Naftiko

Naftiko Capabilities: From APIs to Business Outcomes

Declarative, Standards-Based Integrations Aligned with What Your Platform Can Actually Do

Kin Lane

March 2026

Contents

Executive Summary

The way enterprises expose and consume digital services is undergoing a fundamental shift. For two decades, APIs have served as the primary interface between systems, teams, and organizations. But as AI agents, automation platforms, and cross-functional workflows become the dominant consumers of these interfaces, the API-centric model is showing its limits.

APIs expose *how* to call a system. Capabilities describe *what* a system can do.

Naftiko Capabilities are open-source, declarative, standards-based integrations aligned with business outcomes within specific domains. They provide what is needed to deliver and automate integrations across a variety of use cases – from composing SaaS tools into multi-step workflows, to bridging legacy enterprise systems that no API gateway or MCP server can reach, to enabling AI agents to discover, reason about, and safely invoke business functions without human orchestration.

This white paper explains why the shift from APIs to capabilities is happening, what capabilities are, how Naftiko implements them as a declarative framework, and where the opportunity landscape spans across industries, roles, and emerging architectural patterns.

The Problem: APIs Were Built for Developers, Not for Agents

Traditional APIs provide precise, machine-executable contracts. A developer knows the endpoint, the HTTP method, the required parameters, and the expected response shape. This model works when a human is in the loop – reading documentation, choosing which API to call, and orchestrating the sequence of calls needed to accomplish a business task.

AI agents operate differently. As Christian Posta observes, “agents don’t start with specs – they start with goals.” An agent begins with a high-level objective and context, then dynamically determines which steps and capabilities to employ. This requires systems capable of self-description beyond standard API documentation.

Mike Amundsen reinforces this shift: “When the client is autonomous, your API has to be more than a list of endpoints. Instead, it needs to be a menu of possibilities.” Traditional method names and URLs are insufficient for clients that “don’t just follow instructions – they observe, infer, and choose.”

The gap is not just about AI agents. Business stakeholders, operations teams, and non-developer roles increasingly need to understand, configure, and govern integrations. Bruno Pedro notes that “the most important thing about APIs is not their operations. The thing potential API consumers value most is understanding what capabilities an API offers – the things that make people’s lives easier.” According to Postman’s 2024 State of the API Report, approximately 30% of API users lack developer backgrounds, and 40% follow business directives when selecting integrations.

The API-centric model creates three compounding problems:

1. **Context overload for AI agents.** When an agent must make three to four separate API calls to accomplish a single business task, every intermediate response enters the LLM’s context window. Raw JSON payloads, XML envelopes, and SOAP responses consume tokens, introduce noise, and increase the probability of hallucination or reasoning errors.

2. **Invisible orchestration logic.** The sequence of API calls required to fulfill an order, triage an error, or provision a lead is implicit knowledge that lives in developer heads, runbooks, or scattered scripts. It is not discoverable, not composable, and not auditable.

3. **Protocol barriers to enterprise systems.** The majority of enterprise backend systems – ERP, HRIS, clinical, regulatory, industrial – expose interfaces via SOAP/XML, OData, EDI, SPARQL, or proprietary protocols. No AI agent can generate a correct SOAP envelope. No MCP server exists for SAP, Workday HCM, Epic EHR, or ACORD claims platforms. These systems are structurally invisible to the agentic ecosystem.

What Is Capability Thinking?

Capability thinking is an approach rooted in enterprise architecture, event-driven design, and modern API strategy. As Daniel Kocot defines it: “The focus shifts from exposing data or technical resources to clarifying what your system can actually do for its consumers.”

Rather than exposing granular resources like `GET /orders/{id}` and `POST /orders`, capability thinking describes discrete business functions: *Ship Order*, *Process Payment*, *Approve Loan*. Instead of five API calls that force consumers to understand order structures, validate stock, assign warehouses, trigger shipment, and update status independently, a single **Ship Order** capability handles the full orchestration internally.

This is not a cosmetic relabeling. As Kocot warns, “Renaming ‘Customer Table API’ as ‘Customer Management Capability’ without genuine abstraction adds no value.” True capabilities possess structural properties that distinguish them from API endpoints wrapped in new terminology.

