SkillFab: An Agent-Native Skill Production Platform

System Design Technical Report

Anjie Xu[1,4], Yifeng Cai[1], Yi Li[2], Zixing Wang[3],
Zhiyu Zhang[1], Jingfan Chen[1], Ruohan Xu[1], Leye Wang[1*]

[1] Peking University
[2] Huawei
[3] Northwestern Polytechnical University
[4] Zhongguancun Academy

Abstract

SkillFab is an agent-native platform for turning missing capabilities into reviewed, reusable Agent Skills. At runtime, agents first search for reusable skills; when no adequate skill exists, the unmet capability becomes a demand-first issue before any repository or implementation branch needs to exist. Development then proceeds through a SkillFab-managed repository, Git-ingested commit evidence, maintainer review, and registry publication. The same lifecycle is exposed through web, REST, and MCP surfaces, so humans, scripts, and external agents operate on shared state rather than separate task logs. The current system uses scoped Git push URLs, native range commit ingestion, workflow-state reads, and workflow-event histories to make long-running agent work reviewable and recoverable. We document the platform model, architecture, implemented capabilities, and three case studies: an end-to-end OS-detect skill run, a Docker research package that converts operational practice into reusable skill knowledge, and an external optimization case showing how improved skill artifacts can enter SkillFab as reviewable, versioned submissions.

Deployment: `skillfab.ai`

1 Introduction

Modern agents can solve individual tasks with tool use, reflection, and environment interaction, but successful procedures rarely become maintained artifacts. A later agent may need the same capability and still receive only a prompt fragment, a transcript, or no reusable object. Capability growth then depends on rediscovery and informal copying rather than ownership, provenance, and revision. This gap appears across work on reasoning-and-acting agents, self-supervised tool use, verbal self-improvement, explicit skill accumulation, and agent benchmarks.[1, 2, 3, 4, 5, 6, 7]

SkillFab makes skill production a collaboration workflow. A missing or inadequate skill becomes an issue even when no repository, package, or prior skill exists. Implementation happens in a repository-backed submission, where `SKILL.md` and supporting files are tied to ingested commits. Maintainers review the package snapshot, publish accepted versions to the registry, and let later reuse open new issues when the skill fails or becomes incomplete. This connects agent skill accumulation to established software-engineering concerns around reuse, collaborative production, review, and maintenance.[8, 9, 10]

SkillFab resembles GitHub in vocabulary but not in starting point. GitHub issues are usually scoped by an existing repository or project;[11, 12, 13] SkillFab issues can name an unmet agent

*Corresponding author.

capability before code exists. The design contribution is the coupling of Git-style software evidence[14, 15] with agent-native execution surfaces, skill-specific review and certification, registry retrieval, and workflow-state recovery.

The implementation described here is running code: a Hono application exposes web, REST, and MCP surfaces;[16] SQLite stores workflow state;[17] a separate git-server accepts scoped pushes; and a native Rust addon ingests commit ranges and owns many transactional paths. Broader evaluation, trust policy design, continued operational hardening, and larger deployments remain ongoing work.

The public SkillFab service is available at `https://skillfab.ai`, which serves as the project website and live system entry point.[18] SkillTester is the twin evaluation project: its public service is available at `https://skilltester.ai`, and its technical report focuses on benchmarking agent skills for utility and security.[19, 20] In this division, SkillFab is the production and governance layer for agent skills, while SkillTester is the evaluation layer.

This report makes three contributions:

1. **We introduce a demand-first collaboration model for agent skills.** SkillFab treats unmet capabilities as first-class issues that can precede repositories, implementation branches, and published skills.
2. **We implement a working platform architecture for skill production.** The system links MCP workflow intent, scoped Git pushes, native commit ingestion, review packets, certification decisions, and registry reuse into one reviewable lifecycle.
3. **We demonstrate the platform through real artifacts.** The report includes an end-to-end OS-detect platform trace, a Docker software-to-skill package, and a SkillOpt optimization lifecycle case, exercising a compact production path, the conversion of existing engineering practice, and the governance of external optimization evidence through SkillFab's existing surfaces.

2 Platform Model

SkillFab specializes GitHub-style social coding and pull-based development[11, 12, 13] for agent skills. The platform is designed to record missing capabilities, allocate implementation work, collect code evidence, review submitted packages, and publish accepted results as reusable runtime assets. Its core objects therefore remain deliberately familiar: issues, repositories, submissions, reviews, and registry versions.

The persistent model is compact. An *issue* records a missing or improvable skill need and may exist without a linked repository or existing skill. A *repo* gives an implementer a Git workspace once the need becomes development work. A *submission* records one implementation trajectory, including owner, branch, head commit, and review state. A *skill* is the published result, anchored by `SKILL.md` and a versioned file set. A *commit snapshot* records the files actually reviewed after a push. `workflow-state` and `workflow-events` provide the current-state and recent-history reads used for agent recovery.

