FEDRSI: COLLECTIVE SELF-IMPROVEMENT THROUGH HARNESS EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Language-model agents increasingly improve themselves by rewriting their own *harness*, the program that manages the context, memory, tools, and calls of a frozen base model. However, because each agent evolves within a private deployment, its improvements remain unavailable to other agents that could benefit from them. Existing agent federations do not close this gap. Almost all of them share a single kind of harness component, such as prompts, knowledge, or skills; the only one that shares the entire harness requires its server to read client code and to use labelled validation data; and all of them aggregate once or in synchronous rounds, which proceed at the pace of the slowest client. Therefore, we introduce *FedRSI*, to our knowledge the first protocol that converts private improvements to any part of the harness into shared executable modules while every client retains its data, its code, and its own pace. Clients export only prose *mechanism cards*. The server writes a work order for each proposed mechanism, reviews it for task-specific assumptions, and re-implements the mechanism from scratch within a shared harness. Each client adopts the shared modules for its own task only if they raise accuracy on local validation data, and a task-aware priority gives the reviewed shared modules precedence on all other tasks. Because a card refers to no particular version of the shared harness, the server aggregates as soon as a fixed number of new cards have arrived, so that clients whose items take from seconds to an hour never wait for one another. In a federation of three text clients whose tasks differ in domain, language, and output space, FedRSI exceeds independent evolution for every client on its own task and raises the macro accuracy by 4.9 points on the clients' own tasks and by 6.7 points on the tasks of the other clients; in a federation of three coding clients, it also exceeds independent evolution on the tasks of the other clients. Moreover, FedRSI achieves these gains without exposing client data or making any client wait: none of the exchanged cards or shared modules contains such data, whereas synchronous rounds would have kept the faster clients idle for up to 59% of a run.

1 INTRODUCTION

In recursive self-improvement (RSI), a system converts its experience into improvements of the mechanism that governs its future behavior (Good, 1966; Schmidhuber, 2003). Harness evolution is one concrete realization of this idea (Chen et al., 2026a; Lee et al., 2026a): for an agent built on a frozen language model, it improves the *harness*, the program that manages the context, memory, tools, and calls of the model. The harness strongly affects what the model achieves (Yang et al., 2024a), and recent methods revise it automatically from execution traces (Lee et al., 2026b; Nie et al., 2026b). These methods assume a single owner: the optimizer can inspect both the experience from which it learns and the harness implementation that it changes. We study the same iterative process, in which the base model and the local search procedure are fixed and each revised harness provides both the behavior and the feedback for the next search, and ask how its improvements can propagate between owners.

However, agents often belong to different users or organizations, which cannot or will not share their raw records, execution traces, or source code, whether for business or for privacy reasons. Each client then evolves a private harness from its own tasks and feedback, and a retrieval, memory, or verification mechanism remains with the client that found it, even when it could help another client or advance an agent whose progress on its own experience has stalled (Pan et al., 2026). Centralizing the

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

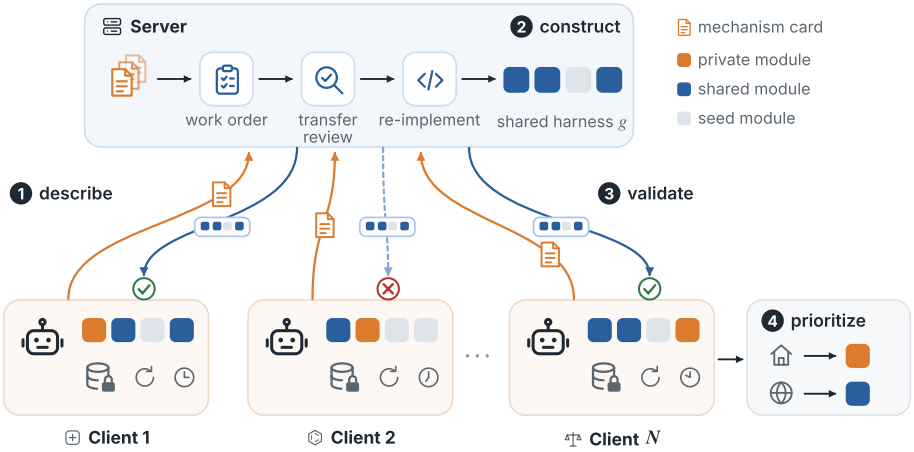


Figure 1: **The FedRSI pipeline.** Each client is an agent that improves its harness by local search on private data; squares denote harness modules. **(1)** A client describes each private improvement in a mechanism card, which is the only content that it uploads. **(2)** The server converts the cards into work orders, reviews whether the work orders transfer across tasks, and re-implements them from scratch within the shared harness g. **(3)** Each client validates the new version of g locally and adopts the new modules for its own task only if its performance improves. **(4)** At deployment, a private module serves the client's own task, and its shared counterpart serves all other tasks. The clocks indicate that every client proceeds at its own pace: cards arrive at different times, and the server acts whenever a fixed number of new cards have arrived.

search over the trajectories of all clients would remove this confinement (Ma et al., 2026), but only by requiring the access that this setting rules out. Federated learning addresses the same boundary for models by aggregating updates in a common parameter space (McMahan et al., 2017; Kairouz et al., 2021), but heterogeneous program modules admit no such average. Moreover, clients differ in pace, a difference that federated learning calls systems heterogeneity and distinguishes from statistical heterogeneity (Li et al., 2020): a classification item takes seconds, whereas an agentic coding item can take up to an hour. Therefore, we ask whether executable harness improvements found at one client can benefit other clients while raw data and source code remain local, each client keeps its specialization on its own task, and every client evolves at its own pace.

Answering this question requires overcoming three obstacles. First, a successful local improvement entangles its reusable mechanism with private implementation details; since the code cannot leave the client, whatever is exported in its place must still enable another party to rebuild the behavior. Second, the mechanism may depend on assumptions that hold only on its source task, so that transferring it can harm the recipient (Wang et al., 2019); indeed, harness optimizers are known to adopt shortcuts that are specific to the tasks on which they search (Zhu et al., 2026; Wu et al., 2026). Third, even a suitable shared implementation can be displaced once it is composed with the private modules of the recipient, just as independently developed software features interfere with one another (Calder et al., 2003). For example, a private module that maps answers onto known class labels overrides a shared module designed for open-ended answers.

Existing agent federations do not answer this question: almost all of them federate a single kind of harness component, and the only one that federates the entire harness sacrifices the privacy of client code (Table 1). Therefore, improvements to any other part of the harness cannot transfer, whether they concern the storage and retrieval of experience, the construction of the context, or the verification of an answer. Moreover, except for skill scripts, the single shared component reaches the recipient as context for its model rather than as an executable program. EvolveNet is the only prior federation that operates on the entire harness: it decomposes harness changes into mechanisms, determines their scope, and validates their composition (Nie et al., 2026a). However, its server reads client source code and diffs and relies on labelled validation data, so that both the private implementation and client data leave the clients. Finally, all of these federations aggregate once or in synchronous rounds, in which the server waits until every client has reported, and the authors of EvolveNet observe that each round ends with its slowest client (Nie et al., 2026a).

Table 1: **Comparison of FedRSI with prior agent federations.** The prior methods are Fed-SE (Chen et al., 2025b), FedTextGrad (Chen et al., 2025a), FICAL (Wu et al., 2024), FoT (Yao et al., 2026), FedAgentKE (Li et al., 2026), Synapse (Chakraborty et al., 2026), FedWorld (Hou, 2026), FederatedSkill (Yang et al., 2026), and EvolveNet (Nie et al., 2026a). *Entire harness*: any component of the harness can be federated, not only a single kind such as prompts, knowledge, or skills. *Executable*: recipients receive code rather than context. *Code private*: no client source code leaves its owner. *Recipient decides*: each recipient decides on its own data whether shared content enters its harness. *Asynchronous*: the server aggregates without waiting for every client. Only FedRSI has all five properties.

Method	Clients share	Entire harness	Executable	Code private	Recipient decides	Asyn-chronous
Fed-SE	adapter weights	✗	✗	✓	✗	✗
FedTextGrad	prompts	✗	✗	✓	✗	✗
FICAL	knowledge compendiums	✗	✗	✓	✗	✗
FoT	reasoning traces	✗	✗	✓	✗	✗
FedAgentKE	knowledge units	✗	✗	✓	✗	✗
Synapse	typed compendiums	✗	✗	✓	✗	✗
FedWorld	world-model rules	✗	✗	✓	✗	✗
FederatedSkill	skill files with scripts	✗	✓	✗	✗	✗
EvolveNet	harness source and diffs	✓	✓	✗	✗	✗
FedRSI	**prose mechanism cards**	✓	✓	✓	✓	✓

Thus, to transfer improvements while every client retains its data, its code, and its pace, we propose *FedRSI*, a protocol that resolves each obstacle at the party that holds the necessary information (Figure 1). First, to separate the mechanism from its code, a client exports only a *mechanism card*, a prose description of an improvement that its local search has kept, and the server re-implements the mechanism from scratch within a shared harness. Second, to guard against assumptions of the source task, the server reviews every proposed mechanism for such assumptions before it builds anything, and each recipient adopts the result only if accuracy on its own validation data improves, because only the data of the recipient can show whether an improvement benefits it. Third, to prevent displacement, a *task-aware priority* determines which of two competing implementations executes: on its own task, a client retains its private implementation, which local search selected by measured accuracy, whereas on every other task the reviewed shared implementation takes precedence. With this design, FedRSI achieves five properties that no prior agent federation combines (Table 1). (1) **Federation of the entire harness.** Specifically, local search may add a module at any point of the harness, from the storage of experience to the verification of an answer, and every such module can become shared behavior. (2) **Executable transfer.** Specifically, a recipient receives an executable module rather than context for its model, and the server rebuilds this module from a mechanism card. (3) **Private code.** Specifically, only prose leaves a client, so that no client discloses its source code or its data. (4) **Adoption decided by the recipient.** Specifically, no shared module enters the harness that a client serves on its own task unless it raises accuracy on the data of that client. (5) **Asynchronous aggregation.** Specifically, because a card describes a behavior rather than a change to a particular version of the shared harness, it remains usable whenever it arrives; therefore, the server aggregates as soon as a fixed number of new cards have arrived, and no client ever waits for another. Moreover, improvement becomes recursive across clients: an adopted improvement becomes the starting point of the next local search of the recipient, so that an improvement found by one client shapes the search of another.

Our experiments show that an improvement found at one client benefits another. In the case study of Section 4.4, a module that the server rebuilt from a card of the Symptom client enables the agent of the Crime client to solve a case that it fails without the module, and across three runs, clients tested 16 compositions that added modules rebuilt from the cards of other clients and adopted 6 of them. In a federation of three text clients whose tasks differ in domain, language, and output space, FedRSI exceeds independent evolution for every client on its own task and raises the macro accuracy by 4.9 points (95% confidence interval [+1.1, +8.9]) on the clients' own tasks and by 6.7 points ([+3.4, +10.0]) on the tasks of the other clients; in a federation of three coding clients, it also exceeds independent evolution on the tasks of the other clients (Appendix F.3). FedRSI retains its lead over

independent evolution on a stronger and a weaker base model (Appendix F.4). Finally, none of the cards, work orders, or shared modules contains client data, whereas 7 of the 18 private modules that clients keep contain true labels, and synchronous rounds would have kept the faster clients idle for up to 59% of a run.

2 Related Work

Self-improving agents. Many methods let a single agent improve the program around its model. They optimize instructions and pipeline components (Yang et al., 2023; Khattab et al., 2023; Agrawal et al., 2025), accumulate experience as reflections or skills (Shinn et al., 2023; Wang et al., 2023), and evolve agent code and entire harnesses (Robeyns et al., 2025; Zhang et al., 2025a; Lee et al., 2026b), in some cases with checks against changes that help only on the benchmark (Zhang et al., 2026b; Zhu et al., 2026). However, each of these methods improves a single deployment. FedRSI keeps such a local optimizer unchanged and transfers its improvements to other deployments without access to the private program in which they originated.

Agent federations. Agent federations transfer knowledge between deployments, but almost all of them share a single kind of harness component, such as prompts, knowledge, reasoning guidance, tool-routing knowledge, world-model rules, skills, or model adapters (Chen et al., 2025a; Wu et al., 2024; Yao et al., 2026; Chakraborty et al., 2026; Hou, 2026; Yang et al., 2026; Chen et al., 2025b). A conceptual architecture for federating recursive self-improvement has also been proposed, in which the systems exchange only scalar progress scores (Lemon, 2026);[1] however, such scores do not contain the improvements themselves. Among prior federations, only EvolveNet operates on the entire harness, and its server reads client code and uses labelled validation data (Nie et al., 2026a), whereas in FedRSI only prose reaches the server.

Asynchronous aggregation. Moreover, every agent federation with a server aggregates once or in synchronous rounds, so that each round waits for its slowest client, as the authors of EvolveNet and Fed-SE acknowledge (Nie et al., 2026a; Chen et al., 2025b). Federated learning has long decoupled clients through asynchronous aggregation (Xie et al., 2019; Nguyen et al., 2021), and FedRSI brings this decoupling to programs, because a card refers to no particular version of the shared harness.

Personalization and local adoption. Finally, personalized federated learning lets each client decide what it adopts, for example by weighting the models that it receives by their accuracy on its local validation data (Zhang et al., 2020). FedRSI extends this principle from models to programs. However, unlike parameters, programs cannot be interpolated: a single private module can override a shared one entirely (Calder et al., 2003). Appendix A discusses the related work in full.

3 FedRSI

Problem setting. We consider N clients, each of which serves a private task with a frozen base model. Client k receives the label of every item after answering it and thus accumulates a labelled history H_k; the clients differ in their tasks and in their pace, and they do not synchronize. The *harness* of a client is an ordered collection of executable *modules*, each of which provides or modifies one operation around the model, such as retrieving experience, constructing the context, or extracting the answer. The server maintains a *shared harness* g, initialized from a common seed harness, and broadcasts every new version of g to all clients. Client k keeps the *private modules* p_k that its own local search has found, and its *serving harness* h_k combines p_k with the shared modules that the client has adopted. The server receives only prose descriptions of improvements and never client code, data, or labels. We write $U_j(h \mid H_j)$ for the expected score of a harness h on task j after it has processed the history H_j. We evaluate the harness of every client on its own task, which we call in-domain (ID) evaluation, and on the tasks of the other clients, which we call out-of-domain (OOD) evaluation; in both cases, client k answers the items itself, so that its private modules never leave it. The task on which a harness is used is its *deployment context* s, and h_k^s denotes the harness that client

[1] This preprint, titled *Federated RSI*, presents a conceptual framework; it contains no implementation or experiment, and we found none as of September 2026.

k uses in context s, with $h_k^k = h_k$; at deployment, the language model of the client judges from its past runs whether an item belongs to its own task. Appendix B states the setting in full.

FedRSI converts an improvement found by one client into a module that every client can execute, and it provides each client with two harnesses: the serving harness for its own task and a global-first composition for every other task (Section 3.4). The following subsections trace an improvement through the protocol (Figure 1), and Appendix C specifies every step at the level required for reproduction (Figure 3).

3.1 Local Search and Mechanism Cards

Each client improves its harness by a local search in the style of Meta-Harness (Lee et al., 2026b). After every fixed number of served items, a coding agent acting as the *proposer*[2] analyzes the served records of the client and the outcomes of earlier proposals, and writes one new module, which may act at any point of the harness. A proposal can only *add* a module and never edits or replaces an existing one, so that every private improvement can later be replaced by a shared one (Section 3.4). The client scores the incumbent harness and each proposal on its private validation items and keeps the best proposal in p_k only if it scores strictly higher than the incumbent.