The Properties of a Capability

Drawing on the work of Kocot, Posta, Amundsen, Pedro, and Lane, a capability is defined by the following properties:

- **Business Aligned** – A capability speaks to business outcomes, not system design. It uses language that business stakeholders recognize: *Process Refund*, *Triage Error*, *Provision Lead*.
- **Clear Boundaries** – Each capability has a well-defined scope. *Approve Loan* is distinct from *Disburse Loan*. Boundaries prevent scope creep and enable governance.
- **Composable** – Capabilities can chain together. *Ship Order* composes with *Generate Invoice* and *Notify Customer*. Complex workflows emerge from simple, well-bounded building blocks.
- **Reusable** – A single capability like *Validate Customer Identity* supports onboarding, fraud detection, and account recovery without duplication.
- **Discoverable** – Capabilities are documented in catalogs with rich metadata, enabling both humans and agents to find what they need.
- **Technology Independent** – *Send Notification* remains valid whether the underlying transport is SMS, email, or push notification. The capability abstracts the protocol.
- **Declarative** – Capabilities are defined as configuration, not code. The specification *is* the implementation contract.
- **Machine and Human Readable** – A capability is something any person can understand at a glance and any machine can parse, input to, and output from.

Kristof Van Tomme, upon reading the capabilities literature while preparing a developer portal maturity assessment, arrived at a complementary insight: “Capabilities are affordances that have been productised.” Where traditional interfaces provide multiple affordances implicitly, capabilities separate and materialize them into granular, composable assets whose boundaries coincide with the boundaries of the tool that provides them.

This framing connects capability thinking to the broader evolution of developer portals, platform engineering, and API product management. Capabilities are not just a technical architecture pattern – they are a unit of value that can be discovered, documented, governed, and monetized.

How Naftiko Implements Capabilities

Naftiko is a declarative framework for defining, publishing, and executing capabilities as YAML specifications. Each capability is a self-contained file that declares what it does, what it consumes, what it exposes, and how data flows between systems.

The Anatomy of a Capability

Every Naftiko capability has two sides:

Exposes – What the capability makes available to consumers (agents, workflows, humans). Today this is an MCP (Model Context Protocol) tool surface: a namespace, tool name, description, input parameters, and output parameters that any MCP-aware agent can discover and invoke.

Consumes – What upstream services the capability calls. Each consumed service declares its base URI, authentication method, resources, and operations. Reusable adapter files (`consumes-slack.yml`, `consumes-github.yml`, etc.) prevent duplication across capabilities.

Between these two sides, the capability defines **steps** – the deterministic sequence of operations that transform a business intent into executed actions across multiple systems.

Example: Triage Sentry Error

A concrete example illustrates the model. The `triage-sentry-error` capability takes a Sentry issue ID and performs three operations in sequence:

1. Fetches the error details from Sentry
2. Creates a GitHub issue with full error context
3. Posts a notification to Slack

```
naftiko: "0.5"
info:
  label: "Triage Sentry Error"
  description: "Given a Sentry issue ID, creates a GitHub issue with
    full context and notifies in Slack."
capability:
  exposes:
    - type: "mcp"
      namespace: "devops"
      tools:
        - name: "triage-sentry-error"
          steps:
            - name: get-sentry-issue
              call: "sentry.get-issue"
            - name: create-github-issue
              call: "github.create-issue"
            - name: notify-slack
              call: "slack.post-message"
  consumes:
    - import: "shared-sentry.yml"
    - import: "shared-github.yml"
    - import: "shared-slack.yml"
```

Without this capability, an AI agent would need to make three separate MCP tool calls. Each intermediate response – the raw Sentry payload, the GitHub API response – enters the LLM context window. With the capability, the agent makes one call. The raw payloads stay server-side. Only the declared output parameters cross the boundary into the agent's context.

The Output Projection Model

This is the architectural property that unifies all Naftiko capabilities: **the raw payload never reaches the LLM**.

```
HTTP API (JSON / XML / SOAP / CSV / SPARQL)
         |  [ Naftiko consumes: outputParameters + mapping ]
     clean, named JSON fields  <-- only projected fields
         |
    MCP tool response           <-- agent receives this
```

For composite capabilities (SaaS tools with existing MCP servers), this eliminates intermediate API responses from LLM context. For bridge capabilities (enterprise systems with no MCP server), it makes entire categories of systems accessible to agents for the first time – converting a format and protocol integration problem into a specification declaration.

