



adaptTo()

EUROPE'S LEADING AEM DEVELOPER CONFERENCE
28th – 30th SEPTEMBER 2026

AI-Driven Content Migration for EDS

Jack Jin and Katie Phillips

Introduction



Katie Phillips

Senior Product Manager
Best Buy



Jack Jin

Associate Director
Accenture

Ctrl

C

COPY

Ctrl

V

PASTE

**Every migration project ends
with the same two keystrokes.**

Different budgets. Different teams.
Different tech. Same shortcut.

Headless
content

Rich
experiences

Platform
migrations

Redesigns &
refreshes

Content Migration is the Drag

Site build



weeks

**Content
migration**



months

SCATTERED SOURCE CONTENT

PDFs

Legacy CMS

Static sites

React apps

Copy docs

AI Can Boost the Content Supply Chain

01

**Reads like
a human**

02

**Understands
intent**

03

**Adapts to
the source**

04

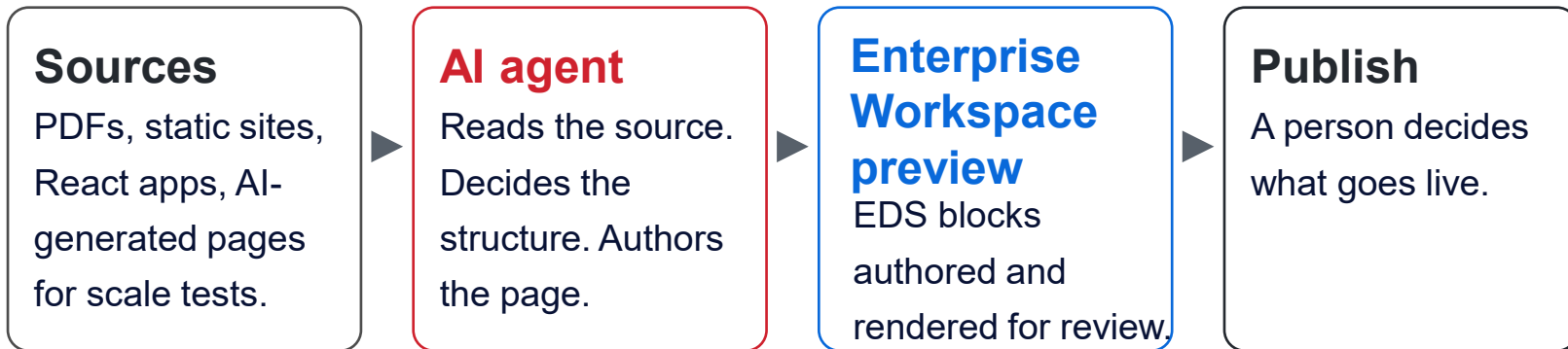
**Improves
itself**

The Vision

One agent-driven migration system, purpose-built for Adobe EDS, reducing the operational overhead of a traditional migration.

0 - the target system

Many messy inputs, one standard output, one person at the end.



 a person  a deterministic step  an agentic step  a mix of both

Core Capabilities

MEMORY

What the agent has learned

- Lives in Enterprise Workspace - readable, not a black box
- Humans edit it directly, in markdown
- Gets richer with every run

BLOCK LIBRARY

What good output looks like

- The canonical definition of every EDS block
- What each block can and can't do
- Authored examples the agent can match

How Do You Know It Worked?

STRUCTURE

The right EDS blocks, in a sane heading order.

ACCESSIBILITY

Contrast, alt text, ARIA, keyboard order.

CONTENT

Nothing lost from the source.
Nothing invented.

VISUAL

Renders the way it should, in a real browser.

Scored outside the migration flow - the thing being measured never grades itself.

The Demo

THE SETUP

Katie's Travel Journal:
real writing, trapped in a
hand-built 2004 blog site.

DESTINATION:

The Wilderness Journal
(on Enterprise
Workspace aka DA.Live)



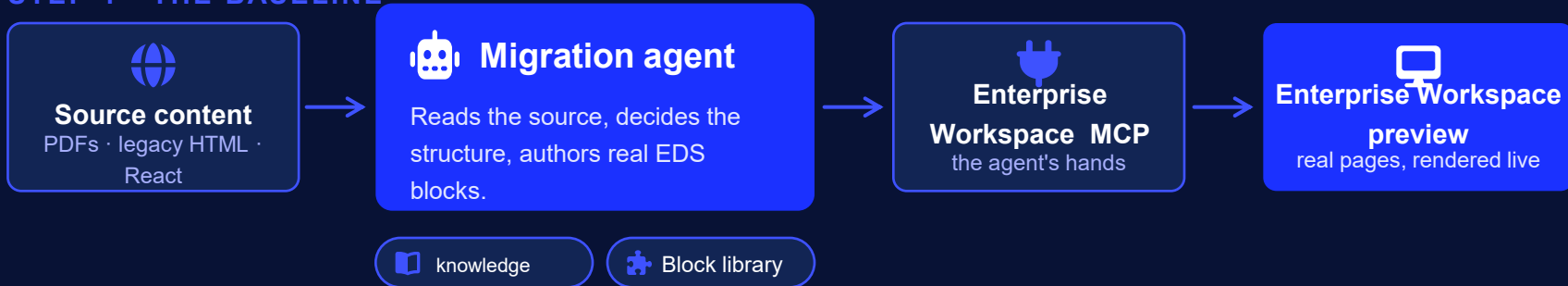
How It's Built

Open source · github.com/jackzhaojin/azure-da-mcp

First: One Good Migration Agent

An agent that knows da.live and the block library - then taught to check its own work.

STEP 1 · THE BASELINE



STEP 2 · SELF-CHECK

NEW



Playwright MCP - in the cloud

The agent opens its own preview in a real browser - and fixes what looks off before a person ever sees it.

THE CHECK LOOP



Verification runs against the rendered page, not the intended markup.

Then: Evals, and a Content Generator

Scores to sharpen the prompts. Synthetic pages to test against.



Evals

NEW

So we could improve the prompts on evidence.

- Every run scored on four dimensions: structure, accessibility, content, visual
- Deterministic tools, plus an agentic pass
- Scores feed straight back into the prompts



Content generator

NEW

A simple way to make test input on demand.

- Picks a topic, writes a brief
- Renders a synthetic source page
- Known ground truth to measure against



But each system had its own front door - and each one started by hand.

But Everything Was Separate

Migration and evals worked - they just never talked to each other.



Migration agent

Enterprise Workspace + Playwright
MCP



Evals

its own app, its own run



Generator

started by hand



Output moved between systems because a person moved it.

