record is absent, the settings document is malformed, or the host does not answer, the probe is over.

Nameserver-pattern matching runs second. The NS RRset for the domain is compared against the patterns each registered adapter declares. This is the probe that answers the question §3 opens with — that registrar and DNS operator are separate roles, and the user usually names the wrong one. A domain registered at GoDaddy but delegated to `ns1.digitalocean.com` matches the DigitalOcean adapter, and the interface offers DigitalOcean, whatever the user believes.

A domain matching neither classifies as manual. That is a real outcome, not a fallback taken in shame, and §7.4 treats it as a first-class rail.

Discovery is best-effort on purpose. Any failure returns `Supported: false` rather than raising, because the caller's next move is identical either way: drop a rail and continue with the ones that remain. An exception thrown by a discovery probe would convert a missing optional capability into a failed connection.

Detection answers who runs the zone. A separate **capability layer** answers what that provider will accept: apex support, subdomain support, wildcards, CNAME flattening at the apex, SPF-override tolerance, CAA support, TXT values over 255 bytes, and a conflict-tolerance threshold that predicts when an automated write will silently degrade to manual.

Route 53 is the instructive case, and it is the case in the right branch of Figure 3.2. Route 53 has no ALIAS record type. It has an `<AliasTarget>` element in its API, which cannot point at an arbitrary external hostname. Its flattening capability flag is therefore false, and the connection check says so before the user commits to an apex connection. Reporting an incapability at check time is better than promising an apex and refusing it at apply time, when the user has already told their team the domain is going live.

7.3 The server computes the records

The browser never derives a record. The control plane emits the RRset and the widget renders it. One authority decides what "correct" means, and every rail — including manual — consumes the same answer.

This is what makes the manual rail verifiable. If the widget computed records for display and the server computed different records for the automated write, the manual path would be checked against an instruction set nobody validated.

Apex connections use a provider-agnostic `APEXCNAME` record type. It is rewritten immediately before the write into whichever native spelling the target provider uses: a flattened CNAME, an ALIAS, or an ANAME. Twelve providers currently have that mapping. The persisted and verified record stays `APEXCNAME`, so verification never has to know which dialect the write used — which matters, because the dialect can change under you when a provider ships a new record type.

One provider-specific rule is worth naming because it is not obvious. Cloudflare records written by the platform are forced to `proxied: false`. The platform's own edge already terminates TLS for the hostname; a second proxy in front of it breaks routing rather than adding a layer of protection.

7.4 Three rails over a manual floor

Three automated write rails sit above a manual floor.

Domain Connect sync is a signed browser redirect. The control plane signs an apply URL, the user lands on their own DNS provider, the provider authenticates them and performs the write. The platform holds no credential at any point.

Domain Connect async exchanges an OAuth code for a grant and applies the template server-side. When a grant-encryption key is configured the grant is stored encrypted, so the template can be re-applied or reverted later without dragging the user back through consent. Without a key, the apply is use-once.

Direct provider write is the platform writing records itself, using either a scoped API token the tenant supplies or a provider OAuth credential the end user authorizes inside the widget. Every adapter on this rail implements the same libdns-shaped contract [R19]: `GetRecords`, `SetRecords` as an idempotent create-or-update against an exact RRset, and `DeleteRecords`.

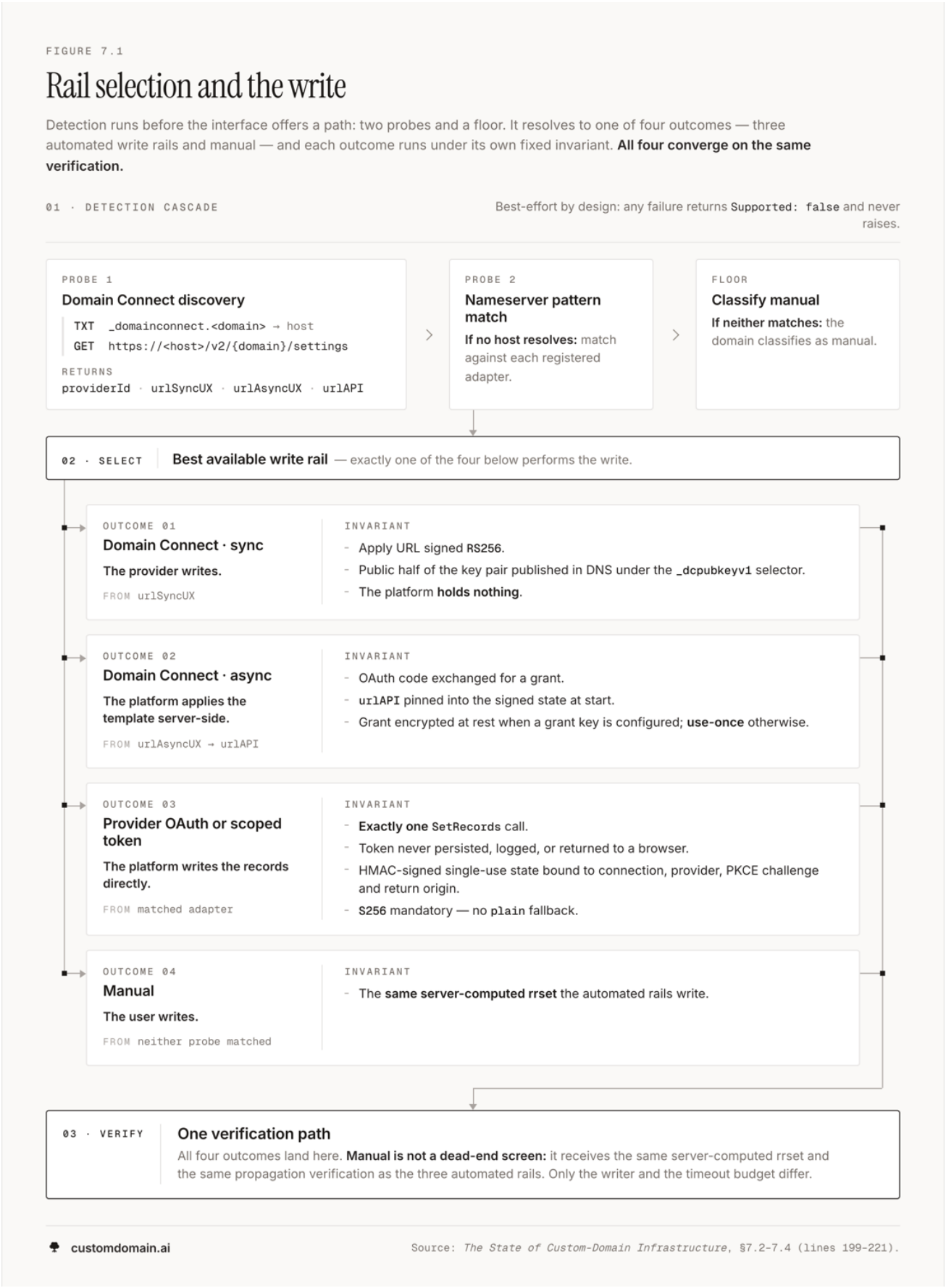


Figure 7.1 — Rail selection and the write. Detection (§7.2) chooses the rail; the record set is computed once by the control plane (§7.3) and is identical on every branch, including the manual floor. The rails differ in who performs the write and in what the platform holds afterwards, not in what gets written.

Rails degrade rather than fail. A provider with no registered OAuth client reports OAuth unavailable and the flow falls back to the next rail down. The sync-redirect rail is restricted to providers with a published, verified template, because an unpublished template strands the user inside someone else's control panel with an error message the platform did not write and cannot change.

Where a rail does run, it runs under fixed invariants: HMAC-signed, single-use, short-TTL state bound to the connection, the provider, the PKCE challenge and the return origin; mandatory `S256` with no `plain` fallback; a provider access token used for exactly one `SetRecords` call and never persisted, logged, or returned to a browser; and, on the async rail, the provider API base pinned into the signed state at start so that no attacker-influenced discovery result can redirect a token exchange. §9.2 explains why that last one is not paranoia.

Manual is not a dead-end screen. It receives the same computed records, the same guarded state machine, and the same propagation verification as every automated rail. It differs in two respects: who performs the write, and the timeout budget — 72 hours pending on manual against 24 hours propagating on the automated rails, because a human has to find a control panel and an API call does not. The widget carries a distinct terminal status for manual completion, because a manual success is a success and reporting it as a degraded automated success would make the product's own metrics lie.

7.5 Routed, shipped, one-click: three different numbers

The census records the best rail each provider is **routed** to. That is not the same as the rail that **executes** in a given deployment, and neither is the same as the number of providers where a user gets a genuine **one-click** experience. Conflating the three is how coverage claims in this category become unfalsifiable (§11, row one).

Across the 63 providers in the parity target, the census routes 38 of them to the manual floor, 16 to a tenant-supplied scoped API token, 6 to native provider OAuth, and 3 to Domain Connect. Six adapters expose provider-hosted OAuth: Cloudflare, DigitalOcean, DNSimple, Netlify, Vercel and WordPress.com.

Quantity	Value	What it counts
Providers in the parity census	63	Every name on the incumbent's published provider list but World4You
Routed to the manual floor	38 of 63	Provider-specific copy-paste instructions; also the default for any provider not on the list
Routed to a tenant-supplied API token	16 of 63	The tenant's customer mints a token by hand at their provider
Routed to native provider OAuth	6 of 63	Requires a client id and secret registered with each provider first
Routed to Domain Connect	3 of 63	GoDaddy, IONOS, NameSilo
Routed to some automated rail	25 of 63	16 + 6 + 3
Executing today	23 of 63	Two of the 25 are routing, not working: see below
Genuine one-click, best case	at most 7	And only after per-provider app registration, which is an owner action

Table 7.1 — Four different quantities that a single coverage headline would collapse into one. The census routes; a deployment executes; a user experiences one click. Figure 1.2 plots the same three numbers.

The shipped configuration is narrower than the routing table, and the gap is worth stating precisely.

Of the three Domain Connect providers, only NameSilo can complete a redirect out of the box. The sync rail is disabled by default and re-enabled only for providers measured to serve our template catalogue. That excludes GoDaddy and IONOS, because both hand-curate their template sets and serve none of ours — as of 2026-07-28, GoDaddy served 0 of our 18 published templates. The async rail ships with no partner credential at all. Of the six OAuth providers, each needs a client id and secret registered with the provider before its rail runs; DigitalOcean's app is still unregistered. So 25 is what the census routes and 23 is what executes today, and the number to design around is not 63 but at most 7 one-click, reached only after per-provider app registration. Everything else lands on an API token the tenant pastes in, or on manual.

A note on the numeral 38. It appears twice in this paper with two different meanings, and they must not be read as the same fact. In this section and in §8.1 it means 38 of the 63 providers in the parity census are routed to the manual floor. In §4 it appears as an adapter-build count — the number of DNS providers for which adapters were built when surveying which ones expose a provider-hosted OAuth flow. The first is a routing verdict about coverage. The second is a denominator for a survey. Any sentence that lets a bare "38" stand alone is ambiguous, and this paper does not use one.

