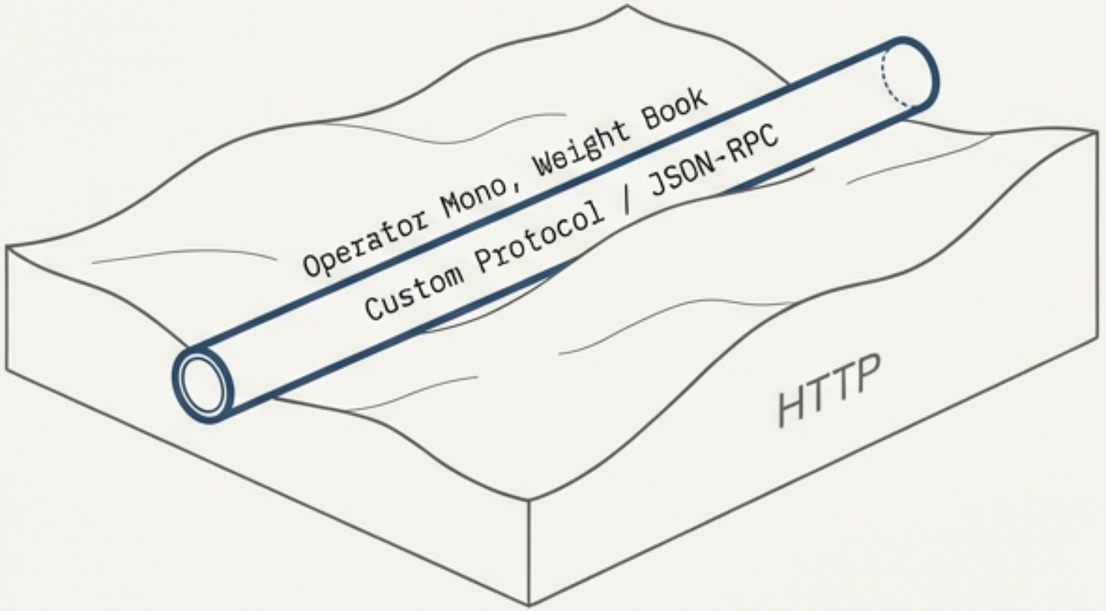
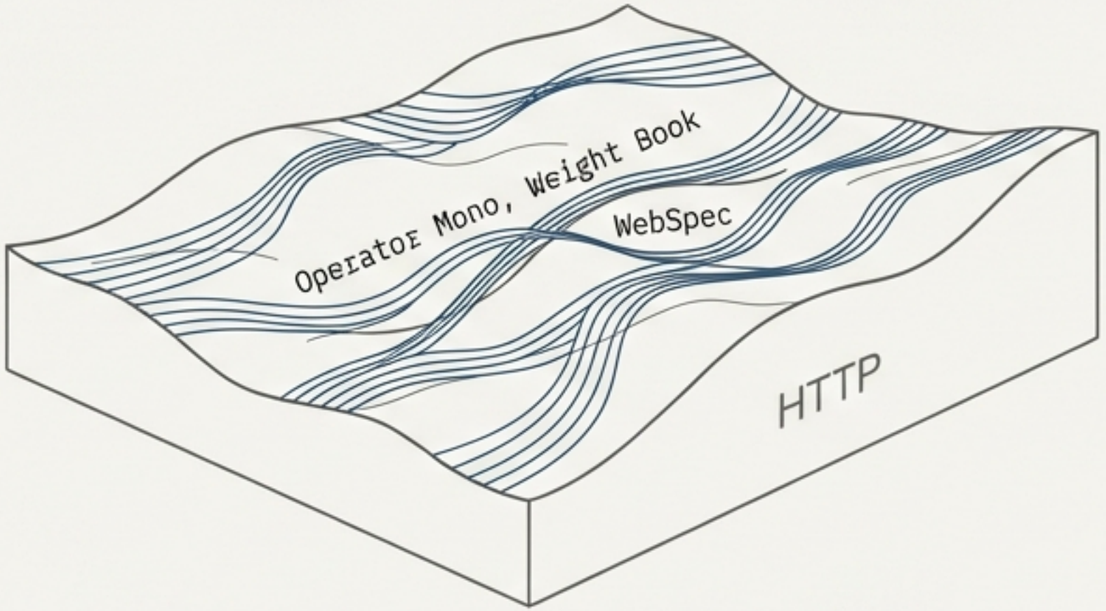
There are two ways to build on the web.



