Orchestrating LLMs as Hierarchical Multi-Agent Reinforcement Learning System for Automotive Software Development

Anonymous Authors[1]

Abstract

Software-defined vehicles depend on firmware that must evolve continuously and safely, yet general-purpose LLM coding agents lack the architectural mechanisms required for safety-critical cyber-physical systems. We introduce **AutoEvolve**, a Hierarchical Multi-Agent Reinforcement Learning (H-MARL) framework with three contributions: (i) jointly learned Orchestrator and sub-agent (Data, Requirements, Code) policies, where an *adversarial* Requirements Agent rejects unsafe candidates rather than merely critiquing them; (ii) an offline-to-online curriculum that initializes via SFT on historical development trajectories and refines via PPO in a *shadow-mode* deployment running parallel to human engineers, treating their commits as delayed supervision; and (iii) a dual-reward decomposition $R_{fast} + \lambda R_{slow}$ that anchors policies to deterministic verification while regularizing toward maintainable, human-aligned style. On an internal benchmark, AutoEvolve attains the highest Success Rate (**61.4%** vs. 53.5–55.1% for MetaGPT and SWE-Agent at the same Llama-3-70B backbone) and reduces the Requirement Violation Rate to **1.6%** – $\sim 7\times$ fewer violations than task-agnostic multi-agent frameworks and $\sim 10\times$ fewer than a Monolithic ReAct baseline. The architectural contribution alone (SFT-Only AutoEvolve at 4.7% violations) drives most of the safety gain; online RL provides the remaining performance lift. Active Development Cycle Time drops from ~ 5 days to < 6 hours, framing safety-critical software evolution as *safe* agentic coding, not raw generation.

1. Introduction

Modern automotive software is no longer a static artifact; it is the dynamic nervous system of a living fleet. Evolving user expectations and competitive pressures necessitate continuous feature enhancements and performance optimizations. Yet, this system is perpetually at risk. Hardware components degrade and physical properties shift - seals stiffen, thermal conductivity changes, and battery chemistry alters - causing optimal control parameters and calibrations to drift from factory baselines (Smith et al., 2015; Kömmling et al., 2020). Simultaneously, latent software defects emerge in the wild: race conditions in sensor fusion, deadlocks in state machines, or unhandled edge cases in diverse operating environments. To address this entropy, we propose **AutoEvolve** (Figure 1), a Hierarchical Multi-Agent Reinforcement Learning (H-MARL) system designed to catalyze a paradigm shift from *Software Development* to *Software Evolution*, aligning with the industry-wide transition towards software-defined vehicles (Otto et al., 2025).

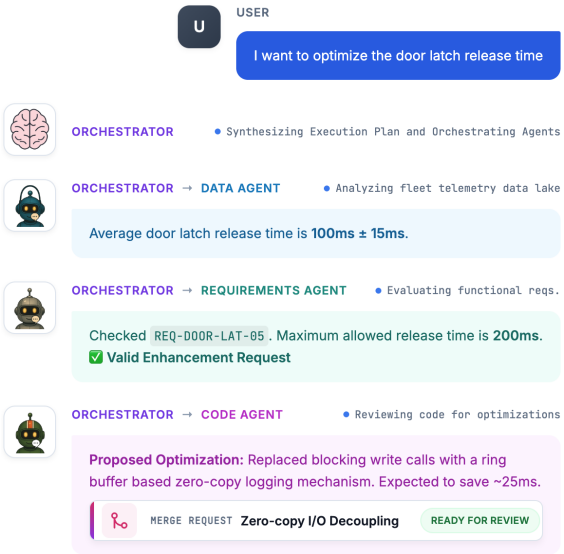
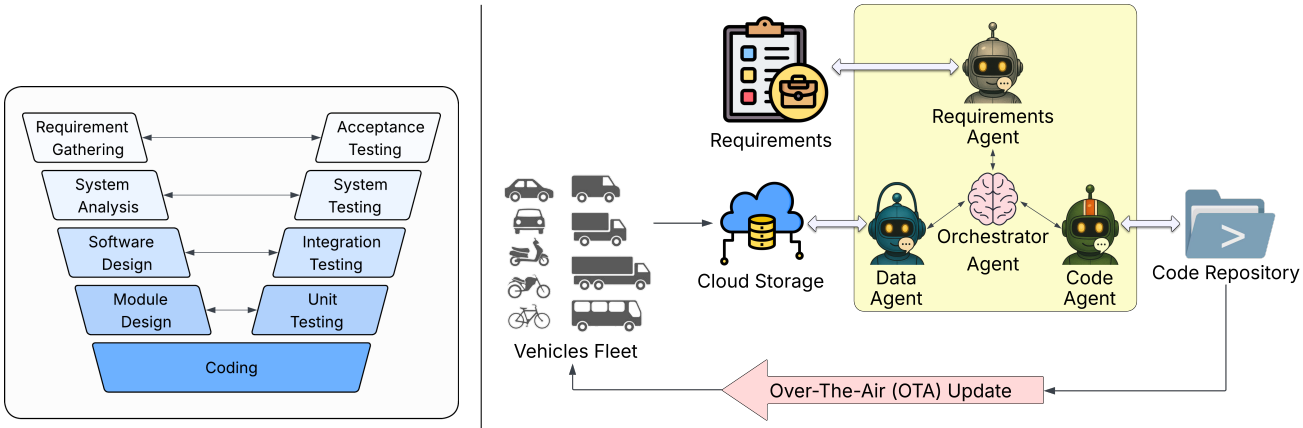


