EVOGENOME: GENOTYPE-PHENOTYPE EVOLUTION FOR COMPOSITIONAL AGENT SKILLS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model agents increasingly use reusable skills that encode procedural knowledge for tool use, artifact manipulation, verification, and failure recovery. Existing skill-evolution methods search over free-form text, files, or complete skill packages. As a result, a single edit can modify unrelated behaviors, which complicates the attribution and recombination of successful changes. We introduce **EvoGenome**, a genotype–phenotype framework that represents compositional agent skills as typed, functionally defined, and independently heritable units. Its **SkillGenome** encodes activation triggers, preconditions, planning logic, tool-use policies, executable modules, postconditions, verification procedures, recovery behavior, and adaptation interfaces. A materialization compiler translates each genome into a conventional executable skill package, separating structured search from deployment while maintaining compatibility with existing agent harnesses. Type- and interface-aware operators perform constrained mutation, homologous crossover, module specialization, pruning, and cross-domain transplantation while preserving dependencies among skill components. EvoGenome uses execution traces and lineage differences to attribute behavioral and fitness changes to inherited genome segments, allowing subsequent variation to prioritize genes that provide consistent benefits. We compare EvoGenome with free-form rewriting and file-level crossover on a skill benchmark and a dynamic tool-use environment under fixed execution budgets. We evaluate validity, executability, behavioral innovation, fitness improvement, regression, and cross-domain transfer. Ablation studies assess the effects of type constraints, lineage-based credit, crossover, non-textual genes, and genome granularity.

1 INTRODUCTION

Large language model (LLM) agents increasingly operate in environments that require general reasoning and procedural knowledge to coordinate tools, manipulate artifacts, validate intermediate results, and recover from failures. Agent skills encode this knowledge as reusable packages of instructions, executable programs, templates, and auxiliary resources. Skills are inspectable, revisable, transferable, and loadable at inference time without updating model weights. Recent evaluations show that well-designed skills improve performance across diverse, expertise-intensive tasks, with gains depending strongly on skill focus and composition.(Li et al., 2026) This observation raises a broader question: how should reusable skill units be represented and improved automatically?

Recent work frames skill construction as artifact-level learning. EvoSkill discovers and refines structured skill folders through failure analysis and validation-based selection,(Alzubi et al., 2026) CoEvoSkills couples skill generation with a co-evolving surrogate verifier,(Zhang et al., 2026b) and SkillEvolver iteratively updates skill prose and code using deployment feedback from independent agents.(Zhang et al., 2026a) These approaches use feedback-driven search to improve external artifacts without modifying the underlying LLM. Their search spaces, however, largely consist of prompts, files, code, or complete skill packages. Free-form rewriting can simultaneously alter triggers, planning logic, tool policies, executable modules, verification rules, and recovery behavior, resulting in discontinuous search. An offspring may contain a useful local innovation but fail because an unrelated interface has changed, and observed fitness gains cannot be reliably attributed to their causal components. File-level crossover faces the same limitation because file boundaries do not necessarily correspond to functional or semantically compatible units. Existing methods therefore

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.

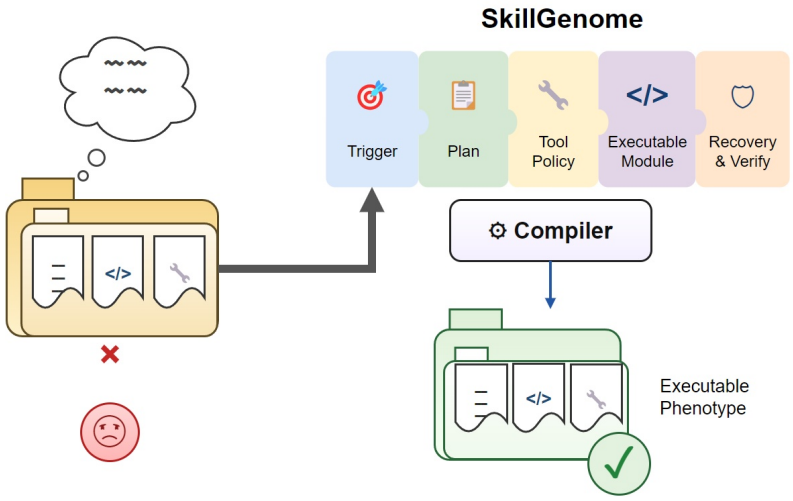

Figure 1: Overview of the EvoGenome framework for compositional agent-skill evolution.

leave three questions unresolved: what constitutes a heritable skill unit, when can such units be safely recombined, and how should their effects be credited across a lineage?

We formulate compositional skill evolution through explicit genotype–phenotype separation. The genotype exposes typed, functionally meaningful units that can be inherited independently, while the phenotype remains a conventional package executable by existing agent harnesses. This representation preserves dependencies among triggers, preconditions, plans, tools, executable code, postconditions, and recovery behavior. It constrains mutation and crossover through semantic interfaces and links behavioral changes to the responsible units, enabling the reuse of successful innovations and the suppression of unproductive changes. Under this formulation, a skill becomes a compositional object that supports controlled variation, interpretable selection, and module-level transfer.

We propose **EvoGenome**, a framework for **genotype–phenotype evolution of compositional agent skills**. EvoGenome represents each skill as a typed **SkillGenome** with genes for activation triggers, preconditions, planning, tool-use policies, executable modules, postconditions and verification, failure recovery, and cross-task adaptation. A materialization compiler translates the genome into an executable phenotype containing the instructions, scripts, templates, and resources required by a conventional agent harness. This design enables search over functional units while grounding evaluation in actual execution. EvoGenome provides type- and interface-aware operators for trigger recombination, homologous module crossover, recovery transplantation, module duplication and specialization, and dead-gene pruning. By retaining compatible dependencies, these operators reduce destructive edits caused by jointly rewriting unrelated skill components. EvoGenome further associates execution traces and fitness changes with genome differences along each lineage. This segment-level credit guides future mutation probabilities and identifies modules whose effects persist across descendants. Transfer adapters expose task-specific interfaces, allowing recovery, verifier, or executable genes evolved in one domain to be transplanted and evaluated in another without transferring the complete source skill. Figure 1 illustrates this genotype–phenotype evolution process and its principal components.

Our experiments evaluate whether this representation improves evolution under a fixed execution budget while controlling for the number of generated candidates. We compare free-form rewriting, file-level crossover, and genome-level evolution on SkillsBench and a dynamic tool-use environment. The evaluation measures syntactic validity, executability, behavioral change, fitness-improving mutation rate, and regression frequency. Cross-domain transplantation tests whether recovery and verifier genes retain utility outside their source domains. Ablations remove type constraints, lineage-based credit attribution, crossover, and non-textual genes. We also vary genome granularity to identify when compositional structure facilitates or constrains search. Together, these experiments assess whether typed, attributable variation produces more useful offspring and supports more reliable module reuse than unconstrained artifact rewriting.

Our contributions are:

- We formulate agent-skill evolution as genotype–phenotype search and introduce the SkillGenome, a typed representation that decomposes multi-file skill packages into functionally meaningful, heritable units.
- We propose a materialization compiler and semantic operators for constrained mutation, homologous crossover, module specialization, pruning, and cross-skill transplantation while preserving executable phenotypes.
- We introduce trace- and lineage-based gene credit attribution that links behavioral fitness changes to inherited genome segments and uses this evidence to guide subsequent variation.
- We provide an evaluation protocol that separately measures candidate validity, executability, behavioral innovation, fitness improvement, regression, and cross-domain transfer, enabling direct comparison with free-form and file-level alternatives.

2 Related Work

2.1 Externalized Procedural Knowledge for Language Agents

LLM agents externalize procedural knowledge through prompts, demonstrations, memories, tool specifications, and programs. ReAct interleaves reasoning with actions(Yao et al., 2022), ART retrieves demonstrations and composes external-tool calls(Paranjape et al., 2023), Toolformer learns to insert API calls(Schick et al., 2023), and Gorilla specializes models for API selection and argument generation(Patil et al., 2023). HuggingGPT delegates subtasks to externally selected models(Shen et al., 2023), while Large Language Models as Tool Makers separates reusable tool construction from application(Cai et al., 2023).

Interaction-based approaches retain experience as textual knowledge. Reflexion stores verbal feedback(Shinn et al., 2023), Self-Refine iteratively critiques and revises outputs(Madaan et al., 2023), ExpeL abstracts lessons from trajectories(Zhao et al., 2023), and Language Agent Tree Search combines reflection with value-guided search(Zhou et al., 2023). Their persistent state, however, is typically an untyped collection whose retrieval, consistency, and attribution depend on contextual interpretation.

Executable libraries provide greater structure. Voyager synthesizes, verifies, indexes, and reuses code-based behaviors(Wang et al., 2023), while SkillsBench defines skills as packages containing instructions, scripts, templates, and references(Li et al., 2026). Such packages support compositional reuse but introduce dependencies among instructions, schemas, runtime interfaces, verification logic, and recovery procedures.

2.2 Feedback-Driven Optimization of Prompts and Agent Artifacts

Prompt optimization searches over natural-language instructions. Automatic Prompt Engineer generates and ranks candidates(Zhou et al., 2022), OPRO conditions generation on evaluated solutions and scores(Yang et al., 2023), Automatic Prompt Optimization combines textual critiques with beam search(Pryzant et al., 2023), and TextGrad propagates language feedback through textual computation graphs(Yuksekgonul et al., 2024). Evolutionary variants maintain populations: PromptBreeder co-evolves task and mutation prompts(Fernando et al., 2023), EvoPrompt applies genetic and differential-evolution operators(Guo et al., 2023), and GEPA combines trace-based reflective mutation with Pareto selection(Agrawal et al., 2025). Free-form rewriting can nevertheless alter several procedural functions simultaneously.

DSPy structures language-model programs as modules with explicit signatures and optimizable instructions and demonstrations(Khattab et al., 2023), although compatibility and component-level credit remain difficult. Recent systems optimize persistent skills directly: EvoSkill revises structured skill folders through failure diagnosis and validation(Alzubi et al., 2026), CoEvoSkills jointly improves skills and a separated surrogate verifier(Zhang et al., 2026b), and SkillEvolver updates instructions and code from deployment feedback(Zhang et al., 2026a). These systems optimize multi-file artifacts using execution evidence, but mutation units often follow textual or file boundaries rather than stable procedural functions.

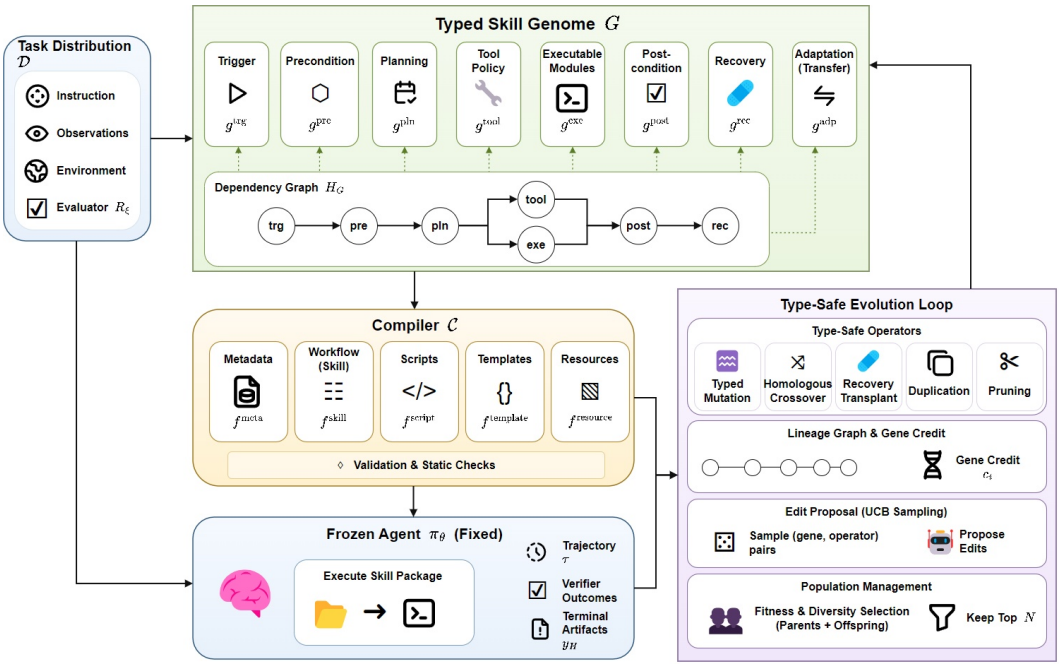

Figure 2: Overview of the EvoGenome framework and its type-safe evolutionary loop.