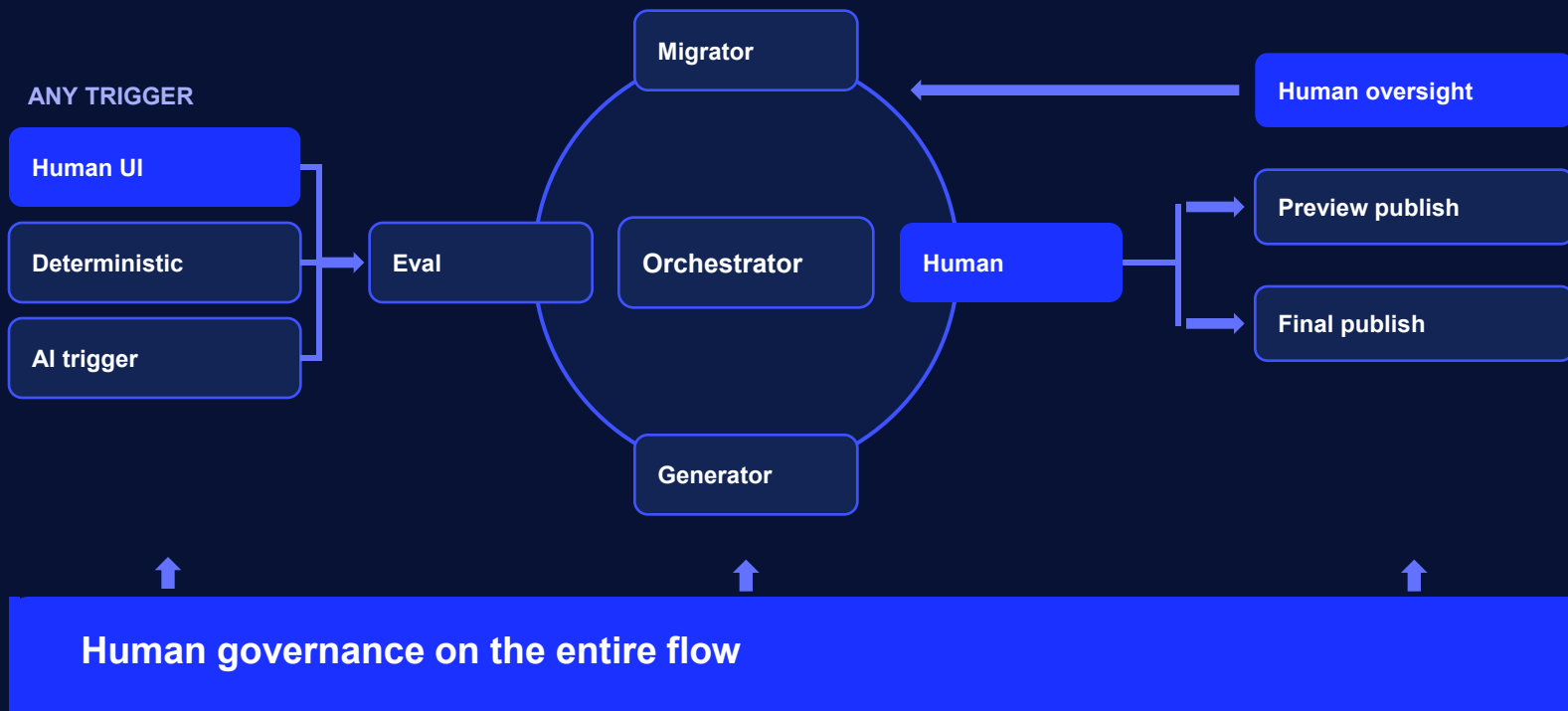
“We automated the copy-and-paste out of migration - then did the copy-and-paste of URLs between our own tools.”



We wanted something integrated, flexible, modern, and fits in just about everywhere with content operations

Conceptually: One Framework, Any Content Process

Any trigger, any route through the agents - and four places the human can sit.



The Rebuild Started Small

Two agents, one open protocol. Nothing else.



Coordinator

A plain route table: evaluate, migrate, generate + migrate, or the full loop. No model calls inside.

A2A

open protocol



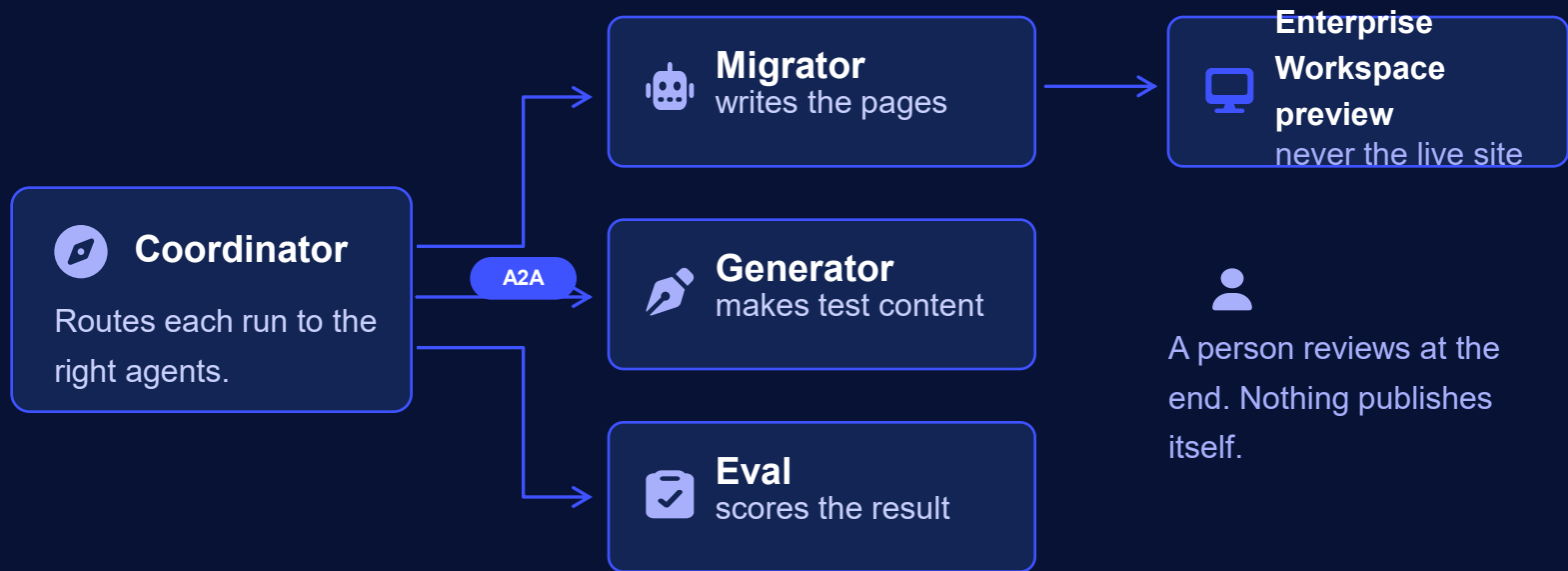
Migrator

One contract: migration.run. The model and the backend behind it stay swappable.

Everything from the baseline comes back next - same protocol, same contract style.

Then Every Piece Came Back - as an Agent

One mesh, one protocol, shared state. Run a single agent, or the whole loop.