The Tunnel

- Treats the web as a dumb pipe.
- Reinvents discovery, negotiation, and security *inside* the pipe.
- Requires custom clients and SDKs to speak its language.
- Example: MCP (Model Context Protocol) optimizes for integration depth within a specific host application.



The Mesh

- Treats the web as the protocol.
- Inherits discovery, negotiation, and security *from* the web itself.
- Any standard web client is a native client.
- Example: WebSpec optimizes for universal interoperability across the entire ecosystem.

WebSpec's Meshing Principle: The web IS the protocol.

WebSpec is not a protocol built *on top of* the web; it is a disciplined application of the web's own native architecture. It maps every layer of the protocol directly to an existing, battle-hardened web primitive.

> "Every security boundary we inherit is code we don't write, bugs we don't create, and audits we don't need. The browser does the heavy lifting."

1. Inherited Security

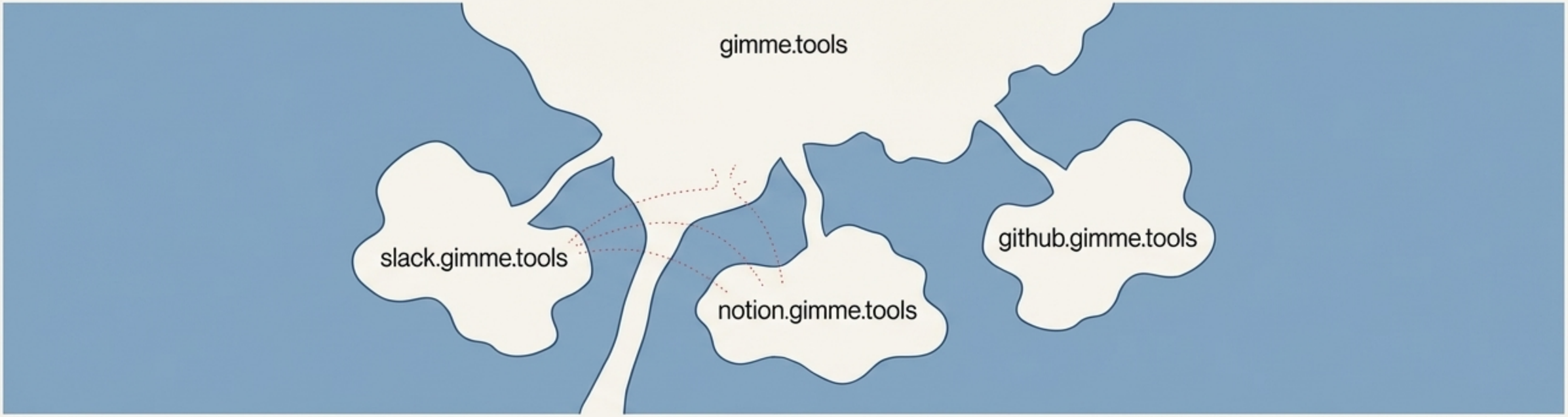
Subdomains and the Same-Origin Policy provide natural, robust isolation.

2. Semantic Clarity

HTTP's own grammar (methods, paths) provides a universal, self-documenting language for tools.

3. Universal Accessibility

Any client that speaks HTTP—from `curl` to a browser—is a native WebSpec client. No SDK required.