Shared Adapters and Reuse

Naftiko's `consumes` import model means that common service connections are defined once and reused across capabilities. The framework includes 31 shared adapter files covering services like Slack, GitHub, Jira, Salesforce, Datadog, PagerDuty, Workday, Stripe, and HubSpot. Each adapter declares the authentication method, base URI, and available operations for a service.

When a new capability needs Slack notifications, it imports the shared Slack adapter rather than redeclaring connection details. This creates a compounding effect: each new adapter makes the next capability cheaper to build, and each new capability validates and hardens the adapters it imports.

Three Capability Patterns

The Naftiko capability landscape spans three complementary patterns, each addressing a distinct integration challenge.

Pattern 1: Agentic Composites

Composite capabilities chain together SaaS services that already have MCP servers. The value is not in making individual services accessible – they already are – but in collapsing multi-step, multi-service workflows into single, deterministic operations.

Thirteen composite capability candidates have been identified across DevOps, Sales, Marketing, HR, and IT operations:

Capability	Services Composed	Business Outcome
Triage Sentry Error	Sentry, GitHub, Slack	Error to tracked issue in one call
Provision Lead	HubSpot, Stripe, Slack	CRM contact to billing customer with writeback
Enrich Incident	PagerDuty, Datadog, Slack	Alert to enriched context summary
Publish Release Notes	GitHub, Notion, Slack	PR list to published changelog
Employee Onboarding	Workday, ServiceNow, Teams, SharePoint	New hire to provisioned accounts
Marketing Campaign Digest	Google Analytics, HubSpot, Confluence, Slack	Multi-platform performance summary
Sales Intelligence Enrichment	LinkedIn, Salesforce, HubSpot, Slack	Prospect to enriched CRM record
Cloud Cost Anomaly Responder	Datadog, Salesforce, Slack	Cost spike to investigated alert
CI/CD Failure Observability	New Relic, Jira, Slack	Build failure to tracked investigation

Each composite collapses three to five tool calls into one, keeping intermediate payloads server-side and eliminating the context window cost of multi-step orchestration.

Pattern 2: Protocol Bridges

Bridge capabilities make enterprise systems accessible to AI agents for the first time. These are systems with no MCP server, often communicating via SOAP/XML, OData, EDI, SPARQL, XBRL, or CSV – formats that no LLM can generate or parse reliably.

Fourteen bridge capability candidates cover enterprise backends across ERP, HRIS, healthcare, financial, regulatory, logistics, and industrial domains:

Capability	System	Protocol	Without Naftiko
SAP Purchase Order Lookup	SAP S/4HANA	OData/XML	Impossible (no MCP server)
Workday Worker Profile	Workday HCM	SOAP/XML	Impossible (SOAP envelope construction)
FHIR Patient Context	Epic / FHIR R4	REST+XML	Raw XML x3 resources in context
SEC Company Filings	EDGAR	XBRL/XML	Raw XBRL in context
Insurance Claim Status	ACORD Platform	SOAP/XML	Impossible (SOAP envelope)
Logistics Shipment Tracking	EDI 214 Gateway	EDI-to-XML	Impossible (EDI segment parsing)
Clinical Trial Status	CDISC CTMS	ODM-XML	Impossible (NCT-to-OID crosswalk)
Maritime Vessel Position	MarineTraffic AIS	REST+XML	Raw AIS XML in context
Smart Meter Consumption	IEC CIM AMI	CIM XML	Raw IEC CIM XML in context

The bridge pattern applies Naftiko's output projection model to normalize XML, SOAP, EDI, and CSV responses into clean JSON fields. The agent receives `{status: "shipped", tracking_id: "1Z999AA10123456784"}` – not a SOAP envelope or an EDI 214 segment.

Pattern 3: Emerging Architectural Patterns

Beyond composites and bridges, Naftiko's declarative model enables a set of emerging patterns that address strategic enterprise requirements:

Self-Integrating Agents – An agent that encounters a missing integration can draft a Naftiko capability specification, submit it for human review via a pull request or Slack approval, and – once approved – deploy it into its own tool surface. The declarative YAML surface is machine-writable and schema-validated before human review.

Human-in-the-Loop Action Gates – Running workflows can suspend at declared steps for human approval before executing destructive or high-value actions. This enforces structural governance: an agent cannot bypass the gate because suspension is part of the capability specification, not a runtime convention.

