

xskill: Team-Level Skill Distillation, Sharing, and Evolution for Coding Agents

Anonymous Author(s)
Anonymous Institution

Abstract

Coding agents increasingly rely on *skills*—reusable markdown documents that encode procedural knowledge for recurring tasks. In team settings, skills learned by one developer remain trapped in individual session histories. We present **xskill**, an open-source framework that operates as a management layer above coding-agent runtimes to automatically distill skills from real user trajectories, share them across a team, and evolve them through data-driven canary deployment. xskill’s three-agent pipeline (TaskAgent, TaskClusterAgent, SkillEditAgent) decomposes trajectories into atomic task segments, clusters them into skill candidates with cumulative evidence thresholds, and produces versioned SKILL.md artifacts via git commits. A canary evaluation protocol—drawing on established online controlled experiment methodology—routes a subset of users to staging skill versions and compares behavioral UX scores against the main branch, promoting or freezing candidates based on statistical evidence. A cross-sectional source-code-level survey of 10 trajectory-to-skill systems reveals that none implements canary-based A/B evaluation, and that skill quality across all systems is ceiling-bounded by the authoring LLM’s capability. We provide a deployment scenario analysis using real skill inventory data (139 loaded skills, 26,448-character listing budget overflow) and formalize the UX scoring function and canary statistical protocol. xskill is the first system to combine asynchronous cross-agent trajectory collection across five ecosystems, cumulative-evidence skill production, canary-based evaluation, and multi-runtime skill distribution.

1 Introduction

The rise of agentic coding assistants—Claude Code [1], Codex [2], OpenCode [3], and others—has transformed software development workflows. These agents leverage *skills*: reusable, markdown-formatted procedural documents (typically named SKILL.md) that encode domain-specific knowledge, best practices, and step-by-step instructions for recurring tasks. The concept parallels long-standing observations in organizational knowledge management [21]: valuable procedural knowledge is generated through practice but easily lost without systematic capture and distribution mechanisms.

In team settings, the skill problem compounds. Developer A spends an afternoon teaching her agent a deployment workflow; Developer B, facing the same task, starts from scratch. The knowledge is trapped in individual session histories. Even when a workflow is manually documented as a skill, there is no mechanism to:

- **Automatically distill** successful agent trajectories into reusable skills;

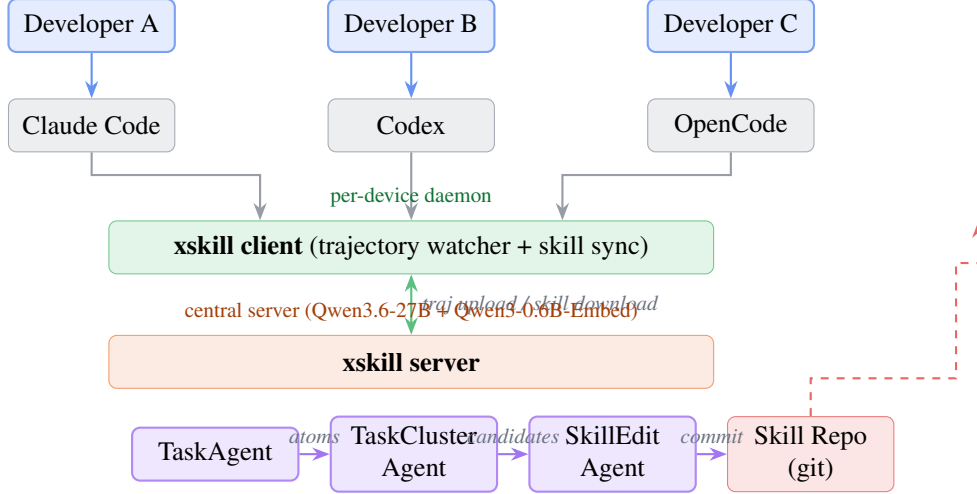


Figure 1: **xskill system overview.** Developers use their preferred coding agents. The xskill client daemon watches trajectory directories and uploads sanitized deltas to the central server. The server’s three-agent pipeline decomposes trajectories into atomic tasks, clusters them into skill candidates, and produces versioned SKILL.md artifacts via git commits. Evolved skills are synced back to each developer’s agent runtime.

- **Share and distribute** skills across team members with appropriate sanitization;
- **Evolve** skills based on real usage data, retiring those that degrade and promoting those that help.

We present **xskill**, an open-source system that addresses all three challenges. xskill operates as a *management layer* above existing coding-agent runtimes (Figure 1). It does not replace any agent’s skill loader; instead, it shapes what those loaders read from disk. The key insight is that the *lifecycle* of a skill—creation, evaluation, evolution, retirement—is orthogonal to the *runtime* that loads and injects skills into agent prompts, and should therefore be managed by a dedicated system.

Our contributions are:

1. A **cross-agent trajectory collection** architecture supporting 5 coding-agent ecosystems through a unified normalizer interface (§3.1).
2. A **three-agent pipeline** with cumulative evidence thresholds for automatic skill distillation, formalized as Algorithm 1 (§3.2).
3. A **canary evaluation protocol** using git branching, behavioral UX scoring, and statistical significance testing—the first such mechanism in the trajectory-to-skill literature, formalized as Algorithm 2 (§3.3).
4. A **comprehensive source-code-level survey** of 10 trajectory-to-skill systems across 5 design dimensions, providing the first systematic comparison for this emerging field (§2, §4).
5. A **deployment scenario analysis** with real skill inventory data and a worked example demonstrating the full skill lifecycle (§5).

Table 1: **Landscape of trajectory-to-skill systems.** Columns indicate whether the system produces SKILL.md files, supports online (per-turn) triggering, implements deduplication, performs canary/A-B evaluation, and targets production coding-agent deployment.

System	SKILL.md	Online	Dedup	Canary	Production
OpenSpace	✓	✓	✓	×	✓
EvoSkill	✓	×	×	×	✓
AutoSkill	✓	✓	✓	×	✓
Hermes	✓	×	×	×	×
GEPA-gskill	✓	×	×	×	×
Trace2Skill	×	×	×	×	×
AgentEvolver	×	×	×	×	×
MemSkill	×	×	×	×	×
EvoAgentX	×	×	×	×	×
SE-Agent	×	×	×	×	×
SkillRL	×	×	×	×	×
xskill	✓	✓	✓	✓	✓

2 Related Work

We surveyed 10 systems that convert agent trajectories or experiences into reusable procedural knowledge. Table 1 presents the landscape overview. All assessments were verified against source code (with path:line citations maintained in supplementary material) rather than documentation claims alone. Below we discuss the most instructive comparisons and connect xskill to the broader research landscape.