Figure 1. The vision of software evolution. The Orchestrator coordinates specialized agents (Data, Requirements, Code) to autonomously resolve a user-reported performance issue: it verifies the current state against fleet telemetry, validates safety constraints, and proposes a verified patch via a shadow PR.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.



(a) The Traditional V-Model

(b) AutoEvolve Architecture

Figure 2. Development paradigms. (a) The V-Model enforces sequential verification but resists rapid adaptation. (b) AutoEvolve closes the loop: fleet data and evolving requirements drive the Orchestrator to coordinate sub-agents (Data, Requirements, Code) for automated diagnosis, repair, and enhancement.

This vision contrasts with the prevailing "V-Model" methodology (Gräßler et al., 2021) (Figure 2a), which enforces strict sequential verification: every modification must descend through requirements/design and ascend through unit/integration/system testing, often spanning months while the fleet operates sub-optimally; Figure 2 juxtaposes this V-Model against AutoEvolve's closed-loop paradigm.

We formalize software evolution as a Reinforcement Learning (RL) problem parameterized by multiple LLMs as agents: the fleet is a stochastic environment, the codebase is a policy, and engineering becomes continuous policy updates that maximize a reward function over fleet health, safety, and user satisfaction. Applying RL directly to safety-critical firmware presents three challenges: (1) *Complexity*, since the action space of all code edits is combinatorial; (2) *Safety*, since unconstrained exploration on physical vehicles is unacceptable; and (3) *Privacy*, since proprietary firmware and telemetry require air-gapped, open-weights models (e.g., Llama-3 (Dubey et al., 2024)) rather than public APIs (e.g., GPT-4 (Achiam et al., 2023)).

To overcome these challenges, AutoEvolve decomposes the maintenance task into a cooperative game:

1. **Hierarchical Coordination:** We formalize the problem as a Hierarchical Markov Decision Process (H-MDP). A top-level *Orchestrator* learns a policy π_{orch} to plan the workflow, delegating sub-tasks to specialized agents (Data, Requirements, Code) that learn low-level policies π_{sub}.

2. **Offline-to-Online Learning:** We pre-train agents via Supervised Fine-Tuning on historical development trajectories (Behavior Cloning), then refine them via PPO in a *shadow mode* that runs parallel to human engineers and treats their final commits as a delayed supervision signal – enabling safe adaptation on live issues without risking the physical fleet.

3. **Human-in-the-Loop Gating:** Any active deployment to the physical fleet is strictly gated by manual approval. The agent acts as a force multiplier for engineers, never as an unsupervised operator.

Empirically, AutoEvolve reduces Development Cycle Time from months to hours and cuts the Requirement Violation Rate from 16.5% (Monolithic), 12.6% (MetaGPT), and 11.8% (SWE-Agent) to **1.6%** on our internal benchmark (Table 2).

To our knowledge, this is the first study of multi-agent LLM systems deployed in shadow mode parallel to a production engineering team for safety-critical automotive firmware. As is typical for industrial AI-coding work in safety-critical domains, the underlying fleet data and firmware are proprietary; we therefore frame our contribution as a portable methodological recipe (section 8, subsection A.9) rather than a public-benchmark result.

2. Related Work

Our work sits at the intersection of agentic software engineering, hierarchical RL, and safe adaptation in cyber-physical systems.

LLM Agents for Software Evolution: Beyond isolated code generation (Chen et al., 2021), recent agentic frameworks like ChatDev (Qian et al., 2024) and MetaGPT (Hong et al., 2023) use multi-agent simulations of software companies, while SWE-agent (Yang et al., 2024) and OpenDevin (Wang et al., 2024) resolve GitHub issues benchmarked on

SWE-bench (Jimenez et al., 2023); related lines explore tool orchestration (Su et al., 2025) and verbal reinforcement (Shinn et al., 2023). AutoEvolve differs along three axes critical for safety-critical CPS: *coordination* (we jointly *learn* hierarchical policies via SFT→PPO rather than relying on fixed SOPs or prompt templates), *safety enforcement* (we add an adversarial Requirements Agent that can reject candidates against formal constraints *before* acceptance, rather than relying on post-hoc tests), and *online adaptation* (shadow-mode policy-level RL using deterministic verification plus delayed human commits, vs. verbal-feedback test-time learning). Detailed comparisons appear in subsection A.7; empirically (Table 2), these differences yield $\sim 7\times$ fewer requirement violations than MetaGPT or SWE-Agent at the same backbone and tool access.