7.6 Propagation is polled, and the state machine is guarded

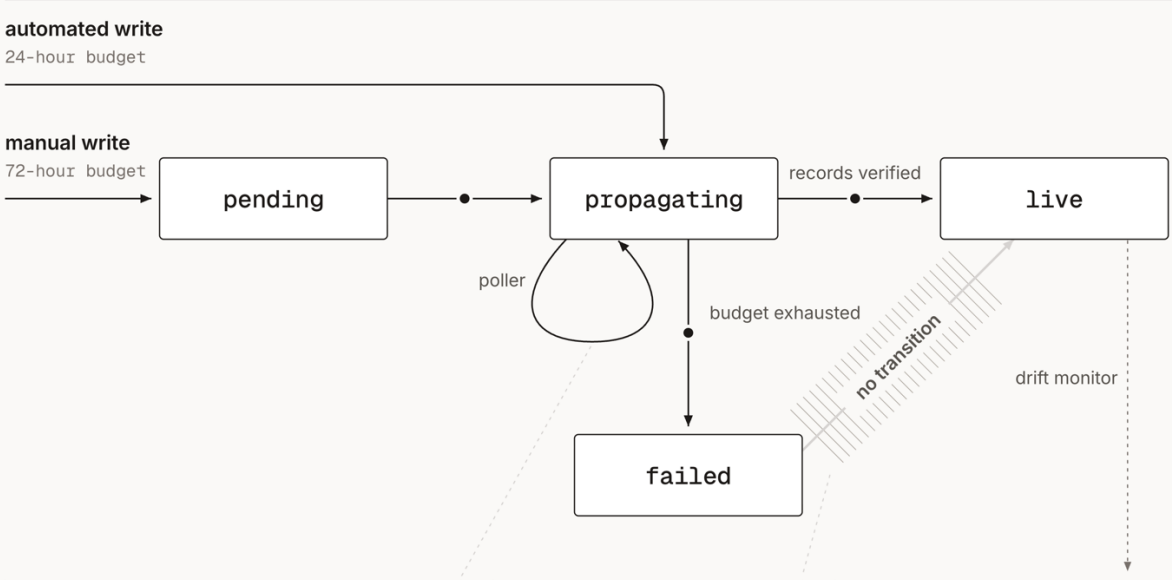
There is no propagation event to subscribe to (§3, Figure 3.1). Authoritative servers hold the record immediately; each recursive resolver learns it whenever its own cached entry expires, and a negative answer cached before the write persists for the zone's negative TTL. The only honest implementation is to poll public DNS and to model the wait as a state.

Statuses are `pending`, `propagating`, `live` and `failed`, with transitions validated by a single owner.

FIGURE 7.2

The connection state machine

Manual writes enter at **pending**, automated writes at **propagating**, and a single owner validates every transition. The edge drawn in grey under hatch is the one the machine does not have.



THE POLLER

Runs on the **propagating** self-loop.

60 s	poll interval, backing off geometrically to 900 s
100	records claimed per batch, at most
2 min	lease held over a claimed batch
8	concurrent workers

THE ABSENT EDGE

`failed → live`

A retry cannot skip re-verification.

Drawn in grey under hatch to mark an edge the machine does not have.

THE DRIFT MONITOR

Every pass returns one of four per-record verdicts. Two of them leave the system.

no change	–
missing	→ domain.record_missing
changed	–
restored	→ domain.record_restored

Both leave through a durable outbox, each webhook signed HMAC-SHA256.

SOURCE PAPER LINES 225-229 · 241-243 · 375

A solid marker on a transition denotes validation by the single owner. The grey hatched edge is drawn to show what the machine does not have; there is no transition from **failed** to **live**. Budgets are shown on the write that opens the connection.

Figure 7.2 — The connection state machine. The edge from `failed` to `live` is deliberately absent: a retry re-enters through re-verification or not at all.

The missing `failed` → `live` edge is the load-bearing design decision in the diagram. Its absence means that no retry path, no support-tool button and no manual override can promote a connection to live without the propagation check running again and passing. A state machine that allows an operator to "just mark it live" will have that transition used, and it will be used on exactly the connections where verification was failing for a reason.

A background poller advances applied connections by reading public DNS on a one-minute tick with geometric backoff out to fifteen minutes. It claims batches of up to 100 connections under a two-minute lease so that a horizontally scaled control plane does not double-poll the same work, and it runs eight concurrent workers. Backoff matters for the reason §3 gives: querying a name too early teaches resolvers a negative answer they will then hold for the negative TTL, so an aggressive poller makes the wait it is measuring longer.

One design note from building this. The connect engine ships a mock DNS adapter that accepts any credential without validating it, which is correct behaviour for a test double. It is excluded from the default registry. With it registered, a caller could persist a "written" record set while holding no DNS access whatsoever, and the poller — finding matching public DNS that was already there — would promote the connection to live. Test doubles that reach production registries manufacture false positives in exactly the component whose entire job is to prevent them.

7.7 Certificates issue at the handshake, fail-closed

The edge orders certificates on demand during the TLS handshake using ACME [R6], with the TLS-ALPN-01 challenge answered inline on the same `:443` listener and HTTP-01 on `:80`. There is no provisioning queue and no batch job; the first request for a hostname is what triggers issuance.

That design puts an unauthenticated party — whoever opened the connection — in control of when the platform talks to its CA. The host policy therefore defers to a gate that calls the control plane's `POST /internal/ask` before anything else happens. Unknown host, malformed response, timeout, control plane unreachable: every one of those resolves to **deny**.

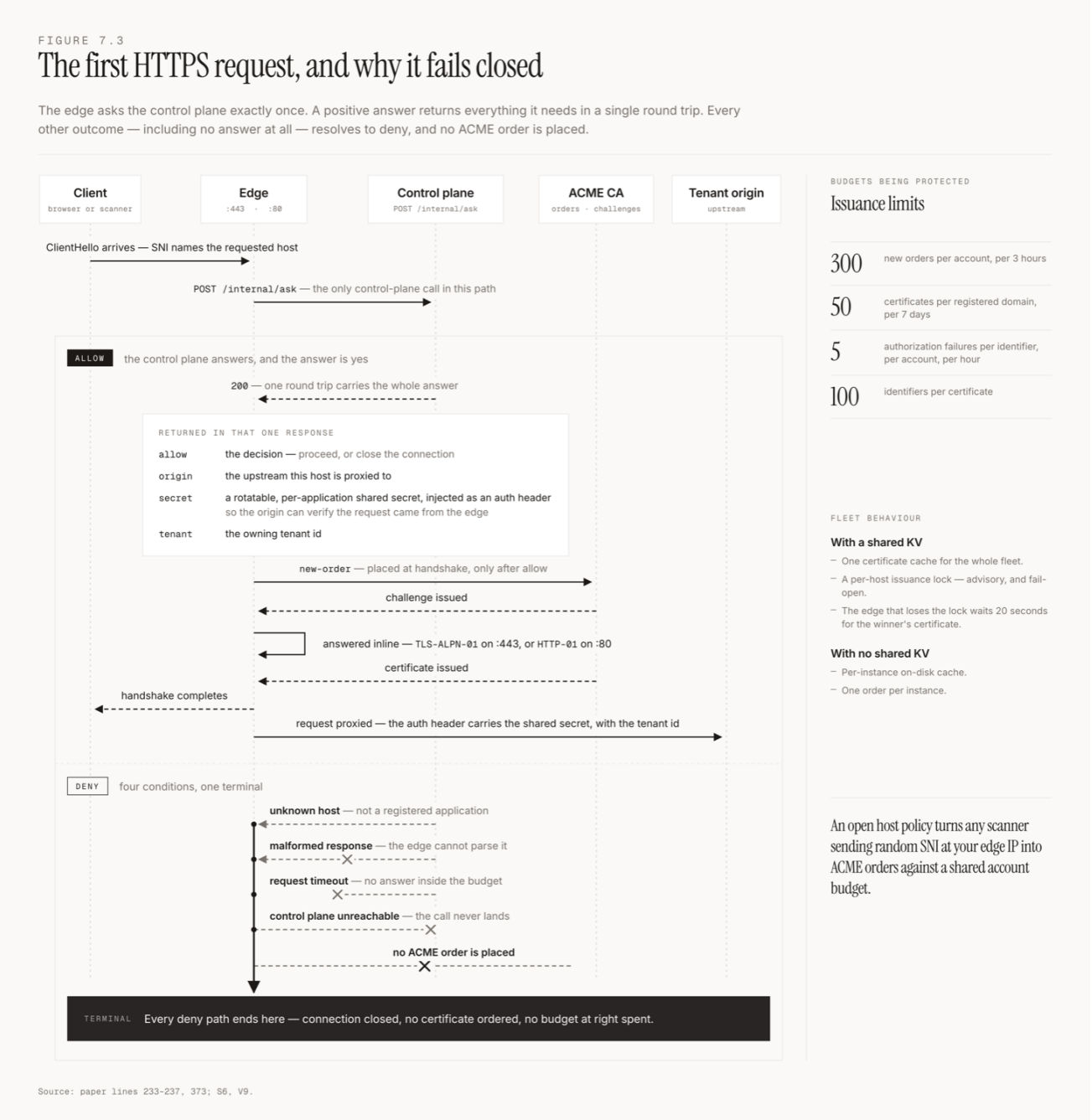


Figure 7.3 — The first HTTPS request for a newly connected hostname, and the three places it can fail: the ask gate denies an unapproved host before any ACME order exists, the CA refuses on a CAA RRset that omits the issuer, and the authorization-failure budget locks the identifier out for the rest of the hour.

Fail-closed here does two jobs. It keeps unapproved hosts off the edge, and it keeps unapproved hosts out of the ACME order budget. Let's Encrypt allows roughly 300 new orders per account per three hours and 5 authorization failures per account per hostname per hour [R21]. An open host policy converts any scanner sending random SNI at your edge IP into ACME orders against those budgets — which is not a hypothetical attack so much as ambient internet weather.

CAA is the other silent blocker (§3; [R7], [R22]). A pre-existing CAA record that does not authorize the issuing CA blocks issuance inside the CA, while a CAA set that does list the issuer does not. So the pre-flight runs a live CAA lookup and warns only where the existing set omits the issuer. It does not warn on the mere presence of CAA, because a correct CAA record is a good thing and a check that cries wolf on it will be ignored.

A positive answer from the ask gate returns everything the proxy needs in one round trip: the allow decision, the upstream origin, a rotatable per-application shared secret injected as an auth header so the origin can verify that traffic arrived through the edge rather than from the open internet, and the owning tenant id.

With a shared KV configured, a fleet of edges uses one certificate cache and a per-host issuance lock, so a newly seen hostname normally produces one ACME order regardless of how many instances saw it. The lock is advisory and fail-open: an edge that does not acquire it waits 20 seconds for the winner's certificate to land in the shared cache, then issues for itself rather than serving an error. With no shared KV configured, each instance keeps its own on-disk cache and there is no cross-edge coordination at all — one order per instance, which is a real cost against the 300-per-three-hours account limit at fleet scale.