One shared store - every agent writes full task state, so runs survive drops and cold starts.
Full diagrams in appendix

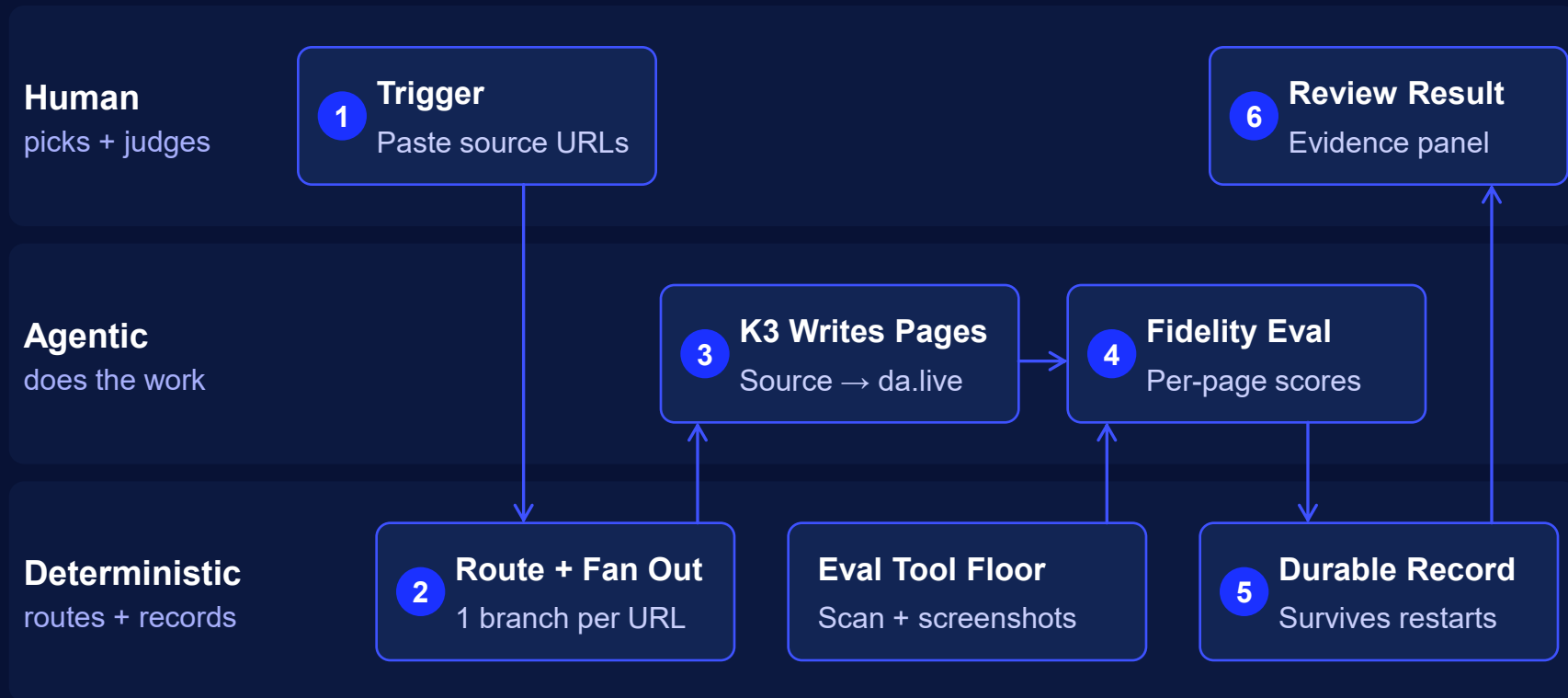
Five Models, One Eval Process: Memory On vs Off

Model	Structure	Accessibility	Content	Visual	Overall	Rules	Blocks
Kimi K3	93 ▲3	92 =	85 ▲3	90 ▲2	90 ▲2	8 of 8	5
Kimi K2.7	90 ▲4	92 =	84 ▲6	86 ▲1	88 ▲3	8 of 8	4
Claude Sonnet 5	95 ▲13	92 =	78 =	88 ▲3	88 ▲4	8 of 8	5
Claude Opus 5	95 ▲13	92 =	81 ▼2	93 ▲5	90 ▲4	8 of 8	5
Claude Haiku 4.5	90 ▲5	92 =	76 ▲2	79 ▼4	84 =	3 of 8	5

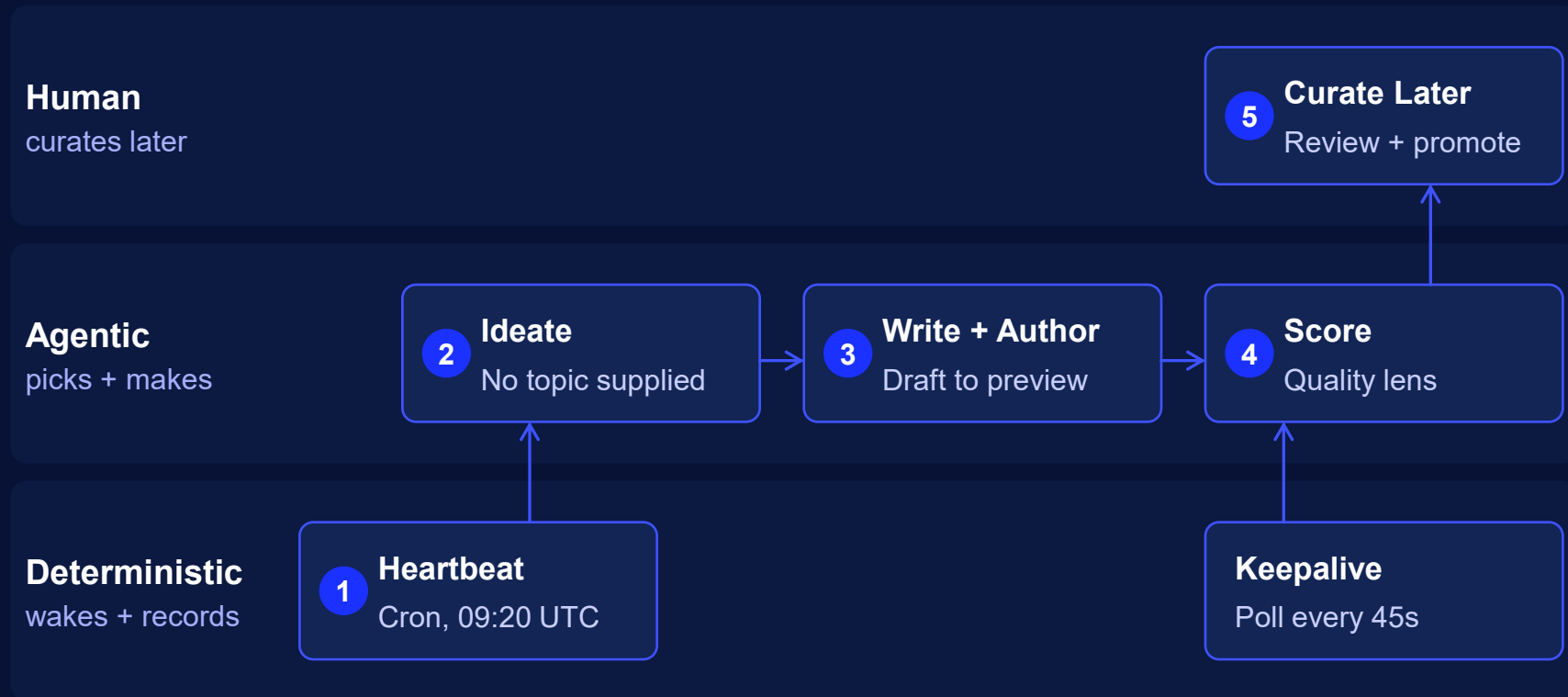
Scores are the 2026-09-08 run with agent memory on; ▲▼ = change against the 2026-08-29 run without memory, = no change (one run each, 2-4 points is within variance). Rules = memory rules the model applied, of 8. Blocks = block-library pages opened before authoring, from logged tool calls.