Hierarchical Reinforcement Learning (HRL): HRL decomposes long-horizon tasks into manageable sub-routines (Barto & Mahadevan, 2003; Vezhnevets et al., 2017). We frame the software evolution lifecycle as a hierarchical dependency: the Orchestrator's action space consists of instantiating sub-agents with specific natural language goals. This mirrors the Manager-Worker architecture (Vezhnevets et al., 2017), but applies it to a novel domain where the "worker's" action space is the infinite set of possible code edits.

Safe RL and Offline-to-Online Learning: Deploying learning agents in safety-critical loops is a major challenge (Garcıa & Fernández, 2015; Dalal et al., 2018). While Offline RL (Levine et al., 2020) mitigates exploration risks by learning from static datasets, it suffers from distribution shift when deployed. AutoEvolve bridges this gap via a *shadow mode* deployment strategy. By running parallel to human engineers, we treat the human's final commit not just as a safety fallback, but as a real-time supervision signal (label) for online policy updates (Nair et al., 2020; Ball et al., 2023; Lee et al., 2022), enabling continuous adaptation without physical risk.

Automated Program Repair (APR): Traditional APR approaches like GenProg (Le Goues et al., 2012) and Angelix (Mechtaev et al., 2016) utilize genetic algorithms or symbolic execution to generate patches. While effective for well-defined logical bugs, they struggle with high-level semantic changes or ambiguous requirements (Xia et al., 2023). AutoEvolve bridges this gap by leveraging the semantic reasoning of LLMs while retaining the rigorous feedback loops characteristic of search-based APR.

3. Problem Formulation

We formulate continuous vehicle software evolution as a Hierarchical Markov Decision Process (H-MDP) $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ with discount $\gamma \in [0, 1)$.

State $s_t = [\mathcal{D}_t, \mathcal{C}_t, \mathcal{K}_t, \mathcal{G}_t]$: fleet telemetry distribution $\mathcal{D}_t$ (sensor readings, error codes), firmware code state $\mathcal{C}_t$ (AST embeddings), active constraints $\mathcal{K}_t$ (e.g., "Battery temp $< 45°$C"), and the active goal $\mathcal{G}_t$ from a human operator or triage system.

Action: The hierarchical action space comprises a high-level Orchestrator action A_{orch} – selecting a sub-agent $k \in \{\text{Data}, \text{Req}, \text{Code}\}$ with a natural-language instruction i – a low-level sub-agent action A_{sub} executing primitive operations (SQL query, RAG retrieval, code generation) to produce output o_k, and a terminal action submitting candidate firmware C' for deployment.

Transition: In the offline phase, transitions are observed from historical logs (commits → new repository states). In the online shadow phase, the environment is the live CI/CD pipeline; transitions (test results, linter errors) are stochastic and determined by compilation and verification tools.

Reward: To handle the latency of human ground truth, we decompose $R_t = R_{fast} + \lambda R_{slow}$, where R_{fast} is the dense CI/CD verification signal and R_{slow} is the sparse, delayed semantic alignment with the human-authored solution; full decomposition is in subsection A.1.

Episodes: An episode is a single issue-to-PR lifecycle: it begins when $\mathcal{G}_t$ is dispatched and ends with either a candidate C' submission, a Layer-1 violation, or a step budget $T_{max} = 25$ being reached. Successful episodes empirically use 5–15 sub-agent invocations (median 9); rewards propagate to earlier steps via GAE with $\gamma = 0.99, \lambda_{GAE} = 0.95$ (subsection A.3).

4. Methodology

AutoEvolve consists of a central Orchestrator and three specialized sub-agents (Figure 2b); Figure 3 shows the offline-to-online curriculum.

4.1. Hierarchical Policy Architecture

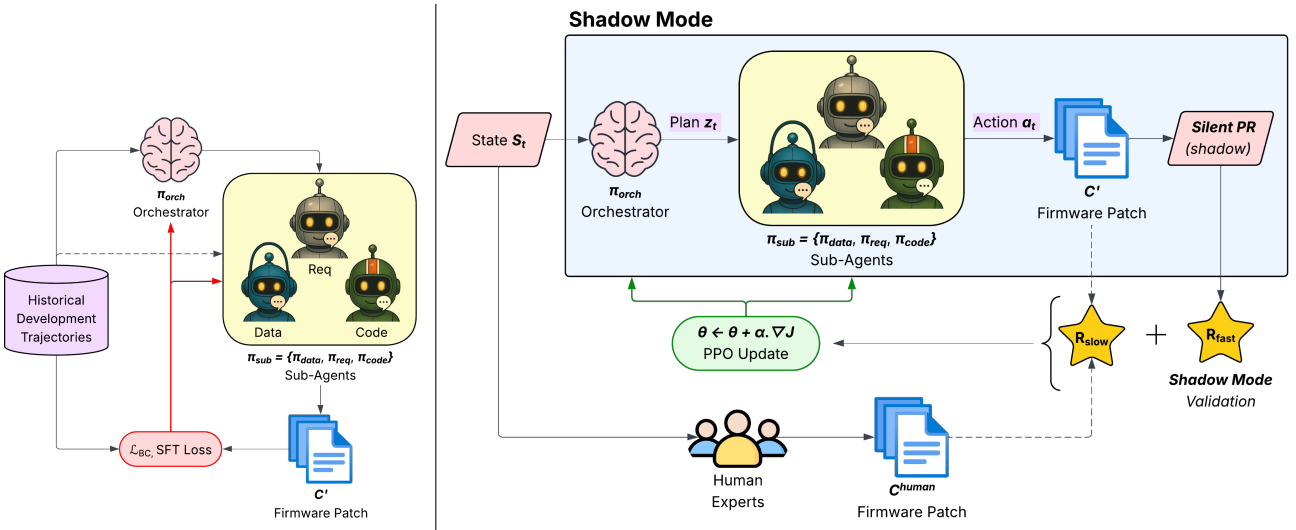
We decompose the global policy Π into a two-level hierarchy: an *Orchestrator policy* π_{orch} and *sub-agent policies* $\pi_{sub} = \{\pi_{data}, \pi_{req}, \pi_{code}\}$.