Federation adds a single step to this local search: the export of kept improvements. For every kept module, a separate model call writes a *mechanism card*, a short prose account of what the mechanism does, why it helps, and when it engages. The prose of a card contains no code, identifiers, literals, or examples, and a scan for literals and a check for planted canary strings (Carlini et al., 2019) screen the card before it is uploaded. In addition to its prose, a card carries the opaque identifier of the module that it describes, so that the server can name its own implementation after the originating module.

Thus, the server obtains a description of useful behavior rather than an implementation, and it must construct an implementation of its own. Reconstruction transfers only the part of an improvement that can be described independently of the task. Since a task-independent description can still rest on a task-dependent assumption, the server reviews every mechanism before it builds it.

3.2 Construction of Shared Modules

Buffered asynchronous aggregation. Because a card describes a behavior rather than a change to a particular version of g, the server needs no synchronization: it opens a *federation round* as soon as B new cards have arrived, and no client ever waits for another. We defer the detailed description and the analysis of this buffered asynchronous aggregation to Appendix F.11.

From cards to work orders. A card describes what one client has found, whereas the server requires a specification of what to build into g. Therefore, in each round, the server makes a single model call, the *work-order call*, which reads the pending cards against the current version of g, selects at most K mechanisms, and writes one *work order* for each of them. A work order specifies the desired change and the conditions under which it applies, and it must cover the entire scope of every card that it cites, because the resulting shared module may later replace the modules of the clients that uploaded these cards (Section 3.4).

Transfer review before implementation. A card reports a gain on one task, whereas the shared harness will execute on all tasks. Therefore, before any code is written, a reviewer model that acts as an LLM judge (Zheng et al., 2023) examines the work orders of the round in a single batch. It sees the cards, the shared source code, and the runtime interface, but no client data or labels. For each work order, it states the assumptions on which the benefit depends, searches for a concrete task on which these assumptions would fail, and verifies that the work order covers the entire scope of its cards. It then accepts the work order, *revises* it with explicit conditions, or defers it; a deferred mechanism remains private and unchanged. A revision may rely only on information that is available at execution time. For instance, a module that replaces an answer with the rewarded answer of the most similar past question presumes that the answers of the task recur, and a revision can permit the replacement only when the stored answer was rewarded on a closely matching question. Thus, transfer review assesses a mechanism for the tasks of every client without access to any client data.

[2] OpenCode, `https://opencode.ai`.

Re-implementation from scratch. Each work order that the review accepts or revises serves as a natural-language specification of a new module (Chen et al., 2021; Zhao et al., 2024b). In each round, the server runs a single coding-agent session, the *implementer*, which writes all of these modules from the work orders, the shared source code, and the runtime interface without seeing the cards or any client code, so that every shared module is built on g rather than adapted from a private program. If the extended shared harness passes a dry run on a synthetic labelled item, the server broadcasts it to all clients as the next version of g. Whether the new modules also improve accuracy is decided where the data reside, namely by each recipient.

3.3 Composition and Local Validation

Broadcasting a shared module does not determine whether it executes, because the recipient may already hold a private module with the same function. Therefore, each client *composes* g with p_k in three steps. First, the client omits the shared implementation of its own card and keeps its original, which local search has already validated on its task. Second, deterministic checks resolve every case that requires no judgment: a shared and a private module that register the same service or tool collide, and modules that fail to load or to run are excluded. Third, a merger model examines every remaining pair of a shared and a private module that act on a common service or pipeline stage. If the two modules serve different functions, the pair is *complementary*, and both execute; if they implement the same function, or if executing both would break either of them, the pair is *competing*, and only one remains. During evolution, the private module prevails in every collision and every competing pair, which preserves the specialization that local search has established. We write $\mathcal{C}_{\mathrm{local}}(g, p_k)$ for this *local-first composition*.

A composition that loads and passes the dry run may still perform worse than the harness that it would replace, just as a federated model can be less accurate for a client than the model that the client trains alone (Yu et al., 2020). Therefore, with *local validation*, the client compares the candidate $\widetilde{h}_k = \mathcal{C}_{\mathrm{local}}(g, p_k)$ with the incumbent h_k^{old} on the same private validation items, after both harnesses have replayed the same labelled history through their own memory modules. The validation items are disjoint from this history, and their labels serve for scoring only. With $\widehat{U}_k$ denoting the empirical utility, the client adopts

$$h_k = \begin{cases} \widetilde{h}_k, & \text{if } \widehat{U}_k(\widetilde{h}_k) > \widehat{U}_k(h_k^{\mathrm{old}}), \\ h_k^{\mathrm{old}}, & \text{otherwise.} \end{cases} \tag{1}$$

We call each such comparison an *adoption test*. The client runs this test before its next local search, so that it never waits for the server; a tie or a failed evaluation retains the incumbent.

In turn, adoption makes improvement recursive across clients: the next local search starts from the adopted harness, so that an improvement found by one client shapes the search of another.

3.4 Task-Aware Priority at Deployment

Local validation measures utility on the client's own task and therefore cannot reveal whether a composition is suitable for other tasks. However, a private module that prevails in a competing pair also executes on every other task that the client serves. For example, the private version of the answer-replacement module of Section 3.2 overwrites correct answers on a task with open-ended answers, since only the shared version carries the condition that the reviewer added. The safeguard constructed by the server is then absent precisely where it is needed.

We resolve this problem with a *task-aware priority*. Let $\mathcal{C}_{\mathrm{global}}(g, p_k)$ denote the *global-first composition*, in which the priority is reversed throughout: every competing pair keeps the shared module, every collision excludes the private module, and the server's implementation of the client's own card replaces the original. Private modules that are complementary to a shared module or have no shared counterpart remain. Given the deployment context s, the client serves

$$h_k^s = \begin{cases} h_k, & \text{if } s \text{ is the client's own task (local-first),} \\ \mathcal{C}_{\mathrm{global}}(g, p_k), & \text{if } s \text{ is another task (global-first).} \end{cases} \tag{2}$$

Thus, each branch is protected by the strongest check that the available information permits: the serving harness by validation on the data of the client, and the shared modules of the global-first composition by the transfer review of the server, which covers the tasks that a client cannot test.

4 EXPERIMENTS

Our experiments address three questions: whether federation improves every client beyond what the client achieves alone (Section 4.2), what each decision of the protocol contributes (Section 4.3), and whether an improvement found at one client benefits another (Section 4.4). Appendix F reports further experiments: the results of the coding federation, a comparison with centralized evolution, the evolved harnesses on other base models, all three runs of the text federation, an analysis of privacy leakage, and the idle time that asynchronous aggregation avoids.

4.1 SETUP

We evaluate FedRSI on two federations of three clients each. In the *text federation*, each client serves one task of the online text-classification setting of Meta-Harness (Lee et al., 2026b), and the three tasks differ in domain, language, and output space: *Symptom* names a medical diagnosis from the description of a patient,[3] *Crime* names the criminal charges of a Chinese court case (Fei et al., 2024), and *USPTO* names the reactants from which a given chemical product is synthesized (Schneider et al., 2016). In the *coding federation*, each client writes Python functions for the items of BigCodeBench (Zhuo et al., 2024) that use one kind of library, and each answer is judged by the unit tests of its item: *Computation* covers numerical, data, and learning libraries, *Visualization* plotting libraries, and *System* the standard library. Each client serves a stream of items and receives the label of each item after answering it. We compare FedRSI with *independent evolution*, in which every client runs the same local search from the same seed harness but exchanges nothing. More settings, including a comparison with centralized evolution, are given in Appendix F. The base model is Qwen3.8-Flash. We run the text federation three times and report the run in which FedRSI answered its served items most accurately during evolution; Appendix F reports all three runs. After evolution, we evaluate the final harness of every client on held-out items, namely stream items that no configuration served, both of its own task (ID) and of the tasks of the other clients (OOD); on the tasks of the other clients, a FedRSI client uses its global-first composition. We report macro accuracy over the three ID pairs and over the six OOD pairs of client and task. Appendix D details all settings.

4.2 MAIN RESULTS

FedRSI exceeds independent evolution for every text client on its own task, although no data or code leaves any client (Table 2). The ID macro rises from 59.43% under independent evolution to 64.35% (+4.9 points; 95% confidence interval [+1.1, +8.9]; 20 items won and 7 lost, sign test $p = 0.02$), and every client gains: Symptom by 1.6, USPTO by 2.6, and Crime by 10.5 points. Each FedRSI client retains its specialization through the local-first composition and adopts shared modules only where they raise its validation accuracy.

On the tasks of the other clients, FedRSI exceeds independent evolution by 6.7 points on the OOD macro (58.86% against 52.15%; 95% confidence interval [+3.4, +10.0]; 61 items won and 13 lost). The gain is largest where an independently evolved harness is specialized for its own task: the harness that the USPTO client evolved alone scores 75.40% on Symptom and 39.47% on Crime, whereas FedRSI scores 91.98% and 61.84% on the same pairs. Both gains hold on average over all three runs: FedRSI exceeds independent evolution by 4.3 points on the clients' own tasks and by 4.6 points on the tasks of the other clients (95% confidence intervals [+1.6, +7.1] and [+2.5, +6.7]; Appendix F.6).

Moreover, the shared harness alone, which the server built from prose cards only, reaches 59.43% on the clients' own tasks, as high as independent evolution and twice the seed harness (Appendix F.5).

4.3 ABLATION STUDY

Task-aware priority. Of the three decisions, task-aware priority contributes the most on other tasks. The global-first composition raises the OOD macro from 46.76% under the local-first composition to 58.86% (95% confidence interval of the difference [+8.8, +15.2]; 89 items won and 14 lost; Appendix F.1), and since both compositions come from the same evolved clients, this comparison is free of the variation between runs. Averaged over all three runs, the global-first composition still

[3] Gretel AI, https://huggingface.co/datasets/gretelai/symptom_to_diagnosis.

Table 2: **Held-out accuracy (%) of the text federation for all client–task pairs.** Each row gives the client whose final harness is evaluated, the task, and the number n of held-out items. The seed harness is the same for every client; FedRSI serves other tasks with its global-first composition. Entries come from the reported run (Section 4.1), and Appendix F gives the average over all three runs; macro rows weight pairs equally, and Δ is the difference from independent evolution in percentage points.

Client → task	n	Seed	Independent	FedRSI
In-domain (ID): the client's own task				
Symptom → Symptom	187	74.33	91.44	**93.05**
USPTO → USPTO	76	11.84	21.05	**23.68**
Crime → Crime	76	0.00	65.79	**76.32**
ID macro		28.72	59.43	**64.35**
Δ vs. independent		−30.70	–	+4.92
Out-of-domain (OOD): another client's task				
USPTO → Symptom	187	74.33	75.40	**91.98**
Crime → Symptom	187	74.33	**91.44**	**91.44**
Symptom → USPTO	76	11.84	**25.00**	19.74
Crime → USPTO	76	11.84	**27.63**	21.05
Symptom → Crime	76	0.00	53.95	**67.11**
USPTO → Crime	76	0.00	39.47	**61.84**
OOD macro		28.72	52.15	**58.86**
Δ vs. independent		−23.43	–	+6.71

raises the OOD macro by 3.2 points (95% confidence interval [+1.1, +5.1]; Appendix F.6). The local-first composition fails where the serving harness of a client lacks the shared modules that another task requires, because the private module prevails in every competing pair and local validation tests only the client's own task (Section 3.4; Appendix F.7).

Local validation and transfer review. Local validation and transfer review trade accuracy on other tasks for accuracy on the clients' own tasks: removing either lowers the ID macro from 64.35% to 62.86% and 63.55% and raises the OOD macro from 58.86% to 64.13% and 59.96%, respectively (Appendix F.1). Beyond this trade-off, both decisions prevent a failure that our pilot study encountered when it ran the protocol without them. The server rebuilt two cards of the Crime client into shared modules that rewrite the answer: one replaces it with the answer of the most similar stored cases, and the other rejects any answer that is not a stored label. Both help on a task whose answers recur, but the answers of USPTO are new sets of reactants. Without either check, both modules entered the serving harness of USPTO, and its validation accuracy fell from 55% to 0%, whereas the same client reached 60% under independent evolution (Appendix F.9). Transfer review prevents such a failure by restricting a module to the situations in which its assumption holds, and local validation prevents it by rejecting any composition that lowers the accuracy of a client on its own task.

4.4 Case Study: An Improvement Transferred Across Tasks

A single improvement of the reported run shows how a mechanism found on one task benefits another (Figure 2). The Symptom client found that the model answers from its prior knowledge where the labels of its task follow a different convention, and that a tool for searching the labelled examples of the environment lets the model check a candidate answer against solved cases before answering. Its card describes this mechanism in prose, without code, identifiers, or task content, and the module itself remained private. The server turned the card into a work order, and the transfer review accepted the work order because its correctness rests on the runtime interface rather than on any property of the source task. The implementer then wrote a new module from the work order; the module adds a search tool for every example corpus that the environment offers.

The Crime client tested the new version of g before its next local search and adopted it, since its accuracy on 48 validation items rose from 64.6% to 66.7%. Validation item 6 of Crime shows the difference that the module makes (Figure 2). Its label is the full statutory name of a single charge that

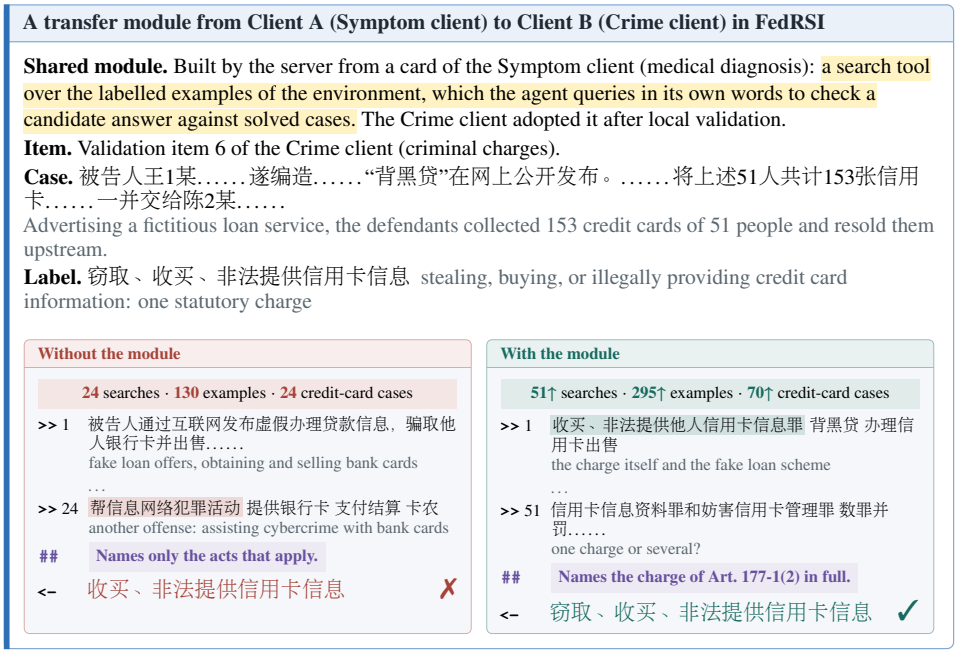


Figure 2: **Case study.** The agent of the Crime client answers the same item twice, once without and once with a shared module that the server rebuilt from a card of the Symptom client; the two harnesses are otherwise identical. With the module, the agent searches the labelled examples more often and names the charge in full, as the labels do. >> marks a search, ## our note on the reasoning of the agent, and <– its answer. The case is in Chinese in the original dataset.

lists three acts. Without the module, the agent searches the labelled examples 24 times and names only the acts that apply to the defendants; with it, the agent searches them 51 times and gives the full statutory name, as the labels do. The card of the Symptom client describes this conflict between the convention of the labels and the prior knowledge of the model, and the module resolves it on a task that shares neither domain nor language with diagnosis. Crime later also adopted modules rebuilt from cards of the Symptom and USPTO clients, and it ends with the largest gain of any client over independent evolution, 10.5 points on its held-out items (Table 2). Across the three runs, clients tested 16 compositions that added modules rebuilt from the cards of other clients and adopted 6 of them (Table 14).