Patterns and Use Cases

Use Case 1: The Migration Route



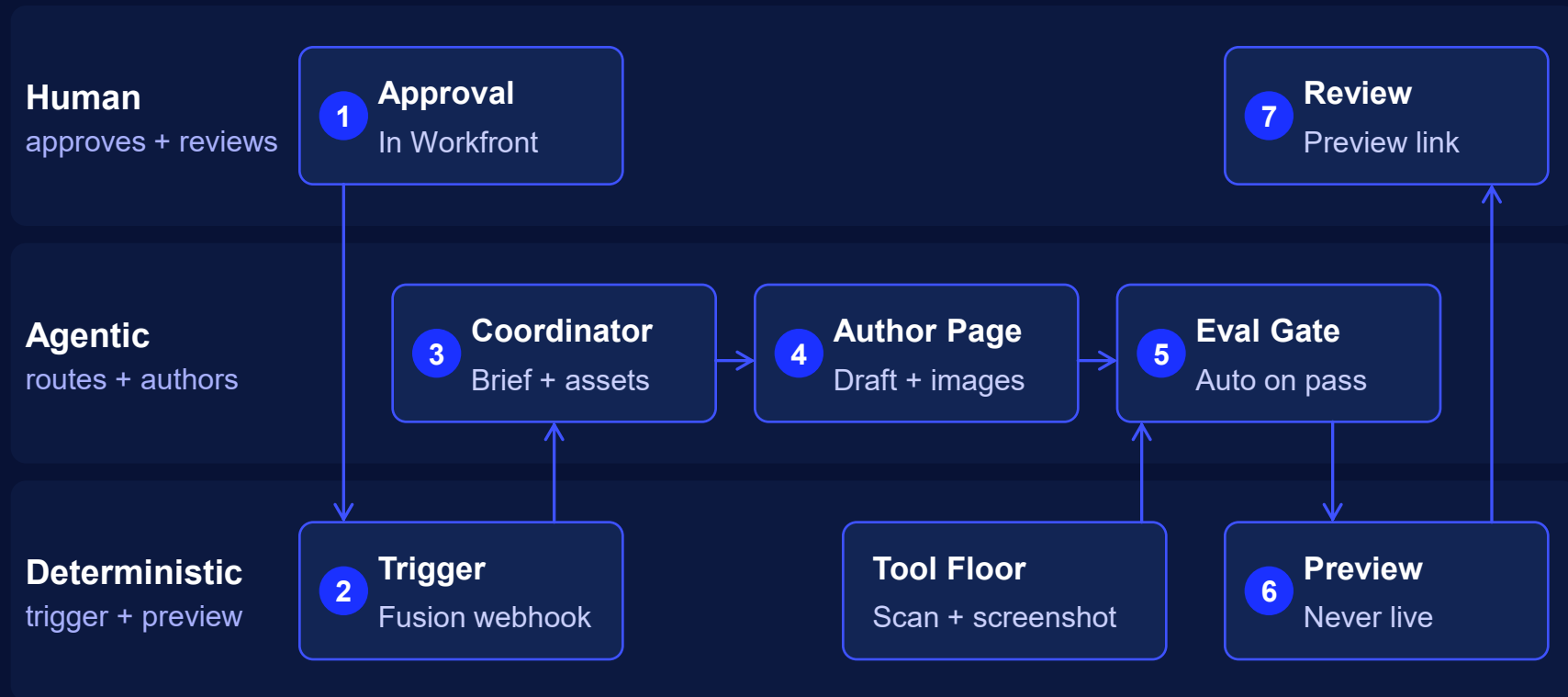
Use Case 2: The Daily Loop



Use Case 3: Trend Research



Use Case 4: Content Supply Chain



Final Takeaways



Agents Do the Work

- Adaptive authoring with Real EDS blocks and default content
- Any model vendor
- AI visual inspection



Evals Prove It

- Objective signal
- Repeatable scores
- Data over vibes



A Decoupled Mesh

- Open protocol
- Swap in config
- Infinite use cases

My Week in Berlin



Photo by [Levin](#) on [Unsplash](#)



Photo by [Aleksas Stan](#) on [Unsplash](#)

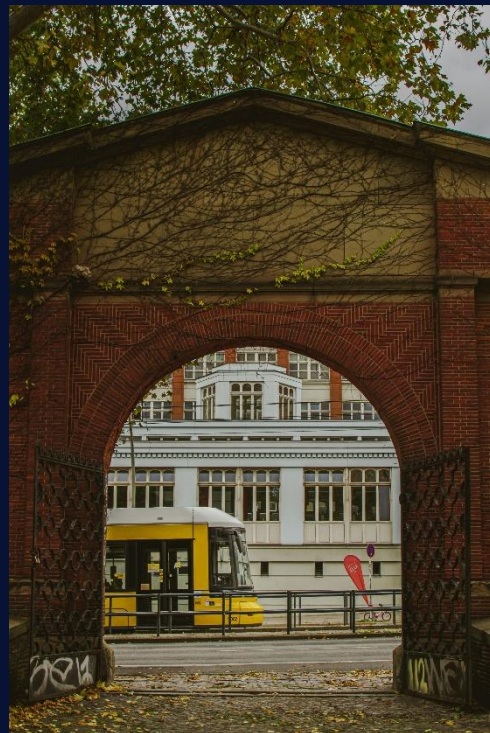


Photo by [Sam Lashbrooke](#) on [Unsplash](#)

Thank You

References



- Links

- Code: <https://github.com/jackzhaojin/azure-da-mcp>
- Demo site: <https://github.com/jackzhaojin/adapt-to-2026-demo>
- Documentation: <https://github.com/jackzhaojin/azure-da-mcp/tree/main/docs/architecture>
- Tools used: <https://github.com/jackzhaojin/azure-da-mcp/blob/main/docs/tools-used.md>
- Everything else (working demo, before and after pages, dashboard): <https://github.com/jackzhaojin/azure-da-mcp/blob/main/REFERENCES.md>

Appendix



Two protocols, two jobs: A2A agent-to-agent, MCP agent-to-tool

A2A is how work moves across the mesh. MCP is how one agent reaches a browser, a CMS, or a database. Same skill file, two model harnesses, both directions.

— A2A (agent to agent) - - - - MCP (agent to tool) plain code path (S3 / REST), no protocol on the wire between agents

Human

Deterministic

Agentic

Mix

Humans - the dashboard, in a browser

Google SSO - four trigger lanes - live activity - evidence - curation
in-the-loop, on-the-loop, over-the-loop: all from one page

UI → /api/trigger → A2A, identity attached

A2A vocabulary - what every agent speaks

Agent Card at /well-known/agent-card.json (discovery)
Task {id, contextid, status, artifacts}, saved on every transition
message/send - message/stream (SSE) - tasks/get - push webhooks
official @a2a-js/sdk, one Express server per agent

same three calls for every agent - the card says what it can do

content-gen :4002

skills: content.ideate - content.brief - synthetic sources
Claude Agent SDK query(), tool-free; template fallback
then plain code writes the source HTML to R2 (S3 API)

tools: none for the model; the artifact store for the code

no MCP at all - the simplest agent proved the chassis first

S3 API, not MCP

R2 - the artifact bytes

synthetic source pages (content-gen),
eval screenshots - public r2.dev URLs
written by code, not by a model tool

Schedules - GitHub Actions, no human identity

daily-content-loop.yml cron → pre-warm/health → coordinate.run full-loop
mesh token; the run is NULL-owned, shared with every signed-in human

Agent Card → /a2a (or /hooks for webhook-only callers)

coordinator :4004 - A2A client AND server

route table + fan-out (deterministic) - ideate + briefs (agentic)
one contextid per run; children's working-notes become the live feed
severed stream? recover from tasks/get against the persisted Task
three doors in, one door out: every caller becomes the same coordinate.run task