4.1.1. High-Level Policy: The Orchestrator

The Orchestrator observes the global state s_t and generates a high-level plan z_t, which is a sequence of sub-tasks (sub-agent invocations).

$$z_t \sim \pi_{orch}(z|s_t; \theta_{orch}) \tag{1}$$

where θ_{orch} are the parameters of the Orchestrator LLM. The Orchestrator's discrete action head selects $k \in \{Data, Req, Code\}$, then samples a natural-language instruction i, giving $z_t = \{(k_1, i_1), (k_2, i_2), \ldots\}$. To prevent infinite refinement, the policy is penalized via R_{cost} (Equa-



(a) Phase 1: Offline SFT Initialization

(b) Phase 2: Online shadow mode RL

Figure 3. H-MARL learning framework. (a) SFT initialization: agents are bootstrapped on historical trajectories via Behavior Cloning ($\mathcal{L}_{BC}$). (b) Online shadow-mode RL: policies π_θ are refined via PPO using a dual-signal reward (R_{fast} and R_{slow}) without physical-fleet risk.

tion 7, subsection A.1), an aggregate penalty over total invocations, redundant tool calls, and no-op steps; the agent never observes its own "progress" as a learnable signal, avoiding a hackable progress proxy. A worked plan trace is given in subsubsection A.1.4.

4.1.2. Low-Level Policies: Specialized Agents

Each sub-agent operates under policy $\pi_k(a|s_t, i; \theta_k)$ (full capabilities in subsection A.2). The **Data Agent** (π_{data}) synthesizes optimized SQL/Spark queries against the fleet data lake and reasons over the results to produce diagnostic summaries, going beyond simple querying. The **Requirements Agent** (π_{req}) is a semantic critic that interprets natural-language safety, homologation, and functional specifications, verifying alignment with $\mathcal{K}_t$ alongside the deterministic guardrails described in subsubsection 4.1.3. The **Code Agent** (π_{code}) synthesizes firmware modifications δ_c over the full repository state and emits AST edit operations to guarantee syntactic correctness.

4.1.3. Layered Safety Architecture

AutoEvolve enforces safety through three independent layers, each strictly subordinate to those below: **Layer 1** – a *deterministic* hard gate (static linters such as MISRA-C/AUTOSAR, compilers, regression tests, memory-safety and timing-constraint analyzers) that no LLM output can bypass; **Layer 2** – a *semantic critic* (the Requirements Agent) that reasons over natural-language requirements via RAG and rejects, but cannot approve, candidates that fail Layer 1; **Layer 3** – a *human-in-the-loop* gate before any deployment to a physical vehicle. Critically, the training signal R_{req} comes *exclusively* from Layer 1 (Equation 7); Layer 2 is used only at evaluation time to compute the reported Requirement Violation Rate (Table 2). Training and evaluation signals therefore come from independent mechanisms, ruling out the circular-reasoning concern. Full layer specifications and the non-circularity argument are formalized in subsection A.8.

4.2. Offline-to-Online Learning

Direct on-fleet RL exploration is unsafe – a trial-and-error update on a physical vehicle can trigger catastrophic behavior. We therefore use a two-phase curriculum.

4.2.1. Phase 1: Offline Supervised Fine-Tuning

From historical development logs (issue tickets, telemetry, codebase snapshots, active requirements) we construct a dataset of expert trajectories $\mathcal{T}_{expert} = \{(s_0, a_0^*, s_1, a_1^*, \ldots, r_{final})\}$ (Figure 3a) and initialize policies via Behavior Cloning:

$$\mathcal{L}_{BC}(\theta) = -\mathbb{E}_{(s,a^*)\sim\mathcal{T}_{expert}}[\log \pi(a^*|s;\theta)] \qquad (2)$$

This embeds the organization's implicit norms into the policy. The *Orchestrator* learns standard operating procedures (*verify requirements before coding*; *quantify telemetry drift before diagnosing a regression*); each sub-agent inherits domain-specific priors – error-code↔telemetry correlations (Data), specification-to-constraint parsing (Requirements), house style and patch idioms (Code) – giving the system an organization-aware initialization before any policy refine-

ment against the verifier.