7.8 Drift after go-live

Go-live is not the end of the record's life. Customers migrate registrars, hand DNS to an agency, or clean up records they no longer recognise, and none of those events notify the platform.

A monitor engine computes per-record verdicts against the **stored baseline** — the exact RRset the control plane computed and confirmed at go-live. The verdicts are `no change`, `missing` and `changed`, with `restored` emitted by the state machine when a previously missing record reappears. Re-check sweeps run through a pluggable resolver seam; the wired deployment reads through the system recursive resolver.

This is DNS-record drift, and it is explicitly not origin uptime monitoring. The monitor answers one question: does the customer's zone still publish the records this connection was verified against. It says nothing about whether the tenant's origin is up, whether the application behind it returns 200, or whether the certificate is about to expire. Those are different systems with different false-positive profiles, and merging them produces alerts that mean "something somewhere," which is the same as no alert.

When drift alerting is enabled, findings leave the system as signed webhooks; otherwise the sweep runs in shadow mode, persisting per-record state without delivering anything. The event set covers the connection lifecycle and drift specifically, including `domain.record_missing` and `domain.record_restored`. Deliveries are HMAC-SHA256 signed over `{timestamp}.{raw_body}`, hex-encoded with a `sha256=` prefix and the timestamp in a companion header, and they queue through a durable outbox rather than a best-effort in-process send.

The outbox is there for one reason. A dropped drift notification is indistinguishable from no drift. Silence has to mean something, and it only means something if the delivery path cannot lose messages when a process restarts mid-send.

§8 Domain connection for AI agents

An agent that provisions a customer's domain hits constraints a human user never notices. The interesting part is that all three of them are consequences of the architecture in §7 rather than properties of agents, which is why they do not go away with a better model.

8.1 Three constraints a human never notices

It cannot click. 38 of the 63 providers in the parity census route to the manual rail (§7.5, Table 7.1): the record has to be typed into a control panel that only a logged-in human can reach. A browser-using agent might get through some of those panels. It will also get through the delete button, in a zone that carries the customer's mail routing. That is not a class of failure worth designing around, and an architecture that depends on it is one bad selector away from taking down a stranger's email.

It must not hold the credential. The obvious shortcut is to ask the user for a DNS API token and let the agent call the provider directly. That token is usually zone-wide or account-wide, it lives in a context window, and nothing about it expires when the task ends. A token pasted into a chat is a token you now have to rotate.

It must not spend money on its own. Domain purchase is a real charge against a real card. An agent that can search availability and an agent that can buy are different security objects, and the second one needs a human in the loop by construction rather than by policy.

8.2 The tool surface and the scope model

The MCP server implements protocol revision 2025-06-18 [R16] and exposes 12 tools: `search-domain-availability`, `generate-domain-suggestions`, `create-domain-order`, `connect-domain`, `check-connection-status`, `check-order-status`, `reapply-connection`, `disconnect-domain`, `discover-provider`, `forward-domain`, `add-email`, and `list-connections`.

Every tool carries a human-readable title, `readOnlyHint` and `openWorldHint`. The five state-changing tools add `destructiveHint` and `idempotentHint`; `disconnect-domain` carries `destructiveHint` without an idempotency hint. The six read-only tools omit the two write hints entirely, because the annotation fields are optional pointers and a nil pointer is dropped from the serialised JSON rather than emitted as `false`. Hosts read these to decide what to auto-run and what to gate. `disconnect-domain` is the only tool whose `destructiveHint` is true.

Annotations are advisory. A host is free to ignore them, and a compromised host will. Enforcement is separate, and it uses a closed three-scope model.

Scope	Tools it covers	Why it is a separate scope
`domains:read`	`search-domain-availability`, `generate-domain-suggestions`, `discover-provider`, `check-connection-status`, `check-order-status`, `list-connections`	Observation only; safe for a host to auto-run
`domains:connect`	`connect-domain`, `reapply-connection`, `forward-domain`, `add-email`, `disconnect-domain`	Writes DNS in a zone the platform does not own, and can remove a live domain
`domains:purchase`	`create-domain-order`	Spends money against a real payment instrument

Table 8.1 — The agent scope model. Three scopes, closed set, one scope per tool. The split between `connect` and `purchase` exists because the failure modes are not comparable: a bad DNS write is recoverable, a purchase is a transfer of funds.

The scope map is stamped at construction time by an invariant that rejects any tool lacking exactly one valid scope. A new tool cannot ship unscoped: `NewServer` panics at construction when a registered tool has no entry in the scope map, and `TestToolScopesExactlyCoverRegisteredTools` fails CI before a binary is ever produced. This is the mechanism that answers the question in §11.6 — are scopes enforced in the request path, or declared and ignored — with something other than an assurance.

Enforcement is live in `tools/call` for delegated agent tokens: a call whose verified ES256 claims lack the tool's required scope returns an `insufficient_scope` error naming the scope it needed.

One limit is worth stating plainly, because it is the kind of thing that reads as a footnote and behaves as a hole. **The scope model binds agent tokens only.** An integrator credential — an `sk_` API key or a client-credentials JWT — carries no scopes and reaches all twelve tools including `create-domain-order`. An integrator that hands its own API key to an agent gets none of the protection described in this section. The scope model is not a property of the MCP server; it is a property of the delegated-agent token, and it protects nobody who routes around it.

8.3 The authorization flow

The delegated-agent surface described here is **off by default**. The control plane registers the routes but reports not-configured until agent auth is switched on with a durable ES256 signing key, and the MCP server's agent-token path stays dark until it is given a JWKS URL.

The design goal is narrow and worth stating before the mechanics: the agent never sees a DNS credential, and never obtains authority without a human granting it in a browser session the agent does not control.

The control plane implements RFC 7591 dynamic client registration at `POST /oauth/agent/register` [R8], a browser authorize endpoint that hands off to a human console consent screen, `POST /oauth/agent/token`, and RFC 7009 revocation [R9]. Agent clients register as public clients with no secret, which is the honest classification: a public client is one that cannot keep a secret, and an agent

running on someone else's machine cannot. PKCE with `code_challenge_method=S256` is mandatory; a request without it is rejected, and the token exchange compares `BASE64URL(SHA256(verifier))` against the stored challenge in constant time.

Three unauthenticated discovery documents describe the server: RFC 9728 protected resource metadata [R10], RFC 8414 authorization server metadata [R11], and a capability manifest at `/.well-known/agent/mcp.json`. Today those carry the integrator client-credentials path only; an agent taking the delegated path is pointed at the control plane's own `/.well-known/oauth-authorization-server`, where the register, authorize, token and revoke endpoints are advertised.

FIGURE 8.1

How an agent gets authority without getting a credential

The delegated-agent surface, end to end. The agent never receives a DNS credential, and it never obtains authority a human did not grant in **a browser session the agent cannot reach**. Every grant in this exchange expires; the lifetimes are on the right, and they are the security argument.

OFF BY DEFAULT · **ENABLED PER DEPLOYMENT** | REQUIRES A **DURABLE ES256 SIGNING KEY** | SCOPES: **DOMAINS:READ · CONNECT · PURCHASE**

WHAT THE AGENT ENDS UP HOLDING — STEP 19

A short-lived, tenant-scoped token that authorizes **tool calls**. It is **not a DNS credential**, and it cannot be replayed against Cloudflare or GoDaddy. Where the connection needs a person, the tool hands back a link — a Domain Connect authorization URL at the owner's own DNS provider where one exists, otherwise a console page carrying the same records — which the person opens themselves.

customdomain.ai

Solid arrows are requests; dashed arrows are responses.
RFC 7591 registration · RFC 9728 resource metadata · RFC 7009 revocation.

Figure 8.1 — How an agent gets authority without getting credentials. The agent registers, then hands the user a link; the human authorizes in a browser the agent cannot reach; the agent receives a 15-minute, tenant-scoped, ES256-signed token that authorizes tool calls and nothing else. No DNS credential crosses the agent boundary at any step.

Access tokens are ES256-signed, tenant-scoped, and valid for 15 minutes. The MCP server verifies them locally against the control plane's published JWKS, with no per-request introspection, which keeps the hot path cheap. The 15-minute ceiling is what makes local verification safe: revoking a grant or a refresh token takes effect within one token lifetime, so the gap between "revoked" and "stops working" is bounded without a round trip on every call. Authorization requests live 10 minutes; codes live 5.

The token authorizes **tool calls**. It is not a DNS credential and cannot be replayed against Cloudflare or GoDaddy — there is nothing in it those providers would accept. When a connection needs a human, the tool returns a link the person opens themselves: a Domain Connect authorization URL at their own DNS provider where one exists, and otherwise a console page carrying the same computed records from §7.3. On the 38-of-63 manual floor, that console page is the product's answer for agents, and it is the same page a human user gets.

8.4 Money is gated twice

Consent granting `domains:purchase` is refused unless the human supplies an explicit positive monthly spend cap. There is no default cap, and a non-positive cap denies outright rather than being treated as unlimited.

At purchase time the cap is enforced by an atomic check-and-record. The ordering matters: a cap checked before an order and recorded after it has a double-spend race, and an agent making concurrent calls is precisely the client that will find it. Over cap, the spend is not recorded, the owner is alerted, and the purchase is denied.

A second gate sits in front of that. Paid orders placed through MCP call back to the integrator over Basic Auth before any order is placed. HTTP 200 authorizes. 401 blocks. Anything else — 500, a timeout, a connection reset, a DNS failure on the callback host — denies. With no authorization URL configured, every purchase is denied.

Registrar purchase is itself behind a deployment flag that defaults off, as is the agent auth surface. In production the control plane refuses to enable agent auth without a durable ES256 signing key, because an ephemeral key would invalidate every live access token and the published JWKS on any restart, or on any second instance coming up behind the load balancer.

Two gates for one action is deliberate, and they protect different parties. The spend cap protects the human owner from their own agent. The callback protects the integrator from a compromised grant. Either one alone leaves a party unprotected against a threat the other one covers.

§9 Security, trust and failure modes

A system that writes DNS records on a domain it does not own is holding borrowed authority. Borrowed authority has a shape: how wide the grant is, how long it survives, who can take it back, and what still runs after it is taken back. Those four answers are the security model.

9.1 What is actually delegated

Three write rails and four grant shapes, and they are not interchangeable.

Signed redirect (Domain Connect sync). The platform holds nothing at all. The control plane signs an apply URL with RS256 using a private key that never leaves it; the DNS provider verifies that signature; the user lands on their own DNS provider; the provider authenticates them and writes. Authority ends at "request". This rail ships behind a deployment flag and a per-provider allowlist, because an unpublished template dead-ends the user at their provider's error page (§7.5).

Domain Connect async grant. An OAuth code is exchanged for a durable grant, encrypted at rest when a grant-encryption key is configured. The grant exists so a template can be re-applied or reverted server-side without dragging the end user back through consent. With no encryption key configured it degrades to use-once: apply, then discard the credential.

Provider OAuth. The end user authorizes their own DNS provider inside the widget. The resulting access token is used for exactly one `SetRecords` call. It is never persisted, never logged, and never sent to a browser.