AI-Native Data Products – Naftiko serves as the MCP-first interface for data mesh products. A data product like "Monthly Revenue by Region" is exposed as a capability that projects exactly the fields the agent needs, with raw BigQuery response envelopes never entering LLM context.

Agent-to-Agent Trust Mediation – When agents call other agents, Naftiko capabilities serve as the declared trust boundary. The specification enforces which operations one agent may invoke on another and which output fields cross the boundary – preventing unauthorized data leakage between agents by construction.

RAG Pipeline Normalization – Every RAG implementation has a retrieval adapter (Pinecone, Weaviate, PGVector). Naftiko normalizes the retrieval step as a governed, reusable capability. When the vector store changes, only the `consumes` block is swapped – every agent and prompt template calling the retrieval tool remains unchanged.

Capability-Driven Evaluations – Naftiko capabilities serve as structured test oracles for AI agent safety testing. The declared input schemas and output projections enable frameworks to validate whether an agent passes schema-valid parameters, selects the correct tool, and respects output boundaries.

The Industry Opportunity

The capability landscape maps across 12 industry verticals, 109 services across three tiers, and 42 protocols and data exchange standards.

Industry Verticals

Each vertical has a distinct integration profile and entry point:

Industry	Bridge Systems	Lead Capability
Financial Services	Bloomberg, Murex, Calypso, FIS, Broadridge	Financial client 360, compliance incident package
Healthcare	Epic EHR, Cerner, Meditech, Availity	Epic-to-Salesforce care bridge, HIPAA incident responder
Insurance	Guidewire, ACORD, Duck Creek, ISO/Verisk	Claims case orchestrator, underwriting enricher
Pharma & Life Sciences	Veeva Vault, Medidata Rave, LIMS, CDISC	GMP deviation orchestrator, regulatory submission tracker
Retail	SAP Retail, Manhattan WMS, JDA/Blue Yonder	Order fulfillment, demand planning
Manufacturing	PTC ThingWorx, Siemens MindSphere, OSIsoft PI	Equipment health, production line observability
Energy	OSIsoft PI, ESRI ArcGIS, Quorum, AspenTech	Pipeline monitoring, field service orchestration
Logistics	FedEx, UPS, DHL, Manhattan TMS, project44	Shipment tracking, transportation management
Telecommunications	NetCracker, Nokia NetAct, CSG Singleview	Network management, subscriber billing
Construction	Autodesk ACC, Oracle Primavera P6, Procore	BIM model status, schedule orchestration
Agriculture	John Deere Operations Center, Climate FieldView	Field telemetry, precision agriculture
Maritime	MarineTraffic AIS, NAVIS N4, DNV Vessel Register	Vessel position, terminal operations

Protocol Coverage

Naftiko bridges 42 protocols and data exchange standards across seven categories:

- **Transport & API**: HTTP/REST, SOAP, JSON-RPC, MCP, OData
- **Authentication**: OAuth 2.0, Basic Auth, Bearer Token, API Key
- **Data Formats**: JSON, XML, CSV, Avro, Protobuf
- **Healthcare**: HL7, FHIR, CDISC/ODM
- **Financial**: XBRL, FIX, ISO 20022, ACORD
- **Logistics**: EDI, EDI 214, EDIFACT
- **Industrial**: OPC-UA, MQTT, NETCONF, SNMP, IEC CIM

This protocol breadth is a structural differentiator. Generic MCP-only platforms cannot operate where enterprise systems speak SOAP, OData, EDI, or CIM XML. Naftiko's bridge pattern makes these systems agent-accessible through specification, not custom integration code.

The Role Landscape

The capability opportunity maps to 139 roles across nine organizational domains. This mapping reveals which roles are served by existing capabilities and where gaps present GTM opportunities.

