Beyond SaaS: Delegate and the Substrate Layer for Runtime-Synthesized Applications

Daniel Phillips, Andrea Phillips
Controlled Mayhem (Applied AI Lab) — controlledmayhem.com
May 2026

"We build systems that move from idea to production, not systems that stay in demos."
Controlled Mayhem, lab principles.

Abstract

Significant categories of software-as-a-service are being displaced. As large language models reach the capability threshold at which they can synthesize purpose-built behavior on demand from natural-language intent, the persistent multi-tenant web application is, for a growing class of use cases, being replaced by what we call ephemeral software: applications that are constructed at runtime against shared primitives, exist for the duration of a user's need, and are not retained as artifacts. Ephemeral software does not require a marketing site, a sign-up flow, a settings page, or a long-lived database. It requires a substrate of callable primitives and a layer of accountable action over the services those primitives reach.

This paper argues that the missing piece of the post-SaaS computing stack is the accountable primitive layer: the substrate that lets a language model take action in the world on behalf of a human, with credentials the model never sees, scopes the human controls, and an audit trail the human can read. We characterize this substrate as composing three types of primitive: mediated primitives, where the principal holds the upstream account and the substrate brokers access; first-party primitives, where the substrate holds the upstream account and the principal interacts only with the substrate; and native primitives, owned outright by the substrate. We present Delegate, a working implementation that exercises the mediated case at scale and demonstrates the architectural path to the first-party case. Delegate exposes a fixed set of seven meta-tools over a single Streamable HTTP endpoint, mediating access to a server-side catalog of integrated services through a discovery-then-execution pattern that decouples agent context cost from catalog size. The architecture composes scoped permissions, AES-256-GCM encrypted credential storage, tenant-isolated audit logging, and per-key service visibility. We use Delegate as a worked example to motivate a broader claim: the substrate layer for runtime-synthesized applications has identifiable structural requirements, and those requirements rule out both the per-service tool pattern dominant in the public Model Context Protocol ecosystem today, and the raw compute-mesh pattern proposed in recent adjacent work.

Keywords: post-SaaS, ephemeral software, large language models, Model Context Protocol, accountable autonomy, third-party authorization.

1. Introduction

The software industry has spent roughly two decades converging on a single dominant pattern. A vendor authors a multi-tenant web application, charges for ongoing access, holds the user's data on their servers, and provides a graphical interface through which the user performs the tasks the application supports. We call this pattern software-as-a-service, or SaaS, and the term is nearly synonymous with the category of business software itself.

We argue that significant categories of this pattern are becoming structurally vulnerable. The forcing function is the capability profile of contemporary large language models, which have crossed a threshold at which they can reliably translate natural-language intent into structured action against external services. A user who wants to send a customized invitation to thirty guests, summarize a quarter of email into a status update, or extract every action item from the last six client meetings, no longer requires a vendor to have built a tool for exactly that task. The user describes the task, and the language model performs it by calling primitives. The persistent product, the configured workflow, the per-user account in a vendor's database, are no longer necessary intermediate artifacts. The behavior the user wants is synthesized on demand and discarded after use.

We are explicit that not all SaaS will be displaced. Categories with deep persistent state (large CRMs, financial systems of record, regulated workflows with retention requirements), categories with strong network effects, and categories where the value is in the curated experience rather than the underlying action will persist for the foreseeable future. The argument of this paper concerns the categories of SaaS whose value proposition is, fundamentally, that they have pre-authored a workflow the user could otherwise have described in a sentence. For those categories, the substrate layer for ephemeral software is necessary.

We refer to the resulting class of computation as ephemeral software. Ephemeral software has structural properties that distinguish it from SaaS:

- **It is not retained.** An ephemeral application exists for the duration of a user's need. Its logic is regenerated when the need recurs, often in a form adapted to the recurrence.
- **It has no front door.** Users do not visit a website, create an account, or configure preferences. The language model is the front door, and the ephemeral application is invoked through natural language against shared primitives.
- **Its capability surface is composed, not selected.** Where a SaaS user selects from a feature menu the vendor has constructed, an ephemeral-software user describes a need and the model composes a path to that need from primitives. The capability surface is the union of available primitives, not the curated menu of a vendor.
- **Its accountability does not live inside the artifact.** A SaaS application carries its own user account, its own audit log, its own permission model. An ephemeral application has none of these. The accountability must live in the substrate, because the application itself does not persist long enough to carry it.

The last property is the central concern of this paper. The substrate layer that ephemeral software runs on must hold the credentials, enforce the scopes, record the actions, and present the audit surface that the application itself cannot. Without a credible substrate, ephemeral software is either useless (the language model has no primitives to call) or dangerous (the language model holds the user's credentials directly and acts without record). The substrate layer is therefore not an implementation detail. It is the load-bearing infrastructure of the post-SaaS era.

Our approach to characterizing this layer follows a principle from our lab's practice: architecture before automation. We do not begin with what an agent should be able to do; we begin with what an accountable substrate must guarantee. The structural requirements derived in Section 3 are the result of that approach, and the Delegate architecture is one realization of them.

This paper makes three contributions. First, we characterize the structural requirements of a primitive substrate for ephemeral software, deriving them from the properties of ephemeral software itself and the asymmetries between human-driven and model-driven authorization. Second, we propose a three-part taxonomy of substrate primitives (mediated, first-party, native) and argue that the substrate's evolution from serving technically fluent principals toward serving non-technical principals corresponds to a shift in composition toward first-party primitives. Third, we present Delegate, a working implementation that exercises the mediated case at scale and demonstrates the architectural path to the first-party case.