2.3 Structured Evolution, Modularity, and Credit Assignment

Genetic programming searches executable symbolic structures(Koza, 1992), automatically defined functions enable reusable evolving subprograms(Koza, 1994), and cooperative coevolution evaluates interacting components through collaborations(Potter & Jong, 2000). These methods expose relational credit assignment because component fitness depends on collaborators and contexts.

NEAT aligns structurally distinct networks using historical markings(Stanley & Miikkulainen, 2002), while HyperNEAT indirectly encodes large connectivity patterns(Stanley et al., 2009). Diversity-oriented methods include novelty search(Lehman & Stanley, 2011), MAP-Elites(Mouret & Clune, 2015), quality-diversity optimization(Pugh et al., 2016), and NSGA-II(Deb et al., 2002), which preserve behavioral niches or objective trade-offs.

Program repair similarly evolves executable artifacts against tests. GenProg searches statement-level variants(Weimer et al., 2009), generate-and-validate analyses identify test adequacy, search-space design, and overfitting as key limitations(Goues et al., 2012), and Prophet ranks patches using features learned from human repairs(Long & Rinard, 2016). This literature demonstrates the importance of executable validation and mutation granularity, while showing that passing tests does not guarantee semantic correctness beyond the validation distribution.

3 Method

3.1 Overview and Problem Formulation

EvoGenome represents an agent skill as a typed genotype and compiles it into an executable phenotype compatible with standard skill-enabled agent harnesses. The representation separates invocation conditions, planning, executable code, validation, and recovery, supporting component-level mutation, crossover, and attribution. Figure 2 summarizes the evolutionary loop: genomes are compiled into multi-file skill packages, executed by a frozen agent, evaluated, and evolved through type-safe operators guided by lineage-aware gene credit.

Let $\mathcal{D}$ denote a task distribution. Each task $\xi \sim \mathcal{D}$ specifies an instruction, observable inputs, an environment, and an evaluator R_ξ. Given a skill package $\mathcal{S}$, a frozen policy π_θ produces a trajectory $\tau = (o_1, a_1, \ldots, o_H)$ and terminal artifacts y_H:

$$\tau \sim P(\tau \mid \pi_\theta, \mathcal{S}, \xi), \qquad r(\tau, \xi) = R_\xi(y_H). \tag{1}$$

Only the external skill evolves, while θ remains fixed. A compiler $\mathcal{C}$ maps a typed genome $\mathbf{G}$ to the phenotype $\mathcal{S}_\mathbf{G} = \mathcal{C}(\mathbf{G})$. The objective is

$$\mathbf{G}^\star \in \arg\max_{\mathbf{G} \in \mathbb{G}_{\text{valid}}} J(\mathbf{G}), \qquad J(\mathbf{G}) = \mathbb{E}_{\xi \sim \mathcal{D}} \mathbb{E}_{\tau \sim P(\cdot \mid \pi_\theta, \mathcal{C}(\mathbf{G}), \xi)}[r(\tau, \xi)], \tag{2}$$

where $\mathbb{G}_{\text{valid}}$ contains genomes that satisfy schema, interface, dependency, resource, and static-parsing constraints. Finite evaluation batches provide estimates of J.

3.2 Skill Genome and Materialization

The genome is an ordered, dependency-aware tuple

$$\mathbf{G} = (g^{\text{trg}}, g^{\text{pre}}, g^{\text{pln}}, g^{\text{tool}}, \mathbf{g}^{\text{exe}}, g^{\text{post}}, g^{\text{rec}}, g^{\text{adp}}), \tag{3}$$

whose components represent triggering, precondition validation, planning, tool policy, executable modules, postcondition checks, recovery, and transfer adaptation. Each gene is defined as $g_i = (z_i, c_i, \mathcal{I}_i, \mathcal{O}_i, \mathcal{D}_i, \nu_i)$, where z_i and c_i denote the type and content, $\mathcal{I}_i$ and $\mathcal{O}_i$ are typed interfaces, $\mathcal{D}_i$ lists dependencies, and ν_i identifies the lineage version. Gene content can comprise instructions, configuration, code, or references governed by a common interface contract.

The induced dependency graph $\mathcal{H}_\mathbf{G} = (V_\mathbf{G}, E_\mathbf{G})$ follows

$$g^{\text{trg}} \prec g^{\text{pre}} \prec g^{\text{pln}} \prec \{g^{\text{tool}}, \mathbf{g}^{\text{exe}}\} \prec g^{\text{post}} \prec g^{\text{rec}}, \tag{4}$$

while g^{adp} can parameterize any gene subject to its interface. A provider g_i and a consumer g_j are compatible if their dependency types are allowed, their output and input interfaces match, and their contracts are jointly satisfiable:

$$\text{Compat}(g_i, g_j) = \mathbb{I}\big[z_i \rightsquigarrow z_j \wedge \mathcal{O}_i \succeq \mathcal{I}_j \wedge \text{SAT}\big(\text{ctr}(g_i), \text{ctr}(g_j)\big)\big]. \tag{5}$$

Genome validity is therefore defined as

$$\text{Valid}(\mathbf{G}) = \text{Schema}(\mathbf{G}) \wedge \text{DAG}(\mathcal{H}_\mathbf{G}) \wedge \bigwedge_{(g_i, g_j) \in E_\mathbf{G}} \text{Compat}(g_i, g_j) \wedge \text{StaticCheck}(\mathbf{G}). \tag{6}$$

The compiler emits

$$\mathcal{C}(\mathbf{G}) = \{f^{\text{meta}}, f^{\text{skill}}, \mathbf{f}^{\text{script}}, \mathbf{f}^{\text{template}}, \mathbf{f}^{\text{resource}}\}, \tag{7}$$

including metadata, workflow instructions, scripts, templates, and resources. It validates the genome and builds the package in a clean workspace. Compilation failures are assigned invalid status without semantic repair, allowing malformed offspring to be distinguished from valid but ineffective offspring.

3.3 Type-Safe Evolution

For a population $\mathcal{P}^{(t)} = \{\mathbf{G}_n^{(t)}\}_{n=1}^N$, an operator $o \in \mathcal{O}$ consumes one parent for mutation or two parents for crossover and declares the gene types it can produce:

$$\widetilde{\mathbf{G}} = o(\mathbf{G}_{p_1}, \ldots, \mathbf{G}_{p_K}; \omega) \in \mathbb{G}_{\text{valid}}, \qquad K \in \{1, 2\}. \tag{8}$$

Typed mutation applies the type-specific operator $\widetilde{g}_i = \mu_{z_i}(g_i, \mathcal{N}_\mathbf{G}(g_i), \mathcal{F})$, conditioned on adjacent contracts and parent failures. The operator preserves the interface unless the same transaction updates all affected dependents. Unchanged genes retain their identities for attribution.

Homologous crossover exchanges genes or connected subgraphs with equal roles:

$$\widetilde{\mathbf{G}} = \text{Cross}(\mathbf{G}^a, \mathbf{G}^b, U^a, U^b), \qquad \text{Type}(U^a) = \text{Type}(U^b). \tag{9}$$

Single-gene replacements require compatible boundary interfaces, and subgraph replacements require matching incoming and outgoing contracts. Recovery transplantation can use g^{adp} to map source contracts to the target; invalid mappings are rejected. Duplication assigns a new lineage identifier and dispatch condition to support subsequent specialization. Pruning removes only unreachable genes or genes whose outputs are neither consumed nor terminal:

$$\text{Dead}(g_i; \mathbf{G}) = \mathbb{I}\big[g_i \notin \text{Reach}(g^{\text{pln}}, \mathcal{H}_\mathbf{G}) \vee \big(\text{OutDeg}(g_i) = 0 \wedge g_i \notin \mathcal{T}_\mathbf{G}\big)\big]. \tag{10}$$

Method	SkillsBench Pass Rate ↑	Dynamic Pass Rate ↑	Valid ↑	Executable ↑	Improving Mut. ↑	Regression ↓
No Skill	31.4±1.8	24.7±2.1	–	–	–	–
Human-Authored Skills	52.8±1.5	39.6±2.4	100.0	96.8±0.7	–	–
One-Pass Generation	36.9±2.7	28.5±2.8	77.6±3.4	68.1±3.8	–	18.9±2.5
Free-Form Rewrite	58.7±2.2	46.3±2.5	73.8±2.9	65.7±3.2	14.8±1.7	16.4±1.9
File-Level Crossover	55.6±2.4	43.1±2.7	81.5±2.5	71.9±3.0	13.1±1.5	14.7±1.8
AST-Level Evolution	62.1±1.9	49.7±2.2	94.2±1.1	87.4±1.8	18.6±1.4	10.8±1.3
EvoGenome	**64.5±1.6**	**54.5±1.9**	**98.1±0.6**	**93.6±1.1**	**23.9±1.3**	**7.2±0.9**

Table 1: Held-out performance and offspring quality under an identical candidate-execution budget. Values are percentages over five evolutionary seeds. Abbreviation: Mut. = Mutation.

Variant	Pass Rate ↑	Valid ↑	Improving Mut. ↑	Regression ↓
EvoGenome	**64.5±1.6**	98.1±0.6	**23.9±1.3**	**7.2±0.9**
w/o Type Constraints	58.1±2.1	82.7±2.4	16.5±1.6	14.1±1.7
w/o Lineage Credit	60.2±2.0	97.8±0.7	18.1±1.5	10.6±1.2
w/o Crossover	61.4±1.8	98.4±0.5	20.2±1.4	8.5±1.0
Text Genes Only	57.3±2.3	96.9±0.9	15.7±1.7	11.9±1.5
Coarse Genome	59.6±2.0	92.3±1.4	17.4±1.5	11.2±1.3
Fine-Grained Genome	62.7±1.7	**98.6±0.5**	21.8±1.2	7.9±0.8

Table 2: SkillsBench ablations under the same backbone and execution budget. Values are percentages over five seeds. Abbreviations: w/o = without; Mut. = Mutation.

3.4 Lineage-Aware Credit and Evolution

A lineage graph records genome versions, applied operators, and edited gene sets. For an offspring $\widetilde{\mathbf{G}}$ and its parent $\mathbf{G}$, let $\Delta(\widetilde{\mathbf{G}}, \mathbf{G})$ contain the added, removed, or modified genes. Their fitness difference on a shared batch is $\delta J = \widehat{J}(\widetilde{\mathbf{G}}) - \widehat{J}(\mathbf{G})$. Single-gene edits receive direct credit. For multi-gene edits, the activation $a_i(\tau) \in [0,1]$ and linked verifier evidence $q_i(\tau) \geq 0$ define

$$\rho_i = \frac{\epsilon + \sum_{\tau \in \mathcal{B}} a_i(\tau) q_i(\tau)}{|\Delta| \epsilon + \sum_{g_j \in \Delta} \sum_{\tau \in \mathcal{B}} a_j(\tau) q_j(\tau)}, \qquad g_i \in \Delta, \tag{11}$$

where $\epsilon > 0$ assigns nonzero responsibility when the traces are inconclusive. Persistent credit is updated as $c_i^{(t+1)} = (1-\eta) c_i^{(t)} + \eta \rho_i \delta J$, where $\eta \in (0,1]$, while unedited genes retain their credit and imported genes retain their source lineage identifiers.

Credit guides edit proposals but does not determine population selection. Compatible gene–operator pairs $(i, o) \in \mathcal{A}(\mathbf{G})$ are sampled according to

$$p_t(i,o) \propto \exp\left(\left[c_i^{(t)} + \lambda \sqrt{\frac{\log(1+t)}{1 + n_{i,o}^{(t)}}} \right] / T \right), \tag{12}$$

where $n_{i,o}^{(t)}$ counts prior applications, $\lambda \geq 0$ controls exploration, and $T > 0$ is the temperature.

At each generation, EvoGenome compiles and evaluates all genomes on a sampled batch, selects parents based on fitness and diversity, and proposes edits using Eq. (12). Invalid offspring are rejected before agent execution. Valid offspring are evaluated on the parent batch for credit updates. Selection retains at most N genomes from the parents and offspring, prioritizing task fitness and preserving distinct edited gene sets among tied candidates. After exhausting the evaluation budget, EvoGenome recompiles the highest-fitness valid genome in a clean workspace and returns the resulting package. The Appendix provides procedural details.