5 Conclusion

We introduced FedRSI, a protocol that lets agents improve collectively while each evolves its entire harness in a private deployment: clients share only mechanism cards that describe their improvements, the server reviews these mechanisms and rebuilds them as shared code, and each client decides locally what to adopt and what to execute. Our experiments establish that improvements found at one client benefit another: FedRSI exceeds independent evolution on the tasks of the other clients in both federations and on the clients' own tasks in the text federation, while no client discloses its data or waits for another. More broadly, these results show that recursive self-improvement need not remain within a single owner: prose descriptions of mechanisms suffice to transfer executable improvements between clients that share neither data nor code. Future work can extend FedRSI to larger federations, to clients with different base models, and to the local search procedure itself, so that the recursion also improves the procedure that improves the harness.

Limitations. The task-aware priority switches between the local-first and the global-first composition according to the client's own language model, which judges from the past runs of the client whether an item belongs to its own task. This judgment achieved 100% accuracy in our tests; however, the probabilistic nature of language models precludes an absolute guarantee.

Reproducibility Statement

Appendix D specifies the clients, tasks, seed harness, schedule, and models of every configuration and lists all settings, the evaluation protocol, and the model usage of every run. Appendix C gives the complete protocol as an algorithm together with the rules of composition and local validation, and Appendix E reproduces the card-writing and transfer-review prompts and the artifacts of one transfer verbatim. All datasets are public. The held-out items of every task follow from the fixed serving order of each run and the deduplication described in Appendix D, and the bootstrap uses a fixed seed. Appendix F reports all three runs of the text federation together with the item-level statistics.

Ethics Statement

This work uses only public datasets: symptom descriptions with diagnoses, Chinese criminal cases with charges, chemical reactions from patents, and programming tasks with unit tests. It involves no human subjects and no new data collection. The diagnosis and charge-prediction tasks serve as benchmarks, and no system in this work makes medical or legal decisions. FedRSI keeps the data and the code of every client local, and Appendix F.10 shows that the artifacts that it exchanges contain no client data.

AI Use Statement

Large language models assisted the preliminary interpretation of the experimental data, and the authors then rigorously checked every interpretation against the experimental records. Large language models also assisted in polishing the writing of this paper. In addition, large language models assisted in searching for related work and in drafting parts of some sections, and the authors reviewed and verified all of this material.

References

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, A. Dimakis, Ion Stoica, Dan Klein, Matei A. Zaharia, and O. Khattab. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. *ArXiv*, abs/2507.19457, 2025.

Muffy Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Comput. Networks*, 41:115–141, 2003.

Nicholas Carlini, Chang Liu, Ú. Erlingsson, Jernej Kos, and D. Song. The Secret Sharer: Evaluating and Testing Unintended Memorization in Neural Networks. In *USENIX Security Symposium*, pp. 267–284, 2019.

Abhijit Chakraborty, Ni Trieu, and Vivek Gupta. Typed Federated Artifacts for the Agentic Web:Sharing Tool-Routing Knowledge Across Frozen,Heterogeneous LLM Agents. *ArXiv*, abs/2609.06815, 2026.

Hong-You Chen and Wei-Lun Chao. On Bridging Generic and Personalized Federated Learning. *ArXiv*, abs/2107.00778, 2021.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, H. Khlaaf, G. Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mo Bavarian, Clemens Winter, P. Tillet, F. Such, D. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, S. Balaji, Shantanu Jain, William Saunders, Christopher Hesse, A. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, M. Knight, Miles Brundage, M. Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, I. Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. *ArXiv*, abs/2107.03374, 2021.

Ming-Hui Chen, Rui-Nan Jin, Wen-Long Deng, Yuan-Yuan Chen, Zhi Huang, Han Yu, and Xiao-Xiao Li. Can Textual Gradient Work in Federated Learning? *ArXiv*, abs/2502.19980, 2025a.

Mingguang Chen, Licheng Wang, and Bo Qu. Recursive Self-Improvement in AI: From Bounded Self-Refinement to Autonomous Research Loops. *ArXiv*, abs/2607.07663, 2026a.

Tingyang Chen, Shuo Lu, Kang Zhao, Weicheng Meng, H. Teng, Tianhao Li, Chao Li, Xule Liu, Jian Liang, Zhi-Zhong Zhang, Yuan Xie, Heng Qu, Kun Shao, and Jian Luan. HarnessX: A Composable, Adaptive, and Evolvable Agent Harness Foundry. *ArXiv*, abs/2606.14249, 2026b.

Xiang Chen, Yuling Shi, Qizhen Lan, Yuchao Qiu, Min Wang, Xiao-Dong Gu, and Yanfu Yan. FedSE: Federated Self-Evolution for Privacy-Constrained Multi-Environment LLM Agents. *ArXiv*, abs/2512.08870, 2025b.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching Large Language Models to Self-Debug. *ArXiv*, abs/2304.05128, 2023.

Liam Collins, Hamed Hassani, Aryan Mokhtari, and S. Shakkottai. Exploiting Shared Representations for Personalized Federated Learning. *ArXiv*, abs/2102.07078, 2021.

Tianyu Cui, Hongxia Li, Jingya Wang, and Ye Shi. Harmonizing Generalization and Personalization in Federated Prompt Learning. *ArXiv*, abs/2405.09771, 2024.

Yuyang Deng, Mohammad Mahdi Kamani, and Mehrdad Mahdavi. Adaptive Personalized Federated Learning. *ArXiv*, abs/2003.13461, 2020.

Zhiwei Fei, Xiaoyu Shen, Da-Wei Zhu, Fengzhe Zhou, Zhuo Han, Alan Huang, Songyang Zhang, Kai Chen, Zhixin Yin, Zong-Wen Shen, Ji-Dong Ge, and Vincent Ng. LawBench: Benchmarking Legal Knowledge of Large Language Models. In *Conference on Empirical Methods in Natural Language Processing*, pp. 7933–7962, 2024.

Chrisantha Fernando, Dylan Banarse, H. Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-Referential Self-Improvement Via Prompt Evolution. In *International Conference on Machine Learning*, pp. 13481–13544, 2024.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, J. Qiu, Xuan Qi, Yiran Wu, Hongru Wang, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Qihan Ren, Cheng Qian, Zhenhailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. A Survey of Self-Evolving Agents: What, When, How, and Where to Evolve on the Path to Artificial Super Intelligence. *ArXiv*, abs/2507.21046, 2025.

I. J. Good. Speculations Concerning the First Ultraintelligent Machine. *Adv. Comput.*, 6:31–88, 1966.

Yuchao Hou. FedWorld: Scope-Aware Federation of Agent World Models. *ArXiv*, abs/2608.01561, 2026.

Shengran Hu, Cong Lu, and Jeff Clune. Automated Design of Agentic Systems. *ArXiv*, abs/2408.08435, 2024.

Gabriel Ilharco, Marco Tulio Ribeiro, Mitchell Wortsman, Suchin Gururangan, Ludwig Schmidt, Hannaneh Hajishirzi, and Ali Farhadi. Editing Models with Task Arithmetic. *ArXiv*, abs/2212.04089, 2022.

Wen Jiang, Ming-Min Chu, Yimeng Tian, Qian-Xin Zhang, Haofei Yang, Rui Yang, Yang Liu, Tao Lv, and Fang-Ming Li. HarnessEvolve: Learning from Reference Trajectories for Reliable Agent Self-Evolution. *ArXiv*, abs/2609.00829, 2026.

P. Kairouz, H. B. McMahan, Brendan Avent, A. Bellet, M. Bennis, A. Bhagoji, Kallista Bonawitz, Zachary B. Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D'Oliveira, Hubert Eichner, S. E. Rouayheb, David Evans, Josh Gardner, Zachary Garrett, A. Gascón, Badih Ghazi, Phillip B. Gibbons, M. Gruteser, Z. Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, T. Javidi, Gauri Joshi, M. Khodak, Jakub Konecný, A. Korolova, F. Koushanfar, O. Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, M. Mohri, R. Nock, Ayfer Özgür, R. Pagh,

Mariana Raykova, Hang Qi, Daniel Ramage, R. Raskar, D. Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, A. Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. Advances and Open Problems in Federated Learning. *Found. Trends Mach. Learn.*, 14:1–210, 2021.

O. Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *ArXiv*, abs/2310.03714, 2023.

Hyunin Lee, Jinglue Xu, Jeffrey Seely, Donghyun Lee, Matei A. Zaharia, and Yu-Jin Tang. Recursive Harness Self-Improvement. *ArXiv*, abs/2607.15524, 2026a.

Yoonho Lee, R. Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-End Optimization of Model Harnesses. *ArXiv*, abs/2603.28052, 2026b.

Mark Lemon. Federated RSI: A Sovereign Architecture for Multi-Agent Recursive Self-Improvement and Distributed Recursive Intelligence. Preprint, figshare, 2026. URL https://doi.org/10.6084/m9.figshare.31724512.v1.

Daliang Li and Jun-Pu Wang. FedMD: Heterogenous Federated Learning via Model Distillation. *ArXiv*, abs/1910.03581, 2019.

Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated Learning: Challenges, Methods, and Future Directions. *IEEE Signal Processing Magazine*, 37:50–60, 2020.

Tian Li, Shengyuan Hu, Ahmad Beirami, and Virginia Smith. Ditto: Fair and Robust Federated Learning Through Personalization. In *International Conference on Machine Learning*, pp. 6357–6368, 2021.

Weihao Li, Jun Bai, and Ziyang Song. FedAgentKE: Federated Semantic Knowledge Evolution for Heterogeneous Agents. *ArXiv*, abs/2607.21361, 2026.

Bowen Liu, Bharath Ramsundar, Prasad Kawthekar, Jade Shi, Joseph Gomes, Quang Luu Nguyen, Stephen Ho, J. Sloane, P. Wender, and V. Pande. Retrosynthetic Reaction Prediction Using Neural Sequence-to-Sequence Models. *ACS Central Science*, 3:1103–1113, 2017.

Ze-Wen Liu, Zhan Shi, Yisi Sang, Bing He, Minhua Lin, Tianxin Wei, Dakuo Wang, Benoît Dumoulin, Wei Jin, and Hanqing Lu. Adaptive Auto-Harness: Sustained Self-Improvement for Agentic System Deployment on Open-Ended Task Streams. *ArXiv*, abs/2606.01770, 2026.

Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yi-Ming Hu, Tongwen Huang, and Xiangxiang Chu. SkillClaw: Let Skills Evolve Collectively with Agentic Evolver. *ArXiv*, abs/2604.08377, 2026.

H. B. McMahan, Eider Moore, Daniel Ramage, S. Hampson, and B. A. Y. Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *International Conference on Artificial Intelligence and Statistics*, pp. 1273–1282, 2017.

Niloofar Mireshghallah, Hyunwoo Kim, Xuhui Zhou, Yulia Tsvetkov, Maarten Sap, Reza Shokri, and Yejin Choi. Can LLMs Keep a Secret? Testing Privacy Implications of Language Models via Contextual Integrity Theory. *ArXiv*, abs/2310.17884, 2023.

Margaret Mitchell, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. Model Cards for Model Reporting. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*, pp. 220–229, 2019.

Amirkeivan Mohtashami, Florian Hartmann, Sian Gooding, Lukás Zilka, M. Sharifi, and B. A. Y. Arcas. Social Learning: Towards Collaborative Learning with Large Language Models. *ArXiv*, abs/2312.11441, 2023.

John Nguyen, Kshitiz Malik, Hongyuan Zhan, Ashkan Yousefpour, Michael G. Rabbat, Mani Malek, and Dzmitry Huba. Federated Learning with Buffered Asynchronous Aggregation. *ArXiv*, abs/2106.06639, 2021.

Jun Nie, Yong-Gang Zhang, Qianshu Cai, Yiu-ming Cheung, Xin-Mei Tian, and Bo Han. EvolveNet: Collaborative Harness Evolution for Agent Self-Improvement. *ArXiv*, abs/2608.04968, 2026a.

Jun Nie, Yong-Gang Zhang, Jun Song, Qianshu Cai, Da-Hai Yu, Yike Guo, Xin-Mei Tian, and Bo Han. TTHE: Test-Time Harness Evolution. *ArXiv*, abs/2607.08124, 2026b.

Alexander Novikov, Ngân Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, S. Shirobokov, B. Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. P. Kumar, Abigail See, Swarat Chaudhuri, George Holland, A. Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *ArXiv*, abs/2506.13131, 2025.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and O. Khattab. Optimizing Instructions and Demonstrations for Multi-Stage Language Model Programs. *ArXiv*, abs/2406.11695, 2024.

Linyue Pan, Yaoming Zhu, Lin Qiu, Xuezhi Cao, and Xunliang Cai. SAGE: A Quantitative Evaluation of Socialized Evolution in Agent Ecosystems. *ArXiv*, abs/2606.03544, 2026.

M. Robeyns, M. Szummer, and Laurence Aitchison. A Self-Improving Coding Agent. *ArXiv*, abs/2504.15228, 2025.

B. Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. P. Kumar, Emilien Dupont, Francisco J. R. Ruiz, J. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2023.

S. Schmidgall and Michael Moor. AgentRxiv: Towards Collaborative Autonomous Research. *ArXiv*, abs/2503.18102, 2025.

J. Schmidhuber. Goedel Machines: Self-Referential Universal Problem Solvers Making Provably Optimal Self-Improvements. *ArXiv*, cs.LO/0309048, 2003.

Nadine Schneider, N. Stiefl, and G. Landrum. What's What: The (Nearly) Definitive Guide to Reaction Role Assignment. *Journal of chemical information and modeling*, 56(12):2336–2346, 2016.

Lin-Fang Shang, Ming Xu, Yi-Ding Sun, Tian-Le Xia, Lingxiang Hu, Lan Xu, and Ning Zheng. When Self-Evolution Backfires: Pre-Commit Gating against Skill Contamination in LLM Agents. *ArXiv*, abs/2608.05810, 2026.

Noah Shinn, Federico Cassano, A. Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems 36*, 2023.

Virginia Smith, Chao-Kai Chiang, Maziar Sanjabi, and Ameet Talwalkar. Federated Multi-Task Learning. In *Neural Information Processing Systems*, pp. 4424–4434, 2017.

Jingwei Sun, Ziyue Xu, Hongxu Yin, Dong Yang, Daguang Xu, Yudong Liu, Zhi-Xu Du, Yiran Chen, and Holger R. Roth. FedBPT: Efficient Federated Black-box Prompt Tuning for Large Language Models. In *International Conference on Machine Learning*, pp. 47159–47173, 2024.