A2A message/stream (SSE) - bearer mesh token - one contextid across the run

migration :4003 - one card, swappable backends

dryrun - makecom - opencode (Kimi K3) - sdk (Claude Agent SDK)
same prompt + SKILL.md + the same two MCP servers under both
harnesses - the vendor swap never touches the tools

**tools: harness built-ins (skill, bash, read, write, edit, webfetch)
+ da.live.* x6 + playwright browser.* x5 (listed below)**

skill-as-a-service: da-live-author-playwright, reused verbatim

MCP over HTTP (streamable)

da.live MCP - the CMS as tools

our functions/ server on Azure Functions, streamable HTTP
self-authenticates: S2S technical account, anonymous inbound
**list_da.live_content - get_da.live_content
create_da.live_content - save_da.live_content
preview_publish_da.live_content - create_folder_da.live**
used by migration only; the caller needs nothing but a URL

REST, S2S

da.live / EDS - the content target

writes to /source/{owner}/(site)/ai-articles/...
then .aem.page preview - never live
the S2S account is a writer on adapt-to-2026-demo

Playwright MCP - a browser as tools

@playwright/mcp over stdio, headless Chromium
migration: --isolated - eval: a browser permit
**browser.navigate - browser.snapshot
browser.take_screenshot - browser.click
browser.fill_form**
migration: read source, validate - eval: live DOM, a11y

one permit

Headless Chromium

inside the container, --no-sandbox where root
eval: BROWSER_PERMITS=3, two merged caches
the shell-out scripts draw from the same pool

Other agentic solutions

any A2A client that reads the Agent Card: another mesh, a bot
webhook-only callers (Make.com, Fusion) use the edge shim

Edge shim - A2A for callers that don't speak it

POST /hooks/agent/skill with flat JSON → 202 (taskid)
the finished Task arrives at callbackUrl as a webhook (A2A push)
edge token, not mesh token - separate blast radius
Make.com, curl, Workfront Fusion: one POST, one webhook

eval :4001 - four dimensions

deterministic tools first, an Agent SDK pass second, then blend
Claude reaches the live page through MCP: DOM, a11y tree,
screenshots - then scores with vision

**tools: harness Bash / Read / WebFetch + playwright browser.*
+ filesystem read_file; the visual pass gets the PNGs as images**

browser permits cap every Chromium, MCP spawns included

MCP over stdio

filesystem MCP

@modelcontextprotocol/server-filesystem
eval reads its own screenshots back
read_file - list_directory
(scoped to the screenshots dir)
spawned per Agent SDK pass, eval only

A person, in Claude Desktop

"how did yesterday's run score, and why?"

MCP over stdio

store-mcp - the mesh exposes MCP too

read-only tools over runs - tasks - eval_reports
@modelcontextprotocol/sdk, no part, no writes

the store - SQLite locally, D1 in the cloud

the flight recorder every consumer can replay

Two harnesses, one skill: the migration prompt, SKILL.md, and both MCP servers are identical under opencode (Kimi) and the Claude Agent SDK.
Swapping the model vendor is a payload field; the tools, the CMS, and the browser never notice.

A2A vs MCP in one line: A2A is agent-to-agent (cards, tasks, streams); MCP is agent-to-tool (a browser, a CMS, a database). We use both, each in the direction it was built for.

Where the tools live: 6 CMS tools behind one MCP URL, 5 browser tools behind one stdio server, a filesystem reader for eval, and the harness built-ins. Storage is plain code: S3 to R2, SQL to the store.



The server side: one Worker, four containers, one database, one bucket

Human

Deterministic

Agentic

Mix

Everything the mesh runs on in production, as configured in agents/deploy/wrangler.jsonc. Scale-to-zero: it sleeps when idle and wakes in 5-30 s.

Cloudflare zone jackzhaojin.com (registrar: Cloudflare) · Workers + Containers · D1 · R2 · Tunnel · deployed by every v2.x tag

Five custom domains, all routed to the one Worker (routes in wrangler.jsonc, custom_domain: true)

content-factory (A2A) · content-factor-dash (the dashboard; the Google-OAuth-registered host) · content-factory-eval · content-factory-gen · content-factory-migrate · all jackzhaojin.com
the five *.xpri.ai twins still resolve in parallel (pending manual teardown) - the Worker preserves Host headers, so Auth.js sees the right origin

Worker "content-factory" - the only thing with bindings and secrets

hostname → Durable Object binding → one Container class each (max_instances 1) · owns the only D1 binding and serves the secret-gated POST /d1/query the containers call back into (~100 ms/query)
13 runtime secrets set once with wrangler secret put (D1 proxy, mesh + edge tokens, R2 keys, Google OAuth, Moonshot, Claude) · vars: R2_*, KIMI_MODEL_ID=k3, DALIVE_MCP_URL, MIGRATION_DEFAULT_BACKEND
/worker-health answers without waking a container (the deploy smoke probe) · observability on · wrangler tail for live logs

coordinator basic

1/4 vCPU · 1 GiB · sleeps after 30 min
coordinator.Dockerfile, Next build baked in
A2A server + Next.js dashboard, one process
restart policy: in-flight runs marked failed

any dashboard visit wakes it; 30 min keeps a demo warm

eval standard-3, the heavy one

2 vCPU · 8 GiB · sleeps after 15 min
non-root appuser, Claude CLI + MCP servers,
two Chromium caches, runtime .claude.json
axe-core, screenshots, four agentic passes

EVAL_CONCURRENCY=1 here: standard-1 starved

content-gen basic

1/4 vCPU · 1 GiB · sleeps after 15 min
non-root appuser, Claude CLI, no Playwright
the agentic writer, template fallback
briefs, ideation, synthetic sources to R2

lite (256 MiB) starved the Node + CLI pair

migration standard-1

1/2 vCPU · 4 GiB · sleeps after 15 min
opencode 1.18.25 pinned, kimi config baked,
skill synced in, Chromium for Playwright MCP
9-30 min agentic turns, streamed live

runs as root, so Chromium gets --no-sandbox

images are linux/amd64, built by wrangler deploy from the four Dockerfiles (build context = agents/), pushed to Cloudflare's managed registry · env reaches a container only as string vars from the Worker · cold start 5-30 s

D1 - a2a-agents the database

runs · tasks (full A2A Task JSON) ·
eval_reports · artifacts · push_configs
reached only via the Worker's /d1/query
same SQL as local SQLite, two drivers

schema by hand: wrangler d1 execute --remote

R2 - a2a-agents-artifacts the disk

public bucket on r2.dev, by design
eval screenshots, synthetic source pages
S3 API (SigV4) from inside the containers
same URL contract as the local /output

the URL is the handoff between agents and clouds

Container disk scratch only

/output, tmp, opencode sessions
gone on sleep, rollout, or replacement
screenshots are copied to R2 before the
task completes; the store keeps the URL

nothing durable lives here

Tunnel a2a-mesh legacy

a2a.xpri.ai → a laptop's :4003
Make.com's callback ingress to a LOCAL
migration agent, not the cloud one
cloudflared, config in ~/.cloudflared

last xpri.ai host; next: a2a.jackzhaojin.com

GitHub Actions

deploy-agents.yml on v2.x tags
daily loop cron, 0920 UTC
agents-e2e.yml on main + PRs

2 repo secrets: CF token + account id

Google OAuth · SSO

Auth.js v5, JWT sessions, no DB
callback on content-factor-dash
optional AUTH_ALLOWED_EMAILS list

runs.user_email names the person

Azure Functions - da.live MCP

jack-mcp-azure-ai-function
azurewebsites.net/api/mcp-streamable
S2S technical account inside

our one piece outside Cloudflare

Model vendors

Moonshot Kimi K3 at api.kimi.com
through the in-process kimi-proxy
Anthropic Claude via the Claude CLI

keys on the Worker, never in git

da.live / EDS - the target

site adapt-to-2026-demo
main--adapt-to-2026-demo--jackzhaojin
aem.page, preview only

written via the MCP, never live

Local parity: the identical code runs on a laptop · SQLite files for D1, localhost ports for the hostnames, /output for R2. The driver is picked by env; behavior never drifts.