4 Experiments

We evaluate typed genotype–phenotype evolution under a fixed candidate-execution budget. The experiments examine four questions: whether EvoGenome improves held-out task performance,

Initialization	Target Pass Rate ↑	Gen. to 95% Final ↓	Regression ↓
Target-Only Evolution	61.8±2.0	6.4±0.7	8.4±1.1
Free-Form File Transfer	57.6±2.5	6.0±0.9	14.8±1.8
Complete-Skill Transfer	59.7±2.2	5.6±0.8	11.5±1.5
Recovery-Gene Transfer	63.2±1.8	4.8±0.6	7.6±0.9
Verifier-Gene Transfer	62.6±1.9	5.0±0.6	7.9±1.0
In-Domain EvoGenome	**64.5±1.6**	**4.4±0.5**	**7.2±0.9**

Table 3: Cross-domain transfer and convergence over five seeds. Abbreviation: Gen. = Generations.

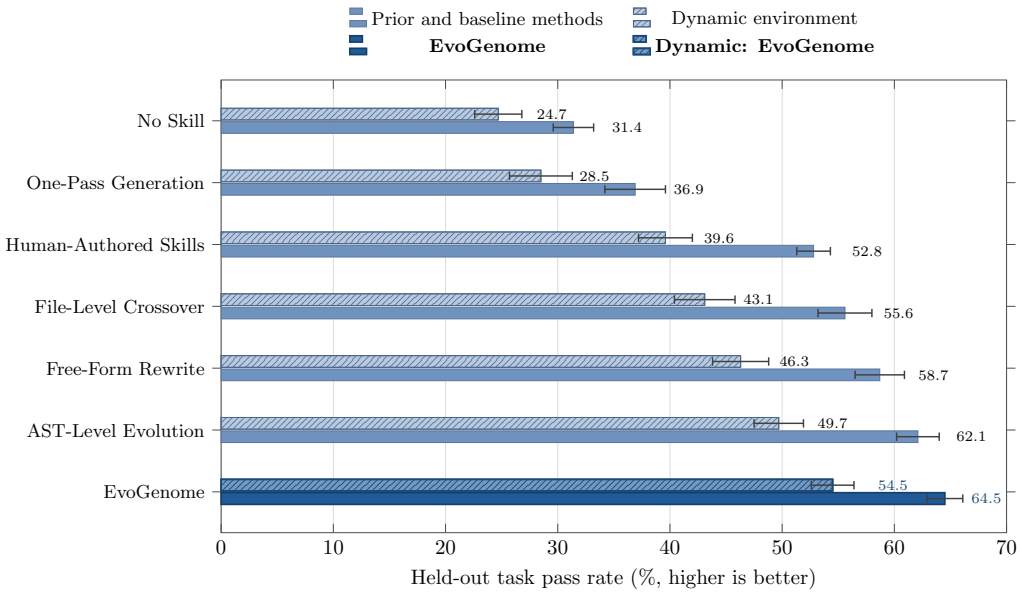

Figure 3: Held-out task pass rates under an identical candidate-execution budget. Bars show the mean and error bars show one standard deviation over five evolutionary seeds on SkillsBench and the dynamic environment.

whether it produces more valid and useful offspring, which representational components contribute to its performance, and whether individual genes transfer more reliably than complete skill artifacts.

4.1 Evaluation Protocol

We evaluate EvoGenome on SkillsBench and a dynamic environment constructed from held-out command-line tasks. SkillsBench covers heterogeneous professional workflows with deterministic verifiers. When an official partition is available, we retain it and reserve 20% of the development tasks for evolutionary selection. Mutation policies, lineage-credit estimates, and fitness evaluations never access the held-out test tasks. The dynamic environment varies tool availability, argument schemas, file locations, and recoverable failures across episodes. This setting tests whether evolved skills encode reusable control logic across harness configurations under varied command interface conditions.

To reduce sensitivity to incidental episode ordering, each method receives the same task sequence within a seed, while sequences differ across seeds. Selection decisions use only development feedback, and held-out scores are computed once after evolution terminates. This separation ensures that repeated candidate testing cannot indirectly tune representations or operators to test-specific verifier outcomes.

The primary metric is task pass rate, defined as the fraction of episodes that satisfy all deterministic verifiers. We also measure syntactic validity, executability, fitness-improving mutation rate, and regression rate. A candidate is valid if it materializes into a well-formed skill package. It is executable if all declared dependencies load and execution reaches a terminal state. A mutation is improving if its validation fitness exceeds that of its parent under paired task seeds. Regression is the fraction of validation tasks solved by the parent but failed by the offspring. We report the mean$\pm$standard deviation over five independent evolutionary seeds.

Overall, paired seeds support comparisons between each offspring and its parent, limiting variance caused by task sampling. Failures encountered during compilation are recorded separately from failures during execution, enabling the analysis to distinguish representation-level incompatibility from ineffective but runnable behavior.

We compare EvoGenome with a no-skill agent, human-authored skills, one-pass skill generation, free-form iterative rewriting, file-level crossover, and AST-level evolution. Free-form rewriting can modify any file. File-level crossover exchanges complete files without semantic compatibility checks. AST-level evolution restricts code edits to valid subtrees but does not assign types to triggers, instructions, recovery policies, or verifiers. All evolutionary methods use the same backbone model, observations, population size, number of generations, candidate evaluations, and oracle calls. EvoGenome uses a deterministic compiler to materialize typed trigger, precondition, planning, tool-policy, executable, verification, recovery, and transfer genes.

4.2 Performance and Offspring Quality

Table 1 and Figure 3 report held-out performance under identical budgets. EvoGenome achieves a SkillsBench pass rate of 64.5%, exceeding free-form rewriting by 5.8 percentage points, file-level crossover by 8.9 points, and AST-level evolution by 2.4 points. In the dynamic environment, EvoGenome reaches 54.5%, 4.8 points above the strongest alternative. The smaller margin over AST-level evolution on SkillsBench indicates that structured code edits already provide benefits when behavior is concentrated in executable modules. The larger gain in the dynamic environment shows the value of coordinating tool policy, verification, and recovery across artifacts.

The process metrics account for these performance differences. Free-form rewriting produces 73.8% valid offspring, of which 65.7% execute successfully, because a single rewrite can disrupt triggers, instructions, scripts, and recovery behavior simultaneously. File-level crossover preserves individual files but can connect incompatible preconditions and postconditions. AST-level evolution preserves code syntax without enforcing consistency between executable modules and natural-language control policies. EvoGenome checks compatibility before environment execution, producing 98.1% valid and 93.6% executable offspring. Compatibility filtering occurs before costly execution, so malformed combinations consume neither task episodes nor verifier time.

EvoGenome also preserves behavioral exploration: 61.7% of its valid offspring produce measurable trace changes, compared with 64.9% for free-form rewriting. At the same time, EvoGenome increases the improving-mutation rate from 14.8% to 23.9% and reduces regression from 16.4% to 7.2%. Typed variation therefore retains behavioral diversity while reducing destructive interactions among unrelated components. On tasks dominated by a single self-contained script, EvoGenome and AST-level evolution differ by less than one point. The margin increases when success requires coordinated tool selection, output verification, and recovery.

4.3 Representation and Search Operators

Table 2 isolates the contributions of typed interfaces, lineage credit, crossover, non-textual genes, and genome granularity. Removing type constraints lowers validity from 98.1% to 82.7%, increases regression by 6.9 points, and reduces pass rate by 6.4 points. Partitioning a skill into named regions alone is insufficient; compatibility constraints between inherited components account for the observed gains. This comparison distinguishes structural decomposition from the stronger guarantee supplied by explicit semantic typing.

Lineage credit primarily improves search efficiency. Disabling it leaves validity nearly unchanged at 97.8%, decreases the improving-mutation rate from 23.9% to 18.1%, and lowers the pass rate to 60.2%. Without evidence from paired traces, the search more frequently modifies genes unrelated to

the observed failure. Removing crossover reduces the pass rate by 3.1 points, indicating that typed recombination combines independently discovered recovery and verification behavior. Restricting evolution to textual genes yields a pass rate of 57.3%, below the 58.7% achieved by free-form rewriting, because instructions cannot co-evolve with executable modules and their contracts.

Genome granularity exhibits a non-monotonic trade-off. Coarse genes weaken attribution and make recombination similar to file exchange, yielding a pass rate of 59.6%. Fine-grained genes maintain high validity but constrain coordinated changes across coupled decisions, reaching 62.7%. The default schema attains 64.5%, indicating that semantic interfaces provide a more effective unit of inheritance than whole files or minimal textual fragments under this setting.

4.4 Cross-Domain Transfer and Efficiency

We transfer recovery genes, verifier genes, complete skills, or free-form files across domains that share interaction patterns but differ in tools, formats, and content. Transfer adapters map declared types and tool names without modifying the imported policy before its first target evaluation. All conditions receive the same target-domain optimization budget.

The transfer results indicate that portability depends on isolating behavior whose contract remains stable across domains. Recovery and verification satisfy this condition more often than complete planning stacks, which retain assumptions about source-specific observations and actions.

Typed transfer improves adaptation and stability. Recovery-gene initialization raises the target pass rate from 61.8% to 63.2% and reduces convergence time from 6.4 to 4.8 generations. Verifier-gene transfer reaches 62.6% after 5.0 generations. Complete-skill and free-form file transfer underperform target-only evolution because source-specific triggers and planning assumptions interfere with the target interface. The gene-transfer conditions remain below the 64.5% pass rate of in-domain EvoGenome, showing that reusable modules accelerate adaptation while target-specific evolution remains necessary.

EvoGenome reaches 95% of its final fitness in 4.4 generations, compared with 6.1 generations for free-form rewriting. Compilation and compatibility checking add 4.7% per-candidate overhead. EvoGenome incurs 27% fewer failed environment executions and completes 11% faster under a fixed wall-clock budget. Under the same evaluation budget, typed and attributable variation produces more executable, behaviorally meaningful, and fitness-improving candidates, leading to higher final task performance.

5 Conclusion

We presented EvoGenome, a genotype–phenotype framework that represents agent skills as typed, heritable functional units and supports controlled evolution through semantic operators, executable materialization, lineage-based credit attribution, and cross-domain transplantation. Experiments on skill evolution and dynamic tool use established that genome-level variation improves candidate validity, executability, behavioral innovation, and fitness while reducing regressions relative to free-form and file-level alternatives. Ablations and transplantation studies further confirmed the importance of type constraints, attributable lineage information, compositional operators, and non-textual genes, validating EvoGenome as an effective foundation for reliable skill evolution and reusable module transfer.

Limitations

Our evaluation uses a fixed agent backbone and two benchmark settings, so results may vary with other models, tool ecosystems, or task distributions. The current genome schema is manually specified and may require modest adaptation for domains with substantially different procedural structures. In addition, gene-credit estimates rely on finite execution batches and observable trace evidence, which can introduce noise when multiple genes interact or when relevant behaviors are only weakly reflected in recorded trajectories.

LLM Assistance Disclosure

We disclose that LLM-based writing assistants were used during the preparation of this manuscript, strictly for the purpose of language polishing. These tools were used to improve the readability of the text, refine technical descriptions, and correct grammatical and stylistic issues. The research contributions of this paper—including the compositional skill genome framework, theoretical analysis, and experimental validation—were entirely conceived and executed by the human authors. LLMs were not used to generate research ideas, develop algorithms, produce experimental data, or interpret results. All AI-assisted modifications were reviewed by the authors to ensure correctness. The authors are solely accountable for the content of this work.

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, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025. URL https://arxiv.org/abs/2507.19457.

Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. Evoskill: Automated skill discovery for multi-agent systems. *arXiv preprint arXiv:2603.02766*, 2026. URL https://arxiv.org/abs/2603.02766.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*, 2023. URL https://arxiv.org/abs/2305.17126.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 2002. URL https://doi.org/10.1109/4235.996017.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023. URL https://arxiv.org/abs/2309.16797.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 2012. URL https://doi.org/10.1109/TSE.2011.104.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Evoprompt: Connecting llms with evolutionary algorithms yields powerful prompt optimizers. *arXiv preprint arXiv:2309.08532*, 2023. URL https://arxiv.org/abs/2309.08532.

Omar 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 preprint arXiv:2310.03714*, 2023. URL https://arxiv.org/abs/2310.03714.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992. URL https://mitpressbookstore.mit.edu/book/9780262527910.

John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. The MIT Press, 1994. URL https://mitpress.mit.edu/9780262111898/genetic-programming-ii/.

Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 2011. URL https://doi.org/10.1162/EVCO_a_00025.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, Chujun Tao, Binxu Li, Xuandong Zhao, Hejia Geng, Xiaojun Wu, Junwei Zhou, Xiaokun Chen, Hanwen Xing, Yubo Li, Qunhong Zeng, Di Wang, Yuanli Wang, Roey Ben Chaim, Penghao Jiang, Haotian Shen, Luyang Kong, Xinyi Liu, Runhui Wang, Xuanqing Liu, Jiachen Li, Xin Lan, Yueqian Lin, Wengao Ye, Junwei He, Songlin Li, Yue Zhang, Yipeng Gao, Yijiang Li, Ze Ma, Liqiang Jing, Tianyu Wang, Kaixin Li, Yiqi Xue, Haoran Lyu, Yizhuo He, Yuchen Tian, Shutong Wu, Bowei Wang, Yixuan Gao, Bo Chen, Litong Liu, Sikai Cheng, Jiajun Bao, Shuaicheng Tong, Shuwen Xu, Terry Yue Zhuo, Tinghan Ye, Qi Qi, Miao Li, Longtai Liao, Zelin Tan, Chang Shi, Xilin Tang, Srinath Tankasala, Boqin Yuan, Yaoyao Qian, Jianhong Tu, Chenguang Wang, Yizhou Sun, Wei Wang, Aaron Taylor, Ziyue Yang, Changkun Guan, Zhikang Dong, Xinyu Zhang, Steven Dillmann, Han chung Lee, and Dawn Song. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026. URL https://arxiv.org/abs/2602.12670.

Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016. URL https://doi.org/10.1145/2837614.2837617.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023. URL https://arxiv.org/abs/2303.17651.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015. URL https://arxiv.org/abs/1504.04909.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023. URL https://arxiv.org/abs/2303.09014.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023. URL https://arxiv.org/abs/2305.15334.

Mitchell A. Potter and Kenneth A. De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 2000. URL https://doi.org/10.1162/106365600568086.

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. *arXiv preprint arXiv:2305.03495*, 2023. URL https://arxiv.org/abs/2305.03495.

Justin K. Pugh, Lisa B. Soros, and Kenneth O. Stanley. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 2016. URL https://www.frontiersin.org/journals/robotics-and-ai/articles/10.3389/frobt.2016.00040/full.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023. URL https://arxiv.org/abs/2302.04761.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *arXiv preprint arXiv:2303.17580*, 2023. URL https://arxiv.org/abs/2303.17580.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023. URL https://arxiv.org/abs/2303.11366.

Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 2002. URL https://doi.org/10.1162/106365602320169811.

Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, 2009. URL https://doi.org/10.1162/artl.2009.15.2.15202.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023. URL https://arxiv.org/abs/2305.16291.

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, 2009. URL https://doi.org/10.1109/ICSE.2009.5070536.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*, 2023. URL https://arxiv.org/abs/2309.03409.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022. URL https://arxiv.org/abs/2210.03629.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic "differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024. URL https://arxiv.org/abs/2406.07496.

Genrui Zhang, Erle Zhu, Jinfeng Zhou, Caiyan Jia, and Hongning Wang. Skillevolver: Skill learning as a meta-skill. *arXiv preprint arXiv:2605.10500*, 2026a. URL https://arxiv.org/abs/2605.10500.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, Xue (Steve) Liu, Xiaoxiao Li, and Philip S. Yu. Coevoskills: Self-evolving agent skills via co-evolutionary verification. *arXiv preprint arXiv:2604.01687*, 2026b. URL https://arxiv.org/abs/2604.01687.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. *arXiv preprint arXiv:2308.10144*, 2023. URL https://arxiv.org/abs/2308.10144.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023. URL https://arxiv.org/abs/2310.04406.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910*, 2022. URL https://arxiv.org/abs/2211.01910.

APPENDIX

A LINEAGE-AWARE EVOLUTION AND CREDIT ASSIGNMENT PROCEDURE

This section specifies the order of compilation, execution, offspring evaluation, lineage-credit updates, and population selection. Persistent gene credit guides edit proposals, while genome fitness determines population selection.

A.1 BATCH-ALIGNED FITNESS AND EDIT RESPONSIBILITY

Let $\mathcal{B} = \{\xi_b\}_{b=1}^{|\mathcal{B}|}$ be a finite batch sampled from $\mathcal{D}$. Executing a valid genome $\mathbf{G}$ on task ξ_b produces a trajectory

$$\tau_b \sim P(\cdot \mid \pi_\theta, \mathcal{C}(\mathbf{G}), \xi_b)$$

with reward $r(\tau_b, \xi_b)$. The batch fitness of $\mathbf{G}$ is

$$\widehat{J}_{\mathcal{B}}(\mathbf{G}) = \frac{1}{|\mathcal{B}|} \sum_{b=1}^{|\mathcal{B}|} r(\tau_b, \xi_b). \tag{13}$$

A parent $\mathbf{G}$ and its offspring $\widetilde{\mathbf{G}}$ are evaluated on the same task batch when computing the fitness difference

$$\delta J_{\mathcal{B}} = \widehat{J}_{\mathcal{B}}(\widetilde{\mathbf{G}}) - \widehat{J}_{\mathcal{B}}(\mathbf{G}). \tag{14}$$

This shared-batch comparison associates the credit update with the applied edit while controlling for changes in the sampled tasks.

Let

$$\Delta = \Delta(\widetilde{\mathbf{G}}, \mathbf{G})$$

denote the set of genes added, removed, or modified by the edit transaction. If $|\Delta| = 1$, the unique edited gene receives responsibility one. If $|\Delta| > 1$, the responsibility of each $g_i \in \Delta$ is

$$\rho_i = \frac{\epsilon + \sum_{b=1}^{|\mathcal{B}|} a_i(\tau_b) q_i(\tau_b)}{|\Delta|\epsilon + \sum_{g_j \in \Delta} \sum_{b=1}^{|\mathcal{B}|} a_j(\tau_b) q_j(\tau_b)}, \qquad g_i \in \Delta. \tag{15}$$

Since $\epsilon > 0$, the denominator remains positive when no edited gene has activation-linked verifier evidence. The responsibilities satisfy

$$\sum_{g_i \in \Delta} \rho_i = 1. \tag{16}$$

The observed fitness difference is therefore distributed across exactly the genes modified by the transaction. Their persistent credits are updated as

$$c_i^{(t+1)} = (1 - \eta) c_i^{(t)} + \eta \rho_i \delta J_{\mathcal{B}}, \qquad g_i \in \Delta, \tag{17}$$

while each unedited gene $g_k \notin \Delta$ retains its credit:

$$c_k^{(t+1)} = c_k^{(t)}. \tag{18}$$

For crossover or transplantation, an imported gene retains its source lineage identifier and source credit before the update in Eq. (17) is applied. A duplicated gene receives the new lineage identifier assigned by the duplication operator.

A.2 Ordered Evolutionary Procedure

For a valid genome $\mathbf{G}$, let $\mathcal{A}(\mathbf{G})$ denote its compatible gene–operator pairs. At generation t, a pair $(i, o) \in \mathcal{A}(\mathbf{G})$ is sampled according to

$$p_t(i, o) = \frac{\exp\left(\left[c_i^{(t)} + \lambda \sqrt{\frac{\log(1+t)}{1+n_{i,o}^{(t)}}}\right] / T\right)}{\sum_{(j,o') \in \mathcal{A}(\mathbf{G})} \exp\left(\left[c_j^{(t)} + \lambda \sqrt{\frac{\log(1+t)}{1+n_{j,o'}^{(t)}}}\right] / T\right)}. \tag{19}$$

The distribution is normalized over the compatible pairs of the selected parent. After applying the sampled operator, the resulting genome is compiled and statically validated before agent execution. An invalid offspring therefore consumes no trajectory evaluation and produces no fitness-difference credit update. For a valid offspring, the application count $n_{i,o}^{(t)}$ records the attempted gene–operator application, and persistent credit is updated after completing the required shared-batch evaluation.

Algorithm 1 specifies the complete procedure. One evaluation corresponds to executing a compiled genome on one task from a finite batch. The evaluation budget defines the stopping condition, so an offspring is evaluated only if the remaining budget can complete the prescribed comparison.

The procedure keeps the policy parameters θ fixed. Its evolving state comprises the population genomes, lineage records, persistent gene credits, and gene–operator application counts. Credit controls subsequent edit proposals through Eq. (19). Population selection uses the fitness and diversity of valid evaluated genomes and does not use gene credit.

Algorithm 1 Lineage-aware type-safe evolution

Require: Initial population $\mathcal{P}^{(0)} = \{\mathbf{G}_n^{(0)}\}_{n=1}^{N}$; task distribution $\mathcal{D}$; frozen policy π_θ; compiler $\mathcal{C}$; operator set $\mathcal{O}$; credit parameters η, ϵ, λ, and T; evaluation budget
Ensure: Compiled skill package associated with the highest-fitness valid genome
Initialize generation index $t \leftarrow 0$
Initialize lineage records, persistent gene credits $c_i^{(0)}$, and application counts $n_{i,o}^{(0)}$ as specified by the initial population
Initialize the number of consumed evaluations to zero
while the evaluation budget is not exhausted **do**
    Sample a finite task batch $\mathcal{B} \sim \mathcal{D}$
    for all $\mathbf{G} \in \mathcal{P}^{(t)}$ whose batch fitness on $\mathcal{B}$ is required **do**
        Compile $\mathcal{S}_{\mathbf{G}} \leftarrow \mathcal{C}(\mathbf{G})$
        if $\mathrm{Valid}(\mathbf{G})$ and compilation succeeds **then**
            Execute the frozen policy with $\mathcal{S}_{\mathbf{G}}$ on each $\xi_b \in \mathcal{B}$
            Compute $\widehat{J}_{\mathcal{B}}(\mathbf{G})$ using Eq. (13)
            Record the resulting trajectories and increase the consumed-evaluation count
        else
            Mark $\mathbf{G}$ invalid and do not execute the agent
        end if
    end for
    Select parent genome or genomes from the valid members of $\mathcal{P}^{(t)}$ according to fitness and diversity
    Choose a parent $\mathbf{G}$ to which the next edit is attributed
    Construct the compatible set $\mathcal{A}(\mathbf{G})$
    Sample $(i, o) \in \mathcal{A}(\mathbf{G})$ according to Eq. (19)
    Apply o to its required one or two selected parents and obtain candidate $\widetilde{\mathbf{G}}$
    Record the operator, parent lineage, and edited set $\Delta(\widetilde{\mathbf{G}}, \mathbf{G})$
    Increment the application count associated with (i, o)
    Compile $\widetilde{\mathcal{S}} \leftarrow \mathcal{C}(\widetilde{\mathbf{G}})$ in a clean workspace
    if $\mathrm{Valid}(\widetilde{\mathbf{G}})$ is false or compilation fails **then**
        Mark $\widetilde{\mathbf{G}}$ invalid and reject it without agent execution
    else if the remaining budget permits evaluation on the shared parent batch **then**
        Execute $\widetilde{\mathcal{S}}$ on every task in the same batch $\mathcal{B}$ used for $\mathbf{G}$
        Compute $\widehat{J}_{\mathcal{B}}(\widetilde{\mathbf{G}})$ and $\delta J_{\mathcal{B}}$ using Eqs. (13) and (14)
        Set $\rho_i \leftarrow 1$ if $|\Delta(\widetilde{\mathbf{G}}, \mathbf{G})| = 1$; otherwise compute all responsibilities using Eq. (15)
        Update the credits of edited genes using Eq. (17)
        Preserve the credits of unedited genes using Eq. (18)
        Add $\widetilde{\mathbf{G}}$ to the valid offspring set and increase the consumed-evaluation count
    else
        Terminate the evolutionary loop without partially evaluating $\widetilde{\mathbf{G}}$
    end if
    Form $\mathcal{P}^{(t+1)}$ by retaining at most N valid genomes from the parents and evaluated offspring, prioritizing task fitness and preserving distinct edited gene sets among tied candidates
    $t \leftarrow t + 1$
end while
Let $\mathbf{G}^\star$ be the highest-fitness valid genome among the evaluated genomes
Recompile $\mathcal{S}_{\mathbf{G}^\star} \leftarrow \mathcal{C}(\mathbf{G}^\star)$ in a clean workspace
return $\mathcal{S}_{\mathbf{G}^\star}$

B Environment Details

B.1 Datasets and Task Settings

We evaluate all methods on SkillsBench and a dynamic command-line tool-use environment. SkillsBench contains heterogeneous professional tasks that require an agent to follow reusable

skill instructions, invoke tools, manipulate files, execute scripts, and produce artifacts satisfying task-specific deterministic verifiers. We retain the official dataset partition whenever available. Twenty percent of the development tasks are sampled once as the evolution split, while the remaining development tasks are used for model and search-parameter selection. The official test split is held out until final evaluation.