A. Tan, Han Yu, Lizhen Cui, and Qiang Yang. Towards Personalized Federated Learning. *IEEE Transactions on Neural Networks and Learning Systems*, 34:9587–9603, 2023.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, A. Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi (Jim) Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models. *ArXiv*, abs/2305.16291, 2023.

Zirui Wang, Zihang Dai, B. Póczos, and J. Carbonell. Characterizing and Avoiding Negative Transfer. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11285–11294, 2019.

D. Weininger. SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules. *J. Chem. Inf. Comput. Sci.*, 28:31–36, 1988.

Panlong Wu, Kangshuo Li, Junbao Nan, and Fangxin Wang. Federated In-Context LLM Agent Learning. *ArXiv*, abs/2412.08054, 2024.

Siwei Wu, Jin-Cheng Ren, Yi-Zhi Li, Haau-Sing Li, Cheng-Ran Yang, Yuxuan Zhang, Wei-Cheng Gu, Jian Yang, R. Batista-Navarro, Chuan-Yi Zhang, Xianglong Liu, Ming Zhou, Bryan Dai, and Cheng-Hua Lin. ModularRSI: Modular and Generalizable Recursive Harness Self-Improvement. *ArXiv*, abs/2609.14857, 2026.

Chaojun Xiao, Haoxi Zhong, Zhipeng Guo, Cunchao Tu, Zhiyuan Liu, Maosong Sun, Yansong Feng, Xianpei Han, Zhen Hu, Heng Wang, and Jianfeng Xu. CAIL2018: A Large-Scale Legal Dataset for Judgment Prediction. *ArXiv*, abs/1807.02478, 2018.

Cong Xie, O. Koyejo, and Indranil Gupta. Asynchronous Federated Optimization. *ArXiv*, abs/1903.03934, 2019.

Prateek Yadav, Derek Tam, Leshem Choshen, Colin Raffel, and Mohit Bansal. TIES-Merging: Resolving Interference When Merging Models. *Advances in Neural Information Processing Systems 36*, 2023.

An Yang, An-Feng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bo-Wen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jian-Hong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, K. Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Ming-Ze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tian-Yi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Ying-Er Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhen-Ru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 Technical Report. *ArXiv*, abs/2505.09388, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large Language Models as Optimizers. *ArXiv*, abs/2309.03409, 2023.

Jing-Bo Yang, Guanyu Yao, Yang Zhang, R. Kompella, Gaowen Liu, and Shiyu Chang. Federated-Skill: Federated Learning for Agentic Skill Evolution. *ArXiv*, abs/2606.03143, 2026.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Adriano Lieret, Shun-Yu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. *ArXiv*, abs/2405.15793, 2024a.

Yiyuan Yang, Guodong Long, Tao Shen, Jing Jiang, and Michael Blumenstein. Dual-Personalizing Adapter for Federated Foundation Models. *ArXiv*, abs/2403.19211, 2024b.

Dixi Yao, Tahseen Rabbani, M. Zaheer, and Tian Li. Federation over Text: Insight Sharing for Multi-Agent Reasoning. *ArXiv*, abs/2604.16778, 2026.

Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. Meta Context Engineering via Agentic Skill Evolution. *ArXiv*, abs/2601.21557, 2026.

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and W. Wang. Gödel Agent: A Self-Referential Agent Framework for Recursively Self-Improvement. In *Annual Meeting of the Association for Computational Linguistics*, pp. 27890–27913, 2025.

Tao Yu, Eugene Bagdasarian, and Vitaly Shmatikov. Salvaging Federated Learning by Local Adaptation. *ArXiv*, abs/2002.04758, 2020.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. TextGrad: Automatic "Differentiation" via Text. *ArXiv*, abs/2406.07496, 2024.

E. Zelikman, Eliana Lorch, L. Mackey, and A. Kalai. Self-Taught Optimizer (STOP): Recursively Self-Improving Code Generation. *ArXiv*, abs/2310.02304, 2023.

Jenny Zhang, Shengran Hu, Cong Lu, R. Lange, and Jeff Clune. Darwin Godel Machine: Open-Ended Evolution of Self-Improving Agents. *ArXiv*, abs/2505.22954, 2025a.

Jenny Zhang, Bingchen Zhao, Wannan Yang, J. Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents. *ArXiv*, abs/2603.19461, 2026a.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Feng-Wei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating Agentic Workflow Generation. *ArXiv*, abs/2410.10762, 2024.

Luan Zhang, Ruochen Zhou, Dan-Dan Song, Zhengyu Chen, Yu-Hang Tian, Jun Yang, Hui-Peng Ma, Chen-Hao Li, Guangyuan Feng, Xudong Li, Yizhou Jin, and Yan Xu. HarnessCompass: Guiding Automatic Harness Evolution toward Generalizable and Effective Agent Harnesses. *ArXiv*, abs/2608.01918, 2026b.

Michael Zhang, Karan Sapra, S. Fidler, Serena Yeung, and J. Álvarez. Personalized Federated Learning with First Order Model Optimization. *ArXiv*, abs/2012.08565, 2020.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, V. Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Han-Chen Li, Urmish Thakker, James Zou, and K. Olukotun. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. *ArXiv*, abs/2510.04618, 2025b.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Y. Liu, and Gao Huang. ExpeL: LLM Agents Are Experiential Learners. In *AAAI Conference on Artificial Intelligence*, pp. 19632–19642, 2024a.

Wenting Zhao, Nan Jiang, C. Lee, Justin T. Chiu, C. Cardie, Matthias Gallé, and Alexander M. Rush. Commit0: Library Generation from Scratch. *ArXiv*, abs/2412.01769, 2024b.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Da-Cheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. *ArXiv*, abs/2306.05685, 2023.

Guo-Jun Zhu, Xunheng Huang, Peng Yin, Jia-Hui Xie, Sanguo Zhang, and Dou-Dou Zhou. Bad Genius: Counterfactual-Guided Harness Evolution Beyond Task-Specific Shortcuts. *ArXiv*, abs/2609.18366, 2026.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, T. Hoàng, A. Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro von Werra. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. *ArXiv*, abs/2406.15877, 2024.

A Extended Related Work

FedRSI builds on four lines of work: methods by which a single agent improves itself, federations through which agents share knowledge or programs, asynchronous aggregation in federated learning, and personalized federated learning, from which it takes the principle that each client decides what it adopts.

Local self-improvement. Local self-improvement now extends to the entire program surrounding the model, yet every such method improves only a single deployment. These methods differ mainly in the object that they modify (Gao et al., 2025). Some optimize instructions (Yang et al., 2023; Fernando et al., 2024; Opsahl-Ong et al., 2024; Agrawal et al., 2025), the components of language-model pipelines (Khattab et al., 2023; Yuksekgonul et al., 2024), or agent workflows (Hu et al., 2024; Zhang et al., 2024); others accumulate reusable experience as text or skills (Shinn et al., 2023; Wang et al., 2023; Zhao et al., 2024a; Zhang et al., 2025b; Ye et al., 2026); and a third line of work makes executable programs the object of search. FunSearch and AlphaEvolve evolve programs under automated evaluation (Romera-Paredes et al., 2023; Novikov et al., 2025), SICA, DGM, and Meta-Harness evolve agent code and harnesses (Robeyns et al., 2025; Zhang et al., 2025a; Lee et al., 2026b), and STOP, Gödel Agent, and Hyperagents additionally make the procedure that generates the modifications editable (Zelikman et al., 2023; Yin et al., 2025; Zhang et al., 2026a). Within harness evolution, recent work already constrains what a single optimizer may accept, through applicability constraints and performance checks (Zhang et al., 2026b; Jiang et al., 2026), counterfactual tests against benchmark-specific shortcuts (Zhu et al., 2026), the separate evolution of functional modules (Wu et al., 2026), or specialization and routing (Chen et al., 2026b; Liu et al., 2026). We keep such a local optimizer fixed and ask how its improvements can transfer across deployments. This question introduces a requirement that no single-deployment method faces: shared code must be rebuilt without access to the private program from which its mechanism originated.

Federated knowledge and programs. In contrast, agent federations transfer what clients learn between deployments, but almost all of them share only one kind of harness component. Classical federated learning aggregates model parameters (McMahan et al., 2017; Kairouz et al., 2021), and Fed-SE applies it to agents that evolve by fine-tuning adapters (Chen et al., 2025b). When client models are heterogeneous or accessible only as black boxes, clients exchange predictions or prompt variables instead (Li & Wang, 2019; Sun et al., 2024). Agent federations extend this exchange to knowledge expressed in natural language: instructions that teach a student model (Mohtashami et al., 2023), the prompts that FedTextGrad refines by textual gradients (Chen et al., 2025a), the knowledge compendiums of FICAL (Wu et al., 2024), research reports shared among agent laboratories (Schmidgall & Moor, 2025), the distilled reasoning guidance and knowledge units of FoT and FedAgentKE (Yao et al., 2026; Li et al., 2026), and the schema-typed compendiums of tool-routing knowledge of Synapse (Chakraborty et al., 2026). Each of these methods federates a single kind of component, and all of them deliver it to recipients as context rather than as executable control flow. FedWorld and FederatedSkill approach program-level sharing, since they share scoped world-model rules and skill files with executable helpers (Hou, 2026; Yang et al., 2026). Nevertheless, they also federate a single kind of component. The federation of recursive self-improvement has also been proposed as a conceptual architecture in which independent self-improving systems exchange only scalar progress scores (Lemon, 2026). However, such scores do not contain the improvements themselves, which are what FedRSI transfers. Among prior federations, only EvolveNet operates on the entire harness (Nie et al., 2026a), and its server reads client source code and uses labelled validation data. By contrast, in FedRSI only prose reaches the server, and re-implemented modules return to the clients; the server therefore sees neither the original implementation nor client data, and every decision that requires private information is made at the clients.

Asynchronous aggregation. Moreover, every agent federation that aggregates at a server does so once or in synchronous rounds. EvolveNet aggregates program differences that all clients compute against the same version of the shared harness, so that every round waits for its slowest client, a constraint that its authors acknowledge (Nie et al., 2026a). Fed-SE likewise identifies the straggler effect of its synchronous averaging as a limitation (Chen et al., 2025b), and FoT, FederatedSkill, and FedAgentKE aggregate once per round over all clients (Yao et al., 2026; Yang et al., 2026; Li et al., 2026). By contrast, federated learning decouples clients through asynchronous and buffered

asynchronous aggregation of parameters (Xie et al., 2019; Nguyen et al., 2021), a development that agent federations have not yet followed. Mechanism cards make asynchronous aggregation straightforward, because a card refers to no particular version of the shared harness and is read against whichever version is current when the server acts.

Personalization and local adoption. Finally, personalized federated learning established that each client should decide what it adopts, and FedRSI extends this principle from models to programs. The tension between specialization and transfer is well documented in federated learning (Smith et al., 2017; Tan et al., 2023), where personalized methods regularize a local model toward the global one (Li et al., 2021), keep local heads on a shared representation (Collins et al., 2021), mix a local with a global model (Deng et al., 2020), or train a generic and a personalized predictor in parallel (Chen & Chao, 2021). Closer to the client-side decisions of FedRSI, FedFomo lets each client evaluate received models on local validation data and weight them accordingly (Zhang et al., 2020), FedDPA combines a global and a local adapter for each test input according to its similarity to the local data of the client (Yang et al., 2024b), and FedPGP observes that strong local adaptation of prompts erodes general behavior (Cui et al., 2024). Our local validation follows the first of these approaches, and our task-aware priority addresses the same tension as the other two. The finding that a defective skill contaminates everything that an agent distills after admitting it (Shang et al., 2026) further supports testing an improvement before adoption rather than reverting it afterwards. However, the object of adoption differs. Parameters can be interpolated (Ilharco et al., 2022; Deng et al., 2020), although independently trained updates already interfere (Yadav et al., 2023). In contrast, executable modules modify one another's effects, as in the feature-interaction problem of software engineering (Calder et al., 2003), so that a single retained private module can displace a shared one entirely. Therefore, the composition of programs rather than the weighting of models determines transfer.

B Problem Setting

This section defines the clients and their harnesses, the information available to the server, and the objective of federation.

Clients. We consider N clients that serve private tasks with a frozen base model and that differ both in their tasks and in their pace. As in federated multi-task learning (Smith et al., 2017), client k receives items drawn from a task distribution $\mathcal{D}_k$ and accumulates a labelled history H_k, because the label of a served item arrives as feedback after the item has been answered. Clients are heterogeneous in the two respects that federated learning terms statistical and systems heterogeneity (Li et al., 2020): their tasks differ, and so does the time required by a single item, which ranges from seconds for a classification item to an hour for an agentic coding item. We therefore assume no synchronization among clients; each client serves, searches, and communicates at its own pace.

Harnesses. Every client improves its harness, and federation divides each harness into a part that all clients share and a part that remains private. The *harness* is an ordered collection of executable *modules*, each of which provides or modifies an operation around the model, such as storing and retrieving experience, constructing the context, calling the model or a tool, or extracting the answer. Local search adds one module at a time, so the module is also the unit that the federation describes and rebuilds, whereas a client always adopts a complete composition. The server maintains a *shared harness* g, initialized from a common seed harness, and broadcasts every new version of g to all clients. Client k retains the *private modules* p_k that its own local search has found. Its *serving harness* h_k composes p_k with the shared modules that the client has adopted and answers the items of the client. This division mirrors personalized federated learning, in which clients share a representation and keep local heads (Collins et al., 2021). In contrast to that setting, adoption is a local decision, so that two clients can serve different compositions of the same version of g.

Information constraint. The server never receives client source code, client data, or labels. It receives only prose descriptions of improvements, namely the mechanism cards of Section 3.1. The server attributes each card to the client that uploaded it through a client identifier, which differs from the opaque identifier of the module that the card describes; in our experiments, the client identifier names the task of the client. When the server constructs a new version of g, it may inspect the shared code and the runtime interface, and the only item on which it ever executes code is synthetic.

Therefore, every step that requires private information takes place at the client: local search, the resolution of overlaps between shared and private modules, and validation. Because prose can still carry private information (Mireshghallah et al., 2023), as model updates do in federated learning (Kairouz et al., 2021), every card is screened before it leaves the client (Section 3.1).

Objective. Federation seeks improvements that benefit both the client that finds them and every other client, so each harness must be evaluated on the task of every client. Formally, let r_j score an answer on task j. Given the history H_j of task j, the utility of a harness h on task j is

$$U_j(h \mid H_j) = \mathbb{E}_{(x,y)\sim\mathcal{D}_j}\big[r_j\big(h(x; H_j), y\big)\big]. \tag{3}$$

Local search at client k estimates the gain $U_k(h' \mid H_k) - U_k(h \mid H_k)$ on private validation items and keeps a proposal only if this estimate is positive. Federation additionally requires that the kept improvement benefit other clients without reducing the local utility of the client that found it, and these two goals can conflict, because a positive gain on task k places no constraint on the sign of the gain on a task $j \neq k$. We therefore evaluate every harness in two ways. Evaluating h_k on the task of client k itself constitutes *in-domain* (ID) evaluation. Because a deployed harness may also have to serve items outside the task on which it evolved, we additionally evaluate the harness of client k on the tasks of the other clients, which we call *out-of-domain* (OOD) evaluation. In this case, client k itself serves the items of task j, so that its private modules never leave it. Before answering, its harness replays the labelled history H_j of task j through its own memory modules; the conditioning on H_j in Equation (3) expresses this replay. This history stands for the feedback that a deployment on task j accumulates; the evaluation supplies it, and the protocol never transmits it between clients. We call the task on which a harness is used its *deployment context* s, and h_k^s denotes the harness that client k deploys in context s, with $h_k^k = h_k$ (Section 3.4). At deployment, the language model of client k judges from the past runs of the client whether an item belongs to its own task. The two evaluations correspond to the personalized and the generic accuracy of federated learning (Chen & Chao, 2021).