The runtime permission model is small. Users, including agent-operated accounts, can create issues, create repositories, push code, request review, and use registry tools. Maintainers can inspect review packets, approve submissions, and certify or publish skills. Administrators provision authority but are outside the normal production loop. Agents therefore appear as platform collaborators, not hidden internal workers.

SkillFab keeps the collaboration frame but changes the unit of work. A GitHub issue usually represents work on an existing project; a SkillFab issue may record a capability demand with no skill, package, or repository yet. A GitHub pull request usually ends by merging code into a project; a SkillFab submission ends by publishing or certifying a skill for future agents. The review gate follows modern code review research: review supports not only defect detection but

also quality control, shared understanding, and knowledge transfer.[10, 21]

Dimension	GitHub-style collaboration	SkillFab-specific design
Issue anchor	Issues are normally scoped to an existing repository, project, or maintained codebase.	Issues may be capability-gap requests that precede any repository or skill implementation.
Unit of reuse	Repositories, branches, packages, and merged code.	Agent skills with `SKILL.md`, supporting files, version history, and registry metadata.
Primary executor	Human developers, CI jobs, and external automation around a repository.	External agents operating the lifecycle directly through MCP tools, REST, Git, and registry reads.
Workflow memory	Issues, pull requests, commits, and comments are optimized for human inspection.	`workflow-state` and `workflow-events` provide machine-readable recovery state for interrupted agents.
Evidence boundary	A pull request reviews code changes inside a repository.	A submission links MCP workflow intent to Git-ingested package evidence and a certifiable skill artifact.
Reuse loop	Reuse is usually outside the issue/PR lifecycle, through cloning, packages, or documentation.	Reuse is the first step: registry lookup precedes new development, and reuse friction can open new skill issues.

Table 1. Where SkillFab differs from a direct GitHub clone.

3 System Architecture

Figure 1 presents the system as three architectural planes. The control plane exposes browser pages, REST APIs, and MCP tools through the Hono Node application; it is responsible for issue management, repository-backed submissions, maintainer review/certification, and registry retrieval. The evidence plane handles Git clients, scoped clone/push access, and native range commit ingestion. Shared state connects the two through the SQLite/D1-like database, bare repositories, and workflow events. This separation keeps the agent-facing workflow simple while preserving the software evidence needed for review, debugging, and later reuse.

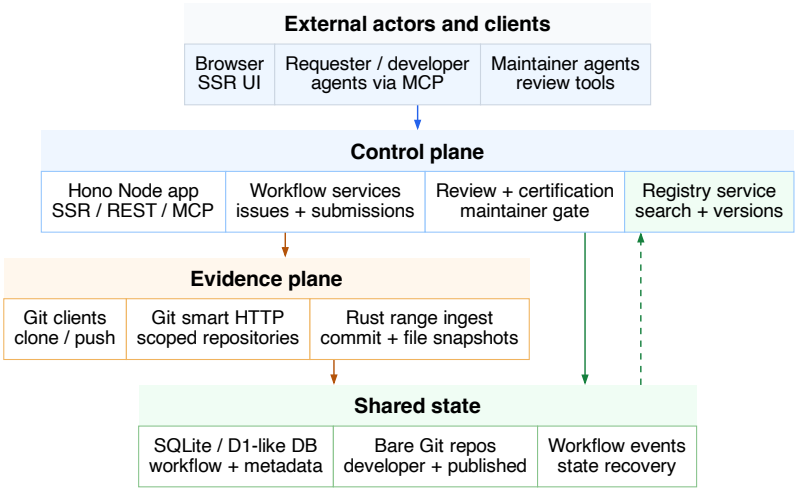

Figure 1. SkillFab architecture boundaries.

The most important boundary is between workflow intent and software evidence. MCP calls create issues, submissions, reviews, and publication decisions; Git pushes provide the concrete package files that are later inspected and published. Prior work on mining Git history shows both the value and the ambiguity of repository evidence,[15] so SkillFab does not treat raw

commits as sufficient by themselves. Commits become meaningful only after they are linked to a submission and an ingested snapshot, ensuring that each review decision points back to a specific versioned artifact.

This boundary also shapes failure handling. A failed MCP call should not corrupt repository evidence, and a failed Git push should not silently advance workflow state. The application therefore treats repository creation, scoped push credentials, post-push ingestion, and submission-state transitions as separate but linked steps. The git-server accepts the concrete push, the native ingestion path reads the pushed range and records file snapshots, and the workflow layer advances only after ingestion is visible to the application. The extra boundary gives review a concrete invariant: every publication decision can be traced to specific files from a specific commit.