The dynamic environment consists of command-line tasks derived from held-out workflow templates. Each task requires multi-step interaction with a sandboxed file system and a task-dependent set of executable tools. Across episodes, we randomize tool availability, command aliases, argument schemas, input-file locations, output directories, and recoverable execution failures. Templates used for test episodes are disjoint from those used during evolution. Each test template is instantiated with five independently sampled environment configurations, and all methods are evaluated on identical configurations and random seeds.

A task instance provides a natural-language instruction, observable input files, a tool manifest, and an executable verifier. The agent must return the requested terminal artifact rather than a free-form short answer. The system prompt specifies the available skill package and requires the agent to inspect intermediate results, execute the relevant tools, and verify the final artifact before termination. No task-specific demonstrations are included. The agent context contains only the current task, the materialized skill package, and observations generated within the current episode.

Evolution and model selection never access test instructions, test artifacts, verifier source code, or execution traces from the held-out split. Fitness is computed exclusively on the evolution split. To reduce contamination risk, we additionally remove development instances whose normalized instructions have a token-level Jaccard similarity above 0.8 with a test instruction, and deduplicate task templates using exact hashes of their canonicalized verifier specifications.

B.2 Preprocessing and Evaluation

Text files are decoded as UTF-8, Unix line endings are enforced, and task paths are replaced with workspace-relative paths before agent execution. JSON and YAML artifacts are parsed and serialized into canonical forms with sorted keys. Whitespace outside quoted strings is ignored when comparing structured text. Numeric fields are compared after conversion to double precision, using the tolerance declared by the task verifier or an absolute tolerance of 10^{-6} when no tolerance is specified. Sets and unordered collections are sorted before comparison, while list order is preserved when it is semantically relevant. For tasks permitting multiple valid outputs, an episode is counted as successful if its artifact satisfies any complete reference specification.

The primary metric is task pass rate,

$$\mathrm{PassRate} = \frac{1}{|\mathcal{T}|} \sum_{\xi \in \mathcal{T}} \mathbb{I}[R_\xi(y_H) = 1] \,,$$

where success requires all deterministic verifier checks associated with an episode to pass. Partial verifier credit is used only as validation fitness during evolution and is not included in the reported test pass rate. Episodes that exceed the interaction limit, fail to produce the required artifact, or terminate with an uncaught tool error receive a score of zero.

We report four complementary offspring-quality metrics. A candidate is *valid* if its genome satisfies the schema, dependency, interface, resource, and static-parsing constraints and the compiler produces a well-formed skill package. A valid candidate is *executable* if all declared dependencies load successfully and the agent reaches a terminal state in a clean sandbox. An offspring is an *improving mutation* if its validation fitness is strictly higher than that of its parent on the same paired task instances and environment seeds. The regression rate is the fraction of paired validation instances solved by the parent but failed by the offspring. Behavioral innovation is recorded when an offspring differs from its parent in at least one normalized tool-call sequence, activated gene, verifier outcome, or terminal artifact hash.

Each stochastic experiment is repeated with five independent evolution seeds. We report the mean and sample standard deviation across seeds. Parent–offspring comparisons use paired task batches and paired environment seeds to reduce execution variance. Final test evaluation uses the best valid genome selected solely by development fitness and recompiles it in a fresh workspace. Statistical com-

parisons use a paired two-sided permutation test over task-level outcomes with $10{,}000$ permutations; no significance test is used for selecting methods or hyperparameters.

The evaluated baselines are a frozen agent without a skill, human-authored skills, one-pass skill generation, free-form iterative rewriting, file-level crossover, and AST-level evolution. All evolutionary methods use the same backbone, initial skill packages, observations, validation tasks, population size, number of generations, candidate-execution budget, and language-model call budget. Failed compilation does not consume an environment execution but is counted as an evaluated candidate.

B.3 Implementation and Hardware

We use `Qwen2.5-Coder-32B-Instruct` as the frozen agent backbone and host it locally with `vLLM`. The same checkpoint is used for task execution, mutation proposal, crossover completion, failure diagnosis, and generation of missing textual or executable gene content. Model weights remain frozen throughout all experiments. Deterministic agent evaluations use greedy decoding with temperature 0, while edit proposals use temperature 0.7, top-$p = 0.95$, and a maximum of $2{,}048$ generated tokens. The maximum context length is $32{,}768$ tokens. Each task permits at most 40 agent–environment interactions and 20 tool executions.

Skills are represented by eight top-level gene types: trigger, precondition, planning, tool policy, executable module, postcondition and verifier, recovery, and adaptation. The compiler materializes each valid genome into metadata, workflow instructions, scripts, templates, and resource files. Python modules are checked with `compileall`, shell scripts with `bash -n`, JSON files with a strict parser, and YAML files with safe loading. Every candidate executes in an isolated container with read-only benchmark inputs, a writable temporary workspace, no network access, two CPU cores, 8 GB of memory, and a 10-minute wall-clock limit.

Evolution uses a population of $N = 16$ genomes for eight generations. Each generation evaluates 16 offspring, giving a budget of 128 candidate evaluations per run, excluding the shared initial-population evaluation. Validation fitness is estimated on a batch of 32 tasks sampled without replacement from the evolution split. Parent and offspring are reevaluated on the same batch whenever lineage credit is updated. Selection retains the highest-fitness candidates, with behavioral diversity used to break ties between candidates whose fitness differs by less than 10^{-4}.

For each offspring, one compatible gene–operator pair is sampled using the lineage-aware distribution in Eq. (12). We set the credit update rate to $\eta = 0.2$, the smoothing constant to $\epsilon = 10^{-3}$, the exploration coefficient to $\lambda_s = 0.5$, and the sampling temperature to $T = 0.8$. Unary typed mutation is selected with probability 0.50, homologous crossover with probability 0.25, recovery or verifier transplantation with probability 0.10, module duplication and specialization with probability 0.10, and dead-gene pruning with probability 0.05. Crossover exchanges one compatible gene or one connected subgraph containing at most three genes. Invalid variants are rejected after static validation without automatic semantic repair.

The coarse-genome ablation merges planning with tool policy, executable modules with postconditions, and recovery with adaptation. The fine-grained ablation splits executable, verification, and recovery genes at function or handler boundaries while preserving their typed interfaces. The text-only ablation evolves trigger, planning, tool-policy, and recovery instructions but freezes scripts, templates, configurations, and other non-textual resources. The no-lineage-credit ablation samples compatible edits uniformly, and the no-type-constraints ablation retains only file-level syntax checks.

Experiments are implemented in Python 3.11 and executed on a Linux cluster. Each evolutionary run uses one node with eight NVIDIA A100 80 GB GPUs, two 64-core AMD EPYC processors, and 512 GB of system memory. Tensor parallelism is set to eight for model inference, with bfloat16 weights and a maximum continuous batching size of 32. Candidate environments run asynchronously on CPU workers, while language-model requests are serialized through the shared inference server. We fix Python, NumPy, PyTorch, environment-generation, and task-sampling seeds for each of the five runs and record all genomes, compiler outputs, tool calls, verifier results, lineage edges, and gene-credit updates for reproducibility.

Table 4: Behavioral diversity and effective offspring yield. Trace change is computed among valid offspring, while effective yield is computed over all generated candidates. Results are mean ± standard deviation over five seeds.

Method	Trace Change ↑	Unique Tool Sequences ↑	Gene/File Diversity ↑	Effective Yield ↑
Free-Form Rewrite	64.9±2.8	41.7±2.6	46.8±3.1	24.6±2.0
File-Level Crossover	56.8±2.5	38.9±2.4	43.5±2.7	27.3±1.8
AST-Level Evolution	58.6±2.1	40.8±2.0	49.2±2.5	38.7±1.7
EvoGenome	61.7±2.0	45.6±2.2	57.9±2.3	**48.1±1.6**

Table 5: SkillsBench pass rate by procedural complexity. Simple tasks require one dominant executable component, medium tasks require two or three coordinated components, and complex tasks require at least four components or explicit recovery.

Method	Simple ↑	Medium ↑	Complex ↑	Overall ↑
Human-Authored Skills	61.8±1.9	53.4±1.7	42.7±2.2	52.8±1.5
Free-Form Rewrite	66.2±2.1	59.1±2.4	49.5±2.8	58.7±2.2
File-Level Crossover	63.8±2.3	56.0±2.5	46.6±2.9	55.6±2.4
AST-Level Evolution	67.4±1.8	62.8±2.0	55.9±2.4	62.1±1.9
EvoGenome	**68.1±1.7**	**65.2±1.8**	**59.9±2.1**	**64.5±1.6**

C Additional Experimental Results

Behavioral innovation and effective offspring yield. Behavioral innovation is measured separately from candidate validity. An offspring is behaviorally distinct when its normalized action-trace distance from its parent exceeds 0.05. Effective yield denotes the fraction of all generated candidates that are executable, behaviorally distinct, and non-regressive. As shown in Table 4, EvoGenome achieves a trace-change rate of 61.7±2.0 and an effective yield of 48.1±1.6, compared with 64.9±2.8 and 24.6±2.0, respectively, for free-form rewriting.

Performance by skill complexity. Table 5 partitions SkillsBench tasks according to the number of procedural components required by the reference workflow. EvoGenome achieves the highest pass rate among the evaluated methods for simple, medium, and complex tasks, with rates of 68.1±1.7, 65.2±1.8, and 59.9±2.1, respectively.

Robustness to reduced execution budgets. Each evolutionary method was evaluated after consuming 25%, 50%, 75%, and 100% of the default candidate budget. As shown in Table 6, EvoGenome achieves the highest pass rate among the evaluated methods at each candidate-execution budget.

Robustness to dynamic environment shifts. The dynamic environment is decomposed into individual perturbation types. Table 7 shows that EvoGenome achieves the highest pass rate among the evaluated methods for tool removal, argument changes, path relocation, recoverable failures, and mixed shifts.

Operator-level analysis. Table 8 reports statistics for the individual EvoGenome operators. Typed mutation has the highest acceptance rate at 31.8±1.9, while homologous crossover has the highest fitness change among accepted offspring at +2.8±0.8 percentage points. Dead-gene pruning has an acceptance rate of 19.6±1.6, a fitness change of +0.6±0.3 percentage points, and a regression rate of 4.2±0.8.

Gene-type attribution. Edit outcomes are aggregated by the primary edited gene type to evaluate lineage credit across components. As shown in Table 9, recovery, executable, and postcondition genes have normalized credits of 0.85±0.05, 0.82±0.05, and 0.79±0.06, respectively. Trigger genes have a normalized credit of 0.34±0.07 and a regression rate of 10.9±1.5.

Table 6: SkillsBench pass rate under different candidate-execution budgets. The full budget contains 128 evaluated candidates per seed.

Method	32 Candidates	64 Candidates	96 Candidates	128 Candidates
Free-Form Rewrite	48.6±2.8	53.7±2.5	56.9±2.3	58.7±2.2
File-Level Crossover	47.9±2.7	51.5±2.6	54.0±2.5	55.6±2.4
AST-Level Evolution	50.7±2.3	56.8±2.1	60.1±2.0	62.1±1.9
EvoGenome	**53.8±2.1**	**59.8±1.9**	**62.9±1.7**	**64.5±1.6**

Table 7: Pass rate under individual dynamic-environment perturbations. Mixed-shift episodes contain two or more simultaneous perturbations.

Method	Tool Removal	Argument Change	Path Relocation	Recoverable Failure	Mixed Shift
Free-Form Rewrite	49.0±2.7	47.1±2.6	48.4±2.8	45.2±2.9	39.8±3.0
File-Level Crossover	46.2±2.8	44.0±2.9	45.1±2.7	41.7±3.1	37.0±3.2
AST-Level Evolution	52.4±2.4	50.8±2.3	51.6±2.5	48.1±2.7	43.5±2.8
EvoGenome	**56.8±2.1**	**54.7±2.0**	**55.9±2.2**	**55.6±2.3**	**49.4±2.5**

Out-of-distribution generalization. The final evolved skills were evaluated on unseen perturbation compositions and on tasks whose surface instructions were paraphrased without changing their verifiers. Table 10 shows that EvoGenome achieves the highest pass rate among the evaluated methods under instruction paraphrases, unseen tool aliases, unseen schemas, and composed perturbations.

Computation and failure cost. Table 11 reports normalized computation relative to free-form rewriting. EvoGenome has a pre-execution overhead of 4.7 ± 0.4, a failed-execution rate of 9.3 ± 1.4, relative tool calls of 0.84 ± 0.03, and a relative wall-clock time of 0.89 ± 0.03.

