BIBFRAME Workshop in Europe 26

AI Agents in an Experimental Technical Services Workflow

https://github.com/jermnelson/tech-services-ai-workflow

Jeremy Nelson (jpnelson@stanford.edu)
Stanford University Libraries

DISCLAIMER: Generative AI was utilized in the creation of this presentation

What are AI Agents?

They act, not just answer — chatbots return text; agents do the work based on goals.

Tools are their hands — APIs, scripts, and commands define what they can reach.

They work in a (ReAct) loop — observe, decide, act, check the result, repeat.

Instructions, not code — you describe the goal in plain language; the agent works out the steps.

Humans stay accountable — people approve the decisions that matter, and can always stop the run.

AI Agentic Harnesses

The harness is the runtime

Around the model. It holds the loop, tools, and permissions.

What they all provide

File access, shell, API & MCP tools, skills, approval gates.

Commercial

Vendor-tuned, tied to their own models: Claude Code, OpenAI Codex, Google Antigravity

Open source

Model-agnostic, self-hostable: OpenCode, goose, pi.dev

Why it matters

Commercial buys capability; open source buys control.

Portable

The workflow is markdown and MCP, not vendor lock-in.

Using AGENTS.md (https://agents.md/)

A README for the agent
how to set up, run, and work in the project.

One file, many harnesses
no required structure; just markdown the agent parses.

Best practice: be specific and correct
exact commands, real file paths, explicit prohibitions ("never commit `.env`"). A stale instruction is worse than none.

Best practice: treat it as code
commit it, review it, update it when the project moves.

In this project
it names the environment variables, the four-stage skill pipeline, and where the authoritative BIBFRAME mappings live.

About SKILL.md (https://agentskills.io/)

A skill is a folder with a SKILL.md

`name` and `description` frontmatter, then instructions.

Progressive disclosures

agents load only the name and description at startup, then the full instructions when a task matches.

Open format

write a skill once, run it in any skills-compatible agent.

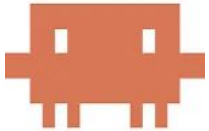
Best practice: the description is the trigger

say *when* to use the skill, not just what it does.

Best practice: keep SKILL.md lean

detail and documentation goes in `references/`, deterministic steps go in `scripts/`, configuration files go in `assets/`

Example Technical Services Workflow



opencode

HUMAN APPROVAL

AGENT HARNESS

opencode

Claude Code

Plans and coordinates

SOURCE SYSTEM

FOLIO Inventory API

Retrieves the record

STRUCTURED OUTPUT

BIBFRAME

Creates Work + Instance

CONTROLLED TOOL

Blue Core MCP

Adds RDF resources

folio

future of libraries is open

bf:

BIBFRAME

Blue Core

folio-inventory-retrieval SKILL.md

Frontmatter Trigger

A `name` and a one-line `description` are all the agent needs to know when to reach for this skill.

Instructions, Not an App

Hands the agent a `FolioClient` snippet, credentials from `.env`, and links to Inventory and Search APIs. No wrapper code.

Three Ways In, One Outcome

UUID or HRID goes straight to Inventory. Otherwise, search by title/author as a fallback and ask user to choose before pulling data.

Demos with Claude Code & Open Code w/Local Model

01. Inspection

Show the 36-line `SKILL.md` — the entire stage, detailed in clear prose.

02. Direct Retrieval

Run with just an HRID; watch the agent authenticate and fetch Instance + Holdings JSON.

03. Search Fallback

Run with only title and author, trigger the interactive cataloger selection prompt.

bibframe-transform SKILL.md

The Mapping Table is the Code

Two markdown tables carry FOLIO Instance and Holdings JSON into BIBFRAME Work and Instance JSON-LD, field by field. No transformation script to maintain.

Cataloging Judgment, Written Down

FOLIO reference-data UUIDs resolve to labels through the same `FolioClient`; unlinked controlled values become blank nodes with `rdfs:label`, not invented URIs.

Hard-Won Constraints

Don't set `bf:hasInstance` before the Work exists, and don't dedupe identifiers that share a value under different types. Quirks are documented, not rediscovered.

Demo with Claude Code & Open Code w/Local Model

01. Inspection

Show the `SKILL.md` — prerequisites, JSON-LD conventions, and the two mapping tables.

02. Direct Transformation

Hand the agent the Instance JSON; watch it resolve reference data and emit Work + Instance JSON-LD.

03. Auditable Mapping

Point at one table row and its triple — mapping is auditable by a cataloger, not just a developer.

bibframe-validation SKILL.md

A Ported Tool, Not a New One

`scripts/validation.py` is a CLI port of the BIBFRAME Interoperability Group's demo validation tool. The derived s SPARQL files from BIG's DCTap TSV files.

The Shapes Select Themselves

BIG shapes pick focus nodes by `sh:targetClass`, so every shape file for one resource kind merges into a single graph and only the applicable shapes fire.

Advisory, Not a Gate

Validation runs with `allow_warnings=True`, so only Violations break conformance. These shapes describe interoperability expectations, not Blue Core ingestion requirements.

Demo of Claude Code & Open Code w/Local Model

01. Inspection

Show the SKILL.md iinput contract, shape-selection table. Then `--summarize` lists every node shape with its constraints.

02. Direct Validation

Run with just an HRID; the Work and Instance graphs are read from `output/` and validated separately.

03. Reading the Report

Interpret validation report of Work and Instance and give confirmation to continue, modify graphs, or re-run bibframe-transform skill.

blue-core-mcp-ingestion SKILL.md

Thirteen Lines and a Pointer

The whole skill is three environment variables and one endpoint: `$BLUECORE_URL/api/mcp`. It doesn't describe the ingestion calls at all — the MCP server advertises its own tools once the agent connects.

A Bridge, Not a Client

The harness speaks MCP over stdio; Blue Core speaks MCP over authenticated HTTP. `blue_core_mcp.py` is a temporary proxy between them — trade the password for a Keycloak bearer token, carry the session id, unwrap the event stream. This will proxy will no longer be needed when latest API is deployed.

Work First, Then Instance

Blue Core mints the real URIs, so the locally generated `@id`s are placeholders. Create the Work, take back the URI the server returns, then point the Instance's `bf:instanceOf` at it. The other order 500s.

Demo with Claude Code & Open Code

01. Inspection

Show the 13-line `SKILL.md`, then the proxy it points at. Nothing here is an ingestion API wrapper — the tool list comes from the server at connect time.

02. The Approval Gate

Read the turtle in `output/a13616108` and approve it. Nothing reaches the datastore until a cataloger says yes; rejecting loops back to the transformation, it doesn't end the run.

03. Two Creates, In Order

Work goes up first; the agent captures the returned URI, writes it into the Instance's `bf:instanceOf`, and creates the Instance second.

04. Same Bridge, Either Harness

The proxy is a plain stdio script, so Open Code launches it exactly as Claude Code does. Only the server registration is harness-specific — the credentials, the token exchange, and the skill are not.

¡Muchas gracias!
Thank-you!

Email: jpnelson@stanford.edu
Source code: https://github.com/jermnelson/tech-services-ai-workflow