2.1 Skill-as-SKILL.md Systems

OpenSpace [4] is the closest system to xskill in scope. It maintains an evolving skill library via an MCP server, with three evolution triggers (per-task analysis, tool-degradation events, and periodic counters) feeding into a 3×3 evolution type matrix (FIX / DERIVED / CAPTURED). However, OpenSpace wraps skill execution inside its own sub-agent—meaning the host agent’s harness (e.g., `CLAUDE.md` rules) does not apply during skill-enhanced execution, a fundamental isolation problem for production use. OpenSpace evaluates skills via success/failure counts and LLM judgments, but never compares skill versions against each other in a controlled setting.

EvoSkill [5] uses git branches as program versions and Pareto frontiers for selection, offering the cleanest version-control story among all surveyed systems. However, it requires offline batch execution and does not support real-time trajectory collection.

AutoSkill [6] is the only system besides xskill that achieves per-turn asynchronous skill extraction during live chat. Its three-layer deduplication sandwich (identity hash \rightarrow semantic+BM25 \rightarrow LLM judge) is the most sophisticated deduplication mechanism we found. AutoSkill also provides the only per-turn usage tracking via an LLM judge that labels each injected skill as `relevant/used`, but this measures skill *selection* quality, not downstream user experience.

Trace2Skill [7] proposes a MapReduce-style patch merging approach and demonstrates that machine-generated skills can outperform human-written ones on SpreadsheetBench. Its key finding—cross-model skill transfer is viable—supports xskill’s design of centralized skill production serving heterogeneous client runtimes.

2.2 Non-SKILL.md Systems and Broader Context

Several systems use alternative skill representations: **MemSkill** [9] trains PPO controllers over operation banks; **EvoAgentX** [10] evolves workflow code; **SE-Agent** [11] replaces per-instance system prompts; **SkillRL** [12] trains RL agents with JSON skill records; and **AgentEvolver** [8] delegates to an external ReMe memory service. These target RL training loops rather than production coding-agent deployments. **GEPA** [13] introduces system-aware three-way merge by common ancestor, analogous to git’s merge algorithm applied to prompt evolution.

xskill also connects to the broader literature on agent self-improvement. **Voyager** [17] pioneered the concept of an automatically growing skill library for embodied agents in Minecraft, demonstrating that LLM-authored programs can compose into increasingly complex behaviors. **Reflexion** [18] showed that agents can improve from trajectory-level verbal feedback, an insight that motivates xskill’s use of trajectory-derived UX signals rather than outcome-only rewards. The prompt optimization literature—including DSPy [19] and OPRO [20]—has established that LLM-generated textual artifacts (prompts, demonstrations) can be systematically optimized; xskill extends this paradigm from prompt tuning to full skill document evolution in multi-agent team settings.

2.3 The Missing Piece: No System Does A/B Evaluation

Across all 10 surveyed systems, *none* implements canary or A/B evaluation. Most evaluate skills via offline benchmarks (EvoSkill, GEPA, MemSkill) or LLM-as-judge on the skill text itself (OpenSpace). Only AutoSkill tracks per-turn `relevant/used` judgments, but this measures skill selection quality, not downstream user experience. The absence of controlled comparison testing means that skill evolution in all existing systems is driven by proxy metrics rather than direct evidence of user benefit—a gap xskill explicitly addresses.

This gap is surprising given the mature literature on online controlled experiments [15, 16], where A/B testing has been standard practice in web systems for over a decade. The canary evaluation protocol (§3.3) adapts these principles to the skill evaluation setting, where sample sizes are smaller (team-level rather than web-scale) but the evaluation target (a versioned SKILL.md document) is well-suited to branch-based experimental design.

3 System Architecture

xskill consists of two components: a **client daemon** running on each developer’s machine, and a **central server** that hosts the three-agent pipeline and the skill repository.

3.1 Cross-Agent Trajectory Collection

A key design decision is that xskill does *not* require agents to push trajectories; instead, the client daemon *pulls* from known filesystem locations (Figure 2). This zero-configuration approach works because all five supported agents write trajectories to predictable, stable paths (detailed in Appendix E).

Each trajectory event is normalized into a `TrajectoryEvent` schema with fields: `source` (agent identifier), `session_id`, `ts` (ISO 8601), `kind` \in {`user_msg`, `assistant_msg`, `tool_call`, `tool_result`, `session_start`, `session_end`, `compact`}, and `data` (native payload preserved for replay). The watcher scans registered directories every 30 seconds; new or appended content triggers the three-agent pipeline.

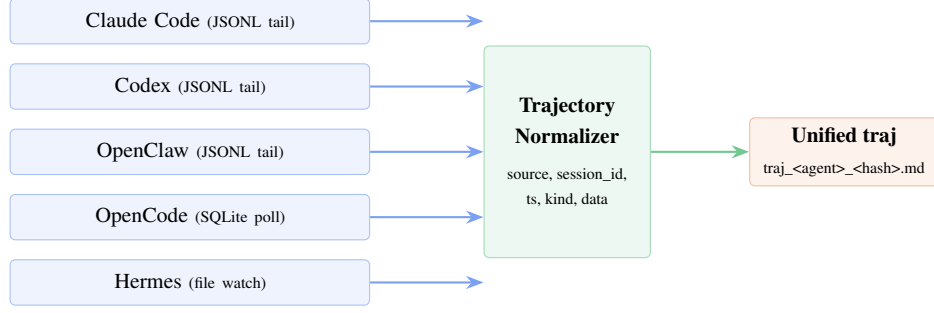


Figure 2: **Cross-agent trajectory collection.** Each coding agent writes trajectories in its native format. The xskill client runs per-agent adapters that normalize events into a unified schema. Four of five agents use appendable JSONL, enabling efficient `inotify/tail` collection; OpenCode’s SQLite database requires polling.