Statistical comparisons. Paired comparisons use per-task outcomes aggregated across matched evolutionary seeds. Table 12 reports bootstrap confidence intervals for pass-rate differences based on two-sided paired permutation tests. For both evaluation environments, the reported confidence intervals for comparisons between EvoGenome and each evolutionary baseline exclude zero.

D Prompt Templates

D.1 Overview

This appendix documents the prompt templates used for skill initialization, agent execution, evolutionary edit proposal, baseline construction, failure analysis, and ablation studies. Variables enclosed in angle brackets are instantiated at runtime. Unless otherwise stated, all methods use the same frozen language-model agent, task observations, tool specifications, and execution budget. Deterministic compilation, static validation, compatibility checking, and task evaluation are performed outside the language model.

D.2 Notation

We use the following placeholders throughout the templates:

- `<TASK_INSTRUCTION>`: the natural-language instruction associated with task ξ.
- `<OBSERVATIONS>`: the observable task inputs, including files, environment state, and prior tool outputs.
- `<AVAILABLE_TOOLS>`: the tools available in the current environment, together with their argument schemas.
- `<SKILL_PACKAGE>`: a materialized skill phenotype containing metadata, workflow instructions, scripts, templates, and resources.

Table 8: Operator-level outcomes within EvoGenome. Fitness change is measured in percentage points relative to the paired parent and is reported over accepted offspring.

Operator	Valid ↑	Accepted ↑	Fitness Change ↑	Regression ↓
Typed Mutation	98.4±0.5	31.8±1.9	+2.3±0.6	7.1±1.0
Homologous Crossover	97.5±0.8	28.6±2.1	+2.8±0.8	7.8±1.1
Recovery Transplant	96.8±1.0	27.4±2.3	+2.5±0.7	6.9±0.9
Duplication/Specialize	98.0±0.7	22.9±1.8	+1.7±0.6	8.0±1.2
Dead-Gene Pruning	99.6±0.3	19.6±1.6	+0.6±0.3	4.2±0.8

Table 9: Edit outcomes and normalized lineage credit by primary gene type. Credit is normalized within each seed before aggregation.

Gene Type	Edit Share	Improving Mut. ↑	Regression ↓	Normalized Credit ↑
Trigger	7.8±1.1	13.6±2.0	10.9±1.5	0.34±0.07
Precondition	9.5±1.3	18.4±1.8	7.7±1.2	0.49±0.08
Planning	16.7±1.6	22.1±1.6	8.1±1.1	0.63±0.07
Tool Policy	14.2±1.5	24.8±1.7	7.4±1.0	0.71±0.06
Executable	19.8±1.8	27.6±1.5	6.8±0.9	0.82±0.05
Postcondition	13.6±1.4	26.3±1.6	5.9±0.8	0.79±0.06
Recovery	14.9±1.5	28.1±1.8	5.6±0.9	0.85±0.05
Adaptation	3.5±0.8	20.7±2.3	7.2±1.4	0.58±0.09

- <GENOME>: the current SkillGenome $G = (g_{\text{trg}}, g_{\text{pre}}, g_{\text{pln}}, g_{\text{tool}}, g_{\text{exe}}, g_{\text{post}}, g_{\text{rec}}, g_{\text{adp}})$.
- <GENE>: the content and interface contract of the gene selected for mutation.
- <DEPENDENCY_CONTEXT>: the types, interfaces, contracts, and contents of genes adjacent to the selected gene.
- <EDITABLE_FILES>: the files that a baseline method is permitted to modify.
- <PARENT_A> and <PARENT_B>: parent skills, files, modules, or genomes used for crossover.
- <EXECUTION_TRACE>: the sequence of observations, actions, tool calls, and intermediate results produced during execution.
- <VERIFIER_OUTCOMES>: deterministic verifier results and associated diagnostics.
- <TERMINAL_ARTIFACTS>: artifacts produced at the end of an episode.
- <FAILURE_SUMMARY>: a structured summary of observed execution failures.
- <OPERATOR>: the selected evolutionary operator.
- <SOURCE_GENE> and <TARGET_CONTRACT>: the imported gene and target-domain interface used for transplantation.

D.3 Main Inference Prompt

Purpose. Execute a materialized skill package for a task using the frozen agent policy.

Inputs. The task instruction, current observations, available tools, and materialized skill package.

Expected Output. A sequence of valid tool calls followed by a concise terminal response. Task completion is determined from the terminal artifacts by deterministic verifiers.

Prompt.

```
You are an agent operating in a tool-enabled environment.

Task:
<TASK_INSTRUCTION>
```

Table 10: Out-of-distribution pass rate on held-out instruction and environment variants. No OOD condition is used during evolutionary selection. Abbreviation: OOD = out-of-distribution.

Method	Instruction Paraphrase	Unseen Tool Alias	Unseen Schema	Composed Perturbation
Free-Form Rewrite	56.4±2.4	48.1±2.7	45.8±2.8	38.7±3.0
File-Level Crossover	53.6±2.6	45.5±2.9	42.7±3.0	35.9±3.1
AST-Level Evolution	59.7±2.1	51.9±2.4	49.8±2.5	42.8±2.7
EvoGenome	**62.3±1.9**	**56.0±2.2**	**55.1±2.3**	**48.6±2.5**

Table 11: Computational efficiency under an identical candidate-evaluation budget. Wall-clock time and tool calls are normalized to free-form rewriting.

Method	Pre-execution Overhead	Failed Executions ↓	Relative Tool Calls ↓	Relative Wall-Clock ↓
Free-Form Rewrite	1.0±0.1	34.3±3.2	1.00±0.04	1.00±0.03
File-Level Crossover	1.3±0.1	28.1±3.0	0.96±0.04	0.97±0.04
AST-Level Evolution	3.1±0.3	12.6±1.8	0.91±0.03	0.93±0.03
EvoGenome	4.7±0.4	**9.3±1.4**	**0.84±0.03**	**0.89±0.03**

```
Current observations:
<OBSERVATIONS>

Available tools:
<AVAILABLE_TOOLS>

Loaded skill package:
<SKILL_PACKAGE>

Follow the loaded skill when its activation conditions are satisfied.
Before acting:
1. Check the skill preconditions against the current observations.
2. Construct a plan using the skill's planning procedure.
3. Select only tools that are currently available and use arguments that
   satisfy their declared schemas.

During execution:
1. Follow the skill's executable procedures and templates.
2. Inspect every tool result before continuing.
3. Apply the skill's postcondition and verification procedures to
   intermediate and terminal artifacts.
4. If an action fails or verification does not pass, apply the documented
   recovery procedure before attempting an alternative.
5. Do not claim success until all required postconditions are satisfied.

Use tools when required. Return a concise terminal response after completing
the task or after determining that no documented recovery action remains.
```

D.4 Baseline Prompts

D.4.1 One-Pass Skill Generation

Purpose. Generate a complete skill package in a single model call without iterative execution feedback.

Inputs. A description of the target task distribution, available tools, package format, and representative task specifications.

Table 12: Paired statistical comparisons between EvoGenome and evolutionary baselines. Differences are absolute percentage points. Confidence intervals use 10,000 stratified bootstrap samples.

Comparison	SkillsBench Δ	95% CI	Dynamic Δ	95% CI
EvoGenome vs. Free-Form Rewrite	$+5.8$	$[3.1, 8.6]$	$+8.2$	$[4.9, 11.4]$
EvoGenome vs. File-Level Crossover	$+8.9$	$[6.0, 11.8]$	$+11.4$	$[7.9, 14.8]$
EvoGenome vs. AST-Level Evolution	$+2.4$	$[0.7, 4.2]$	$+4.8$	$[2.1, 7.4]$

Expected Output. A complete skill package containing metadata, workflow instructions, scripts, templates, and required resources.

Prompt.

```
Create a reusable agent skill for the task distribution described below.

Task distribution:
<TASK_DISTRIBUTION>

Available tools and schemas:
<AVAILABLE_TOOLS>

Required package format:
<PACKAGE_SCHEMA>

Representative task specifications:
<TASK_SPECIFICATIONS>

The skill should:
- specify when it should be activated;
- identify required inputs and preconditions;
- provide a reliable execution plan;
- describe valid tool-selection and argument-construction behavior;
- include any executable modules needed by the workflow;
- verify intermediate results and terminal artifacts;
- define recovery behavior for anticipated failures; and
- avoid assumptions not supported by the task specification.

Return the complete package using exactly the required directory structure.
For each file, provide its relative path followed by its full contents.
Do not include commentary outside the package.
```

D.4.2 Free-Form Rewrite

Purpose. Produce an offspring by jointly revising any part of an existing multi-file skill package using execution feedback.

Inputs. The current skill package, failed task instances, execution traces, verifier outcomes, and editable files.

Expected Output. A complete replacement package in which any file may be modified.

Prompt.

```
Revise the skill package to improve performance on the observed failures.

Current skill package:
<SKILL_PACKAGE>
```

```
Editable files:
<EDITABLE_FILES>

Failed task instances:
<FAILED_TASKS>

Execution traces:
<EXECUTION_TRACE>

Verifier outcomes:
<VERIFIER_OUTCOMES>

Identify the behavior responsible for the failures and rewrite the package
accordingly. You may modify any editable file, including activation
instructions, planning logic, tool-use guidance, scripts, templates,
verification procedures, and recovery behavior.

Preserve behavior that succeeded unless changing it is necessary to resolve
the observed failures. Ensure that references, filenames, tool arguments,
and interfaces remain internally consistent.

Return the complete revised package. For each file, provide its relative path
followed by its full contents. Do not include analysis or commentary outside
the package.
```

D.4.3 FILE-LEVEL CROSSOVER

Purpose. Construct an offspring by selecting complete files from two parent skill packages without semantic compatibility checks.

Inputs. Two parent packages and the required package schema.

Expected Output. A crossover manifest assigning each offspring file to one parent, followed by the assembled package.

Prompt.

```
Construct one offspring skill package from the two parent packages below.

Parent A:
<PARENT_A>

Parent B:
<PARENT_B>

Required package schema:
<PACKAGE_SCHEMA>

For every required relative file path, select the complete file from either
Parent A or Parent B. Do not merge content within a file and do not rewrite
the selected file. Preserve the selected file verbatim.

Return:
1. a crossover manifest mapping each relative path to "A" or "B"; and
2. the resulting package, with each relative path followed by its contents.

Use the following manifest format:
{
  "files": {
    "<relative_path_1>": "A",
    "<relative_path_2>": "B"
```

```
  }
}
```

D.4.4 AST-LEVEL EVOLUTION

Purpose. Revise executable code through syntax-preserving abstract syntax tree mutations while leaving non-code skill components unchanged.

Inputs. The selected code file or subtree, relevant execution traces, verifier outcomes, and the callable interface that must be preserved.

Expected Output. A syntactically valid replacement subtree or code file with the declared interface unchanged.

Prompt.

```
Modify the executable code to address the observed failures.

Current code:
<CURRENT_CODE>

Selected AST subtree:
<SELECTED_SUBTREE>

Required callable interface:
<INTERFACE_CONTRACT>

Execution traces:
<EXECUTION_TRACE>

Verifier outcomes:
<VERIFIER_OUTCOMES>

Propose a syntax-valid replacement for the selected subtree. Preserve the
declared function names, parameters, return types, external resources, and
call sites. Do not modify natural-language instructions, triggers, tool
policies, verifier definitions, or recovery policies.

Return only the replacement code in the source language. Do not include
Markdown fences or explanatory text.
```

D.5 FEW-SHOT AND DEMONSTRATION PROMPTS

No few-shot demonstrations are used. Candidate generation, mutation, and execution prompts contain task-specific observations and execution evidence but no labeled input–output demonstrations.

D.6 REASONING AND RATIONALE PROMPTS

D.6.1 FAILURE ANALYSIS

Purpose. Convert paired execution evidence into a structured failure summary used to condition mutation proposals.

Inputs. The task instruction, execution trace, terminal artifacts, verifier outcomes, and active skill components.

Expected Output. A JSON object identifying the failure stage, evidence, implicated components, preserved behavior, and required behavioral change.