Stated more directly: Delegate is not the application layer. Delegate is the substrate layer that makes application synthesis safe enough to be useful. The remainder of this paper develops the argument for why such a substrate is necessary, derives the structural requirements it must satisfy, and presents Delegate as the working implementation through which we test the thesis.

We use Delegate as evidence that the substrate layer is buildable. We do not claim Delegate is the only viable design. We do claim that any viable design must address the same structural constraints, and that the per-service tool pattern dominant in the public Model Context Protocol [1] ecosystem today does not.

Section 2 reviews related work. Section 3 derives the structural requirements of a primitive substrate for ephemeral software. Section 4 presents the three-part primitive taxonomy. Section 5 describes the Delegate architecture. Section 6 covers the meta-tool aggregation layer and reports a schema-size calculation of context cost reduction, with explicit methodology. Section 7 covers the accountability and credential model. Section 8 describes multi-tenancy and the integration model. Section 9 describes the reference deployment and the trajectory hypothesis from current users to the post-SaaS user. Section 10 discusses design decisions, the threat model, and limitations. Section 11 concludes.

2. Related Work

2.1 The Model Context Protocol Ecosystem

The Model Context Protocol, introduced in late 2024 [1], has become the de facto interface between language models and external tools. The dominant implementation pattern in the public ecosystem is per-service: a developer authors an MCP server that wraps a single backing service, statically registers each operation as a separately schemed tool, and the client loads all tool schemas into the model's context at session start. This pattern works in isolation but degrades when an agent is connected to many services simultaneously, because each tool schema is loaded into context whether the tool is invoked or not. Section 6.2 quantifies the resulting cost.

2.2 Multi-Service Aggregators

Several systems aggregate multiple backing services behind a single MCP-style interface. Zapier MCP exposes user-configured automation flows; Composio [6] offers a managed platform with a large integration library. Both are significant prior art. Both retain the per-tool schema pattern: each enabled integration contributes its tool schemas to the agent's context, and the schema-resident cost scales linearly with enabled integrations. The contribution of Delegate relative to these systems is the meta-tool decomposition that decouples the agent-facing surface from the catalog size, and the three-part primitive taxonomy (Section 4) that distinguishes the substrate from a pure aggregator.

2.3 Personal Agent Frameworks

OpenClaw [7] and its Rust-native reimplementation ZeroClaw [8] are personal AI assistant frameworks that connect a language model to messaging platforms and expose shell, browser, file, and calendar operations as tools. They are substrate-adjacent: both route natural-language requests through a local gateway and dispatch them to execution primitives. Their architectural focus is the single-machine personal assistant, with no concept of remote node registration, third-party service brokering, or principal-scoped accountability across many users. They occupy a different layer of the substrate stack than Delegate: the personal-assistant runtime, not the third-party action layer. The two layers compose naturally, and a personal-assistant framework could plausibly call a Delegate-like substrate for its outbound third-party actions.

2.4 OAuth 2.0 and the Non-Human Caller

OAuth 2.0 [2] was specified for delegated authorization between a human user and a third-party application. Its scope model is defined by the resource server, not the calling application, and not the end user. A user granting an agent gmail.modify cannot, within OAuth itself, restrict the grant further to drafts only or to a specific label. The mismatch between this design and the requirements of model-driven authorization is structural: agents are not enumerable in advance, the human principal cannot predict which scopes a useful agent will exercise, and the consequences of an over-broad grant are amplified by the agent's autonomy. The Delegate permission model (Section 7) composes a finer-grained, proxy-enforced scope on top of the underlying OAuth grant.

2.5 LLM-Orchestrated Compute Meshes

Tran [3] presents GNOT, a distributed execution mesh exposing three primitive capabilities (read_file, write_file, execute_command) orchestrated natively by a language model. GNOT and Delegate share a rejection of the static-tool-registration pattern and a willingness to treat the model as the principal orchestrator. They occupy different layers of the substrate stack: GNOT operates at the compute layer, where any third-party action is reached via shell command on a provisioned node; Delegate operates at the service layer, where actions traverse authenticated APIs through a proxy. The layers are complementary. A compute mesh of any sophistication will eventually need a Delegate-like layer to handle the credential and accountability surface for the third-party calls its nodes make.

2.6 Tool-Use Research and Agent Frameworks

The capability of language models to invoke external tools has been studied under various names: ReAct [5] formalized the reasoning-acting loop; Toolformer [9] demonstrated learned tool-use; MRKL [10] proposed neuro-symbolic composition of language models with external modules. LangChain [4], LlamaIndex, LangGraph, and the broader family of agent frameworks are the application-layer consumers of the primitives a substrate exposes. They are not direct alternatives to Delegate. They are the consumers Delegate is built for.

2.7 Threat Models for Agent-Mediated Action

Greshake et al. [11] characterize prompt injection as a structural class of attack against language-model-driven systems, demonstrating that adversarial content reachable by the model can hijack its instruction stream. The threat is acute for substrate-mediated agents whose tool calls touch arbitrary third-party content. Subsequent work [12] has extended the analysis to indirect injection through retrieved documents and tool outputs. The Delegate threat model (Section 10.2) acknowledges this class of attack and is explicit about which mitigations the current implementation provides and which are deferred.