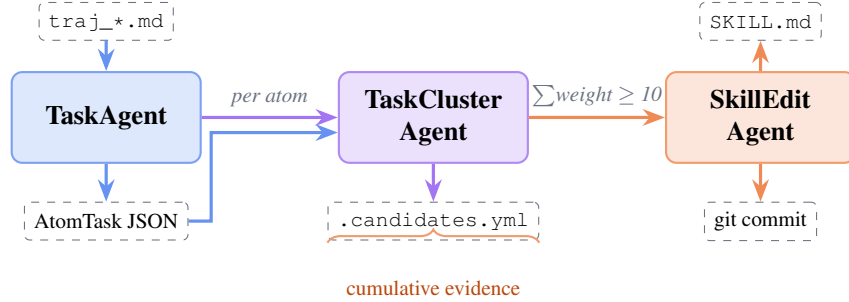


Figure 3: **Three-agent pipeline.** *TaskAgent* decomposes trajectory deltas into AtomTask segments. *TaskClusterAgent* routes each atom to an existing or new skill, appending weighted evidence. *SkillEditAgent* fires when cumulative evidence exceeds a threshold (default: $\text{weight} \geq 10$), producing or updating SKILL.md via git commit.

Sanitization. Before upload, the client applies secret-pattern redaction (API keys, tokens, credentials) using a configurable regex set. This is critical for team-shared trajectories: our ecosystem survey found that only 3 of 10 systems (Hermes, AutoSkill, OpenSpace) implement any form of secret redaction, and none address the subtler question of implicit intellectual property in trajectories.

3.2 Three-Agent Pipeline

The server processes normalized trajectories through three specialized agents (Figure 3). Algorithm 1 formalizes the complete flow.

3.2.1 TaskAgent: Trajectory Decomposition

Raw trajectories are multi-topic conversations that may span hours. The TaskAgent segments each trajectory into **AtomTasks**—coherent units representing a single user intent (e.g., “deploy FastAPI to port 1717” or “fix the SQLite migration”). Each AtomTask records: a unique `atom_id`, semantic metadata (`intent`, `summary`, `tags`), provenance (`offset_start`, `offset_end`), a `ux_score` used in canary evaluation, and linked-list pointers to neighboring atoms for context recovery. The full AtomTask schema is given in Appendix B.

UX Scoring Function. The UX score is the central evaluation signal in xskill’s canary protocol. Unlike LLM-as-judge approaches used by OpenSpace and AutoSkill, the UX score is derived from *observable behavioral signals*. Formally, given an AtomTask a , the score is:

$$s(a) = w_1 \cdot \text{completion}(a) + w_2 \cdot \left(1 - \min\left(\frac{\text{corrections}(a)}{3}, 1\right)\right) + w_3 \cdot \text{attribution}(a) \quad (1)$$

where:

- $\text{completion}(a) \in \{0, 1\}$: whether the atom’s intent was fulfilled, detected by the presence of a natural topic transition vs. session abandonment or explicit user cancellation.
- $\text{corrections}(a) \in \mathbb{Z}_{\geq 0}$: the number of user corrections following the agent’s output, detected by user messages containing negation, redirection, or explicit “no/wrong/redo” patterns within the atom boundary. The score saturates at 3 corrections (any further corrections yield the same minimum contribution).
- $\text{attribution}(a) \in \{0, 1\}$: whether a skill was self-reported as used during this atom (from the `used_skills` field). This signal enables per-skill UX attribution.
- Default weights: $w_1 = 5, w_2 = 3, w_3 = 2$, yielding $s \in [0, 10]$.

This formulation avoids the well-documented biases of LLM-as-judge evaluation [14]—in particular, the tendency of LLMs to overrate their own outputs—at the cost of noisier but less systematically biased signals. The TaskAgent assigns UX scores by analyzing the trajectory text for these behavioral indicators, not by asking an LLM to rate its own performance.

3.2.2 TaskClusterAgent: Skill Routing

Each AtomTask triggers a TaskClusterAgent (TCA) invocation. The TCA receives the atom plus a catalog of existing skills (name + truncated description) and makes one of three decisions: (1) **Route to existing skill**: append the atom to the skill’s `.candidates.yml` with a `weightscore` (0–10) reflecting the atom’s relevance; (2) **Create new skill**: initialize a new skill folder in “baby” state with the atom as first evidence; (3) **Reclassify**: move an atom from one skill to another if the TCA determines a better fit.

The TCA’s routing is LLM-based but constrained by the existing catalog structure, preventing unbounded skill proliferation while allowing organic growth. This positions xskill’s deduplication at a moderate complexity level: more sophisticated than systems with zero deduplication (Hermes, SE-Agent, EvoAgentX) or simple string matching (SkillRL), but deliberately less complex than AutoSkill’s six-layer sandwich. The rationale is that catalog-aware routing combined with cumulative evidence provides sufficient quality control without the engineering complexity of multi-stage dedup pipelines.

3.2.3 SkillEditAgent: Skill Production

When a skill’s `.candidates.yml` accumulates a cumulative `weightscore` $\geq \theta$ (default $\theta = 10$), the SkillEditAgent (SEA) fires. The threshold value was chosen empirically: $\theta = 10$ requires contributions from at least 2–3 distinct trajectories at moderate relevance scores, ensuring that no skill is produced from a single interaction. The SEA reads all candidate atoms and produces or updates the skill’s `SKILL.md`.

The SEA’s commit behavior depends on branch state: (1) **Baby branch** \rightarrow **main**: first public version of a new skill; (2) **Main branch** \rightarrow **staging**: creates a canary candidate for A/B comparison. This branching model ensures that skill updates are never applied directly to main without canary evaluation.