4.2.2. Phase 2: Online shadow mode RL

Behavior cloning is limited by the quality of historical data and suffers from distribution shift. To evaluate and improve beyond human baselines, we transition to an Online RL phase using a shadow mode deployment strategy (Figure 3b). In this setup, the agent operates in the background, parallel to the human engineering team, observing real-time development workflows without intervening. As human engineers address incoming issues or new feature requests (contained within s_t), AutoEvolve independently diagnoses the problem and generates a candidate solution (C').

We define the online environment as a dual-loop evaluation system to handle reward latency:

1. **Shadow Execution & Immediate Reward (R_{fast}):** Upon detecting a code change request, the Orchestrator instantiates the agent team. The generated code C' is immediately subjected to the rigorous internal CI/CD pipeline. The results of these automated checks provide a dense, immediate reward signal R_{fast}:

$$R_{fast} = w_1 \cdot R_{\text{test}} + w_2 \cdot R_{\text{req}} - w_3 \cdot R_{\text{cost}} \quad (3)$$

   Here, R_{test} measures functional correctness via automated tests, R_{req} imposes penalties for safety and functional requirement violations (linters/static analysis), and R_{cost} penalizes inference compute. The weights w_1, w_2, w_3 were determined via a grid search on a held-out validation set to optimally balance safety (critical) vs. feature velocity.

2. **Silent PR & Alignment Regularization (R_{slow}):** If a solution passes all automated checks, we package it as a shadow pull request (PR) on a detached branch. When the human engineer submits their solution (C^{human}), we treat it as the ground truth approval and asynchronously compute alignment with the PR to produce the sparse, delayed signal R_{slow}. We treat R_{slow} as a regularization term for maintainability: in long-lived safety-critical firmware, code readability and adherence to architectural patterns are functional requirements for future human debugging, and R_{slow} prevents convergence to "obfuscated" solutions that pass tests but degrade evolvability. The metric combines functional equivalence (test coverage overlap), structural AST matching, and semantic embedding similarity; if human ground truth is unavailable, $R_{slow} = 0$. This leverages implicit expert supervision without requiring manual review of every agent action.

The total reward R_t blends these signals. For online adaptation, we primarily optimize against R_{fast} to ensure immediate safety and correctness. The delayed signal R_{slow} handles the long-latency feedback loop; it is accumulated to perform periodic, asynchronous policy alignment updates (e.g., weekly batch updates). This ensures the agent's coding style and logic remain aligned with organizational norms without blocking real-time execution.

We evaluate the policy to maximize the expected reward:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t) \right] \quad (4)$$

5. Experimental Evaluation

We evaluate AutoEvolve's efficacy, efficiency, and safety through a series of experiments on historical automotive software issues, comparing its performance against both human benchmarks and monolithic agent baselines.

5.1. System Implementation

Dataset and Environment: Our training dataset comprises real-world historical development artifacts from the OEM under study, collected from a fleet of over 100,000 vehicles. The corpus spans millions of lines of firmware code, thousands of active safety requirements, and petabytes of associated vehicle telemetry data. This captures the full lifecycle of feature development and issue resolution (issue description → PR → merge), paired with the relevant telemetry data ingested from the cloud. These serve as the "expert trajectories" for the Offline SFT phase.

Agent Backbones and Orchestration: We utilize state-of-the-art open-weights models hosted on a secure private cluster. We selected *Llama-3-70B-Instruct* for the Orchestrator and Requirements Agent due to its reasoning and instruction-following capabilities. The Data Agent uses *CodeLlama-34B-Instruct*, and the Code Agent utilizes *CodeLlama-70B-Instruct*, enabling robust code synthesis and large-context understanding. We employ *LangGraph* for hierarchical orchestration, managing stateful interactions and message passing between the Orchestrator and sub-agents.

Training Infrastructure: The system is trained using the TRL (Transformer Reinforcement Learning) library, which provides unified support for both SFT and Proximal Policy Optimization (PPO). We employ Low-Rank Adaptation (LoRA) for efficient fine-tuning of all agent policies, optimizing only $1 - 2\%$ of parameters during both the Offline SFT and Online RL phases (details in subsection A.3). Training was conducted on an internal cluster of GPUs.

5.2. Evaluation Setup

Offline Evaluation (Phase 1): To assess the SFT initialization, we evaluate agents on a held-out test set of historical

issues. We primarily measure the *Pass@1* rate, along with *Success Rate (SR)* and *Attempts to Solution (AtS)* (subsection 5.5), of generated patches against the ground-truth human commits using static analysis and unit tests from the historical state. This validates that the agents have successfully cloned the organization's coding standards and basic debugging procedures.