3. Structural Requirements of a Primitive Substrate

Before describing the Delegate architecture, we derive the structural requirements any viable primitive substrate for ephemeral software must satisfy. These requirements follow from the properties of ephemeral software (Section 1) and the asymmetries between human-driven and model-driven authorization (Section 2.4). We do not claim the resulting list is exhaustive; additional requirements concerning rate limiting, cost ceilings, fallback under upstream failure, and others are plausible and remain open to future work. The eight requirements below are those we have found load-bearing in the design and operation of the reference deployment.

R1. Fixed-size agent-facing surface.

An ephemeral-software substrate connected to many services cannot expose a tool schema for every operation of every service. The model's context window is a scarce resource consumed

by working state, retrieved documents, and ongoing reasoning. A substrate whose agent-facing surface grows linearly with the catalog will, at any non-trivial catalog size, dominate the context budget with schemas it does not need. The agent-facing surface must therefore be constant in size, with the catalog held server-side and queried on demand.

R2. Credentials never observed by the model.

Ephemeral software is, by construction, generated logic the principal has not audited. Granting the synthesized logic direct possession of credentials is a category of risk the substrate must remove. Credentials must be held server-side, referenced by handle, and resolved at execution time by the substrate, never by the synthesized application.

R3. Scopes finer than the resource server provides.

OAuth scopes are defined by resource servers and are typically coarse. The substrate must permit the principal to compose finer-grained restrictions on top of the OAuth grant: read-only refinements, action whitelists, label-level or resource-level constraints. These refinements must be enforced by the substrate, not the synthesized application.

R4. Audit independent of the artifact.

An ephemeral application does not outlive its execution. Its own log, if it had one, would vanish with it. The audit trail of actions taken in the principal's name must therefore be written by the substrate, not the application. The principal must be able to answer the question what was done in my name by querying the substrate, regardless of which synthesized applications produced the actions.

R5. Multi-tenant isolation with per-agent visibility scoping.

A single principal will, in the steady state, operate multiple ephemeral applications concurrently. The substrate must isolate the credentials, audit logs, and visible service catalogs of distinct applications from one another, even when they are operated by the same principal. The minimum unit of isolation is the agent invocation, not the principal.

R6. Runtime extensibility.

The set of services the principal might wish to reach is open-ended. The substrate must accept new service definitions at runtime, scoped to the requesting principal, without engineering effort by the substrate operator. The catalog of a viable substrate is not a pre-curated list; it is an emergent property of what principals, and the models acting on their behalf, register and use.

R7. Provenance of synthesized action.

When the application that issues an action is itself synthesized by a model, the principal's accountability question expands. The principal needs to know not only what was done in their name, but by what synthesizing model, at what version, in response to what prompt. The substrate must record the provenance of synthesized action: which model produced the application or tool-call sequence that resulted in each entry in the audit log. This requirement is the structural answer to the trust-delegation problem inherent in ephemeral software (Section

10.3) and is essential when synthesized applications are produced by models the principal did not author.

R8. Provider opacity.
The synthesizing model and the human principal need not know which upstream provider implements which primitive. The substrate may compose, substitute, or replace upstream providers without changing the primitive surface. This requirement is what permits a single substrate to compose mediated, first-party, and native primitives (Section 4) into a uniform agent-facing surface, and is what permits the substrate's commercial relationship with the principal to be independent of the substrate's commercial relationships with its upstream providers.

The remainder of this paper describes Delegate as one implementation of these eight requirements, and reports on how well the implementation satisfies them in practice.

4. A Three-Part Primitive Taxonomy

The substrate composes three types of primitive, distinguished by the relationship between the principal, the substrate, and the upstream provider that ultimately serves the action. The distinction is not architectural in the sense of changing the agent-facing surface; from the synthesizing model's perspective, all three are simply primitives invoked through the meta-tool layer (Section 6). The distinction is in the locus of the commercial and operational relationship.

4.1 Mediated Primitives

The principal holds an account with the upstream provider. The substrate brokers access, holding the credential server-side under R2, enforcing scopes under R3, and recording actions under R4. The Delegate catalog of integrated services (Gmail, Slack, Linear, Notion, and others, described in Section 8) consists of mediated primitives. Mediated primitives serve the technically fluent principal who already maintains the relevant accounts and wants their agent to act on them without holding the credentials directly. They are the appropriate primitive type for the v1 audience of Delegate.

4.2 First-Party Primitives

The substrate holds an account with the upstream provider. The principal's commercial and operational relationship is with the substrate alone. The upstream provider is invisible to both the principal and the synthesizing model. First-party primitives extend the substrate's responsibility into domains the principal currently fulfills through direct relationships with specialized providers. The categories most immediately relevant to ephemeral software include data persistence, application deployment, transactional communication, and file storage. The defining property of a first-party primitive is not which underlying provider implements it, but that the principal interacts only with the substrate.

First-party primitives serve the non-technical principal who would not create accounts at four different upstream providers in order to run a single ephemeral application. The trajectory hypothesis of Section 9.2 is that the substrate's center of gravity shifts toward first-party primitives as its audience broadens from technically fluent principals to non-technical principals. The Delegate reference deployment does not currently ship first-party primitives at scale; the architectural path to them is described in Section 8.6, and the existing native plugins (Gmail, Slack, fetch_browser) demonstrate the integration pattern at small scale.