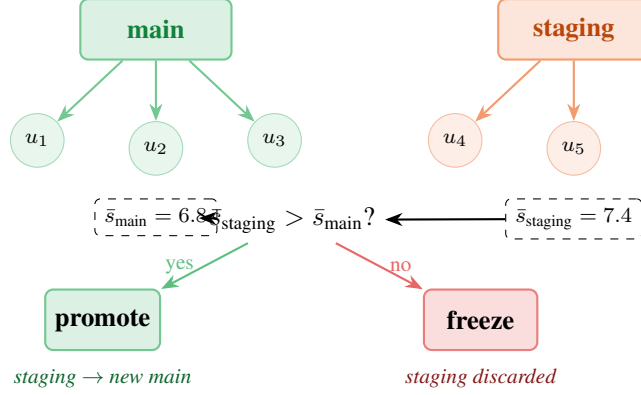


Figure 4: **Canary evaluation protocol.** When the SkillEditAgent produces an updated skill, it commits to a staging branch. A subset of users receives the staging version while the majority continues on main. UX scores from AtomTasks using each branch are aggregated; the staging branch is promoted only if statistically superior.

3.3 Canary Evaluation Protocol

The canary protocol (Figure 4) is xskill’s primary quality gate, drawing on the methodology of online controlled experiments [15, 16] adapted for the team-level skill evaluation setting. Algorithm 2 formalizes the procedure.

Key design decisions in the canary protocol:

Single active canary per skill. At most one staging branch exists for any given skill at any time. If the SkillEditAgent produces a new candidate while a canary is active, the candidate is queued rather than overwriting the current experiment. This prevents evaluation contamination from concurrent canaries—a principle directly borrowed from controlled experiment methodology [16].

Behavioral UX scores, not LLM self-evaluation. The UX score (Eq. 1) is derived from observable signals in the trajectory, not from asking an LLM to rate its own output. This distinguishes xskill from the LLM-as-judge approaches used by OpenSpace (success/failure counts via LLM) and AutoSkill (LLM-labeled relevant/used). While behavioral signals are noisier than LLM judgments, they avoid the systematic upward bias documented in MT-Bench evaluations [14].

Statistical protocol. The minimum sample size ($n_{\min} = 20$ per branch) and significance level ($\alpha = 0.05$, one-sided Welch’s t -test) reflect the team-level deployment context where sample sizes are inherently smaller than web-scale A/B tests. For teams with fewer than 10 active developers, the timeout mechanism (T_{\max} , default 30 days) ensures that experiments do not run indefinitely. When traffic is insufficient for statistical power, xskill falls back to a simple mean comparison with a required minimum effect size ($\Delta_{\min} = 0.5$ points on the 0–10 scale), trading statistical rigor for practical convergence.

3.4 Cross-Agent Skill Distribution

A critical finding from our ecosystem survey is that **SKILL.md with YAML frontmatter is the de facto cross-agent standard**. Five of the ten surveyed systems produce this format,

Table 2: **Trigger mechanisms across 10 systems + xskill.** Eight of ten systems use offline batch or counter-based triggers. Only OpenSpace and AutoSkill achieve user-transparent, per-turn triggering.

System	Async	Per-turn	Transparent	Mechanism
Hermes	–	–	–	Offline CLI
OpenSpace	✓	✓	✓	Hook + counter + event
EvoSkill	–	–	–	Offline batch
AutoSkill	✓	✓	✓	Per-turn + background thread
AgentEvolver	–	–	–	Counter in RL loop
MemSkill	–	–	–	Counter in RL loop
EvoAgentX	–	–	–	Offline batch loop
SE-Agent	–	–	–	Offline iteration
SkillRL	–	–	–	Counter + success gate
GEPA	–	–	–	Offline batch
xskill	✓	✓	✓	30s watcher + per-atom async

and all five supported agent runtimes parse it. xskill exploits this convergence: two filesystem paths—`~/.claude/skills/` and `~/.agents/skills/`—cover 4 of 5 ecosystems; only Hermes requires a dedicated `~/.hermes/skills/` path.

The frontmatter uses a compatible superset where private fields (e.g., `metadata.hermes.tags`, `metadata.openclaw.emoji`) are silently ignored by non-target runtimes—all five do YAML-tolerant parsing—so a single artifact serves all ecosystems. Full distribution path details are in Appendix F.

4 Design Dimensions: A Comparative Analysis

Our survey organizes the trajectory-to-skill landscape into five dimensions. We present key findings per dimension, drawing implications for system design and positioning xskill’s choices.

4.1 Trigger Mechanism

Eight of ten systems use offline batch or counter-based triggers (Table 2). Only OpenSpace and AutoSkill achieve user-transparent, per-turn triggering. This indicates that the research community still largely treats trajectory-to-skill as an offline training concern; the engineering costs of async/concurrent/performance-aware real-time distillation have not been widely addressed. xskill adopts a 30-second polling watcher with per-atom asynchronous processing, achieving transparency without requiring hook registration in the host agent.

4.2 Skill Production: LLM-as-Author is Universal

All 10 systems use LLM-as-author for skill text generation—no system attempts rule-based, template, or AST-based production. This universality has a critical implication: **skill quality is ceiling-bounded by the authoring LLM’s capability**. A skill can only be as good as the model that writes it, regardless of how many trajectories inform the generation.

xskill partially mitigates this ceiling through cumulative evidence: rather than generating a skill from a single trajectory (as OpenSpace, AutoSkill, and SE-Agent can), the Skilled-itAgent waits for multiple weighted atoms to accumulate, providing richer context to the authoring LLM. This does not raise the ceiling, but it helps the LLM approach it more reliably.

Table 3: **Deduplication maturity spectrum.** Systems range from zero deduplication to AutoSkill’s six-layer sandwich. xskill occupies a moderate position with catalog-aware LLM routing.

Level	Systems and Mechanisms
None	Hermes, SE-Agent (overwrite), EvoAgentX (exact string match), SkillRL (set dedup)
LLM self-decision	EvoSkill (prompt constraint), OpenSpace (analysis LLM selects target skills)
Catalog-aware routing	xskill (TCA routes atoms against existing skill catalog via LLM judgment)
Multi-layer sandwich	AutoSkill (identity hash \rightarrow previous-skill hint \rightarrow semantic+BM25 top-5 \rightarrow LLM judge \rightarrow weighted score \rightarrow capability hard gate)
Ancestral merge	GEPA (three-way merge by common ancestor, git-style)

4.3 Deduplication and Merging

Deduplication maturity varies enormously across the landscape (Table 3). xskill’s TCA performs catalog-aware routing—an LLM judgment over existing skill names and descriptions—which provides moderate deduplication without the complexity of AutoSkill’s multi-stage pipeline. The rationale is that xskill’s cumulative evidence threshold already prevents skill proliferation: even if the TCA occasionally creates a near-duplicate skill, it will fail to accumulate sufficient evidence ($\theta = 10$) unless the duplicate genuinely captures a distinct pattern.