Caller-supplied provider API token (BYO token). The same direct-write rail also accepts a provider API token passed on the apply call itself. This is the mode the census records for 16 of the 63 providers.

The widest grant is that last one. It is whatever the integrator's customer minted at their provider — typically zone-wide or account-wide — and the platform cannot narrow it. It is passed per apply call and not persisted, which bounds exposure in time but not in scope. The Domain Connect grant is the only one the platform stores, and its width is bounded by a template the provider itself chose to honour, which is a narrower and more legible boundary than "whatever the customer pasted".

9.2 Pinning, and the parts of discovery an attacker can reach

Domain Connect discovery starts with a TXT lookup on `_domainconnect.<domain>`, which yields a host, which returns a settings document containing `urlAPI`. Everything in that chain is influenced by whoever controls the domain being connected — and the domain being connected is, by definition, supplied by an untrusted caller.

So on the async rail the provider API base is captured at start time, signed into the OAuth state, and pinned onto the stored grant. Token exchange, apply and revert go only to that pinned base. A `urlAPI` from fresh discovery is never trusted after the flow begins. Without pinning, an attacker who controls DNS for a domain they submit could move the API base between the start of a flow and the token exchange, and receive a token intended for someone else's provider.

The provider-OAuth rail carries its own invariants: HMAC-signed, single-use, short-TTL state bound to the connection, the provider, the PKCE challenge and the return origin; PKCE `S256` required with no `plain` fallback; and callback results posted only to an origin vetted at start time against a configured allowlist. An empty allowlist denies everything rather than defaulting to permissive, which is the correct reading of an unconfigured security control.

On the key published in DNS. The sync rail publishes a key in DNS by selector (`_dcpubkeyv1`), and it is worth being explicit about which one, because the naming invites the wrong assumption. What is published is the **public verification key**. The RS256 **signing key is private and never leaves the control plane** — it is not in DNS, not in the widget bundle, and not in any response. The DNS provider fetches the public key at the selector and uses it to verify that an apply URL was signed by the party it claims to be from. Publishing a signing key would hand anyone who can run `dig` the ability to forge applies against every provider that trusts the selector.

9.3 Revocation, and who holds the lever

Revocation from the platform's side depends on which rail wrote the records, and one rail cannot revoke at all.

Domain Connect grants support server-side revert when a grant-encryption key is configured. Without one, the async apply is use-once and there is no stored grant to revert with.

Because the provider-OAuth callback never persists the access token, an OAuth-connected domain is never marked managed. `:reapply` returns HTTP 409 for it, and there is no server-side revert of those records. This is the trade §11.7 asks vendors to state out loud: a one-shot token is the safest posture and it removes the ability to re-apply later. Both answers are defensible; the undefensible move is not knowing which one you made.

The agent authorization surface — off by default, enabled per deployment — is designed around RFC 7009 revocation [R9] and 15-minute ES256 access tokens, so a revoked grant stops working within one token lifetime even though tokens are verified locally against a JWKS and never introspected per request. The embed uses a 15-minute widget JWT minted server-side by the integrator, so no API key reaches a browser. Integrator API keys are stored hashed, with only a short prefix indexed.

Revocation from the provider's side is the uncomfortable half. Under the Domain Connect specification the DNS provider is the only party that writes records, runs discovery, and issues or revokes tokens [R12]. A service provider can only ask. Any DNS provider can sever any aggregator unilaterally by declining applies or revoking tokens, and that is not theoretical.

GoDaddy revised its terms of use in a way that cut off aggregator DNS access. In Entri LLC v. GoDaddy.com LLC, No. 1:24-cv-00569 (E.D. Va.) [R37], Judge Anthony J. Trenga denied GoDaddy's motion to dismiss in a memorandum opinion and order signed **2024-10-10** [R36], writing that the alleged negative tie "cuts off all aggregator services … from competing" — language quoted here as reported by Domain Name Wire on 2024-10-11 [R39]. The parties settled on 2025-02-25 [R40]; a

multi-year access agreement was announced on 2025-06-18 [R41]. Eight months separated the ruling from the access agreement, with the aggregator's automation off in between. Figure 4.2 plots that interval against the standard's own timeline.

The design consequence is blunt. Every automated rail has to degrade to something nobody else can switch off. That floor is manual copy-paste records, and an unrecognised domain classifies to it by default (§7.2). The 38-of-63 manual floor is usually read as a coverage weakness. It is also the only part of the system with no third-party kill switch.

9.4 Tenant isolation

Credentials are keyed per tenant, and every adapter call carries one. There is no shared platform credential at a DNS provider that a tenant's connection could borrow.

The edge's authorization answer (§7.7) returns the owning tenant id alongside the origin URL and a rotatable per-application shared secret, injected as an auth header so the origin can verify that traffic actually arrived through the edge rather than from the open internet. Without that header an origin has no way to distinguish a proxied request from a direct one, and any per-tenant hostname becomes an open door to every other tenant's origin that shares an IP range.

Where the agent surface is enabled, agent tokens are tenant-scoped, and capability is the closed three-scope set of Table 8.1 with the construction-time invariant described in §8.2. Purchase is isolated from the other two scopes because it spends money: consent cannot grant `domains:purchase` without an explicit positive spend cap, and the cap is checked and recorded atomically on the purchase path (§8.4). Both the agent consent flow and registrar purchasing are off by default and must be enabled per deployment.

9.5 Fail-closed defaults

Certificate issuance is gated **before** ACME, not after. The TLS host policy defers to a gate that asks the control plane whether a host is approved. Control plane unreachable, request timeout, malformed response, unknown host: every one of those resolves to deny. A host the control plane has not approved never triggers an ACME order at all, so it cannot consume the account's order budget or its authorization-failure budget [R21].

Two other defaults follow the same rule and are worth naming because each one started out the other way:

An **empty callback-origin allowlist denies everything.** Treating an unset allowlist as "no restriction configured" is the reading that turns a missing config value into a redirect vulnerability.

The **mock DNS adapter is excluded from the default registry.** It accepts any credential without validating it, which is correct for a test double and a silent authorization bypass in production: a caller could persist a "written" record set while holding no DNS access, and the propagation poller would later find matching public DNS and promote that connection to live (§7.6). The adapter is fine. Its presence in the default registry would not have been.

9.6 Honest failure modes

Seven conditions the system cannot fix, and the observable behaviour of each.

Condition	What the system does	What the integrator sees
Provider unrecognised, no adapter	Classifies as manual, renders the exact computed RRset	Setup type `manual`; connection stays `pending` for up to 72 hours
Provider API rejects the write	Connection does not advance; no partial state is recorded	Structured error carrying `{code, title, details}`
Apex requested on Route 53	`cname_flattening` reported false at check time (§7.2)	Capability check fails before apply, not during it
CAA RRset excludes the issuing CA	ACME order fails; no bypass exists	No certificate for that host; the edge logs the failure and re-orders on the next cache-miss handshake
Host not approved by the control plane	Ask gate denies; no ACME order is placed	No certificate, no proxying
Record removed after go-live	Monitor verdict `record_missing` against the stored baseline; the sweep persists the transition	`domain.record_missing` webhook, HMAC-SHA256 signed, from a durable outbox, where drift alerting is switched on
Records never propagate	`propagating` expires at 24 hours	Status `failed`

Table 9.1 — Failure modes stated as behaviour rather than as absence. Each row is a condition with no platform-side fix; what the platform owes the integrator in each case is a legible state, not a retry.

Two rows deserve more than a line.

CAA is the one that surprises teams. Checking it is mandatory for every publicly trusted CA under the CA/Browser Forum Baseline Requirements, so a customer's pre-existing CAA record that omits the issuing CA blocks issuance outright [R7]. The block is loud on the ACME side and silent everywhere else: the order fails with a CAA error the platform can read, and nothing in the customer's DNS or in their browser says why. There is no workaround on the platform side. The honest move is to detect it in pre-flight, report `failed`, and tell the customer which record to change. Two pieces of Let's Encrypt's guidance are worth passing to customers directly: a DNS provider does not need explicit CAA support, only a `NOERROR` response to unknown query types — providers that answer `SERVFAIL` break issuance while holding no CAA record at all — and the `accounturi` parameter limits issuance to a specific ACME account [R22].

ACME rate limits are the other. Fifty certificates per registered domain per 7 days, 300 new orders per account per 3 hours, and 5 authorization failures per account per hostname per hour [R21]. The fleet-level answer is the cross-edge issuance lock over a shared KV described in §7.7, so one

hostname newly seen by twenty edge instances produces one ACME order. Configuring that KV is optional; with none configured, each instance falls back to a per-instance certificate cache and the fleet can place one order per instance.

The per-host answer needs stating precisely, because it is the kind of thing a datasheet would round up. The edge carries a 15-minute issuance back-off for any host the control plane reports as `failed`, consulted only on a genuine cache miss so that a host already holding a valid certificate is never touched. But the edge reads that verdict off the `secure_status` field of the ask answer, and the control plane only populates that field where the Secure surface is enabled. **Secure is not a capability this system ships or sells**; on every deployment described in this paper it is off. With Secure off the control plane sends no status, the back-off treats a missing status as permit, and the shipped behaviour is one ACME order per cache-miss handshake. So on a real deployment the per-host answer is the one nobody controls: a misconfigured domain burns its five authorization failures inside an hour and then waits, and no amount of retry logic in the edge changes that.

Recovery has one rule, and §7.6 already drew it. The connection state machine has no transition from `failed` to `live`. A retry cannot skip re-verification.

§10 Build versus buy

The decision is usually framed as a feature comparison. It is not one. Every line item in this section is something a team can build; several of them are a weekend. The question is which of them you will still be maintaining in three years, and what you gave up to do it.

This section gives you a frame rather than an answer, because the inputs that decide it are yours: the shape of your customer base, how many domains you onboard a month, and what an engineer-month is worth against your roadmap. We have no independent data on any of those and will not invent any.

10.1 What "build" means

Nine line items. They are drawn from the architecture in §7, the agent surface in §8, and the security model in §9, and they are listed in roughly the order a team discovers them.

1. **Provider detection.** A cascade with two probes and a floor: Domain Connect discovery first (a TXT lookup of `_domainconnect.<domain>`, then a settings fetch that yields `providerId`, `urlSyncUX`, `urlAsyncUX` and `urlAPI`), nameserver-pattern matching against your adapter set second, and a manual classification for anything that matches neither [R12]. Detection has to be best-effort: any failure returns "unsupported" rather than raising, because the caller's next move is the same either way, which is to drop a rail (§7.2).

2. **A capability layer, separate from detection.** Detection answers who runs the zone. A second layer answers what that operator will accept: apex support, wildcards, CNAME flattening at the apex, SPF-override tolerance, CAA behaviour, TXT values above 255 bytes, and a conflict threshold that predicts when an automated write degrades to manual. Collapsing these two layers is the most common early mistake, and it is expensive because it moves the failure from check time to apply time (§7.2, §11.2).