Why this shape: the Worker is the only thing with bindings and secrets; the containers are plain Node processes that could run anywhere - which is exactly why the same code runs on a laptop.

What it costs: scale-to-zero, so the first request after idle pays a 5-30 s cold start; rollouts replace containers and kill in-flight runs, so deploy between runs, not during.



Dev ops: trunk-based, tag-driven, and a laptop is never in the release path

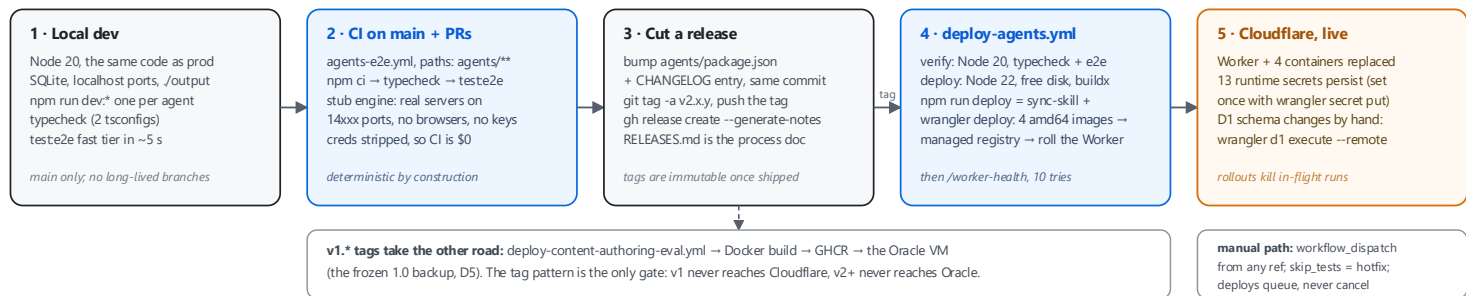
Human

Deterministic

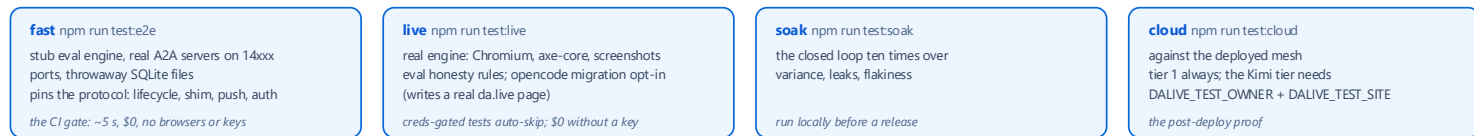
Agentic

Mix

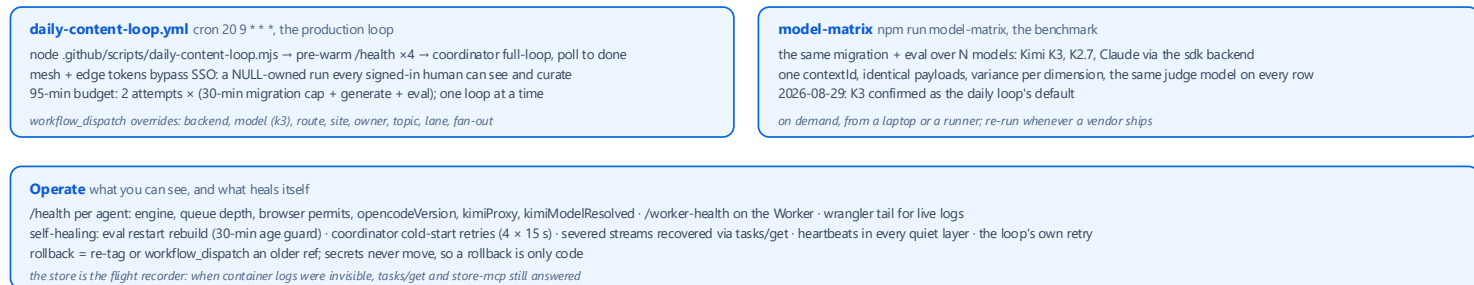
Two tag lines that never cross: v1.* redeploys the frozen 1.0 app on Oracle; v2.x and later builds four containers and rolls the Worker on Cloudflare.



The test tiers real servers, no mocks; each tier a different price



Loops that run without us GitHub Actions as the scheduler



The release is a git tag. Nothing on a laptop is required after the push: GitHub builds the images, wrangler rolls the Worker, the smoke probe passes or the run goes red.

Two secrets in GitHub, thirteen on the Worker: CI only authenticates wrangler; every runtime credential was set once with wrangler secret put and persists across deploys.

Where the human sits: at the keyboard for the tag and at the dashboard for curation; everything between, including the daily production run, is a workflow file.

Migration Orchestration

In the baseline system, Make.com and Workfront Fusion ran the workflow. The agent handled the content.

1.0 — migration orchestration

Orchestration lived in Make.com; intelligence lived in the agent. The separation was intentional — and it's the seed the coordinator grew from.

Central migration agent

understands the source content, reasons about structure and intent, and makes the authoring decisions — not a rules engine, but intelligence applied to each migration task

the judgment stays in one place, on purpose

Playwright MCP — the AI's eyes

vision into live websites: navigate, interact, capture the visual and structural information that static exports miss

capabilities handed to the agent as MCP tools

DA-Live MCP — the AI's hands

read existing content, save new content, preview the output, publish — direct integration, no brittle intermediary steps

the AI operates on the real system

The orchestrated flow — repeatable across thousands of pages

01 · Trigger

manual, on a schedule, or in response to events like content updates

02 · Agent execution

the orchestrator invokes the agent with specific source content and configuration

03 · DA-Live save

content lands directly in DA-Live — version history kept, human review enabled

04 · Preview + validate

automated preview generation gives immediate visual verification of the result

05 · Publish

after validation — to production, or staged for batch publishing

Why this layer mattered: the same workflow, run repeatedly across thousands of pages with consistent quality and traceability — and controlled experimentation: different agent configurations through one orchestration flow, objectively compared.

What survived into 2.0: orchestration separated from intelligence — the coordinator is this idea, given a protocol.

Workflow and content

The orchestrator owned sequencing, retries, and coordination. The agent read the source and wrote the output.

Human review

Trigger, agent execution, save to DA-Live, preview, publish. A person reviewed the preview before anything went live.

Carried forward

The coordinator in the new platform keeps this split: workflow in plain code, judgment in the agents.

Piece one: orchestration and intelligence live in separate places.

Playwright: Giving AI Vision

A remote Playwright MCP let the agent open the migrated page and check its own work in a real browser.

1.0 — Playwright: giving the AI vision

A cloud-hosted remote MCP with Chromium — live browser eyes for the agent, no local infrastructure required.