4.4 Evaluation and Retirement

Most systems evaluate skills via offline benchmarks or LLM self-judgment. The “only increase, never delete” pattern is dominant: 7 of 10 systems never retire skills. Only EvoSkill (`git branch -D` when the frontier is full), AutoSkill (`retrieved ≥ 40 and used = 0` triggers deletion), and MemSkill (bank-full lowest-reward replacement) implement genuine retirement. This is problematic at scale: without retirement, catalog pollution degrades retrieval quality and inflates prompt context.

xskill addresses this through canary evaluation (§3.3) and skill freezing. A skill that consistently fails canary evaluations can be frozen—retained for audit but removed from active distribution. This represents a middle ground between the “never delete” majority and the aggressive pruning of EvoSkill.

4.5 Skill Injection

xskill deliberately avoids injecting skills into agent prompts (Table 4). Instead, it publishes SKILL.md files to the directories each runtime already scans. This preserves native behaviors: Claude Code’s 1% listing budget and compaction, Codex’s progressive disclosure, OpenCode’s full-dump strategy. The management layer shapes *what* is available; the runtime decides *how* to present it.

4.6 Version Control

Version control maturity is a significant differentiator. EvoSkill uses git branches as program versions—the cleanest version-control story, where `evoskill diff 3 7` maps directly

Table 4: **Skill injection strategies.** xskill deliberately avoids injection, publishing to disk and letting each runtime handle loading natively.

Strategy	Systems
Full body in system prompt	OpenSpace, AutoSkill, MemSkill, SE-Agent, SkillRL
Directory listing + Read tool	Codex, OpenClaw
CLI autodiscovery (no injection)	EvoSkill
User message injection	AgentEvolver
Publish to disk (runtime-agnostic)	xskill , GEPA-gskill

to `git diff` between branches. OpenSpace implements a custom SQLite DAG with `lineage_content_diff` (unified-diff strings) and multi-parent `skill_lineage_parents` tables. AutoSkill uses custom semver patches with embedded metadata snapshots (up to 30 historical entries). Five systems (EvoAgentX, SE-Agent, SkillRL, AgentEvolver, Hermes) have no version control whatsoever—only timestamp directories or step-numbered filenames.

xskill adopts git-based versioning following EvoSkill’s lead: each skill version is a commit, staging branches provide canary isolation, and promotion is a git merge. This inherits git’s native support for diff, blame, and rollback—capabilities that would require significant engineering to replicate in a custom system. The key addition over EvoSkill is that xskill’s branches serve a *functional* purpose (canary evaluation), not just a *archival* one.

4.7 Cold Start

A new xskill deployment begins with an empty skill repository. Cold-start strategies vary across the landscape: AgentEvolver initializes by running one full validation pass; AutoSkill supports offline batch scanning of historical trajectories; SkillRL uses a one-time LLM distillation of classified memories. OpenSpace downloads community top- N skills from a cloud pool. Most other systems (Hermes, EvoSkill, MemSkill, EvoAgentX, SE-Agent, GEPA) start from an empty or minimal seed set.

xskill supports batch trajectory import for cold start: historical trajectories from any of the five supported ecosystems can be fed through the three-agent pipeline in bulk. The canary mechanism, however, requires ongoing user traffic to function, meaning that cold-start skills bypass canary evaluation (committed directly to main) until sufficient usage data accumulates to establish baselines.

5 Deployment Scenario Analysis

As a systems paper describing an engineering artifact, we cannot report controlled benchmark results in the style of SWE-bench [22]. Instead, we present a deployment scenario analysis that demonstrates xskill’s mechanisms through real data and a worked example, and discuss throughput characteristics.

5.1 The Scale Problem: Real Skill Inventory Data

To motivate the need for skill lifecycle management, we report statistics from a real Claude Code installation with extensive skill usage. The developer’s `~/ .claude/skills/` directory contains:

Table 5 reveals a $13\times$ budget overflow: the skill listing requires 26,448 characters but Claude Code’s 1% context budget allows approximately 2,000 characters. Under this overflow, Claude Code drops descriptions for least-invoked skills first, meaning that many

Table 5: **Real skill inventory statistics** from a Claude Code power user. The listing budget overflow demonstrates that skill management is not a hypothetical problem.

Metric	Value
Total SKILL.md files loaded	139
Personal skills	87
Plugin skills (dev-workflows, superpowers)	52
Listing total (sum of descriptions)	26,448 chars
Body total (if all expanded)	699,291 chars ($\approx 175K$ tokens)
Claude Code listing budget (1% of 200K context)	$\approx 2,000$ chars
Budget overflow	13\times over budget
Skills with <code>disable-model-invocation</code>	19

skills—potentially including the most valuable recently-distilled ones—become invisible to the agent. This is precisely the scenario xskill addresses: by actively evaluating skill quality through canary testing and freezing underperforming skills, xskill reduces the number of active skills competing for the limited listing budget, ensuring that high-value skills remain visible.

5.2 Worked Example: FastAPI Deployment Skill Lifecycle

We walk through a complete skill lifecycle to illustrate the three-agent pipeline and canary protocol in action.

Step 1: Trajectory capture. Developer A uses Claude Code to deploy a FastAPI application. The session involves cloning a repository, reading the README, configuring the port, installing dependencies, running the server, and debugging a database migration. The xskill client daemon detects the new trajectory content at `~/ .claude/projects/-home-admin-xquiz/3832444b.jsonl` and uploads the sanitized delta to the server.

Step 2: TaskAgent decomposition. The TaskAgent segments the trajectory into three AtomTasks:

- `atom_0001`: “clone and configure FastAPI app” (intent: setup, tags: [deploy, fastapi], `ux_score`: 8.0)
- `atom_0002`: “debug SQLite migration failure” (intent: debug, tags: [sqlite, migration], `ux_score`: 5.5)
- `atom_0003`: “configure reverse proxy and start server” (intent: deploy, tags: [nginx, deploy], `ux_score`: 7.0)

Step 3: TaskClusterAgent routing. The TCA routes `atom_0001` and `atom_0003` to an existing “python-deploy” skill (weightscore 4.0 and 3.5 respectively), and creates a new “sqlite-migration-debug” baby skill for `atom_0002` (weightscore 5.5). The python-deploy skill now has accumulated evidence from this trajectory plus two prior trajectories, reaching a total weightscore of 14.5 ($> \theta = 10$).