3. **Server-side record computation.** The browser never derives a record. One authority computes the rrset and every rail renders the same one, including the manual rail. This is what makes verification uniform (§7.3).

4. **Write adapters, one contract, several rails.** A `GetRecords` / `SetRecords` / `DeleteRecords` contract per provider, with `SetRecords` as an idempotent create-or-update to an exact rrset, plus the Domain Connect signed redirect and the Domain Connect asynchronous OAuth flow, plus provider-hosted OAuth where it exists (§7.4).

5. **A propagation poller and a guarded state machine.** Statuses `pending`, `propagating`, `live`, `failed`, with one owner validating transitions and no edge from `failed` to `live`. In our implementation the poller ticks at one minute with geometric backoff to fifteen, claims batches of up to 100 under a two-minute lease so a horizontally scaled control plane does not double-poll, and runs eight concurrent workers (§7.6).

6. **An edge that terminates TLS for arbitrary hostnames.** On-demand ACME issuance during the handshake [R6], TLS-ALPN-01 answered inline on `:443` and HTTP-01 on `:80`, and a host policy that defers to the control plane and resolves every ambiguous answer — unknown host, malformed response, timeout, control plane unreachable — to deny (§7.7).

7. **Drift monitoring after go-live.** Per-record verdicts against a stored baseline, re-check sweeps driven through a resolver seam that reads public DNS rather than your own write log, and delivery through a durable outbox, because a dropped drift notification is indistinguishable from no drift (§7.8).

8. **The customer-facing surface.** A widget authenticated by a short-lived server-minted token so no API key reaches a browser, provider-specific copy-paste instructions for the manual floor, and a distinct terminal status for manual completion. A manual success is a success, and a product that treats it as an error screen has made the long tail into a support queue (§7.4).

9. **The operational surface.** Per-tenant credential custody, tenant isolation on every adapter call and every edge answer, a machine-readable API description that cannot drift from the handlers, and a test posture in which no mock adapter is reachable from the default registry — an adapter that accepts any credential without validating it lets a caller persist a "written" record set with no DNS access at all, which the poller then promotes to `live` (§9.5).

None of the nine is research. All nine are maintenance.

10.2 The seven line items nobody scopes

These are the ones that do not appear in a build estimate, because they are not visible until the system is carrying real traffic.

1. **ACME rate limits.** Let's Encrypt publishes 300 new orders per account per 3 hours, 5 certificates per identical identifier set per 7 days, 5 authorization failures per account per hostname per hour, 50 certificates per registered domain per 7 days, and 100 identifiers per certificate [R21]. For a custom-domain platform the 300-orders-per-3-hours account limit binds first, because every customer domain is a separate registered domain but they all share one ACME account. The authorization-failure limit does not aggregate across the fleet, which is the good news; it turns one misconfigured customer domain into a one-hour lockout on that domain alone. The fleet-level answer is a shared certificate cache and a cross-edge per-host issuance lock. Both are infrastructure you now operate.

2. **Apex realization, per provider.** RFC 1034 §3.6.2 and RFC 2181 §10.1 make a CNAME at the zone apex illegal [R1][R3], and vendors want a CNAME precisely because their edge addresses change. Providers answer with flattening, ALIAS, or ANAME, and some cannot answer at all. Route 53 is the case that corrects the design: it has no ALIAS record type, only an `<AliasTarget>` element in the API that cannot point at an arbitrary external hostname. Modelling apex as a per-adapter detail does not work. It became tractable only when we treated the apex target as its own record type and rewrote it into each provider's native spelling immediately before the write, so the stored and verified record stays provider-agnostic. Twelve providers currently carry that mapping in our implementation (§7.3).

3. **CAA pre-flight.** CAA checking is mandatory for every publicly trusted CA under the CA/Browser Forum Baseline Requirements [R17], so a customer's pre-existing CAA record that omits your issuer blocks issuance inside the CA, minutes after every DNS record you asked for is correct and resolving. There is no bypass. The only useful engineering is to read CAA before ordering and to say which record is the problem. A related trap costs a day when you hit it: a DNS operator does not need to support the CAA record type, it needs to return NOERROR for unknown query types, and operators that answer SERVFAIL break issuance while holding no CAA record at all [R22].

4. **The propagation state machine.** There is no propagation event to observe. Authoritative servers have the record immediately; each recursive resolver learns it when its own cached entry expires, and if anything queried the name before the record existed, the negative answer is cached for the interval RFC 2308 §5 derives from the SOA. Checking early makes the wait longer. That is a state machine with timeouts, not a spinner, and the timeouts differ by rail: in our implementation 72 hours pending on the manual floor against 24 hours propagating (§3.3, §7.6).

5. **Drift detection.** Go-live is not the end of the record's life. Customers migrate registrars, hand DNS to an agency, or delete records they no longer recognize. The build item is not the alert; it is the discipline that the verdict comes from a public resolver rather than from the fact that your own API call returned 200, plus signed delivery from a durable outbox (§7.8, §11.5).

6. **Credential custody.** Four grant shapes, and they are not interchangeable (§9.1). A one-shot provider OAuth token used for exactly one `SetRecords` call and never persisted is the safest posture and also the one that removes your ability to re-apply the records later. A stored Domain Connect grant buys you server-side re-apply and revert and costs you an encrypted secret with a rotation story. The widest grant is the one you cannot narrow: a provider API token the customer minted by hand, typically zone-wide or account-wide. Deciding this late means deciding it twice.

7. **Provider API churn.** A provider census is a maintained artifact, not a table you write once. Providers change auth models, deprecate endpoints, rename record types and alter what a write to an existing rrset does. Namecheap is the example worth carrying: its API replaces the entire zone in a single call, which is why we route it to manual records rather than to an automated write. Adapters fail quietly, and they fail in a component whose whole job is to be trusted.

10.3 A total cost frame you fill in

The table below is a model, not a measurement. We have no independent data on what any of these lines costs at your company, and the public numbers in this category are thin enough that §5 declines to build an average out of them. Fill in the middle column; the right-hand column says what the paper can and cannot give you as an anchor.

Support load uses the arithmetic from §5.2. Annual DNS support cost is

```
S = T × H × R × 12
```

where `T` is DNS-related tickets per month, `H` is mean handling time in hours, and `R` is your fully loaded support cost per hour. Figure 5.1 parameterises the same identity graphically. The only public anchors for `H` and for the delta in `T` are vendor-published customer claims and should be treated as

such: Fourthwall is reported to have attributed nearly half its support tickets to DNS configuration, at 15 minutes to an hour each, before automating [R30]; Crisp is reported to have gone from five or six DNS tickets a day to about one [R31]. Neither is independent research, and neither is a benchmark for you.

#	Cost line	Your estimate	What this paper can anchor it to
1	Initial build, the nine line items in §10.1	_____ engineer-months	Nothing. No public benchmark exists. Scope it against §7 and §10.1 directly.
2	Ongoing maintenance: adapter and census upkeep (§10.2 item 7)	_____ engineer-months/year	Nothing public. Scale it by how many providers you carry, not how many customers you have.
3	Ongoing maintenance: certificate and edge operations (§10.2 item 1)	_____ engineer-months/year	The published Let's Encrypt limits [R21] tell you which limit binds first at your onboarding rate.
4	Support load today	`T` _____ tickets/mo × `H` _____ hours × `R` _____ /hour × 12	Vendor-attributed claims only [R30][R31]; see §5.2 and Figure 5.1.
5	Support load after automation	Same identity, new `T` and `H`	The residual is set by what fraction of your customers land on the manual floor (§11.1), not by the vendor's provider count.
6	Opportunity cost	_____ engineer-months × _____ value per engineer-month	Only you have this. If you cannot state it, the comparison is not decidable and you should default to the option you can reverse (§11.9).
7	The buy-side alternative	_____ /month at your domain count	The price ladder in §6 and Figure 6.1: edge and TLS sells at roughly ten to thirty cents per hostname per month [R23][R24][R25]; the DNS-configuration job has no settled price [R26][R32].

Table 10.1 — A total cost of ownership frame for build versus buy. This is a model to parameterise, not a result. Lines 1, 2 and 6 have no public anchor in this category and we have not manufactured one; lines 3, 4, 5 and 7 have anchors of the quality stated in the right-hand column.

Two notes on using it. First, lines 4 and 5 are the only ones that shrink when you buy, and they shrink by the fraction of your customers who stop typing — which is a property of your customer base and not of the vendor's coverage headline. Second, line 7 is not a like-for-like substitute for lines 1 through 3. An edge and TLS service replaces line 3 and part of line 1. It does not touch line 4, because it never touches the DNS panel (§6.2).

10.4 When building is the right call

Three conditions, and they compound.

Your customers are technical. The value of an aggregator is concentrated in the non-technical tail: the customer who cannot name their DNS operator, who pastes `_acme-challenge.example.com` into a panel rooted at `example.com`, who opens a ticket four minutes into a negative-cache TTL. If your users run their own zones, hold their own provider API tokens and are comfortable in a control panel, an in-app record table with generated values and an honest verification poll covers most of the ground, and you have written line items 3, 5 and 8 of §10.1 and skipped the rest.

You need subdomains only. If you can require `app.customer.com` and refuse the apex, the single most provider-specific piece of work disappears. A CNAME at a subdomain is uniform across every provider in the census; the apex is where flattening, ALIAS, ANAME and Route 53's `<AliasTarget>` all have to be modelled separately (§3.5, §10.2 item 2). This is a product decision that removes an engineering problem, and it is worth making deliberately rather than discovering after you have shipped.

Your volume is low. ACME account limits do not bind if you are onboarding a few domains a week [R21]. A support queue of two DNS tickets a month is a person who is good at it, not a system. Both of those statements stop being true at a threshold you can compute from Table 10.1 rather than guess at.

There is a fourth case that is not about size. If you already operate an edge — you terminate TLS for arbitrary hostnames today and it works — then the two jobs in §6.2 have already been separated for you, and the only question left is whether to buy the DNS-configuration half. That is a much smaller purchase than the one most vendors quote.

And there is an inverse test worth running before any of this. Take your last few hundred customer domains, resolve their nameservers, and bucket them by operator (§11.1). If most of them land on providers that have no automated write rail from anybody, then buying does not buy you the write. It buys you the widget, the instructions and the verification, which may still be worth the money — but you should know that is what you are paying for, and you should say so out loud in the meeting.

---

§11 Evaluating a vendor

Run this against any vendor, including the version of the vendor that is your own team with two quarters of runway. The questions are ordered by how expensive they are to discover late. §11.10 turns them into a score.

11.1 Coverage: of what, in which mode

Every DNS-configuration vendor leads with a provider count. Edge-only services publish none, because they never touch the provider. The count is close to meaningless on its own, because "supported" spans four mechanics that feel entirely different to a customer: the DNS provider writes the record itself after a Domain Connect redirect or grant; the customer authorizes your platform through the provider's own OAuth; the platform writes with a scoped API token the customer creates by hand and pastes in; or the customer copies and pastes into a panel.

Only the first two are one-click. The third is a DNS panel visit with extra steps. The fourth is the manual floor.