AI needs to see real output

The agent loads the actual page the way a user would — inspecting DOM structure, verifying the render, confirming visual and structural integrity against what was intended.

a static export can't tell you what the page actually looks like

Test, verify, and correct

A visual feedback loop: immediate testing, validation of outcomes, and intelligent self-correction — closing the gap between what the agent intended and what actually shipped.

the agent critiques its own work before any human sees it



Why it mattered: confidence came from the agent validating its own actions in a real environment — understanding what it did, not just doing it.

What survived into 2.0: every authoring backend still runs this loop, and the eval agent screenshots the real render — the mesh never let go of the eyes.

The check loop

Author the page, render the preview, look at it, fix what is off. Verification runs against the rendered page rather than the intended markup.

Real output

Screenshots and DOM checks come from the same preview a reader would see.

Carried forward

Every authoring backend still checks its own preview, and the eval agent screenshots pages in real Chromium.

Piece two: agents verify their output in a browser.

Why Evals Are Part of the System

Evaluation was designed in from the start, and it ran separately from the migration flow.

1.0 — why evals were part of the system

Designed in from day one, not bolted on: objective, repeatable signal about what works and what doesn't.

Measurement over intuition

human review of migrated content is valuable but doesn't scale — automated evaluation gives consistent, objective assessment across thousands of pages, surfacing patterns manual review would miss

a number per page, not a vibe per spot-check

Decoupled evaluation ★

evals intentionally separate from the migration flow — quality is reasoned about independently of how the content was produced, enabling fair comparison between different migration approaches

the most mesh-shaped idea in the whole 1.0 deck

Continuous improvement

every migration run produces measurable outcomes; outcomes inform changes to prompts, models, context allocation, or memory; the next run measures whether those changes actually improved quality

the loop that separates iteration from guessing

What the signal was used for

Assess AI models

which LLM fits which migration task

Optimize strategies

memory + prompt configurations, tuned by evidence

Compare migration approaches

AI-driven vs human-led vs hybrid, judged by one rubric

Why it mattered: this feedback loop is what turns a prototype into an enterprise capability — without it, improvements are guesswork; with it, every change is validated by data.

What survived into 2.0: the starred card became an agent — the eval service is decoupled evaluation given its own address, its own store, and a mode badge on every score.

Different approaches can be optimal for different content types, timelines, or quality bars — universal evaluation lets that be a decision, not an assumption.

Four dimensions

Structure, accessibility, content, visual. Deterministic analysis plus an agentic pass in the baseline app.

Separate on purpose

Evals ran outside the migration flow, so models, prompts, and whole approaches could be compared on the same footing.

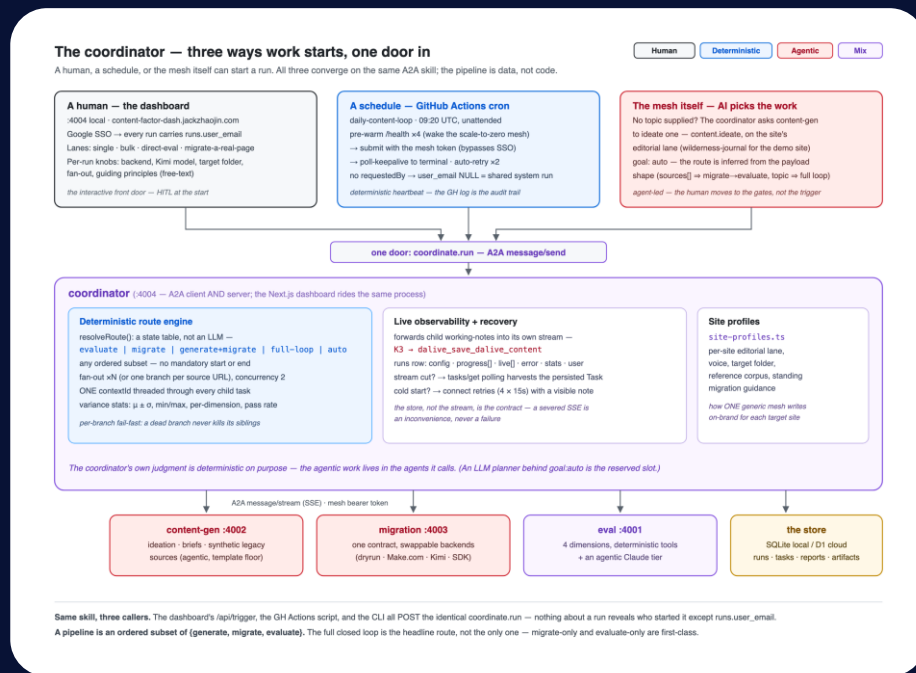
Carried forward

The engine moved into its own agent with its own store. Scores now also record how they were produced.

Piece three: measurement, kept independent of the thing it measures.

The Coordinator

A run can start from the dashboard, from a schedule, or from the mesh itself. All three paths call the same skill, **coordinate.run**.



Three entry points

The dashboard (Google SSO, per-run settings down to the model), a GitHub Actions cron, or the mesh picking its own topic and route.

Deterministic routing

The coordinator is a route table: evaluate, migrate, generate+migrate, full-loop, or auto, in any order. No model calls inside it.

Branches and variance

One contextId per run, N parallel branches, variance stats (mean, sigma, pass rate), and a live activity feed.

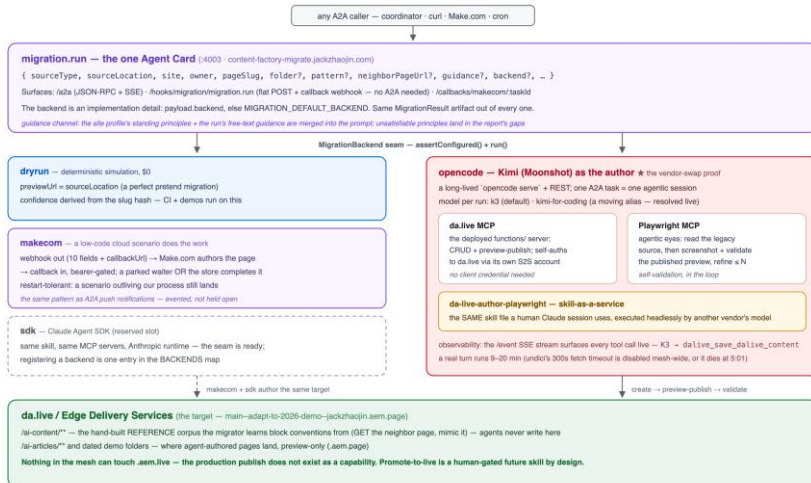
Judgment stays in the agents it calls. The coordinator stays plain code.

The Migration Agent

One contract, **migration.run**. The backend behind it is swappable.

The migration agent — one contract, any runtime, any vendor

A facade: one A2A Agent Card fronts swappable backends. "Claude vs Kimi on the same ten migrations" is a config change, not a rewrite.



The seam is the point. A simulation, a low-code cloud scenario, and a non-Anthropic frontier model all serve the identical migration.run contract — callers cannot tell which ran except by reading the report.
Verified end-to-end: Kimi authored a real da.live page from inside a Cloudflare container and the real eval scored it 91 — one pipeline, two model vendors, zero contract changes.

Four backends

dryrun (a simulation, used by CI), Make.com (low-code scenario), opencode running Kimi (a non-Anthropic model, headless), and an SDK slot.