Step 4: SkillEditAgent production. The SEA fires for the python-deploy skill. It reads 6 candidate atoms spanning 3 trajectories, synthesizes them with the existing SKILL.md body, and produces an updated version. Since a main version already exists, the commit goes to a staging branch.

Step 5: Canary evaluation. The server routes 2 of 8 team members to the staging version. Over two weeks, 24 AtomTasks use the python-deploy skill: 15 on main (mean UX score $\bar{s}_{\text{main}} = 6.8$, $\sigma = 1.2$) and 9 on staging. The staging sample has not yet reached $n_{\text{min}} = 20$, so the canary continues collecting. After 3 weeks, staging accumulates 22 samples with $\bar{s}_{\text{staging}} = 7.4$ ($\sigma = 1.1$). A one-sided Welch’s t -test yields $p = 0.038 < 0.05$; the staging version is promoted to main.

5.3 Throughput Estimation

The xskill server’s three-agent pipeline is bottlenecked by LLM inference. With Qwen3.6-27B on a single GPU ($\geq 40\text{GB}$ VRAM) at approximately 30 tokens/second:

- **TaskAgent:** A typical trajectory (5,000–20,000 tokens) requires 1–3 minutes for decomposition into 3–8 AtomTasks, yielding approximately 60–120 atoms/hour.
- **TaskClusterAgent:** Each atom routing takes 10–30 seconds of inference, yielding 120–360 routing decisions/hour.
- **SkillEditAgent:** Skill production or update requires 1–5 minutes per invocation, but fires only when the evidence threshold is met (~ 2 –5 times per day for a 5-person team).

For a team of 5–10 developers each producing 2–5 meaningful trajectories per day, the pipeline processes the daily trajectory load comfortably within a few hours, well within the 30-second polling interval for near-real-time operation.

5.4 Canary Convergence and Low-Traffic Scenarios

A key practical concern is canary convergence time. With a routing fraction $\rho = 0.25$ (1 of 4 users on staging) and an average of 3 skill-attributed AtomTasks per user per day, reaching $n_{\text{min}} = 20$ for staging requires approximately $20 / (1 \times 3) \approx 7$ days. For main with 3 users, n_{min} is reached in ≈ 3 days. Thus, a typical canary evaluation completes in 1–2 weeks for a 4-person team.

For teams with fewer than 4 active developers, the timeout mechanism ($T_{\text{max}} = 30$ days) provides a safety net: if insufficient data accumulates, the canary falls back to mean comparison with a minimum effect size threshold ($\Delta_{\text{min}} = 0.5$), accepting lower statistical power to avoid indefinite experiments.

6 Discussion

6.1 Design Trade-offs

Pull vs. Push trajectory collection. xskill pulls from known paths rather than requiring agents to push events. This trades real-time latency (30s polling vs. sub-second hooks) for zero-configuration deployment. For Claude Code and Codex, hooks can be registered to reduce latency; the pull mechanism serves as a universal fallback.

Cumulative evidence vs. single-trajectory skill creation. Most systems (OpenSpace, AutoSkill, SE-Agent) can create a skill from a single trajectory. xskill requires cumulative weighted evidence ($\geq \theta$), which delays skill creation but reduces noise. This is especially important in team settings where trajectory quality varies across developers. The trade-off is explicit: OpenSpace and AutoSkill’s approaches are better for rapid prototyping; xskill’s approach is better for stable production deployment.

UX scoring vs. LLM judging. xskill’s behavioral UX scores avoid the systematic biases of LLM-as-judge while accepting higher noise. The correction-count signal ($\text{corrections}(a)$ in Eq. 1) is particularly robust: it requires no LLM inference and directly reflects user dissatisfaction. The attribution signal is weaker, as it depends on agents self-reporting skill usage—a convention not all runtimes follow consistently.

6.2 Limitations

1. **No controlled benchmark evaluation:** xskill targets real-world skill evolution rather than isolated task completion. Designing a benchmark for team-level skill lifecycle management (distillation + sharing + evolution + retirement) remains an open problem. The evaluation in §5 uses real data but is illustrative, not statistically rigorous.
2. **Model dependency:** the server pipeline requires Qwen3.6-27B for generation and Qwen3-0.6B-Embed for embeddings, with minimum 40GB VRAM. Smaller models could be substituted at reduced quality; the pipeline is model-agnostic in principle.
3. **Cold start:** the canary mechanism requires ongoing user traffic. New deployments must bootstrap through batch import and direct-to-main commits, which lack quality gating.
4. **UX score noise:** behavioral signals are inherently noisier than LLM judgments. The correction-count heuristic may misclassify follow-up questions as corrections. Calibration across different user interaction styles remains unaddressed.
5. **OpenCode adapter cost:** OpenCode’s SQLite-only trajectory storage makes it the most expensive ecosystem to integrate, requiring database polling or plugin development.

6.3 Broader Impact

xskill’s team-level skill sharing raises questions about intellectual property and knowledge attribution. The sanitization pipeline strips secrets but does not address the subtler issue of whether automated skill distillation from one developer’s work creates implicit IP obligations. Organizations deploying xskill should establish clear policies about skill ownership and attribution. The `used_skills` provenance field in each AtomTask provides an audit trail, but attribution at the skill level (which developer’s trajectory contributed most to a given skill) is not surfaced to end users.

7 Conclusion

We presented xskill, a system for automatic skill distillation, team-level sharing, and data-driven evolution for coding agents. Through a comprehensive source-code-level survey of 10 trajectory-to-skill systems, we identified that no existing system combines asynchronous cross-agent trajectory collection, cumulative-evidence skill production, canary-based A/B

evaluation, and multi-ecosystem distribution. xskill fills this gap by operating as a management layer above existing coding-agent runtimes, shaping the skill supply without interfering with each runtime’s native loading and injection mechanisms.