Pillar 1: Security is Inherited, Not Invented.

The Same-Origin Policy Gift

- Browsers enforce strict isolation between origins (protocol + host + port).
- By design, `slack.gimme.tools` cannot read cookies, localStorage, or data from `notion.gimme.tools`.
- A compromised or malicious service integration is contained within its own origin.

> "We didn't write this code—decades of browser security did."

What Services CANNOT Do

Attack	Prevented By
Read another service's localStorage	Same-origin policy
Intercept another service's requests	Origin isolation
Steal the master session token	HttpOnly + auth subdomain scoping
Impersonate another service	Audience-bound tokens (`aud` claim)

The Symmetry Extends Locally: Meshing with the OS.

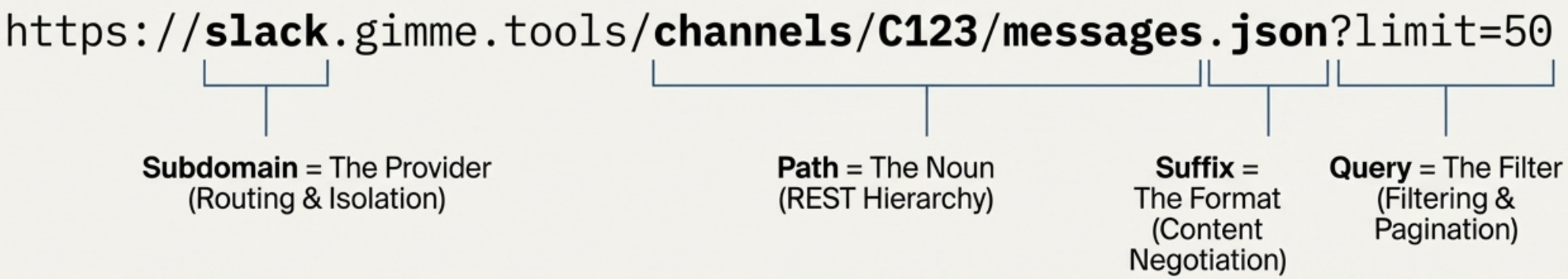
The web's subdomain isolation model has a direct parallel in Unix. The same security philosophy applies, inheriting decades of OS-level hardening instead of inventing a new security model for local tools.

Web Primitive		Unix Primitive	Role
Subdomain (`file.local.gimme.tools`)		Socket Path (`/var/run/gimme/file.sock`)	Isolation Boundary
Same-Origin Policy		Filesystem Permissions / Sandbox	Enforcement
Cookie Scope (`auth.gimme.tools`)		Socket Ownership (uid/gid)	Auth Boundary
Service Token (`aud: "file.local"`)		Socket Access + Token Validation	Capability Proof

Key Insight

Instead of a monolithic daemon running with full user permissions, local WebSpec services run as isolated, sandboxed processes. A compromised file service cannot execute code because the OS itself enforces the boundary. The security model is *infrastructure-defined*, not application-defined.

Pillar 2: The URL Grammar Creates Semantic Clarity



No Redundancy. The subdomain IS the provider, so it never appears in the path.
This simple rule cleans up the entire namespace.

❌ Before (Redundant & Ambiguous)	✅ After (WebSpec: Clean & Clear)
`api.service.com/slack/messages`	**`slack.gimme.tools`**`/messages`
`api.service.com/files?provider=gdrive`	**`gdrive.gimme.tools`**`/files`
`api.service.com/tasks/linear/LIN-42`	**`linear.gimme.tools`**`/issues/LIN-42`

Methods are Verbs. Permissions are Patterns.

Most APIs reduce HTTP to GET and POST. WebSpec restores their full semantic meaning, turning the method into the verb and the path into the noun. This makes permissioning an exercise in simple pattern matching, not parsing a custom vocabulary.