4.3 Native Primitives

The substrate implements the primitive itself, with no upstream provider. Native primitives are the long-horizon endpoint of a substrate's evolution and are not the subject of immediate engineering effort. We include them in the taxonomy for completeness and because the boundary between first-party and native is itself instructive: a sufficiently differentiated first-party primitive (one whose value derives substantially from the substrate's accountability and composition properties rather than the upstream provider's underlying capability) is, in effect, native, regardless of whether an upstream provider exists somewhere in the stack.

4.4 Uniform Surface, Variable Composition

The three primitive types are unified at the meta-tool layer (R1, R8). From the synthesizing model's perspective, all primitives are invoked through the same seven meta-tools, against a catalog whose entries do not distinguish among the types except through metadata the model is not required to read. From the principal's perspective, the distinction surfaces in the billing and connection-management interfaces (mediated primitives require an OAuth flow to the principal's upstream account; first-party primitives do not). From the substrate operator's perspective, the distinction surfaces in the commercial relationship with upstream providers and in the operational responsibility for the primitive's reliability.

5. System Architecture

5.1 Design Principles

Delegate is founded on six principles, mapped to the structural requirements of Section 3:

- **Fixed-size agent-facing surface.** Seven meta-tools, constant in number. (R1)
- **Server-side catalog.** Service and tool specifications held in the proxy, queried by the model on demand. (R1, R6)
- **Credentials by reference.** Tokens held encrypted in the proxy; the agent invokes by connection handle, never by credential. (R2)
- **Layered scopes.** Permission level and per-connection allowed-tools refinement, enforced by the proxy. (R3)

- **Proxy-side audit with provenance.** Every action recorded by the proxy in a tenant-scoped log, with the originating model identified where the calling client provides that metadata. (R4, R5, R7)
- **Uniform interface across primitive types.** Mediated, first-party, and native primitives addressed identically by the agent. (R8)

5.2 Architecture Overview

A Delegate deployment is composed of three logical layers. The transport layer accepts inbound MCP requests on a single endpoint. The proxy layer dispatches meta-tool calls, looks up specifications in the catalog, applies the accountability rules associated with the calling tenant and connection, and resolves credentials. The execution layer carries out the underlying call against the backing third-party service or first-party provider and returns structured output to the proxy for logging and forwarding.

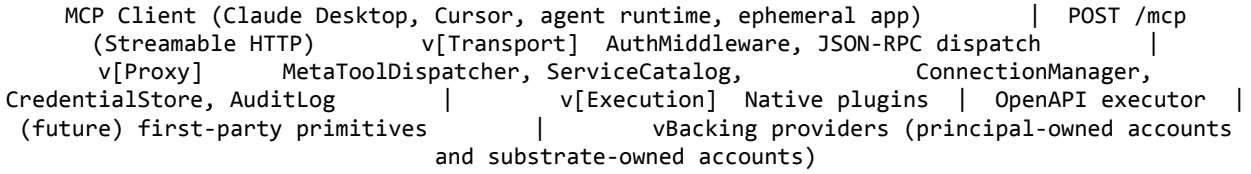

Figure 1. Delegate logical layering. The execution layer composes native plugins, OpenAPI-derived service executors, and (in the trajectory of Section 9.2) first-party primitive providers. The transport and proxy layers are invariant to primitive type.

5.3 Principal Components

Component	Responsibility
MetaToolDispatcher	Routes the seven meta-tool calls to the appropriate handler.
ServiceCatalog	Holds service and tool specifications; supports semantic and substring search. Catalog entries are populated by native plugins shipped with the substrate, OpenAPI specifications imported on demand, and tenant-registered specifications (R6).
ConnectionManager	Tracks per-tenant service connections, permission level, and allowed-tools list.
CredentialStore	AES-256-GCM encrypted storage of OAuth tokens and API keys, persisted with row-level security.
ToolExecutor	Resolves credentials, executes the underlying call against the appropriate backing provider, and returns structured output.
AuditLog (tool_call_logs)	Records every invocation with estimated token cost, status, latency, error class, and (where supplied by the client) the synthesizing model identifier and version (R7); emits alerts to a configured Slack channel for failure classes of interest.

SpecRegistrar	Accepts OpenAPI specifications at runtime and registers them scoped to the requesting tenant.
OAuthBroker	Hosts OAuth flows for mediated primitives, exchanging authorization codes for tokens and persisting them to the CredentialStore.

Table 1. Principal Delegate components and their responsibilities.

6. The Meta-Tool Aggregation Layer

6.1 The Seven Meta-Tools

The agent-facing surface comprises seven meta-tools (Table 2). Each corresponds to an irreducible category of operation an agent must perform to use the system without out-of-band configuration.

Meta-tool	Purpose
list_connected_services	Returns the services the current tenant has connected.
search_services	Returns services matching a query, including catalog services not yet connected.
get_tool_catalog	Returns the list of tools within a specified service.
get_tool_spec	Returns the full parameter schema for a specified tool.
run_tool	Invokes a specific tool with supplied parameters.
connect_service	Initiates an OAuth flow or API-key entry for a new service (mediated primitive) or activates a first-party primitive for the tenant.
add_service_from_spec	Registers a new service from an OpenAPI specification, scoped to the tenant.

Table 2. The seven agent-facing meta-tools.

The set is minimal and complete: four discovery-oriented meta-tools each answer a distinct question the agent must answer to proceed; two action-oriented meta-tools handle the use of an existing connection and the creation of a new one; the seventh supports the case of an agent extending its own capability surface at runtime, which is essential under R6 when the required capability set is not known in advance.