The canary evaluation protocol—grounded in online controlled experiment methodology and formalized with behavioral UX scoring and statistical testing—represents, to our knowledge, the first application of A/B testing to skill evolution in coding agents. The deployment scenario analysis demonstrates that the scale problem (139 skills, 13× budget overflow) is already real for power users, and that xskill’s mechanisms—cumulative evidence, canary testing, skill freezing—address it through data-driven lifecycle management rather than ad hoc curation.

As coding agents become standard development tools, the ability to systematically capture, evaluate, and distribute procedural knowledge across teams will be a critical differentiator. xskill is open-source at <https://github.com/SkillNerds/xskill>.

References

- [1] Anthropic. Claude Code: An agentic coding tool. <https://claude.com/claude-code>, 2025.
- [2] OpenAI. Codex CLI: Lightweight coding agent. <https://github.com/openai/codex>, 2025.
- [3] SST. OpenCode: Open-source coding agent. <https://github.com/sst/opencode>, 2025.
- [4] Chen, Y., Li, J., Wu, Z., and Xia, L. OpenSpace: Automated skill discovery and integration for open-ended coding agents. *arXiv preprint arXiv:2507.04389*, 2025.
- [5] Wang, J., Chen, L., and Zhang, S. EvoSkill: Evolutionary program synthesis for agent skills. <https://github.com/sentient-agi/EvoSkill>, 2025.
- [6] Li, X., Huang, Y., and Wang, Z. AutoSkill: Automated skill management for coding agents. <https://github.com/ECNU-ICALK/AutoSkill>, 2025.
- [7] Qwen Applications. Trace2Skill: From agent traces to reusable skills. <https://github.com/Qwen-Applications/Trace2Skill>, 2025.
- [8] ModelScope. AgentEvolver: Online reinforcement learning with adaptive experience management. <https://github.com/modelscope/AgentEvolver>, 2025.
- [9] Axelsen, V. MemSkill: Memory-augmented skill learning for language agents. <https://github.com/ViktorAxelsen/MemSkill>, 2025.
- [10] EvoAgentX Team. EvoAgentX: Evolutionary agent workflow optimization. <https://github.com/EvoAgentX/EvoAgentX>, 2025.
- [11] Xu, S., Zhang, Y., and Xie, T. SE-Agent: Self-evolving agents for software engineering. <https://github.com/JARVIS-Xs/SE-Agent>, 2025.
- [12] Aiming Lab. SkillRL: Skill-augmented reinforcement learning for language agents. <https://github.com/aiming-lab/SkillRL>, 2025.

- [13] Liu, Z., Yin, D., and Yang, J. GEPA: Generalized evolutionary prompt optimization with system-aware merge. *arXiv preprint arXiv:2507.19457*, 2025.
- [14] Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., et al. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *NeurIPS*, 2024.
- [15] Kohavi, R., Deng, A., Frasca, B., Walker, T., Xu, Y., and Pohlmann, N. Online controlled experiments at large scale. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1168–1176, 2013.
- [16] Kohavi, R., Tang, D., and Xu, Y. *Trustworthy Online Controlled Experiments: A Practical Guide to A/B Testing*. Cambridge University Press, 2020.
- [17] Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.
- [18] Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*, 2023.
- [19] Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Vardhamanan, S., Haq, S., Sharma, A., Joshi, T. T., Mober, H., et al. DSPy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.
- [20] Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q. V., Zhou, D., and Chen, X. Large language models as optimizers. In *ICLR*, 2024.
- [21] Nonaka, I. and Takeuchi, H. *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation*. Oxford University Press, 1995.
- [22] Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. SWE-bench: Can language models resolve real-world GitHub issues? In *ICLR*, 2024.
- [23] Wang, Z., Chen, Y., Li, J., Xia, L., and Huang, C. OpenSpace: An evolving skill library via MCP for code agents. *arXiv preprint arXiv:2507.04389*, 2025.

A Full Comparison Matrix

Table 6: **Cross-system comparison on “skill” definition.** The term “skill” refers to fundamentally different artifacts across systems, complicating direct comparison.

#	System	“Skill” Artifact
1	Hermes	Existing SKILL.md body (mutated in place)
2	OpenSpace	Directory: SKILL.md + scripts/ + references/ + SQLite lineage
3	EvoSkill	.claude/skills/<name>/SKILL.md (YAML frontmatter)
4	AutoSkill	Per-user SKILL.md with semantic semver patches
5	AgentEvolver	Vector-store entry in external ReMe service
6	MemSkill	RL operation (INSERT/UPDATE/DELETE template)
7	EvoAgentX	Workflow code: round_N/{graph.py, prompt.py}
8	SE-Agent	Per-instance system-prompt YAML
9	SkillRL	JSON record: {title, principle, when_to_apply}
10	GEPA	Candidate dict[str,str] (arbitrary optimizable text)
11	xskill	SKILL.md + auxiliary files, git-versioned, canary-evaluated

B AtomTask Schema

```
{
  "atom_id": "atom_traj_cc_dsv4_890da9d9_0001",
  "traj_id": "traj_cc_dsv4_890da9d9",
  "offset_start": 120,
  "offset_end": 2400,
  "intent": "deploy xquiz on port 1717",
  "summary": "agent cloned repo, read README, configured port, started",
  "tags": ["deploy", "fastapi"],
  "used_skills": ["python-deploy"],
  "ux_score": 7,
  "pre_atom_id": null,
  "post_atom_id": "atom_traj_cc_dsv4_890da9d9_0002",
  "context_prefix": "",
  "raw_segment": ""
}
```

C TrajectoryEvent Schema

```
{
  "source": "claudecode|codex|openclaw|hermes|opencode",
  "session_id": "3832444b-2e7c-48be-9277-dcdb8a717678",
}
```



```

"ts": "2026-05-27T14:30:00.000Z",
"kind": "user_msg|assistant_msg|tool_call|tool_result
        |session_start|session_end|compact",
"data": {
    // Native payload preserved verbatim for replay.
    // Structure varies by source agent.
}
}

```

D Canary Guard Conditions

The SkillEditAgent is guarded by three conditions before committing to a staging branch:

1. No staging branch currently exists for this skill (one canary at a time; new candidates queue).
2. Cumulative `weightscore` of candidates $\geq \theta$ (default 10).
3. If on main branch: at least one real `side=main` UX score exists (proves the current main version has been used, establishing a baseline for comparison).