So do not ask how many providers a vendor supports. Ask this instead: **for what fraction of my customers does nobody have to type anything?** That is a question about your domain table, not the vendor's. Compute it:

1. Take your last few hundred customer domains, or your prospect list.
2. Resolve the NS records for each and bucket by operator, not by registrar. Registrar and DNS operator are separate roles, and a domain registered at one and delegated to another is managed at the second (§3.1).
3. Send the vendor the bucket counts and ask which bucket lands in which of the four modes, as a file rather than a slide.
4. Then ask what happens on a domain nobody recognizes. If the answer is an error screen rather than correct provider-specific instructions, the long tail has just become your support queue.

Two public examples show why the headline number does not survive contact. One vendor publishes three different coverage numbers at once across its own surfaces — 50+ providers and a market-share claim on the product page, 60+ with direct API login in the developer overview, and 64 named providers on the developer provider list [R27] [R28] [R29]. Separately, the Domain Connect standard's own site lists nine providers under "Implementation is Live" [R13], while the APNIC introduction to the standard puts implementations at roughly twenty, covering 35% of the `.com` zone as of May 2024 [R38]. The two are not reconcilable from public sources: the site lists only implementations it has been told about and carries no last-updated date, and the post gives a number with no list behind it.

Our own numbers are in Figure 1.2 and we state them the same way. The census routes 63 providers to a mode: 38 of 63 providers are routed to the manual floor, 16 to a customer-supplied API token, 6 to native provider OAuth and 3 to Domain Connect. That is routing. What works in a default deployment is a smaller number, and what is one-click is smaller again — at most 7 of the 63, and only after per-provider application registration, which is an owner action rather than a code path (§1.3, §12).

Ask every vendor for the same three numbers: routed, working, one-click.

11.2 Apex

Root domains break differently from subdomains, and they break at apply time rather than at check time if nobody planned for it. There are three ways to realize an apex alias — flattened CNAME, ALIAS, ANAME — and some providers cannot do it at all.

Ask for the list of providers where apex works, as a list. Then ask the question that actually matters: **when does the user learn it does not?** Honest capability reporting says no during the check, before the customer has committed to a flow. Route 53 is the test case; if a vendor claims apex support there without qualification, ask what it writes, because an `<AliasTarget>` cannot point at an arbitrary external hostname (§3.5, §10.2 item 2).

11.3 CAA

CAA is the failure mode nobody demos, because it does not exist on a fresh test domain. It exists on the domain of a customer who had a certificate vendor relationship three years ago. If the hostname or any ancestor publishes a CAA RRset with an `issue` or `issuewild` property that does not name the issuing CA, issuance fails and the surface symptom is a certificate that never appears [R17][R22].

Ask three things. Does the tool read CAA before ordering, or discover the conflict from a failed ACME order. Does the failure message name the record and the fix, or say "certificate provisioning failed". Does it support the `accounturi` parameter, which Let's Encrypt recommends so that a temporary domain hijack without the ACME account key cannot yield a certificate [R22].

An answer along the lines of "CAA is rare" is a statement about the vendor's test corpus.

11.4 TLS automation

Issuance runs on ACME [R6] and the constraints that bite are the rate limits, not the protocol [R21]. Onboard more than 300 new hostnames in three hours and a build-it-yourself implementation stalls on the new-order limit. Retry a single misconfigured hostname more than five times in an hour and that hostname is locked out for the remainder of the hour.

Ask whether issuance is fail-closed: does a host the control plane has never approved still trigger an ACME order. If it does, one scripted crawl of the vendor's edge sending random SNI converts into orders against a shared account budget. Ask whether the certificate cache and the per-host issuance lock are shared across edge instances, because two edges racing on the same new hostname is how duplicate-certificate limits get hit at exactly the moment you scale out. Ask what happens with no shared store configured — if the honest answer is one order per instance, that is a fine answer, but it should be given rather than rounded up.

11.5 After the connection goes live

Connection is an event. A custom domain is a subscription to someone else's DNS zone, and that zone changes without telling you.

Ask what happens when a customer's record is deleted six weeks later. Ask whether drift is detected by reading public resolvers or inferred from the vendor's own write log. Ask whether "verified" means the vendor re-read DNS or that its API call returned 200. That distinction is not pedantic: a verification pipeline that trusts its own writes manufactures false positives in exactly the component whose job is to prevent them, which is why a test adapter that accepts any credential must not be reachable from a production registry (§9.5).

Ask for the webhook event list, the signing scheme, and whether delivery is backed by a durable outbox or fired in-process and forgotten. Ask whether drift alerting is on by default or ships in shadow mode.

11.6 Agent and API surface

If an agent will ever touch this, the questions sharpen.

Is there a machine-readable API description served by the running service, with contract tests that fail when the served document and the live handlers disagree, or a hand-maintained docs site. If there is an MCP server, are the tools annotated so a host knows which are read-only and which are destructive. Are scopes enforced in the request path or declared and ignored — and which credential classes do they bind, because a scope model that binds delegated agent tokens and not integrator API keys means an integrator who hands its own key to an agent gets none of it (§8.2). If the product can spend money, there must be a spend cap enforced at purchase time by an atomic check-and-record, and a fail-closed default when no cap is set; a cap checked before an order and recorded after it has a double-spend race (§8.4).

Ask which of these surfaces are on by default. Ours are not: the delegated agent authorization surface and registrar purchasing both ship off and must be enabled per deployment (§8.3).

11.7 Credentials and data

Where does a customer's DNS provider token live after the flow completes. A one-shot token used for a single write and never persisted is the safest posture and also the one that quietly removes the ability to re-apply records later. Both answers are defensible. The vendor should state which one it made, and state it per rail, because the answer differs by rail in any system that has more than one (§9.1).

Confirm that no API key reaches the browser and that the embed is authenticated by a short-lived server-minted token. Confirm PKCE with S256 is mandatory rather than optional, and that a callback origin allowlist denies on empty rather than defaulting to permissive. Ask what leaves your network by default, including telemetry, and whether the observability integrations no-op when unconfigured.

11.8 Commercial continuity

This row is new in this edition, and it exists because of a specific sequence of dates.

Under the Domain Connect specification the DNS provider is the only party that writes records, runs discovery, and issues or revokes tokens. A service provider can request. A DNS provider can decline every apply and revoke every token, unilaterally [R12]. That asymmetry has been litigated. GoDaddy revised its terms of use in a way that cut off aggregator DNS access; Entri sued in the Eastern District of Virginia (No. 1:24-cv-00569), filing the complaint on 2024-04-08 [R37]. Judge Trenga denied the motion to dismiss on 2024-10-10 on a negative-tying theory under Sherman Act §1 [R36], a ruling reported the following day [R39]. The parties settled with prejudice on 2025-02-25 [R40], and a multi-year agreement restoring access was announced on 2025-06-18 [R41]. Eight months separated the ruling from the access agreement, and fourteen months separated the complaint from it. Access came back through a private bilateral agreement between two companies. Figure 4.2 shows that timeline against the standard's own.

The evaluation questions follow directly.

- **Which of the rails you are being sold depend on a bilateral arrangement?** Ask for that broken out by provider, the same way you asked for coverage by mode in §11.1.

- **What fraction of your own domain table sits behind those providers?** A rail that covers 2% of your customers lapsing is an incident. A rail that covers 40% of them lapsing is a re-platforming.
- **What does the product do on the morning access lapses?** The only acceptable answer is that it degrades to something nobody else can switch off, which in this category means correct provider-specific manual records and the same verification. If the answer is an error state, the continuity risk is a product-availability risk.
- **What is the notice period, in the contract, for a rail being withdrawn?** Not the vendor's contract with you — the vendor's exposure upstream. You will usually not get an answer. The absence of one is itself information.

This applies to us. Our own Domain Connect rails sit on the same structural asymmetry as everyone else's, and 38 of 63 providers in our census are routed to the manual floor precisely because no arrangement exists to route them anywhere else.

11.9 Exit

Also new, and it is the cheapest section to skip and the most expensive to skip.

The exit cost of a DNS-configuration vendor and the exit cost of an edge vendor are not the same order of magnitude, and the difference is worth understanding before you sign either.

A DNS-configuration vendor writes records into your customer's zone. Those records survive the vendor. If you leave, the records already written keep resolving; what you lose is the ability to write new ones and to verify the old ones. Ask for an export: the record set per connection, the verification baseline, the drift history, in a format you can re-import. Ask what happens to stored Domain Connect grants on termination — they should be revertible or revoked, not orphaned.

An edge vendor is in the request path. If you leave, every customer's DNS record has to be changed to point somewhere else, which puts you back through the onboarding problem this entire paper is about, once per customer, on your timeline rather than theirs. Two practical questions follow. What is the TTL on the record customers point at the edge — that TTL is the length of your migration window, and it is much easier to lower before you need it than during a cutover. And is the certificate portable — it is not; you will re-issue, against the same ACME rate limits in §11.4, for every hostname at once, which is the one scenario where the 300-orders-per-3-hours account limit is guaranteed to bind [R21].

Then the structural hedges. Is the source available and self-hostable, and under what licence. Is the API described by a document the running service serves itself, so that a replacement can be written against something checkable. Is there a single-tenant or self-hosted deployment mode, and has anyone outside the vendor run it.

Prefer the reversible option when the numbers in Table 10.1 are close. They usually are.

11.10 Scoring it

Score each criterion 0 to 4, multiply by a weight you set, and sum. The weights below are a starting point, not a recommendation; the correct weights depend on your customer mix, and the exercise of arguing about them internally is worth more than the total.

#	Criterion (§)	What a 0 looks like	What a 4 looks like	Weight	Score
1	Coverage by mode (§11.1)	One headline provider count	Routed / working / one-click, as a file, mapped to your own domain table	5	___
2	Apex (§11.2)	"Yes", no exceptions list	Per-provider list, realization named, refusal reported at check time	4	___
3	CAA (§11.3)	"CAA is rare"	Pre-flight read, conflict named in the error, `accounturi` supported	3	___
4	TLS automation (§11.4)	Any host can trigger an ACME order	Fail-closed host policy, shared cache and per-host issuance lock, limits stated per deployment mode	4	___
5	Post-go-live drift (§11.5)	Inferred from the vendor's own write log	Read from public resolvers, signed webhooks, durable outbox	3	___
6	Agent and API surface (§11.6)	Scopes declared, not enforced	Enforced in the request path, per credential class, spend gated atomically	2	___
7	Credentials and data (§11.7)	Token lifetime unspecified	Stated per rail, nothing in the browser, S256 mandatory, allowlist denies on empty	4	___
8	Commercial continuity (§11.8)	Rails presented without naming which rest on an agreement	Per-provider dependency disclosed, documented degradation to a floor nobody can switch off	4	___
9	Exit (§11.9)	No export, edge in the path, TTL never discussed	Full export, source available, migration window sized in advance	3	___

Table 11.1 — A weighted scorecard for custom-domain vendors. Apply it to any vendor including us. Scores are 0–4; weights are yours to set, and the defaults shown are a starting point rather than a recommendation.