Online shadow mode (Phase 2): The primary evaluation occurs in a shadow mode alongside the human engineering team. The system monitors the live stream of real-time bug reports, feature requests ($\mathcal{G}_t$), and telemetry alerts ($\mathcal{D}_t$) arriving in the production issue tracking system. For each ticket, AutoEvolve initiates a parallel autonomous evolution loop in the background, utilizing the current codebase ($\mathcal{C}_t$) and existing automated verification tools. The resulting "Shadow PR" is time-stamped and stored, serving as a comparative baseline against the human engineer's eventual solution for functional equivalence, code quality, and time-to-solution.

5.3. Evaluation Scenarios

We evaluate the system across six representative task categories covering the full spectrum of software evolution: (1) **Parameter Tuning and Calibration** (e.g., compensating for ambient temperature variations), (2) **Feature Enhancement** (e.g., implementing new user-facing logic), (3) **Safety-Critical Bug Fixes** (e.g., resolving race conditions), (4) **Resource Optimization** (e.g., reducing CPU/battery usage), (5) **Security Hardening** (e.g., patching vulnerable libraries or tightening access controls), and (6) **Compliance and Localization** (e.g., adapting lighting sequences for new regional regulations). Detailed scenario descriptions are in subsection A.5; the underlying firmware and telemetry are proprietary and our reproducibility recipe (section 8) is designed to enable independent replication on analogous data.

5.4. Comparison and Baselines

We compare AutoEvolve against four agentic baselines plus a human benchmark, all evaluated on the same internal task suite under identical verifier, tool access, and (where applicable) Llama-3-70B-Instruct backbone: (i) **Human Benchmark** – historical DCT for similar issues in the organization's manual workflow; (ii) **Monolithic ReAct** (Yao et al., 2022) – a single Llama-3-70B-Instruct agent with AutoEvolve's tools but no hierarchical orchestration, managing tool use via Chain-of-Thought; (iii) **MetaGPT** (Hong et al., 2023) – a multi-agent framework with fixed SOPs, adapted to our environment; (iv) **SWE-Agent** (Yang et al., 2024) – a state-of-the-art coding agent with an agent-computer interface, adapted analogously; (v) **AutoEvolve (SFT-Only)** – an ablation isolating the architectural contribution from online RL. Closed-source APIs (GPT-4, Claude (Anthropic, 2024)) are excluded due to data-privacy and air-gap requirements; the architecture is model-agnostic.

5.5. Metrics

We define specific metrics to answer our three primary research questions:

- **RQ1 - Efficacy:** *Does the H-MARL architecture perform reliably?* We report **Pass@1** (fraction of issues solved on the first generation), **Success Rate** (eventual completion rate including refinement), and **Attempts to Solution (AtS)** on our internal benchmark (Table 2).
- **RQ2 - Efficiency:** *Can AutoEvolve reduce cycle time?* We track **Development Cycle Time (DCT)**, defined as the total wall-clock time from issue receipt to verified Pull Request.
- **RQ3 - Safety:** *Does the system prevent unsafe modifications?* We monitor the **Requirement Violation Rate** – the percentage of solutions that pass Layer-1 functional tests but violate broader safety, performance, or compliance constraints, measured by the Layer-2 semantic critic against retrieved requirements (subsubsection 4.1.3).
- **Context Overflow:** We also track the frequency of failures caused by the input prompt exceeding the model's maximum token limit, which serves as a proxy for the architecture's ability to handle large-scale codebases.

5.6. Results

5.6.1. Efficiency Gains

We first present the aggregated performance results across the evaluation scenarios in Table 1, focusing on the reduction in Development Cycle Time (see subsection A.6 for detailed measurement methodology).

Table 1. Efficiency comparison: AutoEvolve's wall-clock Active Execution Time vs. estimated Human Active Time and historical Organizational Cycle Time. AutoEvolve eliminates the organizational latency inherent in traditional safety-critical workflows.

Metric	AutoEvolve (Active)	Human (Active Est.)	Human (Org. Cycle)
Diagnosis Time	$\sim$ 10 Mins	$\sim$ 1 Day	1-2 Weeks
Impl. Time	$\sim$ 30 Mins	$\sim$ 2 Days	$\sim$ 1 Week
Validation Time	$\sim$ 2-4 Hours	$\sim$ 2 Days	2-4 Weeks
DCT	**$<$ 6 Hours**	**$\sim$ 5 Days**	**1-2 Months**

As detailed in Table 1, AutoEvolve reduces DCT by an order of magnitude relative to both the human Active Estimate and the human Organizational Cycle (which includes meetings, weekends, and approval-queue latency). Even

against the idealized Human Active Estimate, the system goes from ∼5 days to <6 hours, driven by parallel diagnosis and implementation. Organizational Cycle Time gains reflect a structural advantage: continuous, parallel execution without context-switching overhead achieves throughput substantially higher than sequential human workflows.