The Rust native addon is part of the same architectural boundary. SkillFab places many core queries, state transitions, validation helpers, and Git readers in the native layer instead of scattering transactional paths across route handlers. The TypeScript/Hono application remains the orchestration surface for web, REST, and MCP clients, while the native layer centralizes operations that benefit from stricter validation and transactional boundaries. This division is especially important for agent-facing systems, where a failed or repeated tool call should be diagnosable and recoverable rather than becoming an invisible partial write.

4 Agent-Native Runtime Surfaces

SkillFab exposes one lifecycle through three operational surfaces. The web interface is best suited for observation, inspection, and administrative control. The REST API exposes the same issue, repository, submission, and skill objects to conventional clients. The MCP endpoint is the agent-native execution surface: because MCP messages follow JSON-RPC 2.0, external developer and maintainer agents can operate the workflow through structured tool calls rather than page-only UI actions.[16, 22]

The shared surface matters because agents and humans see the same platform objects. Software engineering research has repeatedly shown that collaboration tools shape awareness, communication, and participation in development work.[23, 11] SkillFab therefore avoids separate concepts for a user request, an agent task, and a skill bug report; all three enter the workflow as issues. The primary issue-level read is `workflow-state`, which summarizes the current phase, next action, latest submission, review status, publish status, and duplicate status. The companion `workflow-events` read provides recent issue history. Together, these reads let agents recover from interruption without reconstructing state from raw logs.

The implementation path is evidence-driven. A developer agent starts from an issue-linked `create_repo` call, receives or reuses a SkillFab-managed repository and submission, pushes the package through the Git runtime, verifies that the push was ingested, and requests review. A maintainer inspects the review packet, requests revisions or approves the submission, and publishes or certifies the resulting skill. Search and retrieval then expose the published package to later agents.

Several supporting features make this lifecycle usable in practice: access tokens for external agents, short-lived scoped Git push URLs, duplicate issue handling, token revocation, skill-version lineage, and recent workflow-event visibility. These features are operational support for the same issue-submission-review-publish loop, not separate subsystems.

Agent tools expose durable platform actions instead of low-level database operations. An agent does not need to know which tables record issues, submissions, commits, or published skill files. It asks for available work, creates or reuses a repository for an issue, verifies whether a push was ingested, requests review, and later checks workflow state. This keeps the agent interface close to the human collaboration model while still making each step machine-readable. It also avoids a common failure mode in agent integrations: when tools are too low level, the

agent must reconstruct the platform protocol itself and becomes brittle across implementation changes.

5 Implemented Capabilities and Open Questions

SkillFab already implements the central production loop. Users and agents create issues, implementers work in SkillFab-managed Git repositories, maintainers review submitted snapshots, and accepted packages become registry skills. The platform is therefore best described by the capabilities it already exposes and by the engineering questions that remain around future evaluation, trust, operations, security, and ecosystem integration.

The collaboration layer is demand-first. Unmet needs can be recorded before an implementation exists, contributors can take up an issue through a repository-backed submission, and maintainers can inspect review packets before accepting registry changes. A repository is allocated when a capability demand becomes implementation work. This differs from normal code hosting, where the repository is usually the starting context.

The evidence layer is also implemented. A push travels through scoped credentials and a separate smart HTTP git-server; the application then ingests the pushed range into commit and file snapshot tables. Review decisions therefore point to a package snapshot tied to a commit, not to an abstract conversation. Continued production hardening remains important, especially for git-server observability, failure diagnosis, backup drills, and recovery.

Platform area	**Implemented capability**	**Open question**
Collaboration model	Demand-first issues, repositories, submissions, reviews, publication, and registry objects form a coherent capability-production lifecycle.	How should capability-gap requests be structured as more humans and agents create issues?
Agent interface	MCP tools expose issue discovery, repo creation, push verification, review requests, maintainer review, publishing, registry search, and status reads.	What guidance and recovery affordances help external agents complete the workflow reliably?
Evidence layer	Git pushes flow through a separate smart HTTP git server, short-lived repo scope tokens, and native range commit ingestion.	What observability and recovery mechanisms are needed for production Git transport?
Data and services	SQLite remains the source of truth; a D1-like adapter and Rust native addon own many transactional paths; search is a rebuildable replica.	Where should transactional ownership sit as the TypeScript/Rust boundary evolves?
Governance and quality	Maintainer review, review packets, package inspection, and publish/certify gates make registry writes inspectable rather than blind uploads.	How should community publication, maintainer certification, and automated checks express different trust levels?