Method as Verb

Natural Language	WebSpec (Method + Path)
“read my messages”	`GET /messages`
“send this to Slack”	`POST /messages`
“update the task title”	`PATCH /issues/LIN-42`
“delete that file”	`DELETE /files/abc`
“does this file exist?”	`HEAD /files/abc`
“what can I do here?”	`OPTIONS /`

Permission as Pattern

The Old Way:

```
<scope>tool.slack.messages.read</scope>
```

The WebSpec Way:

```
scope: ["GET:/messages.*"]
```

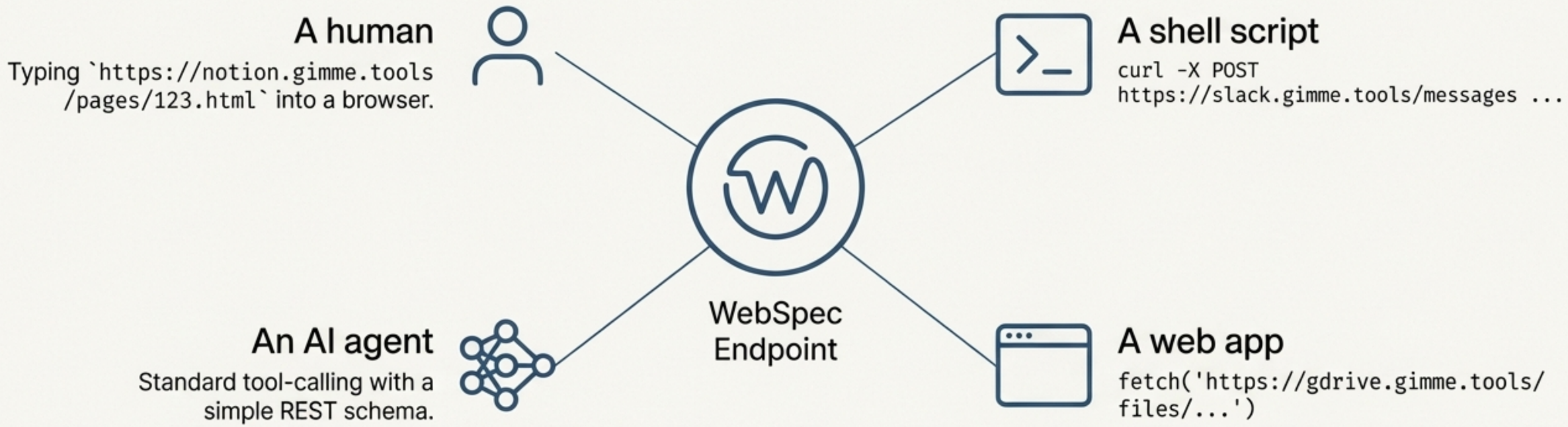
Gateway Enforcement Logic (pseudo-code):

```
// Six lines. Works for any service.
is_allowed = token.scopes.any? do |scope|
  method_match = (scope.method == request.method
         || scope.method == "*")
  path_match = File.fnmatch(scope.path, request.path)
  method_match && path_match
end
```

This demonstrates how authorization becomes trivial pattern matching, universal across all services.

Pillar 3: Universal Accessibility Means No Special Clients.