5.6.2. Comparative Analysis: AutoEvolve vs. State-of-the-Art Multi-Agent and Monolithic Baselines

To isolate the impact of our architectural and algorithmic contributions, we compare AutoEvolve against four baselines, all using the same *Llama-3-70B-Instruct* backbone and equivalent tool access: (i) a *Monolithic ReAct* agent representing the standard single-agent paradigm; (ii) *MetaGPT* (Hong et al., 2023), a multi-agent framework with fixed Standard Operating Procedures (SOPs); (iii) *SWE-Agent* (Yang et al., 2024), a state-of-the-art coding agent with a specialized agent-computer interface; and (iv) an *SFT-Only* ablation of AutoEvolve, isolating the contribution of the hierarchical architecture from the online RL phase. We adapted MetaGPT and SWE-Agent to our environment by providing equivalent tool access and the same evaluation pipeline, while keeping their respective coordination mechanisms intact. All baselines are evaluated on the same internal benchmark of representative software-evolution tasks under the same verification harness.

Table 2. Performance vs. multi-agent and monolithic baselines, all using Llama-3-70B-Instruct with equivalent tool access. AutoEvolve (H-MARL) achieves the highest Success Rate (61.4%) and lowest Requirement Violation Rate (1.6%) – ∼7× fewer violations than MetaGPT/SWE-Agent. The SFT-Only row isolates the architectural contribution: hierarchy alone yields 2.5× fewer violations than SWE-Agent at comparable Success Rate. Bold indicates best per column; MetaGPT achieves the lowest AtS (2.7) reflecting efficient SOP-based coordination.

System	Pass@1	Success Rate	Req. Violation	AtS
Monolithic ReAct	29.1%	44.9%	16.5%	4.8
MetaGPT	37.8%	53.5%	12.6%	**2.7**
SWE-Agent	**45.7%**	55.1%	11.8%	3.2
AutoEvolve (SFT-Only)	38.6%	55.9%	4.7%	3.3
AutoEvolve (H-MARL)	44.1%	**61.4%**	**1.6%**	3.1

Table 2 surfaces two findings that together motivate the AutoEvolve design.

Safe generation, not raw generation, is the contribution. SWE-Agent achieves the highest Pass@1 (45.7%), but its 11.8% Requirement Violation Rate – comparable to MetaGPT's 12.6% and the Monolithic baseline's 16.5% – is unacceptable in production firmware. AutoEvolve trades a small Pass@1 gap (44.1% vs. 45.7%) for an order-of-magnitude reduction in violations (1.6%), because the adversarial Requirements Agent rejects unsafe first attempts and forces re-generation. The result is the highest eventual Success Rate (61.4%): violations are caught *during* the agentic loop rather than after deployment. General-purpose frameworks lack this adversarial-rejection mechanism, so even fine-tuning would likely leave violations elevated – a structural limitation of task-agnostic coordination.

The architecture provides the safety margin; RL provides the performance boost. The SFT-Only row achieves a Success Rate (55.9%) comparable to zero-shot SWE-Agent (55.1%) and MetaGPT (53.5%), but with ∼2.5× fewer requirement violations (4.7% vs. 11.8%–12.6%). Because all three systems share the same backbone and tools, this gap is attributable to the adversarial Requirements Agent. SFT-Only nonetheless plateaus under live distribution shift; the online shadow-mode phase, run for 500 training episodes against the live verifier, then closes the remaining performance gap (55.9% → 61.4% SR, ∼10% relative) and yields a further 2.9× reduction in violations (4.7% → 1.6%).

6. Discussion

The adversarial Requirements Agent penalizes safety violations during training while the delayed reward R_{slow} anchors the policy to human-verified solutions, mitigating specification gaming (subsection A.1); the HITL gate remains a hard constraint for any production deployment. To avoid penalizing structurally novel solutions that diverge from human demonstrations, we prioritize functional correctness ($\lambda \ll 1$) and run periodic manual audits of high-reward, low-alignment trajectories. Generalization beyond training data emerges from the online shadow-mode phase, which adapts policies to novel distribution shifts via real-time verifier feedback. Beyond the algorithmic speedup, the order-of-magnitude DCT reduction reflects a structural shift: the agent operates continuously and asynchronously, parallelizing tasks that are strictly serialized in human workflows.

7. Limitations

We acknowledge five concrete limitations of the present study. *(i) Proprietary evaluation environment:* the benchmark cannot be released, a structural feature of safety-critical CPS deployment rather than a deficit of transparency, and we therefore discharge reproducibility at the recipe level (section 8, subsection A.9) by disclosing the architecture, layered safety design, pseudocode (subsection A.4), hyperparameters (subsubsection A.1.3), a worked episode walkthrough (subsubsection A.1.4), and an open-weights backbone – making no claim of third-party reproduction of exact numbers. *(ii) Open-weights backbone only:* air-gap requirements preclude evaluation against closed-source