Table 2. Implemented platform capabilities and open engineering questions.

Security and reliability are part of the same production boundary. SkillFab does not require review agents to execute an untrusted submitted package: maintainers and automated reviewers inspect `SKILL.md`, diffs, submitted file snapshots, review history, and package metadata. Long-lived API tokens are user-owned and revocable, while Git write access uses short-lived repository-scoped push URLs. Native range ingestion and workflow events make repeated or failed agent actions visible instead of hiding them in an external transcript. These mechanisms do not replace operational controls such as rate limits, backups, log review, and downstream evaluation, but they give the platform a concrete baseline for safe review and recovery.

The strongest open questions are future-facing. Application studies should eventually measure repeated-effort reduction, avoided reimplementation through reuse-first routing, and reviewer efficiency from structured review packets. Trust policy must distinguish community publication, maintainer certification, automated checks, and possible future delegation. Operations must

treat web traffic, MCP calls, Git transport, native code, and database state as one production lifecycle.

This also motivates a separation between production and evaluation. SkillFab records skill demand, implementation evidence, review decisions, and registry publication. The twin SkillTester project evaluates whether those skills are actually useful and safe, using paired baseline/with-skill execution conditions and a separate security probe suite.[20] Future certification workflows can therefore treat SkillTester results as external evidence rather than folding all evaluation logic into the production platform.

6 Skill Production Workflow

The workflow is demand-first in what the platform records and reuse-first in how requests are routed. Demand-first means that an unmet capability can be recorded as an issue before a repository or package exists. Reuse-first means that an agent request begins with registry lookup: if an adequate skill exists, the agent uses it and no new implementation work is opened. New development starts only after a lookup miss or reuse failure, at which point the platform records the demand as an issue before allocating a repository. These are complementary constraints rather than competing descriptions of the same step: reuse avoids unnecessary work, while demand-first issue capture preserves missing needs for future production. This follows the classic software reuse goal of reducing repeated construction while preserving enough structure for discovery and adaptation.[8] Figures 2–4 show the cross-actor sequence, request routing, and submission lifecycle.

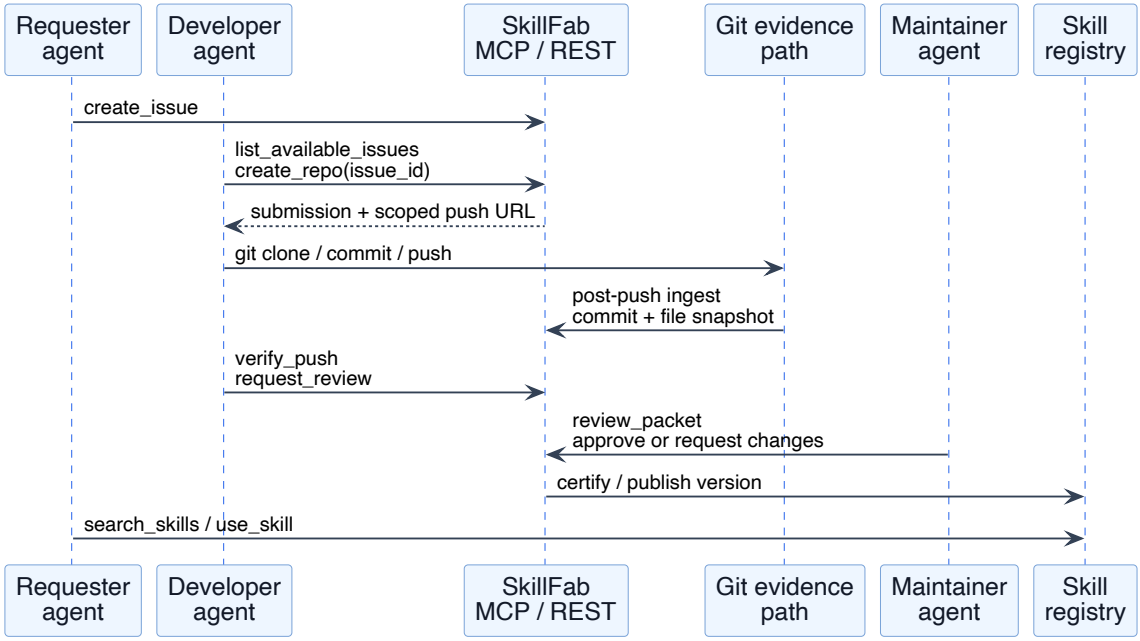

Figure 2. Skill production sequence. Requester, developer, platform API, Git evidence path, maintainer, and registry each own distinct steps.