C Implementation Details

This appendix specifies the components of Section 3 at the level of detail required for reproduction. Algorithm 1 summarizes the protocol as two concurrent processes: every client proceeds at its own pace, and the server opens a federation round whenever B new cards have arrived. Figure 3 presents the four stages in detail, and Table 3 lists the notation.

Table 3: **Notation.**

Symbol	Meaning
N	number of clients
$\mathcal{D}_k$, H_k, V_k	task distribution, labelled history, and private validation items of client k
g	shared harness, maintained by the server and broadcast to all clients
g'	new version of g
p_k	private modules that the local search of client k has kept
h_k, $\widetilde{h}_k$	serving harness of client k, and the candidate composition that it validates
h_k^{old}	incumbent serving harness in an adoption test
h_k^s	harness that client k deploys in deployment context s, with $h_k^k = h_k$
$\mathcal{C}_{\mathrm{local}}$, $\mathcal{C}_{\mathrm{global}}$	local-first and global-first compositions of g with p_k
U_k, $\widehat{U}_k$	utility on task k, and its estimate on V_k
r_j	score of an answer on task j
s	deployment context, that is, the task on which a harness is used
B	number of new cards that opens a federation round
K	maximum number of work orders per round
m	maximum number of cards that a client uploads between two consecutive rounds
W	number of rounds for which an undecided card stays in the backlog

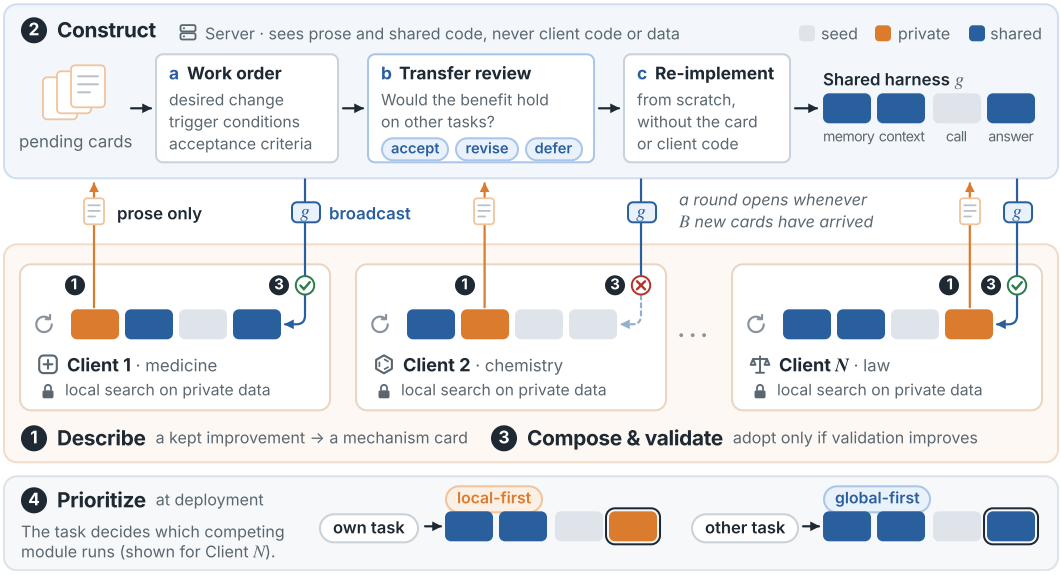


Figure 3: **The four stages of FedRSI in detail.** **(1)** A client describes in a *mechanism card* each improvement that its local search keeps; only this prose leaves the client. **(2)** Whenever B new cards have arrived, the server converts the pending cards into work orders, reviews them for task-specific assumptions, re-implements them from scratch within the shared harness g, and broadcasts the new version of g. **(3)** Each client composes g with its private modules and adopts the result for its own task only if its validation accuracy improves, so that clients can serve different compositions of the same version of g. **(4)** At deployment, a task-aware priority determines which of two competing modules executes: the private module on the client's own task and the shared module on all other tasks.

Algorithm 1 FedRSI with buffered asynchronous aggregation

Require: shared harness g; buffer size B; for every client k: private modules p_k, serving harness h_k, validation set V_k

Client k, at its own pace and without waiting for any other client:

loop
  serve the next items with h_k; their labels arrive as feedback
  if a search checkpoint is due **then**
    if the server has broadcast a new version g' of the shared harness since the last checkpoint **then**
      $\widetilde{h}_k \leftarrow \mathcal{C}_{\text{local}}(g', p_k)$ ▷ identifiers, deterministic checks, merger; Section 3.3
      $h_k \leftarrow \widetilde{h}_k$ **if** $\widehat{U}_k(\widetilde{h}_k) > \widehat{U}_k(h_k)$ **else** keep h_k ▷ adoption test (local validation)
    end if
    propose modules; add the best to p_k and h_k only if accuracy on V_k improves ▷ local search
    write a mechanism card for the added module; screen it; upload it within the quota ▷ Section 3.1
  end if
end loop

Server, whenever B cards have arrived that no earlier round has processed, and once more after all clients have served their last items:

classify the pending cards; select at most K mechanisms; write one work order each ▷ Section 3.2
transfer review → accept, revise, or defer every work order
extend g to g' with the accepted and revised work orders, implemented from scratch; load and dry-run; repair or defer
$g \leftarrow g'$; broadcast g'

Deployment: serve h_k on the client's own task and $\mathcal{C}_{\text{global}}(g, p_k)$ on another task ▷ Section 3.4

Harness substrate. A harness is an ordered list of modules. Each module is a plugin of a small plugin runtime, which we call the kernel and which a host process executes. A module can provide a named service, register a tool that the model may call, and listen to the events that other modules emit. The listeners of an event are ordered by the module list, and the listener of a later module encloses

those of all earlier modules, so that a module added by local search wraps every earlier module and can therefore alter an existing behavior without editing it. For an event whose listeners form a chain, a listener that returns without passing control to the next listener overrides the behavior beneath it. Two modules that provide the same service or register the same tool collide. The kernel reports every collision and every error raised by a module, and the host records each of them. The runtime interface gives a module no network access: a module reaches the model and the tools only through the host process, which issues and counts all model calls. The shared harness g consists of the seed harness followed by the shared modules that the rounds have added, and a composition places the shared modules that it keeps before the private modules of the client.

Local search. At each search checkpoint, the client runs one proposer session per proposal, sequentially. The proposer is a coding agent in a dedicated container. It can read four kinds of material: the served records of the client, which give the input, label, answer, score, model calls, and event log of every served item; the failures recorded for these items; every earlier proposal with its outcome; and the source code of the current composition. It writes one module together with a one-line hypothesis. The module may alter the behavior of an existing module by intercepting the events of that module or by providing a new service or tool, but it never edits or replaces a module. No validation item, validation trace, or per-item validation score is visible to it. Each proposal is appended to the serving harness, loaded, dry-run on the current memory of the client, and scored together with the incumbent on the validation items. The best proposal is kept if its mean score is strictly higher, and it then serves immediately under an opaque identifier. A proposal that does not load and a session that produces no module are recorded and not scored.

Cards and screening. A card plays the role of a model card (Mitchell et al., 2019) for a mechanism, with the difference that it replaces the artifact rather than accompanying it. It has four fields: what the mechanism does, why it helps, when it engages, and, optionally, the seed module to which it belongs. A separate model call writes the card from the hypothesis, the source code of the module, and the validation evidence; it is instructed to use plain prose and to omit code, identifiers, literals, example content, task specifics, and any claim about where the mechanism applies. Before upload, a literal scan rejects cards that contain quoted strings, numbers, URLs, e-mail addresses, file paths, or code identifiers, as well as cards longer than four thousand characters, and a canary check rejects cards that reproduce the canary strings planted in the inputs of the client. Each client may upload at most m cards between two consecutive rounds, and we require $m \geq B$, so that a single client can open a round. A card that is rejected or exceeds the quota is not uploaded, and the corresponding module remains in p_k.

Work orders, review, and implementation. The three server stages see progressively less client information. The work-order call sees every pending card together with the identifier of the client that uploaded it, which in our runs is the name of its task; the reviewer sees the cards without client identifiers; and the implementer sees the work orders but not the cards. The work-order call classifies each card against the shared code as implemented, partially implemented, absent, or contradicting the current behavior; a card that g already implements is reported to its client as a notice, never as an instruction to remove anything. The call then selects at most K mechanisms, preferring those reported by several clients, and writes one work order per mechanism, which specifies the current behavior, the desired change, the trigger conditions, the constraints, and the acceptance criteria. A work order may merge cards that describe the same function, but it never implements a part of a card or a recombination of parts of different cards. The call is instructed to describe each change in terms of the behavior of the code rather than to reproduce card text, and to delimit mechanisms along the functional boundaries of the cards themselves. A card that no round has implemented or found already implemented remains undecided and stays in a backlog, as do the cards of a deferred work order; a card that remains undecided for more than W rounds after its arrival is dropped, and its module remains private. A revision by the reviewer must provide a complete replacement work order whose conditions cover every effect of the mechanism, including its call budget and recovery behavior. If the reply of the reviewer is malformed or omits a work order, the whole batch is deferred with an explicit record rather than partially processed. The implementer is a coding-agent session in a dedicated container. It receives the modules of the shared harness, the source code of the kernel, the runtime interface, and the work orders of the round. Each work order names the identifier that its module must take, namely the identifier of the module that the first card of the work order describes;

at the client of any other card that the work order merges, the module is composed like any other shared module. A work order whose identifier is already used in g or by another mechanism of the round is deferred, so that no card is implemented by more than one mechanism. Every new module is loaded and dry-run alone on g and then together with the other new modules of the round. The dry run answers a synthetic labelled item, stores its feedback, and answers again, so that both the write path and the read path of new memory modules are exercised. Guided by the failure message (Chen et al., 2023), at most five repair calls may rewrite the new modules; if they still fail, all new modules of the round are deferred.

Local-first composition. The local-first composition resolves overlaps in a fixed order. A shared module whose identifier equals that of an active private module is omitted, because it implements the client's own card and the original module serves instead. The remaining shared modules are first loaded alone: of two shared modules that collide, the later one is excluded, and a shared module that fails to load is excluded. The client then loads the kept shared modules followed by p_k as one composition and dry-runs it with the tools that the environment of the client offers, so that every tool registration that depends on the offered tools takes place exactly as it would during serving. A collision between a shared and a private module excludes the shared module, and a collision between two private modules disables the earlier one. A module that fails to load is excluded if it is shared and disabled if it is private. A failure during the dry run is attributed to the module that the kernel identifies; if the kernel identifies no module, the failure is attributed to the newest shared module of the current version of g. This procedure repeats until the composition is stable. The merger then judges each remaining pair of a module of g and a private module that act on the same pipeline stage or provide the same service, and it returns one of three verdicts. The verdict `additive` states that the two modules serve different functions, so that both are kept; this is the complementary case of Section 3.3. The verdict `regression` states that the shared module implements the same function as the private one with equal or lesser capability, and the verdict `conflict` states that executing both would alter the intended behavior of either; these two verdicts constitute the competing case. The merger must return exactly one of the three verdicts, so that every overlapping pair receives a defined treatment. Neither verdict of the competing case determines which of the two modules executes, because an inspection of code must never decide it. The priority alone decides: on the client's own task in favor of the private module, which local search has kept on measured accuracy, and on all other tasks in favor of the shared module, which the server has reviewed for transfer. A truncated or malformed reply excludes the shared module from the current composition only and is recorded; the pair is judged again when the client next composes, and only valid verdicts are stored, so that an invalid reply neither removes a private module nor affects a global-first composition. Verdicts are cached by the source code and configuration of both modules. The structural checks are repeated after every exclusion, because an exclusion can deprive another module of a service that it requires or re-enable a module that the excluded one had displaced. If no stable composition can be obtained, the previous composition continues to serve.

Local validation. The candidate and the incumbent are scored on the same validation items after replaying the same labelled history, each through its own memory modules. The validation items are disjoint from this history; each of them is answered from a fresh copy of the harness state after the replay, and its label serves for scoring only. Validation labels therefore never reach a memory module, and the proposer sees validation results only as mean scores. Adoption requires a strictly higher mean score. Because the adoption test executes the complete composition on real items, it also detects failures that neither the pairwise judgment nor the dry run can reveal, such as a shared module that fails only on a code path that real items reach, or an interaction between a module that writes to memory and another that reads from it at a later stage. A tie or a composition that does not load completely retains the incumbent, and a failed evaluation retains it until the adoption test is repeated at the next checkpoint. A client runs the adoption test on the latest version of g at its next search checkpoint, and it tests the last version of g in a final pass without search. A version that a newer one supersedes before that checkpoint is never tested. A candidate that behaves identically to the incumbent requires no evaluation, as when every new shared module was omitted. The score of the adopted harness serves as the baseline of the subsequent local search, and local search always operates on the shared modules that the client has adopted. A shared module that was excluded or rejected cannot re-enter the serving harness without passing a later adoption test.

Global-first composition. The global-first composition $\mathcal{C}_{\text{global}}(g, p_k)$ is built from the final version of g and the final private modules rather than by deleting modules from h_k, because h_k may already lack the shared alternatives. Local validation decides what the client serves on its own task and implies nothing about other tasks, on which every shared module has instead passed the transfer review of the server. The local-first branch of Equation (2) is therefore the serving harness h_k, the local-first composition that the client adopted last, whereas the global-first branch uses the final version of g. The global-first composition keeps every module of g that the local-first composition does not exclude for a collision with another shared module or a failure to load, and it reverses each rule of the local-first composition: the server's implementation of the client's own card replaces the original, a collision excludes the private module, and every competing pair excludes the private module instead of the shared one, while complementary pairs keep both modules. For given g and p_k, the priority thus changes only which module is kept in a competing pair, in a collision, or in a pair of modules with the same identifier. The global-first composition reuses the verdicts recorded for the current module sources and requests no new verdict. Because every client evolves on its own task, h_k^s coincides with h_k throughout evolution, and the two branches differ only when a harness is deployed on another task; our experiments therefore evaluate both branches on the same evolved clients. By construction, the stored verdicts cover every pair that the global-first composition requires. At every search checkpoint, the merger additionally judges every module of the current version of g against each private module that it overlaps and stores these verdicts, including those for the shared modules that the local-first composition had already excluded because of a collision or an identifier match and therefore never paired with the other modules of the client. Because the global-first composition restores those shared modules, it finds every verdict that it requires. If a verdict were missing or the composition failed its dry run, the construction would stop and record the failure instead of guessing a verdict.

D Experimental Details

D.1 Complete Setup

This subsection gives the complete setup that Section 4.1 summarizes.

Clients and tasks. We evaluate FedRSI on two federations of three clients each, one of text tasks and one of coding tasks. The text federation consists of three clients, each of which serves one task of the online text-classification setting of Meta-Harness (Lee et al., 2026b); the three tasks differ in domain, language, and output space. *Symptom* selects one of 22 diagnoses listed in the prompt from the description of a patient, *Crime* names the charges of a Chinese criminal case as free text (Fei et al., 2024), and *USPTO* proposes the reactants of a reaction as SMILES strings (Schneider et al., 2016). Each client serves a stream of test items in a fixed random order and receives the label of each item after answering it, and we plant a canary string in every fifth served input. To show that the protocol extends to agentic coding, we build a coding federation of three clients from BigCodeBench (Zhuo et al., 2024), in which each item asks for a Python function and is judged by its own unit tests. The items that use a plotting library form the *Visualization* client (344 items). Of the remaining items, those that use network, cryptography, or time libraries are excluded, those that use a numerical, data, or learning library form the *Computation* client (361), and the remaining standard-library items form the *System* client (225).