Two cautions about the total. First, it is meaningless unless row 1 was computed against your own domain table rather than the vendor's marketing page — a strong score on rows 2 through 9 describes a system that is good at the customers it can reach. Second, rows 1 and 8 are better treated as gates than as weights. A vendor can score well overall and still fail on the single row that covers most of your customers, and a weighted average is designed to hide exactly that.

§12 Conclusion and disclosure

The pricing floor in this category is set by services that terminate TLS and reverse-proxy arbitrary hostnames: roughly ten to thirty cents per hostname per month [R23][R24][R25]. Those services solve TLS and routing. They do not touch the DNS configuration step, and they do not claim to. Everything priced above that floor is priced on the other job — getting a record written correctly, on the first attempt, in a panel neither party controls — and that job has no settled price at all, which is why the same widget sells for $249 a month before the first domain at one vendor [R26] and nothing under fifty domains at another [R32]. The spread between those floors is a claim about provider coverage and the arrangements behind it, not a measurement of what the work costs (§6.2, Figure 6.1).

The checklist in §11 is one question asked nine ways: how much of the DNS problem does this vendor absorb, and how does it behave on the day it cannot.

Disclosure. We build customdomain.ai, and the architecture in §7 through §9 is ours. The checklist is published in vendor-neutral form because it is the one we run on ourselves, and the rows we score worst on are stated here rather than left for a reader to find.

Our census routes 63 providers — every name on the incumbent's published provider list but World4You — to a mode. 38 of 63 providers are routed to the manual floor and receive provider-specific copy-paste instructions, as does any provider not on the list at all. The remaining 25 are routed to an automated path: 16 to a customer-supplied API token, 6 to native provider OAuth, 3 to Domain Connect. Two of those 25 are routing rather than working. GoDaddy is one of the three Domain Connect entries, and as of 2026-07-28 it served 0 of our 18 published templates, because it hand-curates its template set and we have not been onboarded. DigitalOcean is the one of the six OAuth entries whose application is still unregistered. So 25 is what the census routes and 23 is what works today, and the number to design around is neither: at most 7 of the 63 are one-click, and only after per-provider application registration, which is an owner action rather than a code path (Figure 1.2).

Two further limits belong in the same paragraph. The Secure and Power surfaces are not products we sell today, and one behaviour in §9.6 depends on Secure being enabled — with it off, the per-host issuance back-off has no verdict to read and the shipped behaviour is one ACME order per cache-miss handshake. Our own homepage still says "63 DNS providers, auto-configured" without the breakdown above, and that is the failure row §11.1 exists to catch.

The point of stating this plainly is not modesty. A majority of DNS providers have no automated write rail available to any third party, and that is the shape of the ecosystem rather than a competitor's weakness. It applies to us. It applies to every vendor in §6. Any paper claiming otherwise is describing a provider list, not a customer base. The design consequence is the one in §9.3: every automated rail has to degrade to something nobody else can switch off, that floor is manual copy-paste, and a system that treats the floor as an error state has mistaken its own coverage for the world's.

And there is a version of this where the right answer is to build it yourself. If your customers are technical, if you can require a subdomain and refuse the apex, and if your onboarding volume never approaches the ACME account limits, then the nine line items in §10.1 reduce to three and the seven in §10.2 mostly do not fire. Table 10.1 will tell you that faster than a sales call will. Run row 1 of the scorecard against your own provider table before you run it against anyone's deck, including ours.

Appendix A — The 63-provider census by mode

The census is a file in the connect engine, `services/connect/census.go`. It holds one row per DNS provider, and each row records the best write rail that provider is routed to. The population is 63 providers: every name on the incumbent's published developer provider list [R29] except World4You. Counts in this appendix are as measured on 2026-08-02.

One distinction governs everything below. A census row says which rail the engine would choose for a domain on that provider. It does not say that the rail has a credential registered, that the provider serves our templates, or that the end user reaches a single button. Those are three further conditions, and the paper reports three different numbers because of them (§1, Figure 1.2).

The four modes

Mode	Providers routed	Who performs the write	What the mode requires of the end user
Domain Connect	3	The DNS provider	Approve the apply at their own provider
Native provider OAuth	6	The platform, with a one-time token	Authorize once, inside the widget
BYO scoped API token	16	The platform, with the tenant's token	Mint a scoped token in the DNS panel and paste it
Manual floor	38	A human, by hand	Type the records into the provider's control panel
Total	**63**		

Table A.1 — The census by mode. The mode names are the census's own; the "who writes" column is the reason the modes are not interchangeable, and it is the same axis §4 uses to classify vendors (Figure 4.1).

Figure 1.1 renders Table A.1 as a unit dot matrix, one mark per provider, which is the honest way to show that three in five providers sit on the manual floor.

Which providers are named, and which are not

v1 names the Domain Connect and OAuth sets in full, because both are small enough to enumerate and both are load-bearing claims. It does not enumerate the API-token set or the manual floor. Rather than reconstruct those lists, this appendix states the gap.

Mode	Named in the source material	Not named
Domain Connect (3 of 63)	GoDaddy, IONOS, NameSilo	— all three are named
Native provider OAuth (6 of 63)	Cloudflare, DigitalOcean, DNSimple, Netlify, Vercel, WordPress.com	— all six are named
BYO scoped API token (16 of 63)	none	all 16
Manual floor (38 of 63)	Namecheap	the other 37

Table A.2 — Provider names by mode. The source material enumerates 9 of 63 rows. The remaining 54 are counted but not listed, and the census file is the authority for them.

Two providers appear in the paper by name without being placed in a mode, and it is worth saying which is which. Namecheap is on the manual floor by decision, not by absence of an API: its API replaces the entire zone in a single call, so an automated write is a whole-zone overwrite of a stranger's DNS. Route 53 appears in §3 and §7 as a capability case — no ALIAS record type, only an `<AliasTarget>` element that cannot point at an arbitrary external hostname, so its CNAME-flattening flag is false — and the source material does not state which mode routes it. A capability flag and a mode are different columns.

Anything not on the list at all classifies to manual by default. That is the design rule, not a gap: an unrecognized domain gets provider-generic copy-paste records and the same verification every automated rail gets (§7).

Routed, working, one-click: three counts of three different things

Count	Value	What it means
Routed to an automated path	25	16 API token + 6 OAuth + 3 Domain Connect
Not known-broken today	23	The routed count minus GoDaddy and DigitalOcean
One-click out of the box	≤ 7	NameSilo's Domain Connect sync plus the six OAuth providers, and only after per-provider app registration

Table A.3 — The three counts, and why they differ. Figure 1.2 is the same data as a chart; it exists because the three numbers are routinely quoted as one.

The two subtractions behind 23 are specific and dated. GoDaddy is one of the three Domain Connect rows, and as of 2026-07-28 it served 0 of our 18 published templates, because it hand-curates its template set and we have not been onboarded. DigitalOcean is the one of the six OAuth rows whose provider app is still unregistered.

The ceiling of 7 is lower than 23 because it counts rails that finish without an owner action. The Domain Connect sync rail ships disabled and is re-enabled only for providers measured to serve the template catalog, which excludes GoDaddy and IONOS for the same reason — both hand-curate and serve none of ours. The async rail ships with no partner credential. Each of the six OAuth rails needs a client id and secret registered with that provider before it runs at all, which is an owner action rather than a code path. On the paper's own figures, IONOS is therefore routed to a rail that cannot complete out of the box for the same reason GoDaddy's cannot, while the 23 removes GoDaddy and DigitalOcean only. Read 23 as the upper bound on what is not known to be broken, and 7 as the count of rails that finish unattended.

The homepage claim "63 DNS providers, auto-configured" carries none of this breakdown. That is the first row of the evaluation checklist failing against its own author, which is why it is printed here rather than left out.

Disambiguating "38"

The number 38 appears in this paper in two unrelated senses. Neither is a restatement of the other, and a bare 38 should never be read without its population.

Sense	Population	Statement
Manual floor	63 providers in the census	38 of 63 providers are routed to the manual floor
Adapter survey	38 providers we built adapters against	6 of those 38 expose a provider-hosted OAuth flow usable for third-party DNS writes

Table A.4 — The two 38s. They coincide numerically and describe different sets: one is an output of the census, the other is the size of the adapter set surveyed for OAuth support.

The six providers found in the adapter survey are the same six that Table A.2 lists under native provider OAuth. That is the only overlap between the two rows.

Appendix B — The RFCs this paper leans on

Seven documents carry the mechanical claims in §3 and §7. Each is cited where it is used; this is the one-line version, and the section column says where the constraint actually bites.

Document	What it governs	Where it bites
RFC 1034 §3.6.2 [R1]	A name that holds a CNAME should hold no other data	The apex cannot take a CNAME, because it must carry SOA and NS (§3, Figure 3.2)
RFC 1035 §3.3.14 [R2]	TXT RDATA is one or more character-strings, each at most 255 bytes	Long TXT values must be split into multiple strings, and providers disagree about who does the splitting (§3)
RFC 2181 §10.1 [R3]	Restates the CNAME restriction as a hard prohibition	Turns the apex problem from a recommendation into a rule vendors must design around (§3)
RFC 2308 §5 [R4]	Negative answers are cached, for the lesser of the SOA MINIMUM field and the SOA record's own TTL	There is no propagation event; querying too early lengthens the wait (§3, Figure 3.1)
RFC 7208 §3.2 [R5]	Exactly one SPF policy record per domain	A second record yields permerror and costs delivery under a DMARC quarantine or reject policy (§3)
RFC 7208 §4.6.4 [R5]	The 10-lookup limit on SPF evaluation	Bounds the correct operation, which is merging an `include:` into the existing string rather than adding a record (§3)
RFC 8555 [R6]	ACME: how certificates are ordered, challenged and issued	Issues certificates; configures none of the records that route traffic. DNS-01 is the DNS-writing problem under a different name (§7, Figure 7.3)
RFC 8659 §3 [R7]	CAA: how a CA locates the relevant RRset by walking up the tree, and what `issue` and `issuewild` mean	A customer's pre-existing CAA record blocks issuance inside the CA, minutes after every DNS record verified correctly (§7)

Table B.1 — The seven documents and the failure each one explains. CAA checking is mandatory for publicly trusted CAs through the CA/Browser Forum Baseline Requirements [R17] rather than through RFC 8659 itself; the RFC defines the record, the Baseline Requirements make reading it compulsory.

One operational note that belongs with RFC 8659 rather than in it: Let's Encrypt states that a DNS provider does not need to support the CAA record type, only to return NOERROR for unknown query types [R22]. Providers that answer SERVFAIL for unknown types break issuance while holding no CAA record at all.

The agent surface described in §8 leans on a second, unrelated set. It is off by default and gated per deployment, so these documents describe a surface an operator switches on, not one that runs out of the box.

Document	What it governs
RFC 7591 [R8]	Dynamic client registration: how an agent client registers with no pre-shared secret
RFC 7009 [R9]	Token revocation: how a grant is withdrawn
RFC 9728 [R10]	Protected resource metadata: how a client discovers which authorization server guards a resource
RFC 8414 [R11]	Authorization server metadata: how the register, authorize, token and revoke endpoints are advertised

Table B.2 — The OAuth-family documents behind the delegated-agent flow in §8 (Figure 8.1).