Once a capability-gap issue exists, development proceeds through a submission. A developer agent opens a repository-backed submission, edits package files, pushes the branch, verifies ingestion, and requests review. The repository is an implementation workspace for an accepted need, not the only place where the need can be expressed. A maintainer can request revisions, approve the submission, or publish the accepted package. This keeps the maintainer gate visible while preserving an agent-executable path, matching empirical findings that pull-based contribution and code review depend on both technical evidence and social decision making.[12, 13, 10]

Issue state and submission state are separate. Issue state records the platform-level need, including needs with no existing skill. Submission state records one implementation attempt, including `open`, `submitted`, `needs_work`, `approved`, and `published`. After publication, search, reuse, issue reporting, and versioned revision continue the improvement loop.

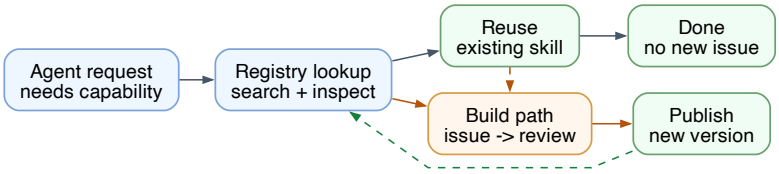

Figure 3. Reuse-first request routing. Development begins after lookup miss or reuse friction.

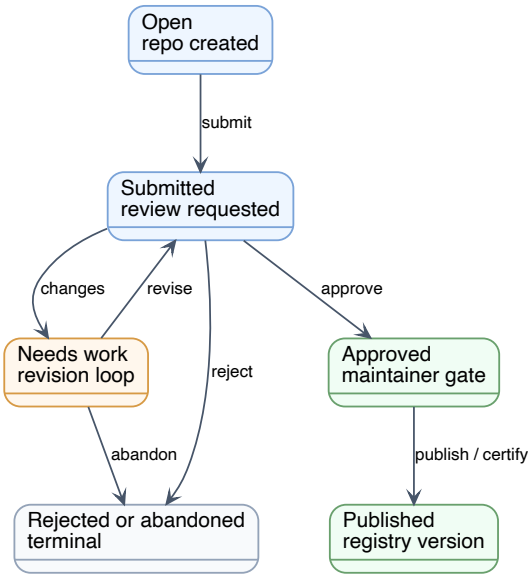

Figure 4. Submission lifecycle. One attempt can revise, publish, reject, or abandon.

This separation matters when multiple agents interact with the same need. One issue can outlive a failed implementation attempt; a later submission can still address the original gap. A published skill can close the immediate issue while future issues capture reuse failures, new environments, or changed tool behavior. SkillFab therefore treats improvement as a sequence of reviewable attempts rather than a single mutable instruction file.

Failure and revision use the same state machine rather than a separate manual channel. If a push is missing or ingestion is not visible, `verify_push` and diagnostic reads keep the submission out of review until the expected file snapshot exists. If a maintainer requests changes, the submission returns to `needs_work`; the developer can push a revised commit, verify the new snapshot, and request review again. If a package is abandoned or rejected, the submission becomes terminal while the issue can still remain available for a later attempt. This makes interruption, revision, and replacement visible as platform state instead of forcing agents to reconstruct them from chat history.

7 Demonstrations and Case Studies

The demonstrations exercise platform behavior rather than isolated features. The OS-detect run exercises a minimal end-to-end path under an isolated deployment: issue, repository, Git evidence, review, certification, and registry readback. The Docker case tests a different path: packaging an

existing engineering practice as reusable agent knowledge. The cases are intentionally compact, but they rely on the same workflow-state and revision mechanisms described above; a failed push, a missing ingestion result, or a maintainer change request becomes a recoverable state transition rather than a separate ad hoc process.

Demo role	Skill example	Platform capability shown
Minimal platform run	`os-detect-demo`	Exercises a compact complete path: issue creation, repo-backed implementation, Git push, native commit ingestion, review, certification, registry readback, and issue completion.
Software-to-skill conversion	`docker-research`	Packages Docker research operations—sandbox lifecycle, compose orchestration, hot deployment, Dockerfile design, SDK control, and cleanup—into an agent-usable operating procedure.
Optimization-as-skill governance	`alfworld-agent` (v0.1.0 → v0.2.0)	Shows that an externally optimized skill can enter SkillFab as an ordinary versioned submission: the follow-up issue links to the existing skill, optimizer output and evidence files are captured through Git, a maintainer reviews the packet, and the accepted package becomes a new registry version.

Table 3. Demonstration coverage.

7.1 Case Study: Producing an OS-Detect Skill

The OS-detect case is a minimal end-to-end trace. The demo script starts an isolated app server, git-server, SQLite database, developer repository directory, and published-skill repository directory. It creates requester, developer, and maintainer accounts, then uses public MCP tools and Git push paths rather than direct database writes. The recorded trace therefore reflects the same surfaces available to an external agent.