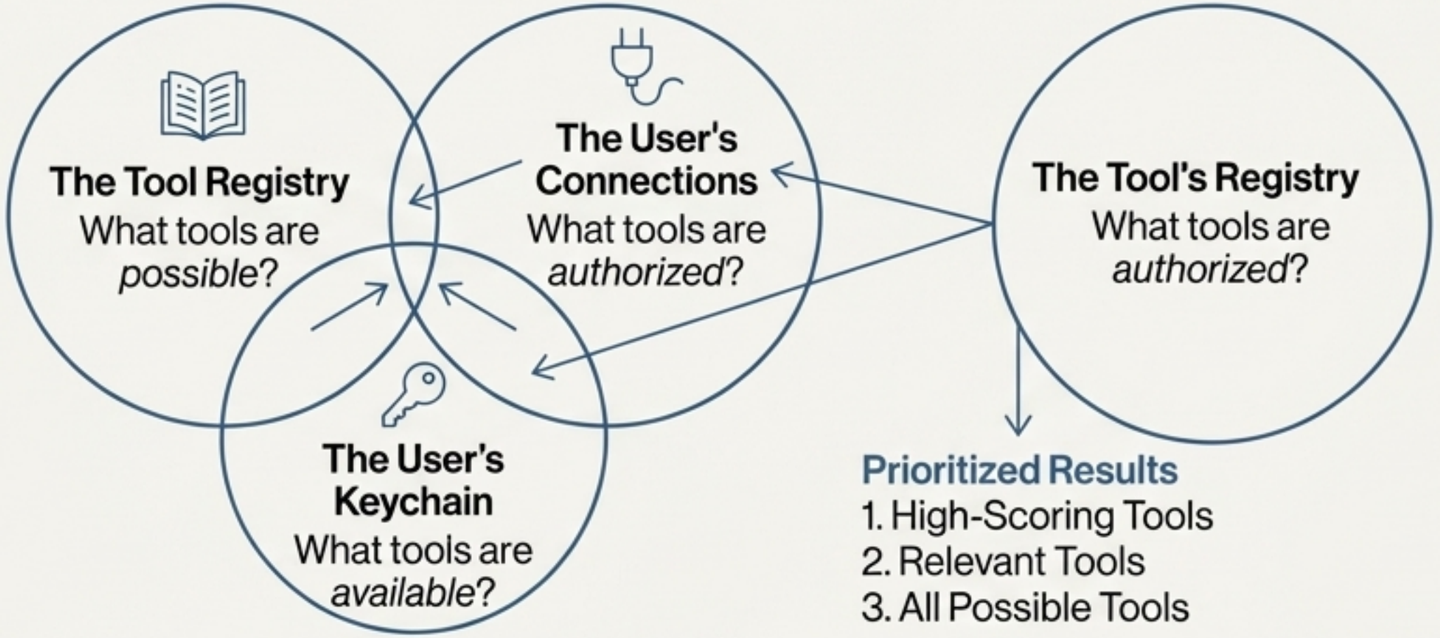
Because WebSpec is isomorphic to HTTP, it requires no special client libraries, SDKs, or protocol adapters. Anything that can speak the language of the web is a native WebSpec client.



The client problem is solved by dissolving the problem. Instead of forcing clients to learn a new protocol, WebSpec speaks the one protocol they already know. This dramatically shrinks the surface area for bugs and increases flexibility.

The Three-Way Join: Matching Intent to Capability.

Tool discovery isn't just a keyword search. WebSpec intelligently matches user intent against three distinct data sources to find the most relevant and frictionless path to execution.



Result Categories & UX

The algorithm scores and ranks results based on their connection status, leading to a prioritized UI.

Status	Description	UX
CONNECTED	OAuth already authorized	[Use Slack] ✅
KEYCHAIN	Found in password manager	[Connect with FaceID] 🔐
AVAILABLE	In registry, not yet used	[Sign Up / Connect] 🔗

This turns discovery from a cold start problem into a warm handshake.

The Gateway: Where Language Becomes Protocol.

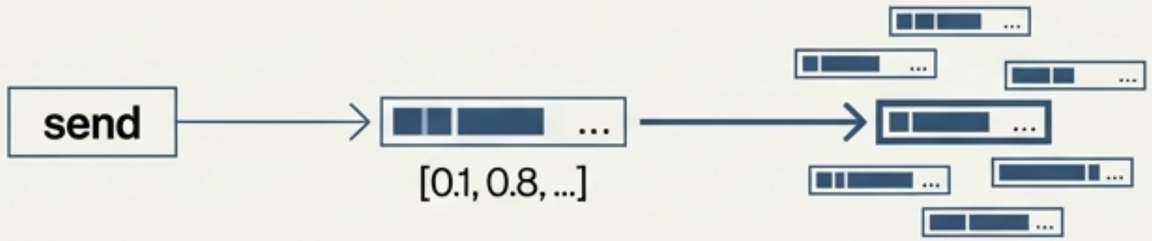
The Flow of Resolution

1. NLP Extraction

The gateway first normalizes the user's intent into canonical predicates and objects.