Prompt.

```
Analyze the execution failure using only the evidence provided below.

Task:
<TASK_INSTRUCTION>

Execution trace:
<EXECUTION_TRACE>

Terminal artifacts:
<TERMINAL_ARTIFACTS>

Verifier outcomes:
<VERIFIER_OUTCOMES>

Active skill components:
<ACTIVE_COMPONENTS>

Determine:
1. the earliest stage at which behavior became inconsistent with the task;
2. the direct evidence for that diagnosis;
3. the skill components that were active at that stage;
4. behavior that succeeded and should be preserved; and
5. the minimal behavioral change required to avoid the failure.

Do not propose unrelated improvements. Do not infer failures unsupported by
the trace or verifier outcomes.

Return valid JSON with exactly this schema:
{
  "failure_stage": "<trigger|precondition|planning|tool_policy|execution|postcondit⌋
  ↪  ion|recovery|environment>",
  "evidence": ["<trace or verifier evidence>"],
  "implicated_components": ["<component identifier>"],
  "preserve": ["<successful behavior to preserve>"],
  "required_change": "<minimal behavioral change>",
  "confidence": "<low|medium|high>"
}
```

D.6.2 Typed Gene Mutation

Purpose. Propose a localized mutation to a selected gene while preserving its typed interface and dependency contracts.

Inputs. The selected gene, its dependency context, failure summary, relevant traces, and operator specification.

Expected Output. A JSON mutation proposal containing the replacement content and an explicit interface-preservation record.

Prompt.

```
Propose one localized mutation to the selected SkillGenome gene.

Selected operator:
<OPERATOR>

Selected gene:
<GENE>
```

```
Gene type:
<GENE_TYPE>

Required input interface:
<INPUT_INTERFACE>

Required output interface:
<OUTPUT_INTERFACE>

Dependency context:
<DEPENDENCY_CONTEXT>

Observed failure:
<FAILURE_SUMMARY>

Relevant execution evidence:
<EXECUTION_TRACE>

Modify only the selected gene. Preserve its input and output interfaces unless
the operator explicitly authorizes a coordinated transaction over all affected
dependents. The edit should address the observed failure while preserving
successful behavior. Do not modify unrelated policies or introduce undeclared
tools, files, dependencies, or resources.

Return valid JSON with exactly this schema:
{
  "gene_id": "<selected gene identifier>",
  "gene_type": "<gene type>",
  "operation": "<replace|specialize|prune>",
  "replacement_content": "<complete replacement content>",
  "input_interface": <preserved input-interface object>,
  "output_interface": <preserved output-interface object>,
  "dependencies": ["<dependency identifier>"],
  "addresses": ["<evidence item from the failure summary>"]
}
```

D.6.3 COORDINATED TYPED MUTATION

Purpose. Update multiple coupled genes in one transaction when a valid local mutation requires interface changes.

Inputs. The selected connected subgraph, boundary contracts, failure summary, and execution evidence.

Expected Output. A JSON transaction containing all changed genes and the resulting boundary interfaces.

Prompt.

```
Propose one coordinated mutation over the selected connected genome subgraph.

Selected genes:
<SELECTED_SUBGRAPH>

Incoming boundary contracts:
<INCOMING_CONTRACTS>

Outgoing boundary contracts:
<OUTGOING_CONTRACTS>
```

```
Internal dependency graph:
<DEPENDENCY_GRAPH>

Observed failure:
<FAILURE_SUMMARY>

Relevant execution evidence:
<EXECUTION_TRACE>

Modify only genes in the selected subgraph. The resulting subgraph must remain
acyclic, satisfy every internal dependency, accept the declared incoming
interfaces, and provide the declared outgoing interfaces. Include every
dependent gene whose contract changes. Preserve successful behavior not
implicated by the evidence.

Return valid JSON with exactly this schema:
{
  "operation": "transactional_replace",
  "changed_genes": [
    {
      "gene_id": "<identifier>",
      "gene_type": "<type>",
      "replacement_content": "<complete replacement content>",
      "input_interface": <input-interface object>,
      "output_interface": <output-interface object>,
      "dependencies": ["<dependency identifier>"]
    }
  ],
  "incoming_contracts": <incoming-contract object>,
  "outgoing_contracts": <outgoing-contract object>
}
```

D.6.4 HOMOLOGOUS CROSSOVER PROPOSAL

Purpose. Select homologous genes or connected subgraphs from two parents for type-compatible crossover.

Inputs. Two parent genomes, candidate homologous regions, boundary interfaces, and lineage credit.

Expected Output. A JSON crossover proposal identifying exchanged regions without rewriting their contents.

Prompt.

```
Select one homologous crossover between the two parent SkillGenomes.

Parent A:
<PARENT_A>

Parent B:
<PARENT_B>

Candidate homologous regions:
<HOMOLOGOUS_REGIONS>

Boundary interfaces:
<BOUNDARY_INTERFACES>

Lineage credit:
<GENE_CREDIT>
```

```
Select either:
- a single-gene replacement with the same functional role and compatible
  boundary interfaces; or
- a connected-subgraph replacement whose incoming and outgoing contracts
  match those of the replaced subgraph.

Do not rewrite gene contents. Do not select a pair with incompatible types,
unsatisfied dependencies, or unmatched boundary contracts.

Return valid JSON with exactly this schema:
{
  "donor_parent": "<A|B>",
  "recipient_parent": "<A|B>",
  "donor_gene_ids": ["<gene identifier>"],
  "recipient_gene_ids": ["<gene identifier>"],
  "gene_types": ["<gene type>"],
  "incoming_contract_match": true,
  "outgoing_contract_match": true
}
```

D.7 Data Generation and Pseudo-Labeling Prompts

No prompt-based data generation or pseudo-labeling is used. Evolutionary selection and gene credit are computed from environment execution, paired task seeds, execution traces, and deterministic verifier outcomes.

D.8 Evaluation Prompts

No LLM-as-a-judge prompt is used. Task pass rate, validity, executability, improving-mutation rate, regression rate, and transfer performance are computed using deterministic compilers, runtime checks, and task-specific verifiers. The language model does not assign evaluation labels or scalar fitness values.

D.9 Ablation Prompt Variants

D.9.1 Without Type Constraints

Purpose. Generate gene-level edits without enforcing type, interface, or dependency compatibility before execution.

Inputs. The selected named genome region, failure summary, and execution evidence.

Expected Output. A replacement for the selected region. Compatibility is not asserted in the output.

Prompt.

```
Revise the selected skill region to address the observed failure.

Selected region:
<GENE>

Observed failure:
<FAILURE_SUMMARY>

Execution evidence:
<EXECUTION_TRACE>
```

```
Modify only the selected named region and preserve successful behavior when
possible. You may change its assumptions, references, inputs, outputs, or
dependencies. Compatibility with adjacent regions will not be checked before
materialization.

Return valid JSON:
{
  "region_id": "<selected region identifier>",
  "replacement_content": "<complete replacement content>"
}
```

D.9.2 WITHOUT LINEAGE CREDIT

Purpose. Propose a typed mutation without conditioning gene or operator selection on lineage-based credit.

Inputs. A uniformly sampled compatible gene–operator pair, dependency context, failure summary, and execution evidence.

Expected Output. The same structured mutation object as the main typed mutation prompt.

Prompt.

```
Propose one localized mutation to the sampled SkillGenome gene.

Sampled operator:
<OPERATOR>

Sampled gene:
<GENE>

Required input interface:
<INPUT_INTERFACE>

Required output interface:
<OUTPUT_INTERFACE>

Dependency context:
<DEPENDENCY_CONTEXT>

Observed failure:
<FAILURE_SUMMARY>

Relevant execution evidence:
<EXECUTION_TRACE>

Modify only the sampled gene. Preserve its declared interfaces and
dependencies. Address the observed failure without introducing undeclared
tools, files, dependencies, or resources. No historical gene-credit
information is available.

Return valid JSON with exactly this schema:
{
  "gene_id": "<sampled gene identifier>",
  "gene_type": "<gene type>",
  "operation": "<replace|specialize|prune>",
  "replacement_content": "<complete replacement content>",
  "input_interface": <preserved input-interface object>,
  "output_interface": <preserved output-interface object>,
  "dependencies": ["<dependency identifier>"],
```

```
  "addresses": ["<evidence item from the failure summary>"]
}
```

D.9.3 WITHOUT CROSSOVER

Purpose. Restrict evolution to unary mutation, specialization, transplantation, and pruning operators.

Inputs. One parent genome, the selected unary operator, failure summary, and dependency context.

Expected Output. A single-parent mutation proposal using the structured schema of the selected unary operator.

Prompt.

```
Propose one unary edit to the parent SkillGenome.

Parent genome:
<GENOME>

Selected unary operator:
<OPERATOR>

Selected gene or subgraph:
<SELECTED_REGION>

Dependency context:
<DEPENDENCY_CONTEXT>

Observed failure:
<FAILURE_SUMMARY>

Relevant execution evidence:
<EXECUTION_TRACE>

Use only the selected parent. Do not import, exchange, or combine content from
another genome. Preserve all required interfaces and dependencies, and modify
only the selected gene or connected subgraph.

Return the structured mutation object required by the selected unary operator.
Do not include explanatory text.
```

D.9.4 TEXT GENES ONLY

Purpose. Restrict evolutionary edits to textual trigger, precondition, planning, tool-policy, postcondition, recovery, and adaptation content.

Inputs. A selected textual gene, immutable executable modules, failure summary, and dependency context.

Expected Output. A replacement textual gene that does not modify executable code or non-textual resources.

Prompt.

```
Revise the selected textual gene to address the observed failure.

Selected textual gene:
```

```
<GENE>

Immutable executable modules and resources:
<IMMUTABLE_NON_TEXTUAL_GENES>

Dependency context:
<DEPENDENCY_CONTEXT>

Observed failure:
<FAILURE_SUMMARY>

Relevant execution evidence:
<EXECUTION_TRACE>

Modify only the textual content of the selected gene. Do not create, delete,
or change scripts, executable modules, templates, binary resources, schemas,
or callable interfaces. Preserve references to immutable components exactly.

Return valid JSON:
{
  "gene_id": "<selected gene identifier>",
  "gene_type": "<textual gene type>",
  "replacement_content": "<complete replacement text>",
  "dependencies": ["<unchanged dependency identifier>"]
}
```

D.9.5 COARSE GENOME

Purpose. Mutate a coarse region that combines multiple functional roles into a single heritable unit.

Inputs. A coarse genome region, its package-level boundary contract, failure summary, and execution evidence.

Expected Output. A complete replacement for the coarse region.

Prompt.

```
Revise the selected coarse genome region.

Coarse region:
<COARSE_REGION>

Functions contained in this region:
<CONTAINED_FUNCTIONS>

Package-level boundary contract:
<BOUNDARY_CONTRACT>

Observed failure:
<FAILURE_SUMMARY>

Relevant execution evidence:
<EXECUTION_TRACE>

Return a complete replacement for the region. You may jointly revise any
triggering, planning, tool-use, execution, verification, or recovery behavior
contained within it. Preserve the package-level boundary contract and do not
modify content outside the selected region.

Return valid JSON:
{
```

```
  "region_id": "<coarse region identifier>",
  "replacement_content": "<complete replacement content>",
  "boundary_contract": <preserved boundary-contract object>
}
```

D.9.6 FINE-GRAINED GENOME

Purpose. Mutate a minimal semantic fragment while preserving all local interfaces.

Inputs. A fine-grained fragment, its immediate consumers and providers, failure summary, and execution evidence.

Expected Output. A replacement for exactly one fragment with unchanged local interfaces.

Prompt.

```
Revise exactly one fine-grained genome fragment.

Selected fragment:
<FINE_GRAINED_FRAGMENT>

Immediate providers:
<PROVIDERS>

Immediate consumers:
<CONSUMERS>

Required local interface:
<LOCAL_INTERFACE>

Observed failure:
<FAILURE_SUMMARY>

Relevant execution evidence:
<EXECUTION_TRACE>

Modify only the selected fragment. Do not change adjacent fragments, dispatch
conditions, interfaces, dependencies, tool schemas, or shared resources.
Make the smallest edit that directly addresses the supplied evidence.

Return valid JSON:
{
  "fragment_id": "<selected fragment identifier>",
  "replacement_content": "<complete replacement content>",
  "local_interface": <unchanged local-interface object>
}
```

D.9.7 RECOVERY-GENE TRANSPLANTATION

Purpose. Adapt a source-domain recovery gene to a target-domain contract without modifying its imported recovery policy before the first target evaluation.

Inputs. The source recovery gene, source and target contracts, and target tool mapping.

Expected Output. A transfer-adapter specification that maps interfaces and tool names while preserving the imported policy.