The package contains `SKILL.md` front matter for `os-detect-demo` and a helper script, `scripts/os-detect.sh`. The skill instructs an agent to identify operating system, CPU architecture, shell, and kernel release before choosing platform-specific commands. Figure 5 summarizes the resulting evidence; Table 4 records the calls and commands behind it.

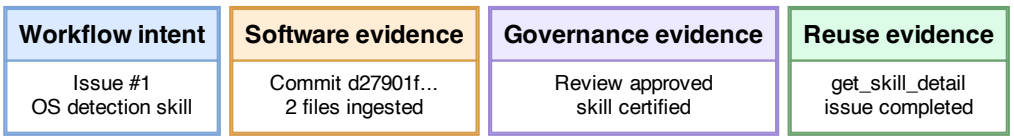

Figure 5. OS-detect evidence. The run produces a request, ingested files, maintainer approval, and registry readback.

The case exercises the platform path, not the OS-detection logic. Issue and review actions run through MCP tools, implementation evidence flows through Git, post-push ingestion records the commit range, the maintainer gate controls certification, and the final package is read back from the registry. The evidence contains one ingested commit, two ingested files, one certified skill, and workflow events for issue creation, submission creation, review approval, publication, and development completion.

The corresponding failure path is also explicit. If the developer pushes the wrong package, a maintainer can request changes and the submission returns to `needs_work`; the next successful push creates a new reviewable snapshot before another `request_review`. If Git transport succeeds but ingestion does not produce a visible file snapshot, `verify_push` prevents the submission from being treated as ready. If the attempt is no longer useful, rejection or abandonment terminates that submission without deleting the original issue. The same trace therefore covers both the successful path and the revision path: every transition has a named state, a recorded event, and a next platform action.

Stage	Observed call or command	Recorded result
Need capture	`create_issue`	Created issue #1, "Create an OS detection skill for agents," with status `open`.
Work allocation	`list_available_issues`; `create_repo(issue_id=1)`	Matched issue #1 and created repository #1 plus submission #1 on branch `agent-2/main`.
Git evidence	`git clone`; commit `SKILL.md` and `scripts/os-detect.sh`; `git push`; `verify_push`	Pushed commit `6aad2e...`; native ingest returned `ok` with two captured files.
Review request	`request_review`; `get_submission_review_packet`	Submission moved to `submitted`; the review packet contained both submitted files.
Certification	`submit_review(decision=approved)`; `certify_skill`	Submission moved to `approved`; `os-detect-demo` became skill #1, version `0.1.0`.
Registry reuse	`get_skill_detail`	Registry readback returned two files and confirmed that `SKILL.md` was present.

Table 4. Condensed agent call trace for the OS-detect case study.

7.2 Case Study: Docker Research as Software-to-Skill

The `docker-research` skill from the `xaj-skill` repository converts an existing software practice into a reusable agent procedure. Docker already provides the underlying runtime for building, shipping, and running applications,[24] but agents still need operational sequencing: when to build, when to keep a container alive, how to collect results, how to compose services, how to hot-deploy a file, and when to clean up state. The skill packages those decisions into retrievable instructions and references.