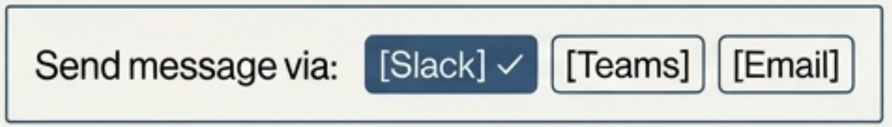
2. Semantic Search

These canonical terms are converted to vector embeddings and compared against the tool registry to find the closest semantic matches based on meaning, not keywords.



3. Ranking & Disambiguation

The results are ranked using the Three-Way Join score. If ambiguity remains (e.g., "send message" could be Slack, Teams, or Email), the user is prompted with the most likely options.



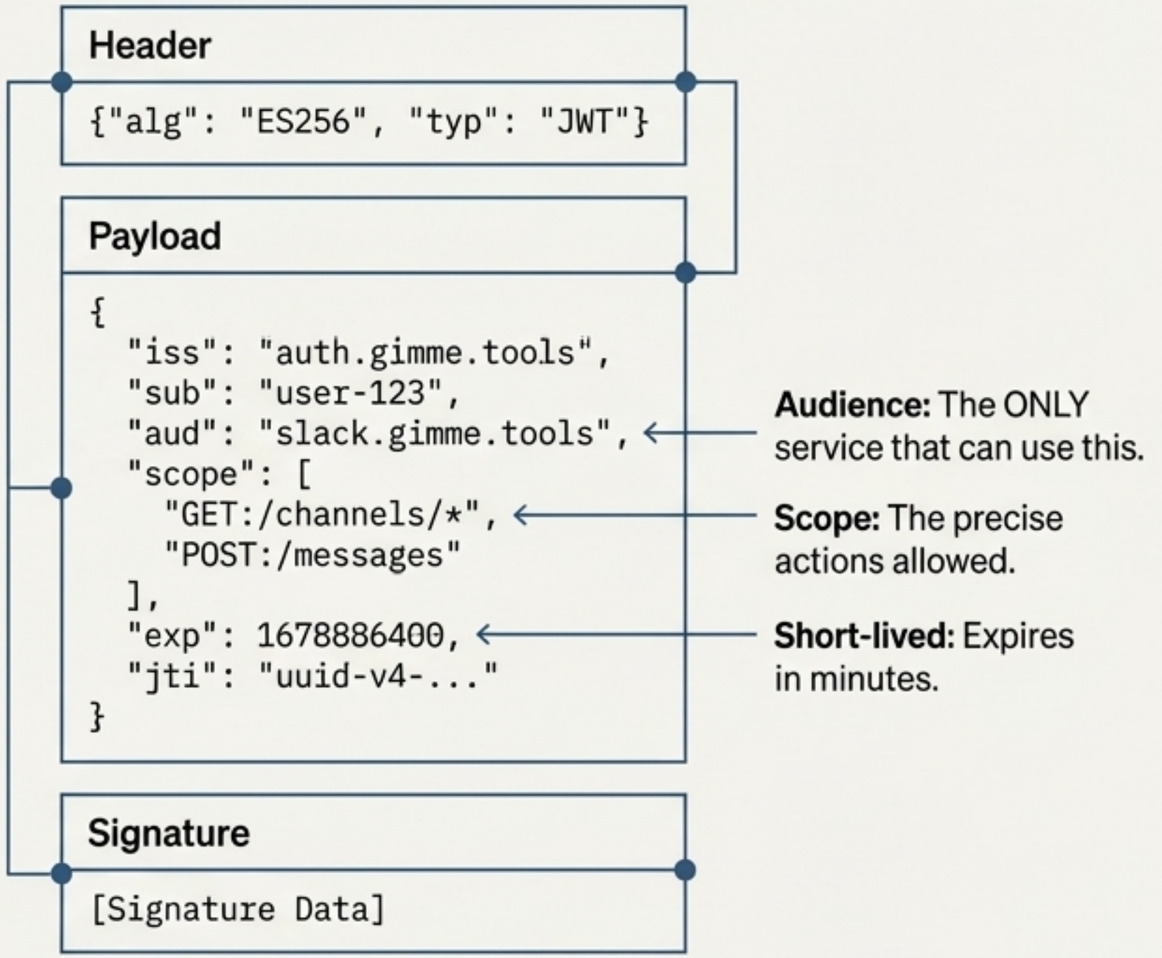
Authentication is Layered and Scoped.