Prompt.

```
Construct a transfer adapter for the imported recovery gene.

Source recovery gene:
<SOURCE_GENE>

Source contract:
<SOURCE_CONTRACT>

Target contract:
<TARGET_CONTRACT>

Available target tools:
<AVAILABLE_TOOLS>

Map source input types, output types, tool identifiers, and resource
references to target-domain equivalents. Do not rewrite the imported recovery
policy. Reject the transfer if any required source capability has no
compatible target mapping.

Return valid JSON:
{
  "source_gene_id": "<source gene identifier>",
  "target_gene_type": "recovery",
  "input_type_map": {
    "<source type>": "<target type>"
  },
  "output_type_map": {
    "<source type>": "<target type>"
  },
  "tool_map": {
    "<source tool>": "<target tool>"
  },
  "resource_map": {
    "<source resource>": "<target resource>"
  },
  "policy_modified": false,
  "compatible": <true|false>,
  "rejection_reason": "<empty when compatible>"
}
```

D.9.8 VERIFIER-GENE TRANSPLANTATION

Purpose. Adapt a source-domain verifier gene to a target-domain artifact contract while preserving its verification logic before the first target evaluation.

Inputs. The source verifier gene, source and target artifact schemas, and target resource mapping.

Expected Output. A transfer-adapter specification for the imported verifier.

Prompt.

```
Construct a transfer adapter for the imported verifier gene.

Source verifier gene:
<SOURCE_GENE>

Source artifact contract:
<SOURCE_CONTRACT>

Target artifact contract:
```

```
<TARGET_CONTRACT>

Target resources:
<TARGET_RESOURCES>

Map source artifact fields, types, paths, and resource identifiers to
target-domain equivalents. Do not rewrite the imported verification policy
or alter its acceptance criteria. Reject the transfer if the required
evidence cannot be represented under the target contract.

Return valid JSON:
{
  "source_gene_id": "<source gene identifier>",
  "target_gene_type": "postcondition_verifier",
  "field_map": {
    "<source field>": "<target field>"
  },
  "type_map": {
    "<source type>": "<target type>"
  },
  "resource_map": {
    "<source resource>": "<target resource>"
  },
  "policy_modified": false,
  "compatible": <true|false>,
  "rejection_reason": "<empty when compatible>"
}
```

D.10 Output Formatting

All executable agent actions conform to the native tool-call schema exposed by the evaluation harness. Prompts that produce evolutionary edits return JSON and are parsed before compilation. A response is accepted only if it contains one JSON object with the exact fields required by the corresponding template, uses valid JSON syntax, and introduces no text outside the object. Gene contents containing code or multiline instructions are represented as escaped JSON strings. File-generating baselines return complete relative paths and file contents, enabling reconstruction in a clean workspace. Compiler validation, static parsing, dependency checks, interface checks, and deterministic task verification are performed after parsing and are not delegated to the language model.

E Case Study

We present representative execution traces to illustrate how the typed genome changes an agent's behavior at the level of concrete tool calls, intermediate artifacts, verification decisions, and failure recovery. The cases are selected to expose interactions among genes rather than to summarize aggregate performance. In each case, the frozen agent receives the same task instruction and observations; only the external skill package differs.

E.1 Coordinated Recovery from an Unavailable Command-Line Tool

The first case comes from the dynamic tool-use environment. The task asks the agent to extract selected fields from a newline-delimited JSON file, order the records by timestamp, and write a clean CSV artifact. The environment exposes standard shell utilities and Python, but the commonly used JSON processor `jq` is unavailable. The input also contains one malformed line between otherwise valid records.

A free-form skill contains the instruction "use `jq` to parse JSON efficiently" and a shell script that directly invokes the command. Its recovery paragraph recommends retrying a failed command after checking the file path. Consequently, the agent interprets the nonzero exit as a possible path problem, lists the working directory, and invokes the same script again with an absolute input path. The second

attempt reproduces the original failure because the missing executable, rather than the path, is the cause. The agent then switches to a short Python snippet, but reads the input with `json.load`, treating the entire file as a single JSON document. Parsing terminates at the malformed line, and no output artifact is produced. The free-form skill therefore contains individually plausible instructions while failing to coordinate tool selection, input-format assumptions, and recovery.

The evolved genome separates these responsibilities. Its tool-policy gene declares `jq` as an optional provider of the capability `jsonl_record_iterator`, while the executable gene accepts any provider implementing that interface. The precondition gene probes tool availability before execution and identifies the input as JSON Lines by inspecting multiple nonempty lines rather than the filename extension alone. When the preferred provider is unavailable, the recovery gene selects a Python iterator with the same output contract. The postcondition gene then validates the generated CSV independently of the parsing implementation.

The relevant portion of the materialized skill is shown below.

```
capability: jsonl_record_iterator
preferred_provider:
    command: jq
    availability_check: command -v jq
fallback_provider:
    runtime: python3
    entrypoint: scripts/stream_jsonl.py

input_contract:
    unit: one JSON object per nonempty line
    malformed_record_policy: quarantine
    required_fields:
        - id
        - timestamp
        - status

output_contract:
    format: csv
    columns:
        - id
        - timestamp
        - status
    ordering:
        key: timestamp
        direction: ascending

recovery:
    missing_tool:
        action: select_compatible_fallback
        preserve:
            - input_contract
            - output_contract
    malformed_record:
        action: append_to_reject_log
        continue_if:
            - remaining records are independently parseable
            - required output can still be constructed

verification:
    - parse output with a CSV reader
    - require the declared header exactly once
    - require every emitted row to contain all declared columns
    - check nondecreasing timestamp order
    - require a reject log when malformed input is observed
```

This decomposition changes the trajectory before the initial failure occurs. The agent first runs the declared availability check, observes that `jq` cannot be resolved, and chooses the Python provider without attempting an invalid command. The fallback reads the file line by line, records the malformed line in `rejected.jsonl`, and continues processing valid records. It normalizes timestamps before sorting, writes the CSV through a temporary file, and atomically renames the temporary file only after serialization completes.

The verifier subsequently reopens the CSV with a parser rather than checking only for file existence. It confirms the exact header, verifies that each row has the expected arity, and compares adjacent normalized timestamps. It also checks that the reject log contains the original malformed line and its line number. The final agent response reports both artifacts and explicitly distinguishes successful records from quarantined input. The important behavior is not merely the substitution of Python for `jq`; it is the preservation of the same typed input and output contracts across the substitution. The recovery gene can therefore change the implementation without silently changing what downstream verification expects.

E.2 Verifier-Guided Repair of a Structurally Valid but Semantically Incorrect Artifact

The second case concerns a spreadsheet-like tabular transformation in which the requested output must retain the original row order while adding a normalized category column. The input contains category values such as `''North-East''`, `''north east''`, and `''NE''`. The task instruction requires these aliases to map to a canonical label, while unknown categories must remain present and be marked for review.

An early skill phenotype produces a syntactically valid CSV, but its planning gene instructs the agent to "clean the data and remove invalid entries." The executable module consequently drops rows whose categories do not occur in its alias dictionary. Its verifier checks that the file exists, that the new column is present, and that all emitted canonical values belong to an allowed set. These checks pass even though unknown rows have disappeared. The artifact is structurally well formed but violates the preservation requirement.

The execution trace localizes the discrepancy to the relationship between the postcondition and the executable module. The trigger, input discovery, and CSV-writing behavior are all exercised normally. The verifier, however, never compares input record identities with output record identities. EvoGenome therefore edits the postcondition gene rather than rewriting the entire skill. The evolved verifier introduces a conservation check and returns a structured failure that identifies missing keys:

```
postcondition:
    preserve_records:
        input_identity: source_row_id
        output_identity: source_row_id
        require:
            - every input identity appears exactly once in output
            - output identities follow input order

    canonical_category:
        known_alias:
            require: canonical label
        unknown_alias:
            require:
                normalized_category: REVIEW_REQUIRED
                original_category: preserved verbatim

failure_payload:
    type: semantic_validation_error
    fields:
        - violated_contract
        - affected_row_ids
        - expected_behavior
        - observed_behavior
```

On the next execution, the verifier rejects the apparently clean artifact and returns a failure payload specifying that several source row identifiers are absent. The recovery gene dispatches semantic validation failures back to the executable module with the violated contract attached. The module is then specialized so that dictionary misses produce `REVIEW_REQUIRED` rather than triggering row deletion. It also copies the original category into a separate provenance column.

The repaired execution preserves the original sequence of row identifiers. Known aliases are canonicalized, while an unseen spelling remains in the output with its original surface form and a review marker. The verifier checks identity conservation before inspecting category values, preventing a degenerate repair that would satisfy the vocabulary constraint by removing difficult examples. This case illustrates why postconditions are represented as independently evolvable genes: a verifier can reveal a semantic omission in an otherwise executable phenotype and can provide a precise repair signal without perturbing unrelated trigger or tool-policy behavior.

E.3 Homologous Crossover without Breaking Boundary Contracts

The third case illustrates crossover between two parent skills developed for different artifact workflows. Parent A reliably discovers input files and parses a manifest, but its recovery behavior overwrites the destination artifact after a partial write. Parent B uses transactional output handling: it writes to a temporary location, validates the result, and commits it only after postconditions succeed. Direct file-level crossover cannot safely combine these behaviors. Copying Parent B's script into Parent A also imports assumptions about directory names, manifest fields, and output extensions. Copying only its instruction file leaves the transactional helper undeclared and causes the materialized package to reference a missing module.

EvoGenome instead identifies a connected recovery subgraph in Parent B consisting of a recovery policy, a temporary-artifact executable module, and a commit postcondition. The subgraph exposes the following boundary interface:

```
input:
    destination_path: Path
    producer: Callable[temp_path -> execution_result]
    validator: Callable[temp_path -> validation_result]

output:
    committed_artifact:
        path: Path
        state: validated
    recovery_report:
        attempted_provider: String
        retained_original: Boolean

guarantees:
    - destination is unchanged if production fails
    - destination is unchanged if validation fails
    - temporary artifacts are removed on terminal failure
```

Parent A already provides a destination path and a producer callable, but its verifier accepts a final path rather than a temporary path. The raw subgraph is therefore not immediately compatible. The crossover operator retains Parent A's trigger, precondition, planning, parser, and domain-specific verifier logic, while an adaptation gene wraps its verifier so that it can inspect the temporary artifact before commit. The compiler checks that the wrapper returns the validation result expected by the imported recovery subgraph and that no consumer still depends on the previous direct-write output.

In the resulting phenotype, the agent discovers the manifest using Parent A's established procedure and constructs the output with its original executable module. The producer receives a generated temporary path rather than the user-visible destination. When validation detects a missing manifest entry, the recovery policy deletes the temporary artifact and leaves any pre-existing destination untouched. It then invokes Parent A's domain-specific repair routine, regenerates the temporary artifact, and reruns validation. Only the validated artifact is moved to the destination.

This behavior differs from textual blending. The offspring does not inherit Parent B's task vocabulary, file-discovery rules, or content assumptions. It inherits only a recovery mechanism whose incoming and outgoing contracts can be satisfied in the target genome. The adaptation gene is explicit in the lineage and can later be revised or pruned independently. The case demonstrates how homologous crossover can transfer a procedural invariant, namely "never commit an unverified artifact," without transferring an entire source skill.

E.4 Lineage Credit Distinguishes an Active Repair from an Incidental Edit

A final trace demonstrates why execution evidence is combined with lineage differences. A candidate offspring modifies both the planning gene and the recovery gene. The planning edit changes the wording of an instruction from "inspect the output" to "inspect the output before returning," while the recovery edit adds bounded retries with error-class dispatch. On a task containing a transient file-lock error, the offspring releases the stale lock, waits, retries the interrupted write, and subsequently validates the artifact.

The trace records that the revised planning branch was not consulted after the initial plan had been formed. In contrast, the recovery gene was activated by the file-lock exception, selected the `transient_resource` branch, and emitted the retry event consumed by the verifier. The successful terminal artifact is therefore linked to concrete activation and verifier evidence for the recovery edit. The planning edit remains part of the descendant, but it does not receive the same causal support from this episode.

A later task exposes an invalid output schema rather than a transient resource error. The bounded-retry branch correctly declines to repeat the operation because the failure is classified as semantic and deterministic. Control is transferred to the schema-repair branch, which reports the unexpected columns to the executable module. This second trace supports the dispatch structure of the recovery gene while avoiding a misleading policy of retrying every failure.

Across these cases, the qualitative advantage of the genome representation arises from explicit behavioral boundaries. Tool substitution preserves capability contracts; semantic verification protects information that a superficial validator would overlook; crossover transfers connected procedures rather than source-specific files; and lineage evidence associates a repaired trajectory with the gene that was actually activated. These mechanisms make the evolved phenotype interpretable at the level of concrete agent actions and terminal artifacts while retaining compatibility with an ordinary skill-enabled agent harness.