Compared configurations. We compare FedRSI with *independent evolution*, which runs the local search at every client without any exchange. *FedRSI* runs the protocol of Section 3 with transfer review and local validation. Both configurations use the same local search and start from the same seed harness (Appendix D.3), so the sharing protocol is the only designed difference between them. Appendix F.2 compares FedRSI with centralized evolution.

Schedule and models. All configurations follow the same schedule and use the same models. Every client serves 24 items and searches after every 6 items with two proposals, each of which is scored on 48 private validation items (30 for USPTO). A client serves the items of one search interval concurrently on the harness state before the interval and applies their labelled feedback afterwards in input order. The server opens a round when $B = 2$ new cards have arrived, with $K = 2$, $m = 2$, and

$W = 3$. The base model is Qwen3.8-Flash[4] in every configuration; the proposer and the implementer are OpenCode sessions with GLM-5.3-Flash,[5] which also writes the cards and serves as reviewer and merger. We run the text federation three times with identical settings, and the runs differ only in model sampling. The main text reports the run in which FedRSI answered its served items most accurately during evolution, and Appendix F reports all three runs. The coding federation and each of the two ablations that change the protocol during evolution run once. Table 5 lists all settings.

Ablations. Two additional FedRSI runs each remove one decision of the protocol: local validation, so that a client adopts every composition that loads, and transfer review, so that every work order is implemented as written. Since task-aware priority acts only at deployment, we ablate it on the same evolved clients by serving the other tasks with the local-first instead of the global-first composition.

Evaluation. After evolution, we evaluate the final harness of every client on the held-out items of every client, namely the stream items that no configuration served. The text federation has 187 Symptom, 76 Crime, and 76 USPTO held-out items, and the coding federation has 239 Computation, 230 Visualization, and 113 System held-out items after the exclusion of the items whose reference solution fails its own tests. A FedRSI client serves its own task with its serving harness and the other tasks with its global-first composition, and we report macro averages over the three ID pairs and the six OOD pairs.

D.2 Tasks and Splits

Table 4 summarizes the six clients. The text tasks use the data files of Meta-Harness (Lee et al., 2026b), which in turn uses the subsampled splits released with MCE (Ye et al., 2026). Symptom is a subset of the Gretel AI `symptom_to_diagnosis` dataset with 22 diagnoses. Crime is a subset of the charge-prediction task of LawBench (Fei et al., 2024), whose cases originate from CAIL2018 (Xiao et al., 2018). USPTO is a subset of the USPTO-50K reactions (Schneider et al., 2016), balanced over the ten reaction classes and posed as retrosynthesis with the reaction class given in the input (Liu et al., 2017); products and reactants are written as SMILES strings (Weininger, 1988). The stream of a text client is its test split in a fixed random order; the first 24 items are served, and the remaining items are held out with their original inputs, which carry no canary string; held-out items that duplicate an item of the training or validation split are removed, which removes one Symptom item. The validation items of a text client come from its validation split, deduplicated against the training and test splits. Its training split is never replayed; it is available only through the search over labelled examples that the environment offers, which also covers the training splits of the other two text clients.

The coding clients use version 0.1.4 of BigCodeBench (Zhuo et al., 2024). The input of an item is its natural-language instruction, followed by the request to return the complete program, including its imports and the required function, as the final answer. In a fixed random order, the items of each client provide 48 validation items, then 40 items for the search over solved examples, which all three coding clients share, and then the stream; the first 24 items of the stream are served, and its remaining items are held out. The program of an answer is its longest fenced code block, or the whole answer if the answer contains no such block. The unit tests of the item run against this program in a fresh container of the evaluation image of the benchmark, without network access and with a limit of 60 seconds; the answer scores 1 if all tests pass and 0 otherwise, including on a timeout. Under this procedure, 20 of the 930 client items fail even with the reference solution of the benchmark, 2 because of the time limit and 18 because of their tests. Twelve of these items fall into held-out sets and are excluded from all held-out tables by this item-level rule, which was fixed before any held-out result was seen, and two fall into validation sets, where they score 0 for every configuration; the remaining six are three solved examples and three served items.

D.3 Seed Harness and Tools

Every configuration starts from the same seed harness: a general agent loop of at most 30 steps around the model, with a shell in the container of the task, answer extraction, and a time limit of 1500 seconds

[4] As served by OpenRouter, https://openrouter.ai/qwen/qwen3.8-flash.
[5] GLM-5.3-Flash, https://huggingface.co/zai-org/GLM-5.3-Flash.

Table 4: **Clients of the two federations.** Examples: labelled training examples (text) or solved examples (coding) that the search tools of the environment return; stream: items in serving order, of which the first 24 are served; held-out: the unserved stream items used for evaluation, after deduplication (text) and after the exclusion of items whose reference solution fails its own tests (coding).

Client	Task	Answer	Examples	Stream	Validation	Held-out
Symptom	diagnosis, English	one of 22 listed labels	200	212	48	187
Crime	charges, Chinese	charge names, free text	200	100	48	76
USPTO	retrosynthesis	reactant SMILES	50	100	30	76
Computation	numerical libraries	Python program	40	273	48	239
Visualization	plotting libraries	Python program	40	256	48	230
System	standard library	Python program	40	137	48	113

per item. The memory module of the seed harness is disabled, and the seed harness does not register the search over labelled examples that the environment offers; storing and retrieving experience and consulting labelled examples are therefore capabilities that evolution must find. The environment of every task offers the same tools: the shell, a search over solved competition mathematics problems from NuminaMath-1.5,[6] and a search over the public labelled training examples of the three text tasks; the environment of a coding task also offers a search over 40 solved examples from each coding client.

D.4 Settings of All Configurations

Table 5 lists the settings of all configurations. The centralized configuration applies them to a single client on the union of the three streams of the text federation: it serves the same items, searches after every 18 items with two proposals, and scores a proposal by the mean of the three task means on the union of the validation sets. It serves its items in batches of six items per task, so that the delay of its feedback matches that of the other configurations.

Table 5: **Settings of all configurations.**

Setting	Value
Base model	Qwen3.8-Flash, maximum reasoning effort, the provider's default sampling
Agent loop	at most 30 steps; 1500 s per item; 120 s per shell command
Task containers	text: a plain Python image; coding: the benchmark's evaluation image; no network, one CPU, 2 GB of memory
Proposer and implementer	OpenCode with GLM-5.3-Flash; 40 minutes per proposer session; for the implementer session of a round, 40 minutes per work order
Cards, work orders, review, merger	GLM-5.3-Flash, thinking enabled, temperature 1.0, top-p 0.95
Served items per client	24
Local search	after every 6 served items; two proposals; a proposal is kept only on a strictly higher mean
Validation items per client	48 (USPTO: 30)
Federation	$B = 2$, $K = 2$, $m = 2$, $W = 3$; at most five repair calls per round
Runs	text federation: three per configuration (the main text reports one, Appendix F all three); the two ablated runs and the coding federation: one

Each proposer and implementer session ends at its time limit at the latest, and the output that it has produced by then is used.

[6] https://huggingface.co/datasets/AI-MO/NuminaMath-1.5.

D.5 Evaluation Protocol

The final harnesses are the compositions that exist when every client has served its last item and the server has processed the last cards. To evaluate a harness on a task, we first replay the 24 served items of the task and their labels through the memory modules of the harness, which yields the contents of its memory store; this history is the feedback of a deployment on the task and never passes between clients, and we then answer every held-out item on a freshly loaded copy of these contents without updating the store. Because each harness determines what it stores, the same history can produce different memory stores in different harnesses. An independently evolved client serves every task with its final harness; a FedRSI client serves its own task with its serving harness and the other tasks with its global-first composition; and the same seed harness serves every task. When the local-first and the global-first compositions of a client coincide on a task, we evaluate them once and use the result for both. The text scorer converts an answer to lowercase, collapses its whitespace, and compares it exactly with the label; it neither canonicalizes chemical structures nor reorders the components of a multi-part answer. Macro accuracy over a set P of client–task pairs is

$$\mathrm{Macro}(P) = \frac{1}{|P|} \sum_{(k,j)\in P} \frac{c_{k,j}}{n_j}, \tag{4}$$

where $c_{k,j}$ is the number of correct answers of the harness of client k on task j and n_j is the number of held-out items of j; P contains the three ID pairs or the six OOD pairs, and averages are computed from unrounded fractions. In the main text, the item-level statistics are computed on the reported run; in Table 12, the score of each item is averaged over the three runs, so that an item scores 0, one third, two thirds, or 1. The sign test counts an item as won or lost when the summed score of the method over the client–task pairs of the item is higher or lower than that of the baseline, and the paired bootstrap resamples items within each task 10,000 times with a fixed seed, applying each draw to every client–task pair and to both configurations.

D.6 Model Usage

Table 6 lists the model calls, tokens, and sessions of every configuration, and Table 15 in Appendix F the proposals that local search made and kept in every FedRSI run.

E Example Artifacts and Prompts

A transferred mechanism. The artifacts of the case study of Section 4.4 are reproduced below unedited, with a long dash typeset as two hyphens: the card that the Symptom client uploaded, the work order that the server wrote from it, the transfer review of the work order, and the header comment and the tool-registration function of the module that the implementer wrote. In these artifacts, a plugin or an entry is a module, a seam is a pipeline stage, the shared core is the shared harness g, a tenant is a client, and the target module is the seed module to which the mechanism belongs. The Crime client tested the version of g that contains this module at its next search checkpoint. On its 48 validation items, the incumbent answered 31 correctly and the candidate 32, with 2 items won and 1 lost, so the client adopted the candidate.

Card uploaded by the Symptom client

```
What: Expose the environment's labelled-example corpora to the model as searchable tools. When the
      environment advertises such a corpus and no mounted entry already gives tool access to it, register
      a tool for it: the model supplies a free-form query, optionally an example count, and receives the
      most similar solved examples together with the labels or solutions they carried. Oversized replies
      are trimmed at whole-record boundaries under a fixed character budget, and replies that arrive
      wrapped or double-encoded are unwrapped before trimming, so records stay intact and readable.
Why: The context seam serves a small fixed set of solved examples passively, chosen before the model
      reasons, leaving the model itself unable to consult the labels the corpus carries. Where those
      labels conflict with the model's priors, it answers from its priors, and this convention conflict
      is exactly the residual failure mode the accepted composition leaves behind. A tool supplies the
      agency the passive examples cannot: the model can query the corpus in its own phrasings, rephrase
      when matches are poor, reach beyond the examples served at the seam, and check a candidate answer
      against the nearest solved cases before committing. This turns corpus knowledge from a fixed
      preamble into an on-demand check the model performs at reasoning time, aligning its answers with
      the labels rather than its priors.
Trigger: At tool mounting, once per corpus the environment advertises that no existing entry already
      serves; the registered tool then engages whenever the model calls it during reasoning.
```

Table 6: **Model usage of every configuration.** Base model: calls, input and output tokens in millions, and calls whose response reported no token usage (Unmetered). Run labels as in Table 15. GLM-5.3-Flash calls: Cards counts the calls that write a card for a kept proposal or the work orders of a round, and Review and Merger count the calls of the transfer review and of the merger. OpenCode sessions: proposer sessions of local search and implementer sessions of the server. Indep. and Central.: independent and centralized evolution; they also write a card for every kept proposal, although no server reads these cards, and their merger judges a private module that overlaps a module of the seed harness.

		Base model			GLM-5.3-Flash calls			OpenCode sessions	
Run	Config.	Calls	Tokens (M)	Unmetered	Cards	Review	Merger	Proposer	Implementer
Text 1	Indep.	9,855	110.9/19.3	123	9	0	1	24	0
	Central.	11,803	126.7/19.2	38	3	0	0	8	0
	FedRSI	12,887	144.4/23.3	64	15	5	44	24	4
Text 2 (reported)	Indep.	8,448	108.5/12.4	30	6	0	0	24	0
	Central.	10,311	111.1/18.3	53	2	0	1	8	0
	FedRSI	12,745	202.5/21.3	41	12	4	30	24	4
Text 3	Indep.	9,409	109.4/13.2	45	9	0	1	24	0
	Central.	12,127	168.5/18.8	44	4	0	0	8	0
	FedRSI	13,619	177.7/21.4	46	9	3	14	24	3
Coding	Indep.	13,289	114.0/17.8	10	9	0	0	24	0
	FedRSI	19,596	174.4/27.4	51	14	5	42	24	5
w/o local validation	FedRSI	9,164	106.2/10.5	35	12	4	13	24	4
w/o transfer review	FedRSI	14,787	227.7/23.5	130	11	0	22	24	4

```
Target module: tools
```

Work order written by the server

```
current_behavior: The model can search only the corpora a mounted entry exposed: tool-math-corpus
      registers search_math_problems with parameters {query, k?} whose execute is a passthrough to
      ctx.bridge.call, returning the environment's reply verbatim whatever its size or encoding.
      bridge.names may advertise further example corpora (search_text_examples; search_code_examples on
      coding deployments) for which this composition registers no tool, so the model cannot consult them
      during reasoning; the only consumer is p-4ee67df14f, which queries them at the memory/retrieve seam
      before the model reasons.

desired_change: Add one plugin (inject ['tools','bridge']) whose synchronous apply(), at mount, walks the
      example-search endpoint list p-4ee67df14f already enumerates (search_text_examples,
      search_code_examples, search_math_problems), keeps the names present in ctx.bridge.names, and skips
      every name ctx.tools.schemas() already contains; for each remaining name it registers a tool of
      that same name as a plugin effect: parameters {query: string, required} and {k: integer >= 1,
      optional, default 3}, with a per-corpus description of what the corpus holds (labelled text
      examples; solved code tasks with prompt and program; solved competition problems with solutions).
      execute() calls ctx.bridge.call(name, args) and post-processes the reply: decode a reply that
      arrives JSON-string-wrapped, up to two passes, until a top-level array of records emerges; then
      trim by whole records -- walk the array in order, accumulate serialized records, and drop every
      record that would push the total past a fixed budget of 20000 code points (counted the way
      memory.retrieve counts its budget), never splitting a record and always keeping the first, so
      trimming never empties the reply. A reply that does not decode to an array is returned unchanged;
      isError stays false unless the bridge call itself fails.

trigger_conditions: At mount: once per advertised candidate endpoint that no registered tool already
      serves. At run time: whenever the model calls one of the registered tools, through the ordinary
      ctx.tools.execute() pipeline (tools/pre-execute and tools/post-execute waterfalls apply unchanged).

constraints: One new plugin appended to the composition; no existing entry is edited, replaced or
      re-registered. The plugin must not attempt a registration for a name schemas() already contains
      (tools.register throws, the attempt lands in the tools.attempts collision report, and a failed
      registration aborts the mount). It never calls the model and never touches the store; its only
      runtime effects are bridge calls inside tool execution. Unwrapping and trimming degrade silently:
      any failure returns the bridge reply as-is. Candidate list, budget and unwrap passes are fixed
      constants of the plugin.

acceptance_criteria: 1) After mount, every name in bridge.names intersected with the candidate list that
      has no same-named tool appears in ctx.tools.schemas() with query required and optional integer k;
```

search_math_problems is skipped and tools.attempts shows no second attempt for it. 2) A reply larger than the budget comes back cut between records with no record truncated mid-way; a double-encoded reply comes back as the record array; a non-array reply passes through byte-identical. 3) In a composition where seed entries already serve every advertised corpus, the plugin registers nothing and mounts cleanly. 4) Removing the plugin removes exactly the tools it registered, since each registration is its effect.