References

One numbering scheme runs across the whole paper. Every in-text citation is an [Rn] into this list, and every URL lives here rather than in body prose.

Three labels are used deliberately. **Attributed claim** marks a figure whose only primary source is the party that benefits from it, including competitors' case studies; it is reproduced because it is the best public evidence available, not because it is independent. **Uncited** marks a widely-repeated number whose primary source could not be traced. **No permalink recorded** marks a document the source material identifies but does not link.

First-party measurements are dated at the point of use rather than listed here: the negative-TTL readings taken with `dig` on 2026-08-02 (§3), the census file `services/connect/census.go` (Appendix A), and the template-catalog check of 2026-07-28 (§1).

Standards and RFCs

- **[R1]** P. Mockapetris, "Domain Names — Concepts and Facilities", RFC 1034, November 1987. §3.6.2. https://www.rfc-editor.org/rfc/rfc1034
- **[R2]** P. Mockapetris, "Domain Names — Implementation and Specification", RFC 1035, November 1987. §3.3.14. https://www.rfc-editor.org/rfc/rfc1035
- **[R3]** R. Elz, R. Bush, "Clarifications to the DNS Specification", RFC 2181, July 1997. §10.1. https://www.rfc-editor.org/rfc/rfc2181
- **[R4]** M. Andrews, "Negative Caching of DNS Queries (DNS NCACHE)", RFC 2308, March 1998. §5. https://www.rfc-editor.org/rfc/rfc2308
- **[R5]** S. Kitterman, "Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1", RFC 7208, April 2014. §3.2, §4.6.4. https://www.rfc-editor.org/rfc/rfc7208
- **[R6]** R. Barnes, J. Hoffman-Andrews, D. McCarney, J. Kasten, "Automatic Certificate Management Environment (ACME)", RFC 8555, March 2019. https://www.rfc-editor.org/rfc/rfc8555
- **[R7]** P. Hallam-Baker, R. Stradling, J. Hoffman-Andrews, "DNS Certification Authority Authorization (CAA) Resource Record", RFC 8659, 2019. §3. https://www.rfc-editor.org/info/rfc8659/
- **[R8]** "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591. https://www.rfc-editor.org/rfc/rfc7591

- **[R9]** "OAuth 2.0 Token Revocation", RFC 7009. https://www.rfc-editor.org/rfc/rfc7009
- **[R10]** "OAuth 2.0 Protected Resource Metadata", RFC 9728. https://www.rfc-editor.org/rfc/rfc9728
- **[R11]** "OAuth 2.0 Authorization Server Metadata", RFC 8414. https://www.rfc-editor.org/rfc/rfc8414
- **[R12]** Domain Connect specification, `Domain-Connect/spec`. The specification text is the source for the claim that the DNS provider is the only party that writes records, runs discovery, and issues or revokes tokens. https://github.com/Domain-Connect/spec
- **[R13]** Domain Connect, "DNS providers", retrieved 2026-08-02. Listed under "Implementation is Live": IONOS, Cloudflare, Domain Chief, Glauca Digital, GoDaddy, NameSilo, Plesk, Vercel, WordPress.com. The page lists only implementations it has been told about and carries no last-updated date. https://www.domainconnect.org/dns-providers/
- **[R14]** IETF, `draft-ietf-dconn-domainconnect-03`, dconn working group, revision dated 2026-07-03. Co-authors P. Kowalik (DENIC), A. Blinn, J. Kolker (GoDaddy), S. Kerola (Cloudflare); milestone targets IESG submission in December 2026. https://datatracker.ietf.org/doc/draft-ietf-dconn-domainconnect/
- **[R15]** `Domain-Connect/Templates` repository. Counted via the GitHub git-trees API, untruncated, on 2026-08-02: 998 root-level templates from 596 distinct provider IDs; Entri publishes 77, GoDaddy 27 across its `godaddy` and `secureserver` provider IDs. https://github.com/Domain-Connect/Templates
- **[R16]** Model Context Protocol specification, revision 2025-06-18. https://modelcontextprotocol.io/specification/2025-06-18
- **[R17]** CA/Browser Forum, Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates. The instrument that makes CAA checking mandatory for publicly trusted CAs. https://cabforum.org/
- **[R18]** "Domain-based Message Authentication, Reporting, and Conformance (DMARC)", RFC 7489, March 2015. Cited in §3.6 for the cost of an SPF permerror under a quarantine or reject policy. https://www.rfc-editor.org/rfc/rfc7489
- **[R19]** `libdns/libdns`. The `GetRecords` / `SetRecords` / `DeleteRecords` provider interface that the direct-write adapter contract in §7.4 is shaped after. https://github.com/libdns/libdns

Vendor documentation and pricing pages

- **[R20]** Google, "Email sender guidelines". Requirements effective 2024-02-01 for senders above 5,000 messages per day to personal Gmail accounts: SPF and DKIM on the sending domain, plus a DMARC policy, which may be `p=none`. https://support.google.com/mail/answer/81126
- **[R21]** Let's Encrypt, "Rate limits", page updated 2025-06-12. 300 new orders per account per 3 hours; 50 certificates per registered domain per 7 days; 5 certificates per identical identifier set per 7 days; 5 authorization failures per account per hostname per hour; 100 identifiers per certificate. https://letsencrypt.org/docs/rate-limits/
- **[R22]** Let's Encrypt, "Certificate Authority Authorization (CAA)". Source for the NOERROR-on-unknown-query-types requirement and for the `accounturi` parameter. https://letsencrypt.org/docs/caa/
- **[R23]** Cloudflare, "Cloudflare for SaaS — plans". 100 custom hostnames included on Free, Pro and Business; $0.10 per hostname per month above that to 50,000; custom certificates, mTLS and wildcard custom hostnames reserved for Enterprise. https://developers.cloudflare.com/cloudflare-for-platforms/cloudflare-for-saas/plans/
- **[R24]** Approximated. Roughly $0.20 per domain per month, $20 minimum, 400 GB bandwidth included; setup described as one A record pointed at a dedicated cluster IP. https://approximated.app/
- **[R25]** SaaS Custom Domains. $0.29 per domain with a 100-domain minimum and automatic discounts up to 50% as volume scales. https://saascustomdomains.com/
- **[R26]** Entri, pricing. Startup $249/month for 600 domains a year; Growth $749/month; tiers above Startup demo-gated. https://www.entri.com/pricing
- **[R27]** Entri, Connect product page, retrieved 2026-08-02. Markets "50+ DNS providers" and "75% of the market". https://www.entri.com/products/connect

- **[R28]** Entri, developer overview, retrieved 2026-08-02. States "60+ DNS providers with direct API login". https://developers.entri.com/connect/overview
- **[R29]** Entri, developer provider list, retrieved 2026-08-02. Enumerates 64 named providers across two tables carrying the same set. This is the list the 63-provider census in Appendix A is drawn from, less World4You. https://developers.entri.com/provider-list
- **[R30]** Entri, "Fourthwall case study". Attributed claim: nearly half of Fourthwall's support tickets related to DNS configuration; 15 minutes to an hour of back-and-forth per ticket; part-time contractors hired to absorb the load; a reduction of as much as 97% reported after adoption. Vendor-published, not independent research. https://www.entri.com/case-studies/fourthwall-case-study
- **[R31]** Entri, "Crisp case study". Attributed claim: five or six DNS tickets per day down to about one, an 83% reduction, with CEO Baptiste Jamin quoted directly. Vendor-published, not independent research. https://www.entri.com/case-studies/crisp-case-study
- **[R32]** Domainee, "Entri alternative 2026". Attributed claim from a competing vendor: 50 domains free, $0.20 per domain after, with the post conceding that Entri's widget is one of the best onboarding components in the market. https://domainee.dev/blog/entri-alternative-2026
- **[R33]** npm registry downloads API, package `entrijs`, week 2026-07-26 to 2026-08-01: 236,511 downloads. https://api.npmjs.org/downloads/point/last-week/entrijs
- **[R34]** IONOS Group SE, "IONOS secures strategic minority stake in Entri", press release, 2025-08-18. Minority stake of under 20%. No permalink recorded in the source material.
- **[R35]** IONOS Group SE, Consolidated Financial Statements 2025. Records €5,028k of additions at FVOCI, Level 3 — the only hard financial number in Entri's public record. No permalink recorded in the source material.

Legal filings

- **[R36]** Memorandum opinion and order denying the motion to dismiss, Entri LLC v. GoDaddy.com LLC, No. 1:24-cv-00569 (E.D. Va. Oct. 10, 2024) (Trenga, J.). Denial on a negative-tying theory under Sherman Act §1; the alleged tie "cuts off all aggregator services … from competing". The opinion is dated 2024-10-10; see [R39] for the reporting one day later, which is the date often quoted for the ruling itself. https://www.courtlistener.com/docket/68429734/entri-llc-v-godaddycom-llc/
- **[R37]** Docket, Entri LLC v. GoDaddy.com LLC, No. 1:24-cv-00569 (E.D. Va.). Complaint filed 2024-04-08; settled with prejudice 2025-02-25. https://www.courtlistener.com/docket/68429734/entri-llc-v-godaddycom-llc/

Reporting and analysis

- **[R38]** P. Kowalik, "Domain Connect: a missing piece in the DNS toolbox", APNIC Blog, 2025-12-24. Source for the claim of roughly twenty Domain Connect implementations covering 35% of the `.com` zone as of May 2024, and for the widely-repeated Microsoft 365 setup figures — six steps, 7 to 15 DNS entries, 16 help sites, 40 minutes of training, 50% abandonment. Uncited: the post cites CENTR for its renewal-rate figures but gives no source for the Microsoft 365 numbers. Treat them as an attributed estimate from a standards author. https://blog.apnic.net/2025/12/24/domain-connect-a-missing-piece-in-the-dns-toolbox/
- **[R39]** "Judge denies GoDaddy's motion to dismiss in antitrust lawsuit", Domain Name Wire, reporting dated 2024-10-11, on the opinion signed 2024-10-10 [R36]. Source of the quoted language as reported. https://domainnamewire.com/2024/10/11/judge-denies-godaddys-motion-to-dismiss-in-antitrust-lawsuit/
- **[R40]** "GoDaddy and Entri settle antitrust lawsuit", Domain Name Wire, 2025-02-25. https://domainnamewire.com/2025/02/25/godaddy-and-entri-settle-antitrust-lawsuit/
- **[R41]** "Entri regains DNS update access at GoDaddy under new agreement", Domain Name Wire, 2025-06-18. Announcement of a multi-year agreement restoring access, eight months after the ruling in [R36]. https://domainnamewire.com/2025/06/18/entri-regains-dns-update-access-at-godaddy-under-new-agreement/

- **[R42]** "Antitrust: GoDaddy.com sued for negative tying restriction", Virginia Lawyers Weekly, 2024-10-21. https://valawyersweekly.com/2024/10/21/antitrust-godaddy-com-sued-for-negative-tying-restriction/