6.2 Context-Cost Reduction: A Schema-Size Calculation

We report the context-cost reduction of the meta-tool decomposition as a schema-size calculation, not as a production measurement of user behavior. The reference deployment is at an early stage (Section 9.1), and the user base is too small at the time of writing to support a

behavioral measurement claim. The calculation below establishes a structural upper bound on what the per-tool registration pattern costs and a lower bound on what the meta-tool surface costs; the realized reduction in any given production deployment falls between these bounds.

The calculation proceeds as follows. Let:

- S = the number of services in the deployment's effective catalog at session start.
- T = the average number of tools exposed per service.
- k = the average tokens consumed by one tool schema (name, description, parameter schema in JSON form).

The per-tool-registration cost, paid in context every model invocation regardless of which tools are used, is approximately S * T * k tokens. For a representative configuration with S = 33 (the current Delegate catalog), T = 30 (mid-range across the catalog), and k = 300 (a conservative estimate for OpenAPI-derived tool schemas), the bound is approximately:

```
33  x  30  x  300  =  297,000 tokens
```

This bound exceeds the working context window of the major commercially deployed frontier models as of early 2026. In practice, an agent author using the per-tool pattern is forced to selectively enable only a small subset of services per session, with the consequence that the agent cannot opportunistically reach for a service it had not been pre-configured to use. For ephemeral software, where the required services are unknown until the principal states their need, this is disqualifying.

The Delegate seven-meta-tool surface registers a fixed schema cost on the order of:

```
7  x  ~150 tokens  =  ~1,050 tokens
```

Taken at the upper bound of the per-tool case, the reduction is approximately 99.6 percent. This figure is not the operationally interesting number. Real deployments do not enable the full catalog; the per-tool author selectively activates a subset of services. A representative realistic comparison takes a working set of five to ten enabled services as the per-tool baseline. Against a five-service baseline (S = 5, T = 30, k = 300, total ~45,000 tokens), the meta-tool surface reduces schema-resident cost by approximately 97 percent. Against a ten-service baseline, by approximately 99 percent. The schema-size calculation therefore supports the claim that the reduction is asymptotically near-total as the catalog grows, with the realized reduction in any specific deployment depending on the size of the working set the per-tool author would otherwise enable.

We do not report a single headline percentage in this paper because the figure depends entirely on the chosen baseline. We instead commit to a production measurement appendix in a future revision, once the user base supports a behavioral study with adequate sample size and across multiple tokenizers. The architectural claim, however, is independent of the precise figure: any per-tool baseline at non-trivial S exceeds the meta-tool surface by at least an order of magnitude, and the gap widens with S.

6.3 Discovery-Then-Execution

A typical interaction follows a discovery-then-execution pattern. The agent calls list_connected_services to establish what is available, then search_services or get_tool_catalog to narrow to the relevant service, then get_tool_spec to retrieve the schema for the specific tool, then run_tool to invoke. A capable agent caches the results of intermediate discovery within its working context for the duration of a session, paying the cost once per task. Subsequent invocations of the same tool require only a run_tool call. The pattern is analogous to lazy capability discovery in service-mesh control planes, with the consumer being a language model reasoning across multiple turns.

7. Accountability and Credential Model

7.1 Permission Levels

Each connection between a tenant and a service carries one of three permission levels: readonly, draft, or full. The readonly level permits invocation of tools that do not mutate state. The draft level additionally permits tools that produce unpublished or non-acting artifacts, such as Gmail drafts. The full level permits any tool the service exposes. Classification is per-tool, encoded in the catalog at registration time. The proxy enforces the classification before dispatching the underlying call.

7.2 Per-Connection Allowed-Tools Refinement

Beyond the three-level grant, each connection carries an optional allowed-tools whitelist. When present, only tools in the whitelist may be invoked through the connection, regardless of permission level. This provides finer-grained restriction below the granularity offered by the backing service's own OAuth scopes, which is the explicit purpose of requirement R3.

7.3 Credential Storage

OAuth tokens and API keys for mediated primitives are encrypted at rest using AES-256-GCM. The encryption key is held separately from the application's authentication secret, permitting independent rotation. Tokens are persisted with row-level-security policies enforcing tenant isolation at the database layer. First-party primitive credentials are held under the same encrypted-at-rest discipline, with the additional property that the upstream provider account is owned by the substrate operator rather than the tenant.

7.4 Audit Log

Every meta-tool invocation, and every underlying tool invocation that follows from a run_tool call, is recorded in tool_call_logs with fields including tenant identifier, API key fingerprint, service, tool, redacted parameters, estimated tokens, status, latency, error class, and (where the calling client supplies it) the identifier and version of the synthesizing model. The log is the principal accountability artifact under R4 and R7. A human principal can answer the question

what was done in my name, by what model, in the last twenty-four hours by querying it, regardless of which ephemeral applications produced the actions.

7.5 Error Escalation

Errors of class authentication, authorization, and quota are escalated to a configured Slack webhook in addition to being recorded in the audit log. This provides a real-time signal when an agent's grant is failing in production, a class of failures that frequently goes undetected with per-service MCP servers because each server emits errors only to its own log.

7.6 Specification Versioning

The catalog is versioned. Each registered service maintains a history of specification revisions, with a tenant-scoped rollback path. This is operationally significant for the dynamic specification registration path (Section 8.4), where a tenant-registered specification may need to be reverted after a regression in the backing service or a misconfiguration of the registered scope.