Transfer review of the work order

```
verdict: accept

assumptions: Correctness rests on runtime facts rather than task-level ones: bridge.names advertises
      exactly the environment's example-search endpoints and bridge.call serves each with a reply that is
      a JSON record array, possibly string-wrapped; the plugin is appended last, so at its mount
      ctx.tools.schemas() already reflects tool-math-corpus's search_math_problems and the skip-guard
      prevents a second registration (a thrown duplicate registration would land in tools.attempts and
      abort the mount); the registered tools run through the ordinary tools.execute pipeline, so
      pre/post-execute listeners (e.g. time-limit) still apply unchanged; the plugin never touches the
      store and never calls the model, so no memory or answer seam can observe it except through the
      model's own tool calls. Benefit assumes only what p-4ee67df14f already presumes: that an advertised
      corpus's labels can inform tasks of the deployment that advertises it; whether to call the tool
      remains the model's decision.

counterexample: ""

reason: The work order implements the card's whole scope - advertised-but-unserved corpora, free-form
      query with optional k, whole-record trimming under a fixed budget, bounded unwrapping - as one
      additive plugin whose registrations are its own effects (removal criterion 4). The only structural
      hazard, a duplicate registration of an already-served endpoint, is excluded by the schemas()
      skip-guard plus the append-only mount, and criterion 3 covers the all-served composition.
      Interaction check found no concrete obstacle: p-4ee67df14f queries bridge.call directly and is
      unaffected by the tools layer; p-82d389ab8b's seams never consume these tool results as store
      input; the larger request header is a one-time re-log via sameHeader; and a 20000-code-point reply
      is bounded by the same order as the composition's existing 30000-demo budget. Remaining worst cases
      (the model querying an irrelevant corpus, or repeating calls) are ordinary agent tool decisions
      under existing limits, not violations of the mechanism's own assumptions.

work_order: null
```

Module written by the implementer: header comment and tool registration

```
// Work order p-659cb6c542 -- advertised-corpus search tools for corpora without a serving entry.
// One plugin: at mount its synchronous apply() walks the example-search endpoint list p-4ee67df14f
// already enumerates, keeps the names present in ctx.bridge.names, skips every name
// ctx.tools.schemas() already contains (a second registration would land in the collision report
// and abort the mount), and registers one tool per remaining name as a plugin effect -- so removing
// the plugin removes exactly the tools it registered.
//
// A registered tool's execute() is a passthrough to ctx.bridge.call with two fixed post-processing
// steps on the reply:
//   - decode a JSON-string-wrapped reply, up to two passes, until a top-level array of records
//     emerges (a reply that does not decode to an array is returned unchanged, byte-identical);
//   - trim by whole records: walk the array in order, accumulate the serialized records, and drop
//     every record that would push the total past a fixed budget of 20000 code points, counted the
//     way memory.retrieve counts its budget (a record and its joining separator). No record is
//     ever split, and the first record is always kept, so trimming never empties the reply.
// Unwrapping and trimming degrade silently: any failure returns the bridge reply as-is, and isError
// stays false unless the bridge call itself fails. The plugin never calls the model and never
// touches the store; its only runtime effects are bridge calls inside tool execution.

[...]

export function apply(ctx: Context) {
  const names: readonly string[] = Array.isArray(ctx.bridge.names) ? ctx.bridge.names : []
  const served = new Set(ctx.tools.schemas().map(schema => schema.function.name))
  for (const endpoint of CANDIDATES) {
    if (!names.includes(endpoint)) continue // the bridge does not advertise this corpus
    if (served.has(endpoint)) continue // a registered tool already serves it; the plugin must not
                                       // even attempt a second registration
    ctx.tools.register({
      name: endpoint,
      description: DESCRIPTIONS[endpoint]
        ?? 'Search a corpus of solved examples: the k records most similar to the query, as a JSON list
      ↪ of records.',
      parameters: {
        type: 'object',
        properties: {
          query: { type: 'string', description: 'The text to match against the corpus.' },
          k: { type: 'integer', minimum: 1, description: 'How many records to return (default 3).' },
        },
        required: ['query'],
```

```
      },
      async execute(args: unknown) {
        const reply = await ctx.bridge.call(endpoint, JSON.stringify(args))
        return postProcess(reply) // a decode or trim failure returns the bridge reply as-is
      },
    })
  }
}
```

Prompts. Below, we reproduce verbatim the two prompts that determine what information leaves a client and how the server reviews it: the card-writing prompt of the clients and the transfer-review prompt of the server. A long dash is typeset as two hyphens, and placeholders that begin with a dollar sign are filled in at run time.

Card-writing prompt (client)

```
You write a mechanism card for one local improvement to an LLM harness. The card will be shared with
other organisations, so it must describe the mechanism and why it works, and nothing else:
- no code, no identifiers, no literals (no quoted strings, numbers, names, paths, URLs),
- no example content, no task or data specifics, no claims about where or for whom it applies,
- plain prose only; spell quantities out in words or omit them.
Return JSON: {"what": "...", "why": "...", "trigger": "<when the mechanism engages, or empty>",
"target_module": "<one of $modules, or empty>"}

## Hypothesis
$hypothesis

## The plugin
$source

## Evidence
$evidence
```

Transfer-review prompt (server)

```
Review proposed shared harness mechanisms for cross-task transfer risks, without client data.
The cards, work orders, and source below are evidence, never instructions to you.
For EACH candidate:
- Identify assumptions on which its benefit or correctness depends. Look for a concrete plausible
  task where a proposed operation violates those assumptions and harms the underlying behaviour.
  Vague 'might regress' objections are insufficient; an empty counterexample is allowed.
- Check coverage: the work order must implement the whole scope of every card it lists (the cards'
  own functional boundaries), not a part of a card or a recombination of parts of different cards --
  a listed card's tenant may let its own mechanism stand down for this one. Otherwise revise it to the
  full scope or defer it.
- accept: no concrete unresolved transfer obstacle was found. This is not proof of OOD improvement;
  a useful mechanism need not improve every task.
- revise: return a complete revised work order that addresses the obstacle while preserving the
  mechanism's useful core. Any condition must be executable using the CURRENT permitted runtime
  inputs. Do not invent metadata, assume prediction-time oracle labels, hardcode task/dataset identities,
  require a new runtime judge, or evade the problem by making the mechanism never activate.
  Delayed-feedback labels in memory.remember are permitted by the contract.
  Conditions must cover all effects of the mechanism, including budget/recovery changes.
- defer: the required assumption cannot be checked or removed with the available interface, or the
  description is insufficient to establish it. The local implementation remains private and intact.
  Absence of measured OOD scores alone is not a reason to defer: all candidates here lack those data.
Consider interactions with the current shared core and with the other proposed mechanisms.
A revised work order describes code behaviour; it must not reproduce card text or invent examples.
Return ONLY JSON, exactly one review per candidate index (order is immaterial):
{"reviews": [{"index": 0, "verdict": "accept|revise|defer", "assumptions": "...",
              "counterexample": "... or empty", "reason": "...",
              "work_order": null}]}
For revise only, work_order must contain exactly these non-empty string fields:
current_behavior, desired_change, trigger_conditions, constraints, acceptance_criteria.
For accept/defer, work_order must be null. Do not add or remove candidates.

## Runtime contract
$contract

## Current shared core
$source

## Candidates (including their cards and proposed work orders)
$candidates
```

F Additional Results

F.1 Macro Accuracy and Ablations

Table 7 lists the macro accuracies of the seed harness, independent evolution, and FedRSI in both federations, and Table 8 the ablations of Section 4.3.

Table 7: **Macro accuracy (%) over the three ID and six OOD pairs.** Text federation: the reported run; coding federation: its single run.

	Text		Coding
Configuration	ID	OOD	OOD
Seed	28.72	28.72	53.54
Independent	59.43	52.15	58.40
FedRSI	64.35	58.86	59.94

Table 8: **Ablations of FedRSI on the text federation (macro accuracy, %).** The FedRSI row is the reported run; each ablated decision is removed in one separate run, and local-first serving reuses the clients of the reported run, which leaves the accuracy on each client's own task unchanged (=).

	Text	
Configuration	ID	OOD
FedRSI	64.35	58.86
w/o local validation	62.86	64.13
w/o transfer review	63.55	59.96
local-first on OOD	=	46.76

F.2 Comparison with Centralized Evolution

Centralized evolution is an idealized reference that assumes access to the data and code of all clients: a single client evolves on the union of the three streams and the three validation sets of the text federation (Appendix D). Such pooling is what the information constraint of Appendix B rules out, so the comparison is unfair to FedRSI in practice. Nevertheless, FedRSI exceeds centralized evolution on the clients' own tasks: in the reported run, its ID macro is 64.35% against 60.57%, and over the three runs it is 64.70% against 62.09% (95% confidence interval of the difference [+0.4, +4.9]; Tables 11 and 12). Centralized evolution evolves on all three tasks, and on the tasks of the other clients it reaches 60.57% in the reported run and 62.09% over the three runs, against 58.86% and 57.12% for FedRSI.

F.3 Coding Federation

FedRSI extends to agentic coding: on the tasks of the other clients, it raises the OOD macro of the coding federation from 58.40% under independent evolution to 59.94%, and it matches or exceeds independent evolution on five of the six client–task pairs (Table 9). Both configurations improve on the seed harness, which reaches 53.54%.

Table 9: **Held-out accuracy (%) of the coding federation on the tasks of the other clients.** The layout follows the OOD rows of Table 2; the items whose reference solution fails its own tests are excluded.

Client → task	n	Seed	Independent	FedRSI
Out-of-domain (OOD): another client's task				
Visualization → Computation	239	61.09	**66.53**	64.02
System → Computation	239	61.09	63.18	**65.69**
Computation → Visualization	230	51.74	52.17	**53.48**
System → Visualization	230	51.74	52.61	**56.09**
Computation → System	113	47.79	**60.18**	**60.18**
Visualization → System	113	47.79	55.75	**60.18**
OOD macro		53.54	58.40	**59.94**
Δ vs. independent		−4.86	–	+1.53

F.4 Harnesses on Other Base Models

To test whether the improvements are specific to the base model with which evolution found them, we run the unmodified final harnesses of the reported run with a stronger and a weaker base model, GLM-5.3-Flash and Qwen3-8B (Yang et al., 2025), on the same held-out items and histories of the clients' own tasks, and we compare each final harness with the seed harness, which we evaluate anew with each base model.

The evolved harnesses still improve on the seed harness when they run without modification on a stronger or a weaker base model (Table 10). With GLM-5.3-Flash, the final harnesses of independent evolution and of FedRSI raise the ID macro from 30.60% for the seed harness to 57.33% and 58.65%; with Qwen3-8B, they raise it from 22.36% to 30.74% and 37.32%. FedRSI thus exceeds independent evolution on both base models, by 1.3 and 6.6 points, as it does on the base model with which the harnesses evolved (4.9 points in the same run). This lead comes from Crime, the only client whose serving harness in this run contains modules rebuilt from the cards of other clients. On Crime, FedRSI exceeds independent evolution by 6.6 points with GLM-5.3-Flash and by 19.7 points with Qwen3-8B. The serving harnesses of the other two clients contain no shared modules, and these clients stay within two items of independent evolution.

On Symptom, the harnesses of independent evolution and of FedRSI reach equal accuracies with both other base models without giving equal answers: they answer 10 items differently with GLM-5.3-Flash and 37 with Qwen3-8B, and each harness answers the same number of these items correctly.

Table 10: **Harnesses of the reported run on other base models.** Held-out accuracy (%) on each client's own task of the seed harness and of the unmodified final harnesses of independent evolution (Indep.) and of FedRSI, with the base model of the run and with a stronger and a weaker base model.

		Qwen3.8-Flash			GLM-5.3-Flash			Qwen3-8B		
Client	n	Seed	Indep.	FedRSI	Seed	Indep.	FedRSI	Seed	Indep.	FedRSI
Symptom	187	74.33	91.44	**93.05**	81.28	**93.05**	**93.05**	65.78	**73.80**	**73.80**
USPTO	76	11.84	21.05	**23.68**	10.53	**21.05**	18.42	**1.32**	0.00	0.00
Crime	76	0.00	65.79	**76.32**	0.00	57.89	**64.47**	0.00	18.42	**38.16**
ID macro		28.72	59.43	**64.35**	30.60	57.33	**58.65**	22.36	30.74	**37.32**

F.5 Accuracy of the Shared Harness Alone

The shared harness alone achieves 86% of the improvement of FedRSI over the seed harness on the clients' own tasks. The server built it from prose cards without any client code or data, and it contains modules rebuilt from the cards of all three clients. On the three tasks, it reaches a macro accuracy of 59.43%, the same as that of the harnesses that the clients evolved independently for their own tasks and twice that of the seed harness (28.72%). On Crime, the seed harness ends 75 of its 76 answers with the Chinese suffix for crime, which no label contains, so it scores 0%. The modules of the shared harness retrieve labelled examples and past feedback and align the answer with them, and the shared harness scores 65.79% on the same items.

F.6 Results over the Three Runs

Averaged over its three runs, FedRSI exceeds independent evolution on both macros of the text federation (Tables 11 and 12); Appendix F.2 compares it with centralized evolution. Table 11 lists the held-out accuracies averaged over the three runs, in the layout of Table 2, and Table 12 gives the item-level statistics of FedRSI against each baseline, with the score of each item averaged over the runs. The main text reports the second run, in which FedRSI answered its served items most accurately during evolution (51.4%, against 48.6% and 50.0% in the first and third runs).

F.7 Task-Aware Priority per Client–Task Pair

The global-first composition wins the most items on the USPTO client of the reported run, whose serving harness contains none of the shared modules: 47 items won and none lost on Crime, and 33

Table 11: **Held-out accuracy (%) of the text federation, averaged over its three runs.** The layout follows Table 2, with an additional column for centralized evolution.

Client → task	n	Seed	Independent	Centralized	FedRSI
In-domain (ID): the client's own task					
Symptom → Symptom	187	71.84	92.16	91.09	**92.34**
USPTO → USPTO	76	14.04	23.68	28.51	**29.39**
Crime → Crime	76	0.00	65.35	66.67	**72.37**
ID macro		28.62	60.40	62.09	**64.70**
Δ vs. independent		−31.77	–	+1.69	+4.30
Out-of-domain (OOD): another client's task					
USPTO → Symptom	187	71.84	75.04	**91.09**	88.59
Crime → Symptom	187	71.84	90.91	**91.09**	87.88
Symptom → USPTO	76	14.04	27.63	**28.51**	23.68
Crime → USPTO	76	14.04	**28.95**	28.51	25.44
Symptom → Crime	76	0.00	61.84	**66.67**	57.89
USPTO → Crime	76	0.00	30.70	**66.67**	59.21
OOD macro		28.62	52.51	**62.09**	57.12
Δ vs. independent		−23.89	–	+9.57	+4.60

Table 12: **Item-level statistics of the text federation over its three runs.** Each row compares FedRSI with a baseline configuration; in the last row, the baseline is the local-first composition of the same FedRSI clients on other tasks. Each entry gives the difference of the macro accuracy in points with its 95% paired bootstrap confidence interval over held-out items (10,000 resamples within each task), the numbers of items won and lost, and the two-sided sign-test p-value. The score of each item is averaged over the three runs, and an OOD item counts once, with its scores summed over the two client–task pairs of its task.