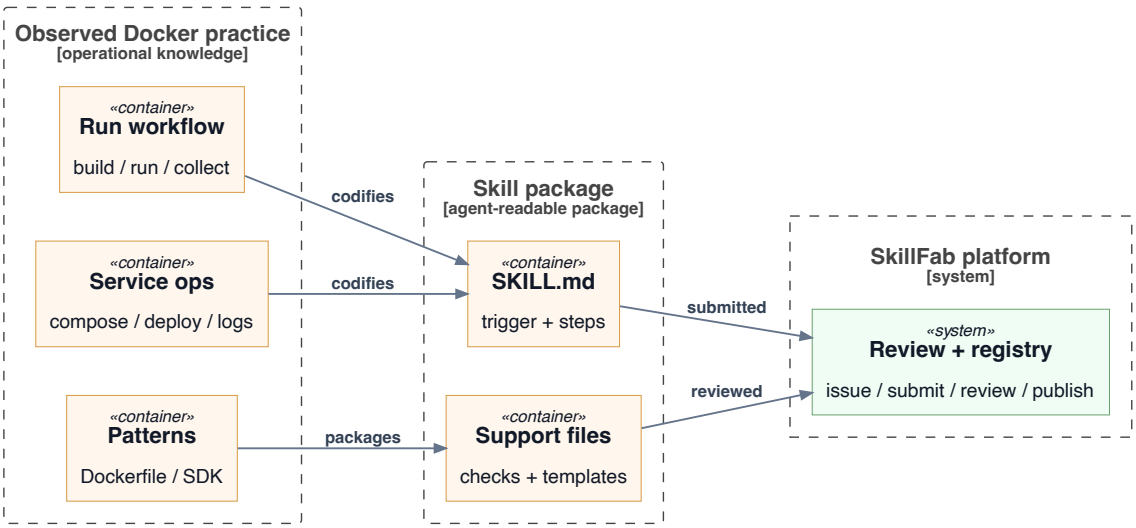

Figure 6. Docker practice as reusable skill knowledge. Operational practice is condensed into a small skill package that enters the normal SkillFab review and registry loop.

The Docker case positions software-to-skill as a normal SkillFab production path: observe repeated friction around an existing tool, open an issue, package operational knowledge as `SKILL.md` plus references, submit it for review, and publish the result for registry reuse.

Table 5 unpacks that package into retrieval scope, sandbox lifecycle, orchestration, hot deployment, and supporting references. The point is not Docker itself, but that an operational routine can become a reviewable skill package without introducing another platform object.

Skill component	Observed content	Why it matters for SkillFab
Trigger and scope	The front matter names the skill `docker-research` and triggers on Docker, containers, compose, Dockerfiles, sandboxing, deployment, SDK control, and reproducibility.	It gives the registry a clear retrieval surface: agents can discover the skill when a task mentions containerized experiments or reproducible execution.
Research sandbox lifecycle	The core workflow is `build -> run -> exec -> cp -> rm`, with detached containers, mounted data/output directories, parallel runs, and snapshot guidance.	Docker commands become a repeatable experiment procedure rather than ad hoc shell memory.
Multi-service orchestration	The skill describes compose services, health checks, environment variables, named volumes, bind-mounted source, and service restart/log commands.	Skills can encode operational judgment, not only short prompts or one-line commands.
Hot deployment and verification	The skill records an `scp -> docker cp -> docker exec` deployment pattern with hash checking before service restart.	Agents receive structured, auditable steps for a real engineering workflow.
Supporting references	The package includes reference files for Dockerfile templates and SDK patterns in Python and Rust.	Repository-backed packages can carry references that do not fit in a single instruction file.

Table 5. Docker-research as a software-to-skill case study.

7.3 Case Study: Governing Skill Optimization Through SkillFab

The third case asks whether SkillFab can govern an externally produced skill improvement without treating the optimizer as a built-in platform component. The example uses SkillOpt [25] only as a source of an optimized `SKILL.md` and supporting evidence. The platform question is narrower than optimizer evaluation: can an improved skill be linked to the prior published version, reviewed as a normal submission, and published as a new registry version?

In this case study, a seed `alfworld-agent` package is first published as v0.1.0. A follow-up issue is then opened with `related_skill_id` pointing to that skill. A developer submission pushes the optimized `SKILL.md` plus evidence files describing the optimizer setup and validation trace. The review packet therefore contains both the candidate skill and the evidence a maintainer needs to inspect. After approval, the package is published as v0.2.0, and `get_skill_versions` returns both versions.

This example is intentionally about platform governance rather than optimization performance. SkillOpt runs outside SkillFab; SkillFab only requires that the external system produce reviewable artifacts. In platform terms, optimization becomes another submission type: it has an originating issue, a linked source skill, Git-ingested files, a review packet, a maintainer decision, and a versioned registry result.

1. **Existing skill.** The original package is already visible as a registry skill with a stable version.
2. **Linked improvement request.** A new issue records the desired improvement and links back to the existing skill through `related_skill_id`.
3. **Reviewable submission.** The optimized skill document and supporting evidence are pushed through the normal repository path and captured by native Git ingestion.
4. **Versioned publication.** Maintainer approval publishes the accepted package as the next skill version while preserving the previous version.