8. Multi-Tenancy and the Integration Model

8.1 Tenant Model

A tenant corresponds to an end-user account. Each tenant holds a set of API keys, each with its own allowed-services scope, a set of service connections, per-connection credentials, and a personal audit log scoped via row-level security.

8.2 API-Key-Scoped Service Visibility

Each API key carries an allowed-services list. The catalog query returns only services on the list, and run_tool refuses invocations against services not on the list. This enables a single tenant to issue distinct keys to distinct ephemeral applications with non-overlapping service grants, satisfying R5 at the per-agent granularity.

8.3 Native Plugins and OpenAPI-Derived Services

The current Delegate deployment ships with native plugins for high-friction integrations (Gmail, Slack, and an HTTP fetch primitive) and OpenAPI-derived integrations for a working set of widely-used productivity, communication, customer-relationship, and developer services. Native plugins handle capabilities that do not map cleanly onto a single OpenAPI operation. OpenAPI-derived services are imported through the SpecRegistrar, which parses each specification, classifies operations by permission level using HTTP-method heuristics with admin-curated overrides, and registers the resulting tools.

We do not enumerate the exact set of integrated services here. The catalog is properly understood as an emergent property of usage rather than a fixed product specification: a service is in the catalog if it has been shipped natively, included in the default OpenAPI import

set, or registered by a tenant. The composition of the catalog is expected to shift continuously as principals and the models acting on their behalf add specifications under R6.

8.4 Dynamic Specification Registration

The add_service_from_spec meta-tool exposes the SpecRegistrar to tenants. A tenant supplies an OpenAPI specification at runtime, registers it against their own catalog, and immediately invokes its tools through the same meta-tool surface. Registration is tenant-scoped: a specification registered by tenant A is not visible to tenant B. This is the satisfaction of R6 (runtime extensibility) without sacrificing the integrity of the global catalog. The current implementation has a known limitation in handling specifications that do not declare server URLs; an explicit base-URL override at registration time is in progress and is the gating dependency for fully model-driven discovery and registration.

8.5 OAuth Flow

For mediated primitives, the connect_service meta-tool returns a URL pointing to the Delegate-hosted OAuth flow for the requested provider. The principal completes the flow in a browser, the callback is handled by the proxy's web layer, and the resulting tokens are encrypted and persisted against the tenant's connection. The agent receives a connection identifier and never observes the token at any point, satisfying R2.

8.6 Architectural Path to First-Party Primitives

First-party primitives extend the existing native plugin pattern. From the substrate's internal perspective, a native plugin already encapsulates connection complexity that the principal does not see; the architectural difference for a first-party primitive is that the upstream account is owned by the substrate operator rather than the tenant, and the principal's relationship is with the substrate alone (R8). The integration of a first-party primitive proceeds analogously to a native plugin: define the primitive's tool surface, implement the executor against the upstream provider's API using substrate-held credentials, register the tools in the catalog, expose them through connect_service for tenant activation. No change to the agent-facing meta-tool surface is required. The Delegate codebase as deployed today implements the integration pattern; activation of first-party primitives at scale is a near-term roadmap item, with priority determined by observed demand from the principal trajectory described in Section 9.2.

9. Reference Deployment and Trajectory

9.1 Deployment

The reference Delegate deployment is a single hosted service exposing POST /mcp as its primary endpoint, with an associated web UI handling OAuth flows, connection management, and audit log inspection. Persistence is provided by a managed Postgres backend with row-level-security policies enforcing tenant isolation. The hosted service is operational and

accepting design-partner traffic at the time of this writing. The user base is at an early stage, with a small number of active design partners; production behavioral measurement at statistically meaningful sample size is deferred to a future revision (Section 6.2).

9.2 Trajectory from Current Users to the Post-SaaS Principal (Working Hypothesis)

The thesis of this paper is forward-looking. We are explicit that the trajectory described in this subsection is a working hypothesis to be evaluated empirically, not a prediction supported by current evidence. The hypothesis is as follows.

The current adopter of Delegate is a software developer building a personal agent that needs to take actions across services. They are technically fluent, have already attempted an OAuth integration, and are seeking a way to make their agent useful without writing infrastructure that is off-mission for their project. This principal is, in our framing, the leading edge of the post-SaaS principal: a person who has internalized the pattern of describing intent to a language model and expecting action, but who currently does so through an agent they have authored themselves.

The hypothesized trajectory from this principal to the broader audience of non-technical principals invoking ephemeral software is one of capability inheritance rather than substitution. The same substrate that satisfies the developer's agent today is hypothesized to satisfy the ephemeral applications that synthesize themselves for non-technical principals tomorrow. The accountability requirements (R3, R4, R5, R7) are not specific to developer principals; they are specific to the asymmetry between human principal and model-driven action. The fixed-size surface (R1), credentials-by-reference (R2), and provider opacity (R8) are likewise universal across the audience expansion.

The contrast between the two principals can be stated sharply: for the technical principal, Delegate removes integration burden; for the non-technical principal, Delegate removes the need to know that an integration exists. The substrate is the same in both cases. What changes is how much of it the principal must consciously perceive.