WebSpec uses standard OAuth 2.0 but applies rigorous scoping at every layer, from the platform down to a single request. This ensures the principle of least privilege is always enforced.

Authorization Layers

Layer	What's Authorized	Revocation Granularity
Platform	`gimme.tools` (OAuth client)	Revoke all sessions
Session	`session-id + device-id`	Revoke one session
Invocation	Per-request confirmation	Real-time control

The Anatomy of a Secure Service Token



The `aud` claim is the security linchpin. It binds the token to a single subdomain. A leaked token is useless anywhere else.

From Protocol to Philosophy

The structure of a permission system is not an arbitrary technical choice. It is a reflection of how we structure relationships and trust. The questions that define permission scoping are the same questions that define relational intimacy.

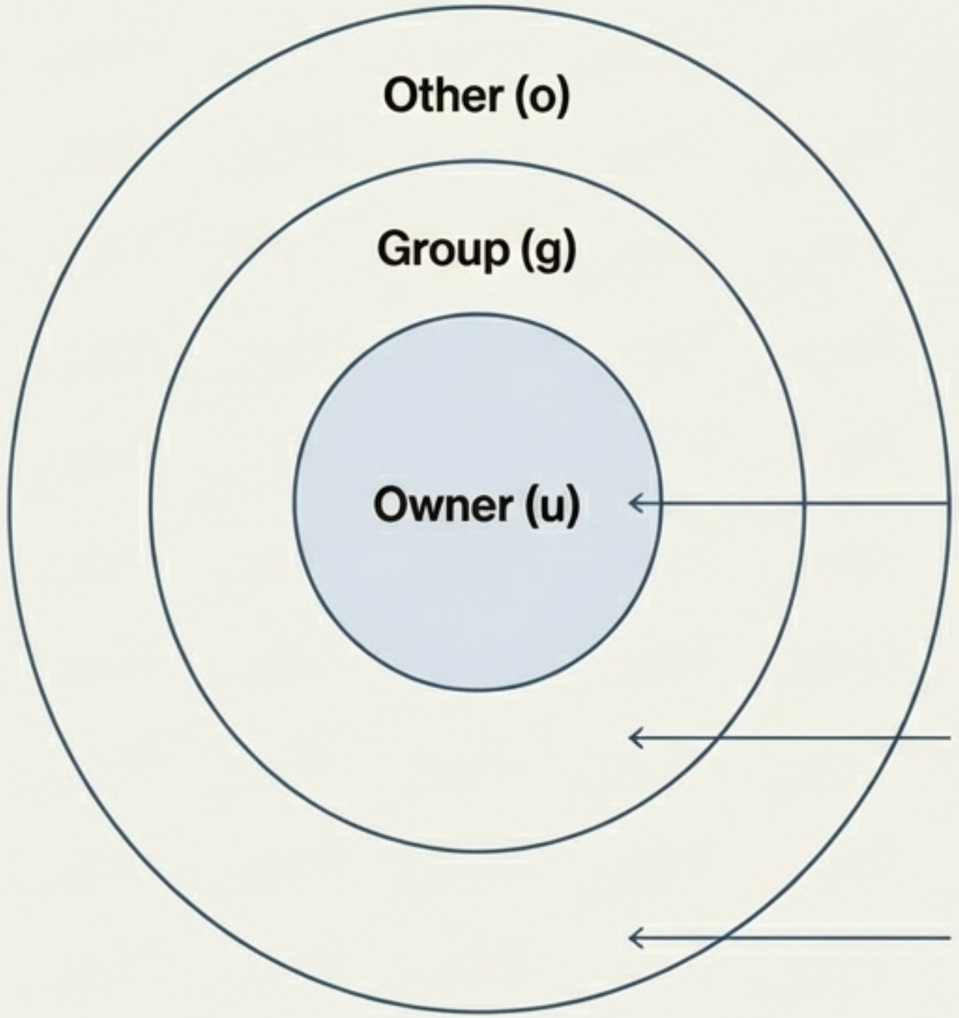
Who can know me? -> `read`

Who can change me? -> `write`

Who can act as me? -> `execute`

Permission systems are codified love boundaries.

Permissions as Trust Gradients: The Philía Model.



Philautia (Self-relation)
What I permit myself.

Philia (Friendship)
Mutual trust with a chosen circle.

Agape (Universal Love)
What I offer freely to all.

Operations as Vulnerability

Permission	Love Expression	What You're Granting
Read (`r`)	Being Known	Vulnerability to observation
Write (`w`)	Being Changed	Vulnerability to modification
Execute (`x`)	Acting As You	Trust to embody your agency

Common Permission Patterns as Relational Statements

The numeric modes of Unix permissions are not just numbers; they are concise statements about trust and boundaries.

Mode	Numeric	Relational Statement
`rw-------`	600	**"Only I may know and change myself."** (Healthy Self-Containment)