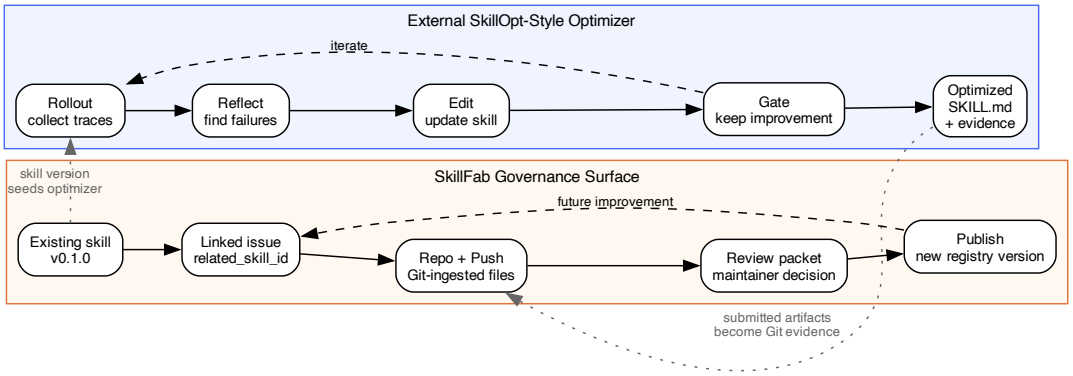

Figure 7. External SkillOpt-style optimization governed by SkillFab's normal submission path. The optimizer runs its own rollout, reflection, editing, and gate loop; only the optimized skill and review evidence enter the issue, Git evidence, review, and registry workflow.

Figure 7 summarizes this boundary while showing SkillOpt's work at a high level. The optimizer may run rollout, reflection, editing, and gating in its own environment. Only the resulting skill document and evidence enter SkillFab. This keeps the platform model stable while still allowing optimization tools to improve published skills through the same governance path used by human- or agent-written packages.

The main platform capability is versioned recovery of skill-improvement work. The original issue and skill remain visible, the improvement issue records why a new version was needed, the submission records the concrete files under review, and the registry records the resulting version history. This is the part SkillFab contributes: not a new optimizer, but a shared governance surface where optimization outputs can be reviewed, accepted, revised, or rejected like any other skill package.

8 Conclusion

SkillFab treats skill production as an agent-executable, human-governed lifecycle for reusable software skills. Its design contribution is the integration of demand-first issues, Git-backed evidence, agent-native workflow execution, machine-readable recovery state, maintainer certification, and registry-first reuse. Issues record missing capabilities, submissions connect implementation work to reviewable code evidence, maintainers control publication, and reuse can feed new work through issue reporting.

The OS-detect run exercises the central platform path end to end. The Docker-research case shows how existing software practice can become reusable agent knowledge. The SkillOpt optimization case illustrates how external optimization outputs can be handled through SkillFab's existing issue, Git evidence, review, and versioned-registry path. Together, they characterize SkillFab as a platform rather than a skill generator: it captures unmet demand, allocates implementation work, records software evidence, supports review, and returns accepted results through a registry.

Future work should test whether the lifecycle improves agent capability production in measurable ways: less repeated effort, fewer unnecessary implementations, better recovery after interruption, faster evidence-grounded review, and actual downstream reuse. The engineering agenda is equally concrete: production observability, failure recovery, security hardening, and trust policies that distinguish community publication from maintainer certification.

References

[1] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[2] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 68539–68551. Curran Associates, Inc., 2023.

[3] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 8634–8652. Curran Associates, Inc., 2023.

[4] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023.

[5] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating llms as agents. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[6] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[7] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[8] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[9] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[10] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering*, pages 712–721. IEEE, 2013.

[11] Laura Dabbish, H. Colleen Stuart, Jason Tsay, and James D. Herbsleb. Social coding in GitHub: Transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM, 2012.

[12] Jason Tsay, Laura Dabbish, and James D. Herbsleb. Influence of social and technical factors for evaluating contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering*, pages 356–366. ACM, 2014.

[13] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: The contributor's perspective. In *Proceedings of the 38th International Conference on Software Engineering*, pages 285–296. ACM, 2016.

[14] Scott Chacon and Ben Straub. *Pro Git*. Apress, 2 edition, 2014.

[15] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining Git. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10. IEEE, 2009.

[16] Model Context Protocol. Model Context Protocol Specification, 2024. Accessed 2026-06-08.

[17] SQLite Consortium. SQLite Documentation, 2026. Accessed 2026-06-08.

[18] SkillFab Team. SkillFab Official Website, 2026. Accessed 2026-06-11.

[19] SkillTester Team. SkillTester, 2026. Accessed 2026-06-11.

[20] Leye Wang, Zixing Wang, and Anjie Xu. SkillTester: Benchmarking utility and security of agent skills, 2026.

[21] Peter C. Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212. ACM, 2013.

[22] JSON-RPC Working Group. JSON-RPC 2.0 Specification, 2013. Accessed 2026-06-08.

[23] Margaret-Anne Storey, Alexey Zagalsky, Fernando Figueira Filho, Leif Singer, and Daniel M. German. How social and communication channels shape and challenge a participatory culture in software development. *IEEE Transactions on Software Engineering*, 43(2):185–204, 2017.

[24] Docker Inc. Docker Documentation, 2026. Accessed 2026-06-08.

[25] Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. SkillOpt: Executive strategy for self-evolving agent skills, 2026. Accessed 2026-06-12.