One shared skill file

Kimi uses the same da-live-author-playwright skill a Claude session uses, plus a da.live MCP with its own service account and a Playwright MCP.

Preview only

No part of the mesh can publish to the live site. That capability does not exist in it. Publishing stays with a person.

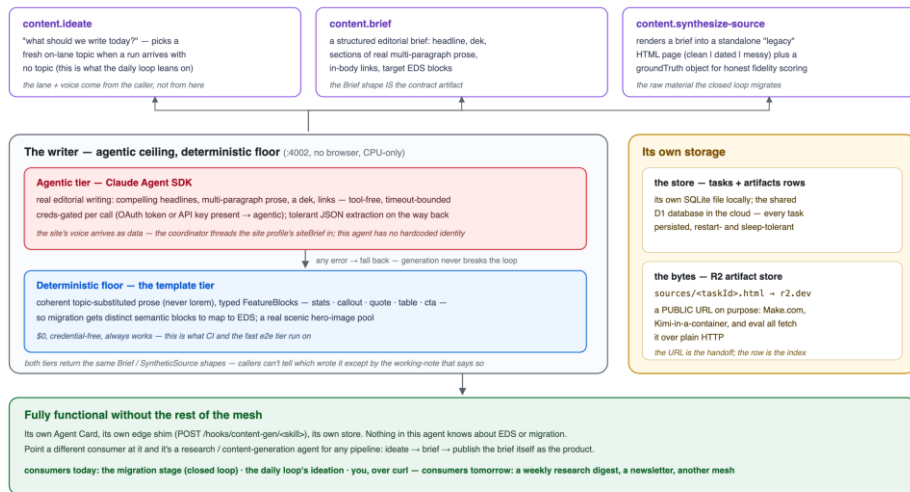
Comparing two models on the same ten migrations is a configuration change.

The Content Generator

Three skills: pick a topic, write a brief, render a synthetic source page with a known ground truth.

The content generator — a complete agent on its own

Three skills, an agentic writer with a deterministic floor, and its own storage. Migration is one consumer of its output — not the owner of it.



Decoupling test: could this agent move to a different repo, a different team, a different company — and keep working? Yes: callers hold only its URL and its contract.

Resilience rule: the agentic tier can time out, return garbage, or have no credentials — the deterministic floor answers anyway. A red generate stage always means something else.

Two tiers

An agentic writer (Claude Agent SDK) over a deterministic template that always works and costs nothing. Agentic failures fall back to the template and are recorded.

Its own storage

Its own store rows, plus artifact files on R2 under public URLs. The URL is the handoff to whatever consumes it.

Standalone

Nothing in this agent knows about EDS or migration. Use case three shows it working alone.

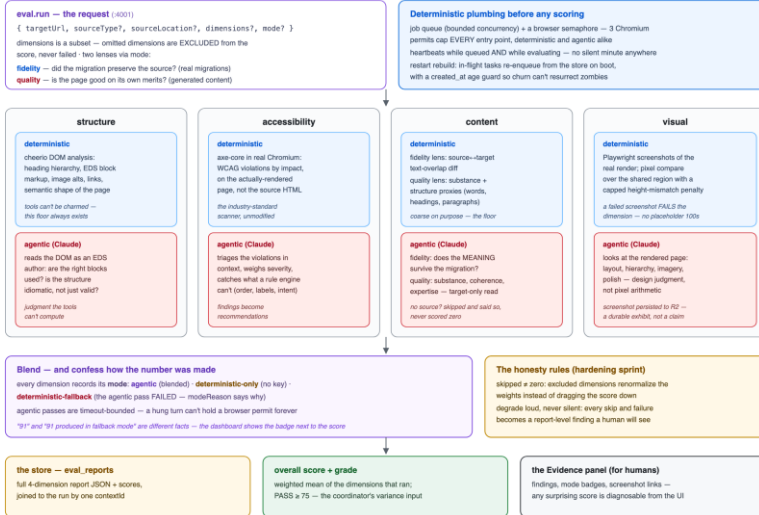
A generation failure never stops the rest of a run.

The Eval Agent

Four dimensions: structure, accessibility, content, visual. Each is scored twice, then blended.

The eval agent — deterministic floor, agentic ceiling, honest blend

Four dimensions, each scored twice — by tools that can't lie and by a model that can judge — then blended, with the blend itself on the record.



Why the two lenses matter: the same generated article scored 67 under fidelity and 88 under quality — fidelity-to-a-throwsaway-synthetic-source was the wrong question. The rule picks the right lens automatically.

The war story: a dead URL once scored visual=100 off a placeholder screenshot, and the agentic tier had silently never run locally. Every silent failure is now loud, visible, and on the record.

Two scoring tiers

Deterministic tools (cheerio, axe-core, Playwright screenshots in real Chromium) plus an agentic pass for what tools cannot measure.

Mode badges

Each dimension records how it was scored: agentic, deterministic-only, or fallback, with a reason. Skipped dimensions renormalize instead of counting as zero.

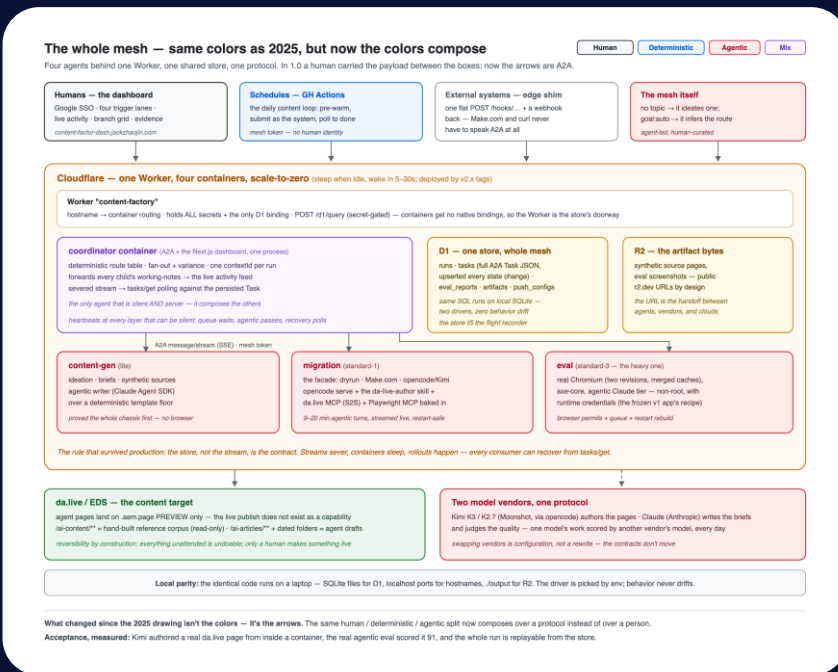
Two lenses

Fidelity asks whether the migration preserved the source. Quality asks whether the page is good on its own. The route picks the lens.

A dead URL once scored visual 100 from a placeholder screenshot. These rules exist because of bugs like that.

The Full Mesh

Four agents behind one Cloudflare Worker, one shared store, artifacts on R2, da.live as the preview-only target.



State lives in the store

Streams drop and containers sleep, so every agent writes full task state on every change. Anything can recover by polling tasks/get.

Laptop and cloud parity

SQLite locally, D1 in the cloud, same SQL and same code. Local runs behave like production.

Measured result

Kimi wrote a real da.live page from inside a container and the agentic eval scored it 91, end to end on Cloudflare.

Same legend as the 2025 drawing. The connections are now A2A calls instead of a person.