`rw-r--r--`	644	**"I am open to being known, but only I may change myself."** (Public Presence, Private Agency)
`rwxr-xr-x`	755	**"I offer myself to be known and used by all, but only I may change my core."** (A well-defined public utility)
`rwxrwx---`	770	**"My chosen circle and I share full intimacy; the world is outside."** (A Private Collaboration)
`rwxrwxrwx`	777	**"I have no boundaries."** (Pathological or Saintly?)

HTTP Methods as Relational Verbs

Method Operator Mono, Weight Book	**Relational Act**	**Essence**
GET	The **Desiring Gaze** (Eros)	Wants to **know**, to consume the body of the other.
POST	The **Generative Act** (Storge)	**Brings into being**. Creates something that did not exist.
PATCH	The **Nurturing Act** (Storge)	**Adjusts, heals, grows**. Respects what exists while fostering change.
PUT	The **Covenantal Act** (Pragma)	**Replaces entirely**. A commitment to a complete new state.
DELETE	The **Act of Release**	**Lets go**. A necessary ending for a new beginning.
OPTIONS	The **Gift of Self-Disclosure** (Agape)	“Here is everything I can offer you. I ask nothing in return.”

Security is the Discipline of Healthy Boundaries.

We did not invent permission systems from pure logic. We built them from the structures of human relating—because those are the structures we understand. A permission structure that feels wrong probably *is* wrong, because it encodes a relational pattern we recognize as unhealthy. Tiempos Text Pro.

Security Concept	Relational Equivalent
Firewall	Knowing where you end and others begin.
Principle of Least Privilege	Not giving more access than a relationship warrants.
Token Revocation	“This relationship has changed; access is withdrawn.”
Scope Limitation	“You may know this part of me, but not all of me.”

When we design these systems, we might ask:

> What kind of love does this enable?