The principal change as the audience broadens is twofold. First, the locus of application authorship shifts from user-authored agents to model-synthesized ephemeral applications, which makes R7 (provenance) operationally critical rather than nice-to-have. Second, the composition of the catalog shifts from predominantly mediated primitives (which assume the principal holds the upstream accounts) toward a growing fraction of first-party primitives (which the substrate operator holds on behalf of principals who do not). The substrate's architectural surface is invariant across both shifts. This is the property that makes the substrate-layer bet defensible across an audience expansion that would otherwise force a product pivot.

We treat the trajectory hypothesis as falsifiable. If, over the next several quarters, the substrate's design-partner traffic remains concentrated among technically fluent principals invoking mediated primitives, with no observed shift toward non-technical principals or first-party

primitive demand, the hypothesis is weakened. If the shift occurs, the hypothesis is strengthened, and the substrate's evolution toward heavier first-party-primitive composition is justified empirically rather than speculatively.

10. Discussion

10.1 Design Decisions

D1. Why seven meta-tools rather than a smaller set?

A minimal three-meta-tool set comprising get_tool_spec, run_tool, and connect_service is feasible but eliminates discovery. An agent that does not know which services exist or which tools exist within a service cannot use the system without out-of-band configuration. The discovery-oriented meta-tools are each irreducible. The seventh, add_service_from_spec, is less universal than the other six and could be hidden behind administrative API keys; we retain it in the agent-facing surface because R6 (runtime extensibility) is essential for ephemeral software whose required capabilities are not predictable in advance.

D2. Why a server-side catalog rather than client-side specification download?

Client-side caching reintroduces the per-tool-schema-in-context problem if the agent caches aggressively, complicates per-tenant catalog scoping, and prevents the catalog from changing during a session. R6 explicitly requires the last property.

D3. Why aggregate behind a single proxy rather than federate across per-service servers?

Federation forecloses the meta-tool aggregation: each federated server retains its own tool surface, and the schema-explosion problem returns at the multiplexer's tool registration step. Federation also disperses the accountability layer across independently authored servers, none of which can guarantee the audit, permission, or provenance properties (R4, R7) that Delegate provides centrally. Finally, federation forecloses provider opacity (R8): a federated multiplexer cannot transparently substitute first-party for mediated implementations of the same primitive.

D4. Why AES-256-GCM at rest rather than a managed key-management service?

Managed key-management service integration introduces deployment dependencies inconsistent with our preference for a low-operational-footprint reference deployment. AES-256-GCM with a separately rotated encryption key, against a row-level-security-enforced database, provides credible isolation for the current audience. Larger deployments will likely require a managed-KMS path; we treat this as future work.

D5. Why three permission levels rather than per-tool boolean grants?

Per-tool boolean grants are the limit case of the allowed-tools whitelist already provided. The three-level abstraction is offered as the default for principals who do not wish to author a tool-by-tool whitelist. It captures the most common access patterns observed in the target audience without requiring per-tool configuration on first use.

D6. Why distinguish mediated, first-party, and native primitives explicitly?

The distinction does not change the agent-facing surface (R1, R8). We make it explicit because it determines the substrate operator's commercial relationships, the principal's billing surface, and the trajectory of substrate evolution as the audience broadens. A taxonomy that hides the distinction risks the failure mode of a substrate that purports to serve non-technical principals while in fact requiring them to maintain accounts at half a dozen upstream providers. Making the distinction explicit forces honesty about which primitives a given substrate actually offers in first-party form versus merely mediates.

10.2 Threat Model

We are explicit about what is in scope for the current version and what is not.

In scope:

- Compromise of the agent's runtime. The agent cannot exfiltrate tokens because the agent never holds tokens.
- Over-broad OAuth grants from the resource server. Permission level and allowed-tools refinement limit practical exposure below OAuth-native granularity.
- Audit-trail evasion by the agent. The audit log is written by the proxy, not by the agent.
- Cross-tenant data exposure at rest. Row-level-security policies enforce isolation at the database layer; AsyncLocalStorage prevents per-request context bleed at the application layer.

Out of scope for the current version:

- Compromise of the Delegate proxy itself. The hosted proxy is a centralized trust point; the current version does not defend against a malicious proxy operator.
- Indirect prompt injection through tool outputs and retrieved content [11, 12]. The substrate does not currently sandbox or rewrite content returned to the synthesizing model from upstream providers. Mitigations are an active area of design.
- Malicious specifications registered via add_service_from_spec. A tenant who registers a specification pointing to an adversarial endpoint can exfiltrate the parameters of their own subsequent calls.
- Hard budget enforcement. The proxy records estimated token costs but does not block actions at a budget ceiling.
- Side-channel timing attacks on token comparison. Most authentication paths use constant-time comparison, but a full audit of timing-sensitive paths is pending.

10.3 Limitations

- **Centralized trust point.** The hosted Delegate proxy is the single accountability node. Self-hosted deployment is on the roadmap.

- **Non-persistent operational state.** Per-request state such as rate-limit counters is held in process memory. A proxy restart loses this state.
- **Single-region deployment.** Multi-region deployment will require attention to the latency cost of cross-region OAuth flows and the consistency model of the catalog.
- **Trust delegation chain.** When ephemeral software is synthesized by a model the principal does not author, the chain of trust extends from the principal through the synthesizing model to the substrate. R7 (provenance) is the substrate's structural answer to this problem, but the provenance record is only as trustworthy as the calling client that supplies it. A fully verifiable provenance chain requires cooperation from the synthesizing model provider and is a longer-horizon research question.
- **Production measurement at scale.** The schema-size calculation reported in Section 6.2 is a structural argument, not a behavioral measurement. A production-data appendix is deferred to a future revision.