Roles by Domain

Domain	Roles	Key Capabilities
Technology & DevOps	12 roles	Triage error, enrich incident, CI/CD observability
Sales & Revenue	20+ roles	Provision lead, sales intelligence enrichment
Marketing	20+ roles	Campaign performance digest, content publishing
Cybersecurity	8 roles	SIEM incident chain, security advisory digest
Data & Analytics	9 roles	Data pipeline digest, ML pipeline observability
C-Suite & Management	7 roles	Executive briefing digest
HR & Operations	Multiple	Employee onboarding orchestrator
Design & UX	5 roles	Design review handoff
Compliance & Legal	Multiple	Regulatory reporting pipeline

Capability Author Personas

Beyond end-users, six author personas create and maintain capabilities:

1. **RPA Developer** – Translates Power Automate flows into Naftiko capabilities
2. **Platform Engineer** – Adds workflow capabilities to internal developer platforms
3. **Solutions Architect** – Turns architecture diagrams into executable capabilities
4. **API Developer** – Writes consumes adapters for existing APIs
5. **MLOps Engineer** – Bridges ML pipelines into incident and observability flows
6. **Marketing Technologist** – Automates campaign performance and content workflows

These author personas are force multipliers. One author creating ten capabilities generates more platform value than fifty end-users consuming a single pre-built capability.

From API Catalogs to Capability Catalogs

Most organizations maintain API catalogs that list endpoints, versions, and technical specifications. These catalogs serve developers but are opaque to business stakeholders. Bruno Pedro advocates a reverse-engineering approach: start with Jobs-to-Be-Done, extract the benefits users seek, and then define capabilities that deliver those benefits.

For a financial management API, the progression looks like:

- **Job-to-Be-Done**: Track expenses in real time
- **Benefit**: Gain instant spending visibility
- **Capability**: Transaction enrichment with categorization

A capability catalog reframes discovery around business outcomes, not technical implementation:

```
Capability: Ship Order
Description: Prepares an order for shipment, assigns carrier,
             generates tracking ID
Inputs: Order ID, shipping preferences
Outputs: Tracking ID, shipment status
Related Capabilities: Generate Invoice, Notify Customer
```

Kin Lane identifies three distinct layers in the API ecosystem that inform this catalog model:

- **API Resources** – Foundational CRUD operations that power digital infrastructure
- **API Capabilities** – Advanced operations that enable sophisticated interactions across platforms
- **API Experiences** – Meaningful combinations of resources and capabilities orchestrated across multiple providers

While quality API resources and capabilities exist abundantly, meaningful experiences remain scarce. Capabilities are the intermediate layer that bridges the gap between raw API resources and the orchestrated experiences that end-users actually value.

Governance and Security by Construction

Naftiko’s declarative model provides governance properties that are enforced structurally, not by convention.

Output Projection as a Security Boundary

The `outputParameters` declaration in every capability is the last structural control point before data reaches an LLM context. Only declared fields cross the boundary. An agent with broad backend credentials cannot inadvertently surface PII, internal system state, or unauthorized data into its context window because the projection is enforced at the framework level.

Role-Based Capability Variants

Today, Naftiko supports identity-scoped access through multi-variant capabilities with shared `consumes` adapters:

```
shared-customer-api.yml           <-- one consumes import
  |
  |-- support-customer-profile    <-- exposes: name, email, status
  |-- finance-customer-profile    <-- exposes: name, billing, balance
  |-- compliance-customer-profile <-- exposes: name, consent, retention
```

Each variant is a distinct capability registered to a specific agent role. The shared `consumes` import means the upstream connection is declared once. The output projection is the access control mechanism.

Determinism and Auditability

Every capability execution follows the same declared step sequence. There is no LLM-driven chaining, no probabilistic tool selection, and no runtime deviation from the specification. This makes capabilities:

- **Auditable** – Every execution follows the declared steps with logged inputs and outputs
- **Testable** – Mock mode enables deterministic evaluation without live systems
- **Reproducible** – The same input produces the same execution path every time
- **Promotable** – `$env` -parameterized base URIs allow the same capability to run against stubs in test and live APIs in production, controlled by a single environment variable

---

The Ecosystem of Capability Thinkers

The capability thinking movement draws on a growing community of practitioners who are converging on the same architectural insight from different vantage points.

Daniel Kocot (Architectural Bytes) has written extensively on the shift from resource-centric API design to capability thinking, arguing that the traditional concept of system edges has become fictional and that architecture should shift toward expressing intent, defining ownership, and aligning value creation with exposure. His work on “Beyond MCP” articulates why capability thinking, ecosystem design, and systems thinking deliver more strategic value than any single protocol.

Christian Posta frames the shift in terms of AI agent requirements, arguing that enterprises should expose "semantically rich declarations encompassing natural language descriptions, structured input/output specifications, execution preconditions, and usage examples" – the exact properties that Naftiko capabilities provide.