These guards prevent evaluation contamination from concurrent canaries and ensure that staging branches compete against a used—not merely published—main version.

E Trajectory Collection Paths

The xskill client daemon pulls trajectories from the following filesystem locations:

- **Claude Code:** `~/.claude/projects/<proj>/<sid>.jsonl` — JSONL, async batch append. Detected via `inotify/tail`.
- **Codex:** `~/.codex/sessions/YYYY/MM/DD/rollout-*.jsonl` — JSONL, stream-append per item (256-slot mpvc channel). Detected via `inotify/tail`.
- **OpenClaw:** `~/.openclaw/agents/<id>/sessions/*.trajectory.jsonl` — JSONL, per-event sync with schema version field. Detected via `inotify/tail`.
- **OpenCode:** `~/.local/share/opencode/opencode.db` — SQLite with Drizzle ORM (4 tables: `session`, `message`, `part`, `session_message`). Requires polling or plugin integration.
- **Hermes:** `<cwd>/trajectory_samples.jsonl` — JSONL, written once at run end. Requires file-appearance watch rather than tail.

F Skill Distribution Paths

The SKILL.md frontmatter uses a compatible superset:

```

---
name: <slug>
description: <one-line, <=1024 chars>
version: 1.0.0
metadata:

```

Table 7: **Skill output paths for 5 agent ecosystems.** Two filesystem paths cover 4 of 5 ecosystems; only Hermes requires a dedicated path.

Agent Ecosystem	xskill Output Path	Also Scanned By
Claude Code	~/.claude/skills/<name>/SKILL.md	OpenCode
Codex	~/.agents/skills/<name>/SKILL.md	OpenClaw
OpenCode	~/.claude/skills/<name>/SKILL.md	Claude Code
OpenClaw	~/.agents/skills/<name>/SKILL.md	Codex
Hermes	~/.hermes/skills/<cat>/<name>/SKILL.md	(exclusive)

```

hermes: {tags: [...]}
openclaw: {emoji: "...", requires: {...}}
short-description: <Codex alias>

```

Private fields are silently ignored by non-target runtimes (all five perform YAML-tolerant parsing), so a single artifact serves all ecosystems.

Algorithm 1 Three-Agent Skill Distillation Pipeline

Require: Trajectory delta δ , skill catalog \mathcal{C} , threshold $\theta = 10$

Ensure: Updated skill repository

```
1: Stage 1: TaskAgent — Decompose  $\delta$  into AtomTasks
2:  $\mathcal{A} \leftarrow \text{TaskAgent.decompose}(\delta)$  {Segment by semantic boundaries}
3: for each atom  $a \in \mathcal{A}$  do
4:   Assign  $a.\text{ux\_score} \leftarrow f_{\text{UX}}(a)$  {Behavioral UX scoring (Eq. 1)}
5:   Record  $a.\text{intent}, a.\text{tags}, a.\text{used\_skills}$ 
6: end for
7:
8: Stage 2: TaskClusterAgent — Route atoms to skills
9: for each atom  $a \in \mathcal{A}$  do
10:   $(\text{action}, \text{skill}, w) \leftarrow \text{TCA.route}(a, \mathcal{C})$ 
11:  if action = CREATE then
12:    Initialize new skill folder in “baby” state with  $a$  as first evidence
13:     $\mathcal{C} \leftarrow \mathcal{C} \cup \{\text{skill}\}$ 
14:  else if action = ROUTE then
15:    Append  $(a, w)$  to  $\text{skill.candidates.yml}$ 
16:  else if action = RECLASSIFY then
17:    Move  $a$  from current skill to better-fit target
18:  end if
19: end for
20:
21: Stage 3: SkillEditAgent — Produce or update skills
22: for each skill  $s \in \mathcal{C}$  do
23:  if  $\sum_{(a,w) \in s.\text{candidates}} w \geq \theta$  and canary guards pass then
24:     $\text{SKILL.md} \leftarrow \text{SEA.generate}(s.\text{candidates}, s.\text{existing\_body})$ 
25:    if  $s.\text{branch} = \text{baby}$  then
26:      git commit to main {First public version}
27:    else
28:      git commit to staging {Canary candidate}
29:    end if
30:  end if
31: end for
```

Algorithm 2 Canary Evaluation Protocol

Require: Skill s with new staging commit, user set U , routing fraction $\rho \in (0, 0.5]$, minimum samples $n_{\min} = 20$, significance level $\alpha = 0.05$, timeout T_{\max} days

Ensure: Decision $\in \{\text{PROMOTE}, \text{FREEZE}, \text{TIMEOUT}\}$

- 1: **Guard conditions:**
 - 2: Assert no active staging branch for s (one canary at a time; new candidates queue)
 - 3: Assert ≥ 1 real side = main UX score exists (baseline established)
 - 4:
 - 5: **User routing:**
 - 6: Route $\lceil \rho|U| \rceil$ users to staging; remainder to main
 - 7:
 - 8: **Score collection:**
 - 9: **while** $|S_{\text{main}}| < n_{\min}$ **or** $|S_{\text{staging}}| < n_{\min}$ **do**
 - 10: **if** elapsed $> T_{\max}$ **then**
 - 11: **return** TIMEOUT (freeze staging, retain for audit)
 - 12: **end if**
 - 13: Collect UX scores s_i from AtomTasks using skill s
 - 14: Attribute each s_i to S_{main} or S_{staging} based on user's branch
 - 15: **end while**
 - 16:
 - 17: **Statistical comparison:**
 - 18: Compute $\bar{s}_{\text{main}}, \bar{s}_{\text{staging}}, \sigma_{\text{main}}, \sigma_{\text{staging}}$
 - 19: Perform one-sided Welch's t -test: $H_0: \mu_{\text{staging}} \leq \mu_{\text{main}}$
 - 20: **if** $p < \alpha$ **and** $\bar{s}_{\text{staging}} > \bar{s}_{\text{main}}$ **then**
 - 21: Merge staging \rightarrow main;
 - 22: **return** PROMOTE
 - 23: **else**
 - 24: Freeze staging (retain for audit, hide from distribution);
 - 25: **return** FREEZE
 - 26: **end if**
-