10.4 Future Work

Targeted improvements include:

- Hard budget enforcement, with refusal above a configurable ceiling.
- Self-hosted deployment artifact for principals with regulatory constraints.
- Indirect prompt injection mitigations: content rewriting, provenance tagging of tool outputs, and structured separation of model-facing and substrate-facing channels.
- Activation of first-party primitives across the categories identified in Section 4.2.
- Production measurement appendix with sample-size-supported context-cost reduction figures across multiple tokenizers.
- Formal characterization of accountability guarantees an agent-mediating substrate can offer to a human principal.

11. Conclusion

Significant categories of software-as-a-service are ending not because they failed, but because their central assumption stopped holding. SaaS assumed that the cheapest way to give a user a capability was to pre-author the capability, sell access to it, and let the user invoke it through an interface. The capability profile of contemporary language models inverts that assumption for an expanding class of use cases. The cheapest way to give a user a capability is now, increasingly, to let the user describe the capability and have a model synthesize it on demand against a substrate of primitives.

Ephemeral software is the resulting class of artifact. It is not retained, has no front door, composes its capability surface rather than selecting from a menu, and carries no accountability inside itself. These properties make the substrate it runs on load-bearing. The substrate must

offer a fixed-size agent-facing surface, hold credentials the model never sees, enforce scopes finer than resource servers provide, write the audit trail the application cannot, isolate at the per-agent granularity, accept new service definitions at runtime, record the provenance of synthesized action, and present a uniform interface across mediated, first-party, and native primitives.

We have presented Delegate as a working instance of such a substrate. The seven-meta-tool aggregation layer decouples agent context cost from catalog size by structural construction. The accountability layer composes scoped permissions, encrypted credential storage, tenant-isolated audit logging, provenance recording, and per-key service visibility into a coherent surface over which a human principal can observe, scope, and revoke model-driven action. The three-part primitive taxonomy clarifies how a single substrate serves both technically fluent principals (predominantly through mediated primitives) and non-technical principals (through a growing composition of first-party primitives) without an architectural pivot. The reference deployment is operational, the v1 audience is the technically fluent leading edge of the post-SaaS principal, and the substrate is invariant for the audience expansion that follows.

We do not claim Delegate is the only viable substrate, nor that its current implementation is the limit of what such a substrate can be. We claim that the requirements we have derived are structural, that the per-service tool pattern dominant in the public Model Context Protocol ecosystem does not satisfy them, that the raw compute-mesh pattern in adjacent work satisfies a different layer of the stack, and that the substrate layer for ephemeral software is the missing piece of the post-SaaS computing stack. Whether built by us or by others, this layer will exist. The question that remains is who builds it well enough to be trusted with it.

Acknowledgments

Delegate is a project of the Controlled Mayhem applied AI lab. We thank the broader Model Context Protocol community for the open specifications and the open-source server ecosystem that made the meta-tool decomposition formulable. The conceptual sharpening of this paper benefited from internal review and disagreement; the resulting argument is stronger for the pushback. Errors that remain are ours.

References

[1] Anthropic. (2024). Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol

[2] Hardt, D. (Ed.). (2012). The OAuth 2.0 Authorization Framework. RFC 6749, Internet Engineering Task Force. https://www.rfc-editor.org/rfc/rfc6749

[3] Tran, Q. V. (2026). GNOT: Generative Node Orchestration Technology. A Minimal-Seed Architecture for LLM-Native Distributed Execution. SSRN.

[4] Chase, H. (2022). LangChain: Building Applications with LLMs Through Composability. https://github.com/langchain-ai/langchain

[5] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. (2022). ReAct: Synergizing Reasoning and Acting in Language Models. arXiv preprint arXiv:2210.03629.

[6] Composio. (2024). Composio: A Managed Tool Platform for AI Agents. https://composio.dev

[7] Steinberger, P. et al. (2026). OpenClaw: Open-source Autonomous AI Agent Framework. https://github.com/openclaw/openclaw

[8] ZeroClaw Labs. (2026). ZeroClaw: Fast, Small, and Fully Autonomous AI Assistant Infrastructure. https://github.com/zeroclaw-labs/zeroclaw

[9] Schick, T., Dwivedi-Yu, J., Dessi, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. arXiv preprint arXiv:2302.04761.

[10] Karpas, E., Abend, O., Belinkov, Y., Lenz, B., Lieber, O., Ratner, N., Shoham, Y., Bata, H., Levine, Y., Leyton-Brown, K., Muhlgay, D., Rozen, N., Schwartz, E., Shachaf, G., Shalev-Shwartz, S., Shashua, A., and Tenenholtz, M. (2022). MRKL Systems: A Modular, Neuro-symbolic Architecture That Combines Large Language Models, External Knowledge Sources and Discrete Reasoning. arXiv preprint arXiv:2205.00445.

[11] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., and Fritz, M. (2023). Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec).

[12] Liu, Y., Jia, Y., Geng, R., Jia, J., and Gong, N. Z. (2024). Prompt Injection Attacks and Defenses in LLM-Integrated Applications. arXiv preprint arXiv:2310.12815.

[13] Anthropic. (2025). Building Effective Agents. https://www.anthropic.com/engineering/building-effective-agents

[14] Phillips, D. and Phillips, A. (2026). Controlled Mayhem: Applied AI Lab Principles. https://controlledmayhem.com