Mike Amundsen (Signals from Our Futures Past) emphasizes that autonomous API clients require documented capabilities rather than method names and URLs, drawing on conversations with Irakli Nadareishvili about adapting API design for AI-agent consumers.

Bruno Pedro (API Changelog) provides the practical methodology for translating user benefits into API capabilities, demonstrating how the Jobs-to-Be-Done framework maps to capability definitions and API documentation that serves both technical and business audiences.

Kristof Van Tomme (Pronovix) connects capability thinking to developer portal evolution and the concept of "solution portals," observing that capabilities are "productised affordances" – a bridge between sociotechnical interface theory and the practical needs of enterprise platform teams.

Irakli Nadareishvili challenges the data-centric habit directly: "The first step in breaking the data-centric habit is to stop designing systems as a collection of data services, and instead design for business capabilities."

Kin Lane (API Evangelist) consolidates these perspectives into a comprehensive framework of 35+ capability attributes spanning metadata, governance, security, observability, lifecycle, and economics.

Getting Started

The transition from APIs to capabilities can happen incrementally:

1. **Map existing APIs to business outcomes.** Identify the multi-step workflows that teams perform manually across multiple systems today. Each workflow is a capability candidate.
2. **Start with composite patterns.** Choose two to three workflows that chain SaaS tools your teams already use. Error triage (Sentry + GitHub + Slack) and lead provisioning (CRM + billing + notification) are proven starting points.
3. **Add a bridge capability for differentiation.** Identify one enterprise backend system (ERP, HRIS, clinical) that is currently invisible to your agent and automation tooling. A single bridge capability demonstrates value that generic MCP-only platforms cannot deliver.
4. **Build the capability catalog.** Document capabilities with business language, not endpoint specifications. Use the Jobs-to-Be-Done framework to ensure each capability entry answers "what problem does this solve?" before "how do I call it?"
5. **Invest in shared adapters.** Every `consumes` adapter you define for a common service (Slack, GitHub, Salesforce, Jira) makes the next capability cheaper to build. The compounding effect of shared adapters is the primary driver of capability ecosystem growth.
6. **Measure by workflow, not by connector.** Track adoption and outcomes by workflow execution – manual steps removed, time saved, failure recovery rate – not by how many integrations are listed.

Conclusion

The shift from APIs to capabilities is not a change in terminology. It is a structural change in how digital platforms describe, expose, and govern what they can do.

APIs answer the question "how do I call this system?" Capabilities answer the question "what can this system do for me?" In an era where the consumer of your digital services is increasingly an AI agent, an automation platform, or a business stakeholder without a developer background, the distinction is decisive.

Naftiko provides the declarative framework for making this shift concrete: YAML specifications that are machine-writable and human-readable, output projections that keep raw payloads out of LLM context, shared adapters that compound with every new capability, and protocol bridges that make enterprise systems accessible to agents for the first time.

As Daniel Kocot writes: “As ecosystems expand and automation becomes more common, being able to articulate a platform’s capabilities clearly will determine how well it can adapt. Moving from APIs to capabilities is not just a shift in terminology. It is a step toward making digital platforms more meaningful and resilient.”

References

- Kocot, Daniel. “Beyond the Edge: What Comes After the Gateway.” *Architectural Bytes*, November 9, 2025.
- Kocot, Daniel. “From APIs to Capabilities.” *Architectural Bytes*, September 2, 2025.
- Kocot, Daniel. “Beyond MCP.” *Architectural Bytes*, August 5, 2025.
- Posta, Christian. “From APIs to Capabilities to Support AI Agents.” *ceposta Technology Blog*, April 10, 2025.
- Amundsen, Mike. “Focusing on Capabilities Is a Win.” *Signals from Our Futures Past*, July 14, 2025.
- Lane, Kin. “What is a Capability?” *API Evangelist*, October 7, 2025.
- Lane, Kin. “API Resource, Capabilities, and Experiences.” *API Evangelist*, February 9, 2024.
- Pedro, Bruno. “Translating User Benefits into API Capabilities.” *brunopedro.com*, August 1, 2023.
- Pedro, Bruno. “Documenting Your API Around Its Capabilities.” *API Changelog*, August 8, 2025.
- Van Tomme, Kristof. “Capabilities Are Productised Affordances.” *LinkedIn*, 2025.