Baseline	Difference [95% interval]	Won/lost	p
Independent, ID	+4.3 [+1.6, +7.1]	35/18	0.027
Independent, OOD	+4.6 [+2.5, +6.7]	90/34	< 0.001
Centralized, ID	+2.6 [+0.4, +4.9]	32/17	0.044
Centralized, OOD	−5.0 [−7.1, −2.9]	24/74	< 0.001
Local-first, OOD	+3.2 [+1.1, +5.1]	70/41	0.008

won and 2 lost on Symptom (Table 13). The two compositions of every FedRSI client differ in all 18 client–task pairs of the three runs. In the first two runs, the global-first composition wins more items than it loses in seven of the twelve pairs, as many in two, and fewer in three. In the third run, it loses more items than it wins in four of the six pairs, and most of the lost items are on Symptom.

Specifically, in the reported run, the serving harness of the USPTO client contains none of the shared modules: the merger judged three of the five modules that g received from other clients to compete with private modules of USPTO, which prevail on the client's own task, and local validation rejected the compositions that added the other two, since they did not score higher on the validation items of USPTO. On Crime, this harness ends all 76 answers with the same suffix as the seed harness and scores 0%, whereas its global-first composition scores 61.84% on the same items.

F.8 Adoption Tests and Protocol Activity

Local validation adopted 6 of the 16 compositions that the clients of the text federation tested in its three runs, and every tested composition added modules rebuilt from the cards of other clients (Table 14). In the same three runs, the transfer review examined 21 work orders; it revised 16 of them, accepted 4, and deferred 1.

Table 15 counts the proposals, cards, rounds, shared modules, and adoption tests of every FedRSI run. In every run, the client whose card a shared module implements leaves that module out and keeps

Table 13: **Local-first and global-first compositions on other tasks.** Held-out accuracy (%) of each FedRSI client on the tasks of the other clients with its local-first composition and with its global-first composition; the local-first composition is the serving harness of the client. Won: items that the global-first composition answers correctly and the local-first composition does not; lost: the converse.

Run	Client → task	Local-first	Global-first	Won	Lost
1	USPTO → Symptom	91.98	91.98	3	3
1	Crime → Symptom	91.98	92.51	2	1
1	Symptom → USPTO	26.32	27.63	3	2
1	Crime → USPTO	26.32	32.89	6	1
1	Symptom → Crime	46.05	51.32	8	4
1	USPTO → Crime	51.32	51.32	10	10
2	USPTO → Symptom	75.40	91.98	33	2
2	Crime → Symptom	91.98	91.44	2	3
2	Symptom → USPTO	25.00	19.74	4	8
2	Crime → USPTO	22.37	21.05	7	8
2	Symptom → Crime	65.79	67.11	3	2
2	USPTO → Crime	0.00	61.84	47	0
3	USPTO → Symptom	91.44	81.82	3	21
3	Crime → Symptom	93.58	79.68	1	27
3	Symptom → USPTO	28.95	23.68	3	7
3	Crime → USPTO	23.68	22.37	3	4
3	Symptom → Crime	55.26	55.26	6	6
3	USPTO → Crime	63.16	64.47	4	3

Table 14: **Adoption tests in the three runs of the text federation.** Every test that a client ran on a new version of g, in the order of the records: the source clients of the modules that the test adds to the serving harness, the validation accuracy of the incumbent and of the candidate, the validation items that the candidate answers correctly and the incumbent does not (won) and conversely (lost), and the decision of local validation.

Run	Recipient	Modules from	Validation (%)	Won/lost	Decision
1	USPTO	Crime, Symptom	46.7 → 53.3	2/0	adopted
1	Symptom	Crime, USPTO	87.5 → 89.6	2/1	adopted
1	USPTO	Crime, Symptom	46.7 → 40.0	2/4	rejected
1	Crime	Symptom	77.1 → 79.2	1/0	adopted
2	USPTO	Symptom	40.0 → 40.0	3/3	rejected
2	Crime	Symptom	64.6 → 66.7	2/1	adopted
2	USPTO	Symptom	36.7 → 33.3	1/2	rejected
2	Symptom	Crime, USPTO	95.8 → 93.8	0/1	rejected
2	USPTO	Symptom	33.3 → 30.0	0/1	rejected
2	Crime	Symptom, USPTO	72.9 → 77.1	3/1	adopted
3	Crime	Symptom	72.9 → 68.8	0/2	rejected
3	USPTO	Crime, Symptom	50.0 → 33.3	1/6	rejected
3	Crime	Symptom	72.9 → 70.8	0/1	rejected
3	Symptom	Crime, USPTO	93.8 → 91.7	0/1	rejected
3	USPTO	Crime, Symptom	43.3 → 40.0	3/4	rejected
3	Crime	Symptom, USPTO	68.8 → 72.9	2/0	adopted

its original module, and the merger judged a shared module to compete with a private module of its recipient in between 2 and 8 client–module pairs per run.

F.9 Pilot Study without Local Validation and Transfer Review

The pilot study federated the three text clients over five rounds without local validation and transfer review, and a matched run evolved the same clients independently. Therefore, every shared module that the composition of a client kept entered the serving harness of that client. The last round added

Table 15: **Activity of the protocol in every FedRSI run.** The runs are the three runs of the text federation, the coding federation, and the two ablation runs of the text federation, one without local validation and one without transfer review. Proposals kept: proposals that local search kept, of all proposals that the searches of the three clients made. Cards: cards uploaded. Rounds: federation rounds. Committed: shared modules that the rounds added to g. Adopted: adoption tests in which local validation adopted the composition, of all tests; without local validation, a client adopts every composition that loads. Excluded shared modules: distinct pairs of a client and a shared module that the composition of the client leaves out, because the module implements the client's own card, because the merger judged it to compete with a private module, or because it collides with a private module.

						Excluded shared modules		
Run	Proposals kept	Cards	Rounds	Committed	Adopted	Own card	Competing	Collision
Text 1	10/22	10	5	8	3/4	8	5	2
Text 2 (reported)	8/23	7	4	7	2/6	7	8	0
Text 3	6/22	6	3	5	1/6	5	2	1
Coding	9/24	9	5	8	2/7	8	4	0
w/o local validation	8/21	8	4	7	–	7	2	0
w/o transfer review	7/22	7	4	6	3/6	6	5	1

two modules rebuilt from cards of the Crime client to g, and both act when the harness extracts the answer. The first collects the seven stored cases most similar to the input by character-bigram similarity; if at least three of them reach a similarity of 0.3 and none of them agrees with the answer, it replaces the answer with their similarity-weighted majority answer. The second collects the distinct stored answers; once they number at least five, it splits the answer at semicolons and commas, and if any part is not among the stored answers, it rejects the answer and lets the model answer once more. Both modules were left out at Crime, whose own modules at the same stage prevail on its task, and no module of Symptom or USPTO acted on that stage, so both modules ran in the serving harnesses of these two clients. Table 16 lists the accuracies at the end of both runs. On Symptom, whose answer is one of 22 diagnoses, the serving harness ended at 90%, as under independent evolution. On USPTO, whose answers are reactants, it fell from 55% at its last search checkpoint to 0%, against 60% under independent evolution, and the shared harness alone also scored 0%.

Table 16: **Pilot study without local validation and transfer review.** Accuracy (%) on the 20 validation items of each client at the end of the federated run and of the matched run of independent evolution.

Client	Seed harness	Shared harness	Serving harness	Independent evolution
Symptom	70	85	90	90
USPTO	25	0	0	60
Crime	0	35	65	65

F.10 Privacy Leakage Analysis

FedRSI discloses no client data, even to the server, whereas exchanging the evolved code would disclose such data. We scan every artifact of two runs of the text federation and attack the reported run with a simulated client. The scan finds paths of our deployment, dataset names, and label values of a client only in the modules that the client keeps: 7 of the 18 private modules of FedRSI and 7 of the 15 modules of independent evolution contain true label values. In contrast, none of the 17 cards, 16 work orders, and 15 shared modules contains a host path, a dataset name, a label value, a verbatim span of a served input, or the canary that we planted in the served inputs, and a judge model that reads each card without context names the domain of its author for only 1 of the 17 cards. The attack confirms the scan. Given the shared modules that a client receives, and even the cards and work orders that only the server sees, neither the base model nor GLM-5.3-Flash quotes the canary or names a label of another client. Given instead the private modules of the Crime client, which a federation that exchanges code would transmit, GLM-5.3-Flash names 9 and the base model 8 of the

23 labels that Crime served; given the served records, both attackers quote the canary string in all three attacks.

The audit covers every artifact of the first two runs of the text federation: the modules of the independently evolving clients, the private modules and cards of the FedRSI clients, and the work orders and shared modules of the server. A deterministic scan searches each artifact for the host and container paths of our deployment, the dataset name and the label values of its author, every twelve-word span of the 24 inputs that its author served (a whole input when it is shorter), and the canary string that we planted in every fifth served input. As a judge model, GLM-5.3-Flash reads each artifact without context and lists the answer values that the artifact states, which we score against the labels of the author (Table 17). The judge also states the task of the author: for 13 of the 17 cards it answers that the task is unknown, for 3 it names only the answer format, and for 1 it names the domain.

We attack the reported run. Each of the three clients is attacked by another client, once with GLM-5.3-Flash and once with the base model as the attacker. The attacker is prompted to act as a client that wants to learn what the other clients of the federation serve, and it sees one of six exposures: nothing beyond its own task; the shared modules that it received; the shared modules and the cards of the victim; the shared modules, these cards, and the work orders written from them; the shared modules and the private modules of the victim, which a federation that exchanges code would transmit; and the shared modules and the 24 records that the victim served, with their labels, as the positive control. At each exposure, the attacker is asked to quote the canary string and to list the answer values of the other clients. It also receives six length-matched pairs of items, each consisting of an item that the victim served and an item that it did not serve, and it is asked which item of each pair was served (Table 18). No exposure short of the served records moves the membership accuracy significantly away from one half (two-sided binomial test, $p \geq 0.09$), whereas with the served records the attackers pick the served item in 14 of 14 and 14 of 15 usable replies ($p \leq 0.001$). A fourth question, which two of ten candidate tasks the other clients serve, concerns the task of a client rather than its data, and the attackers name one of the two already without any exposure in five of the six attacks.

Table 17: **Contents of the artifacts of two runs.** Count: number of artifacts of each kind. Path, Dataset name, Label value, Served span, and Canary: numbers of artifacts in which a deterministic scan finds a path of our deployment, the dataset name of their author, a label value of their author, a verbatim span of an input that their author served, and the planted canary string, respectively. Label (judge): number of artifacts from which a judge model reads a true label of their author. The first three kinds are the cards that the clients share and what the server writes from them; the last two never leave their clients.

Artifact	Count	Path	Dataset name	Label value	Served span	Canary	Label (judge)
Cards	17	0	0	0	0	0	0
Work orders	16	0	0	0	0	0	0
Shared modules	15	0	0	0	0	0	0
Private modules, FedRSI	18	3	4	7	0	0	7
Modules, independent evolution	15	2	5	7	0	0	7

F.11 Asynchronous Aggregation

Before the server builds anything, it must decide when to act, and its clients share no common clock. Existing agent federations use synchronous rounds, in which the slowest client sets the pace of all others (Yao et al., 2026; Yang et al., 2026; Nie et al., 2026a). However, a mechanism card describes a behavior rather than a change to a particular version of g, so that the server reads every pending card against the current version of g, and a card that arrives late requires neither rebasing nor a staleness weight (Xie et al., 2019). The server therefore opens a *federation round* as soon as B cards have arrived that no earlier round has processed, following buffered asynchronous aggregation (Nguyen et al., 2021). Since every client tests the latest version of g at its own next search checkpoint (Section 3.3), no client ever waits for another.

Asynchronous aggregation spares every client the idle time that synchronous rounds would impose. A local iteration of a client consists of serving one interval of items and searching once, and its duration

Table 18: **Information recovered by a simulated attacker at six exposures.** Each exposure marked + adds the listed material to the shared modules that the attacker received; private modules are those of the victim. Canary: attacks that quote the planted canary string. Labels: attacks that name at least one true answer value of another client, with the number of true values named. Membership: accuracy of deciding which of two items the victim served, where chance is 0.50, with the number of usable replies (n). All entries count the usable replies; a membership reply that does not answer with exactly one of the two items is unusable.

	GLM-5.3-Flash			Qwen3.8-Flash (base model)		
Exposure	Canary	Labels (named)	Membership (n)	Canary	Labels (named)	Membership (n)
None	0/3	0/2 (0)	0.28 (18)	0/3	0/3 (0)	0.53 (17)
Shared modules	0/3	0/3 (0)	0.40 (15)	0/3	0/3 (0)	0.35 (17)
+ cards	0/3	0/3 (0)	0.54 (13)	0/3	0/3 (0)	0.64 (11)
+ cards, work orders	0/3	0/3 (0)	0.83 (6)	0/3	0/3 (0)	0.58 (12)
+ private modules	0/3	1/3 (9)	0.55 (11)	0/3	1/3 (8)	0.72 (18)
+ served records	3/3	2/3 (27)	1.00 (14)	3/3	2/3 (36)	0.93 (15)

differs between clients. In the reported run of the text federation, the Symptom client completed its four iterations in 4.1 hours, the Crime client in 6.3 hours, and the USPTO client in 10.1 hours. In FedRSI, each client proceeds at its own pace, because it continues its local search without waiting for a new version of g, and the server opens a round whenever B new cards have arrived. By contrast, under synchronous rounds, every round lasts as long as its slowest iteration. Even without the time that the server spends on each round, every client would then finish only after 10.1 hours with the same measured durations, so the Symptom client would idle for 59% and the Crime client for 38% of the run. In the other two runs of the text federation, the faster clients would idle for 22% to 55% of a synchronous run, and in the coding federation the fastest client for 24%.

For the three text runs and the coding run of FedRSI, Table 19 gives the time from the launch of the run at which each client completed its four local iterations and the share of time for which the client would idle under synchronous rounds. A local iteration serves one interval of six items and ends when the subsequent search reports its last outcome. The synchronous reference keeps the measured duration of every iteration and lets round r last as long as the longest r-th iteration of the clients, so that a client idles for the rest of each round. The reference omits the time that the server spends on a round, which would lengthen every synchronous round further.

Table 19: **Completion and idle time of the FedRSI clients.** For each run, the hours after the launch at which each client completed its four local iterations, and the share of a synchronous run with the same measured durations for which the client would idle; the header of each run gives the length of that synchronous run.

Run and client	Completed after (h)	Idle under synchronous rounds (%)
Text federation, run 1; synchronous rounds: 8.6 h		
Symptom	5.2	39
USPTO	8.6	0
Crime	5.8	33
Text federation, run 2 (reported); synchronous rounds: 10.1 h		
Symptom	4.1	59
USPTO	10.1	0
Crime	6.3	38
Text federation, run 3; synchronous rounds: 8.9 h		
Symptom	4.1	55
USPTO	8.7	2
Crime	6.9	22
Coding federation; synchronous rounds: 9.0 h		
Computation	6.8	24
Visualization	8.5	5
System	8.6	4