APIs (GPT-4, Claude), so the architectural claim is empirically verified only at the Llama-3-70B scale; a stronger backbone would presumably improve absolute performance across all configurations but would not change the relative gains from H-MARL. *(iii) Single-organization study:* all experiments run within one OEM's workflow, and generalization to other OEMs or adjacent domains (aerospace, industrial control) is plausible but unverified, since the SFT prior is organization-specific. *(iv) Soft-real-time scenarios only:* the six evaluation categories in subsection A.5 (parameter tuning, feature enhancement, race-condition fixes, resource optimization, security hardening, compliance) do not exercise hard-real-time deadlines such as chassis-control loops at $\leq$1 ms or ADAS sensor-fusion at sub-50 ms WCET, and extending the framework to those targets requires augmenting the Requirements Agent with WCET analyzers and schedulability checks at Layer 1. *(v) Heuristic time measurement and Layer-2 robustness:* the human Active Time baseline uses a Focus Factor α, with sensitivity analysis (Table 3) showing the order-of-magnitude conclusion is robust over $\alpha \in \{0.50, 0.65, 0.80\}$ but a fully controlled A/B trial remaining future work; separately, the Layer-2 Requirements Agent is itself an LLM subject to standard failure modes (hallucinated constraint matches, prompt-sensitive judgments), so the production safety story relies on the deterministic Layer-1 gate plus the Layer-3 HITL review rather than on Layer 2 in isolation, and we have not stress-tested Layer 2 against adversarial inputs designed to elicit false approvals.

8. Conclusion

We presented **AutoEvolve**, a framework that reimagines vehicle software maintenance as a hierarchical multi-agent reinforcement learning problem. By coupling a high-level Orchestrator with specialized sub-agents and a rigorous shadow mode verification, we demonstrated that it is possible to automate the end-to-end evolution of firmware - from bug fixing to feature enhancement. Our results on a diverse set of real-world scenarios show that AutoEvolve reduces Development Cycle Time by an order of magnitude compared to human workflows, while the hierarchical structure offers stronger safety guarantees ($\sim$7$\times$ fewer requirement violations) than both monolithic and task-agnostic multi-agent LLM baselines. This work paves the way for self-evolving cyber-physical systems that adapt autonomously to the changing needs of their environment and users. Future work will focus on transitioning AutoEvolve from shadow mode to an active assistant role, where agent-generated pull requests are submitted for human review and merge, and on extending the framework to hard-real-time targets such as chassis control and ADAS sensor fusion. As confidence in the system's safety guarantees grows, we envision a gradual shift towards supervised autonomy.

Software and Data

The proprietary nature of fleet telemetry, firmware source, and homologation requirements is a structural feature of safety-critical CPS deployment, not a deficit of transparency. Our position is that *recipe-level* reproducibility – exposing every architectural, algorithmic, and training-pipeline choice in sufficient detail for independent replication on analogous proprietary data – is the appropriate standard. We cannot release the telemetry corpus, firmware source, requirement specifications, fine-tuned LoRA adapters, or the internal verifier toolchain; we *do* disclose the formal H-MDP (section 3), architecture and layered safety design (section 4, subsubsection 4.1.3), absolute results against four baselines including MetaGPT and SWE-Agent (Table 2), Focus Factor sensitivity (Table 3), pseudocode (subsection A.4), full $R_{cost}/R_{fast}/R_{slow}$ decomposition (subsection A.1), all hyperparameters (subsubsection A.1.3), a worked episode (subsubsection A.1.4), and – critically – an open-weights backbone (Llama-3-70B-Instruct) that keeps the recipe portable. A step-by-step replication recipe for analogous domains (DO-178C avionics, IEC 62304 medical firmware, IEC 61131-3 industrial control) appears in subsection A.9. The architecture, offline-to-online curriculum, and dual-reward decomposition are model- and domain-agnostic; only the verifier and SFT corpus are domain-specific. The selected scenarios are illustrative validation cases and do not reflect the aggregate performance or reliability of the production fleet.

Impact Statement

This paper advances ML for safety-critical automotive software, facilitating faster diagnosis, accelerated feature development, and safer iteration. Potential societal benefits include rapid safety patch deployment, continuous feature delivery and optimization, improved fleet performance and user satisfaction, and reduced developer burden through higher levels of automation.

However, automating software evolution in cyber-physical systems introduces risks: non-deterministic LLM outputs or flawed updates could degrade safety, and increased software complexity can expand cybersecurity and accountability challenges. We mitigate these via shadow mode validation that decouples exploration from physical actuation, strict HITL gating for final approval, and periodic audits. Broader deployment should remain subject to rigorous verification, regulatory compliance, and organizational governance.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. Gpt-4 technical report. *arXiv preprint*

arXiv:2303.08774, 2023.

Anthropic. The claude 3 model family: Opus, sonnet, haiku, 2024. URL https://api.semanticscholar.org/CorpusID:268232499.

Ball, P. J., Smith, L., Kostrikov, I., and Levine, S. Efficient online reinforcement learning with offline data. In *International Conference on Machine Learning*, pp. 1577–1594. PMLR, 2023.

Barto, A. G. and Mahadevan, S. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13(4):341–379, 2003.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde de Oliveira Pinto, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv e-prints*, pp. arXiv–2107, 2021.

Dalal, G., Dvijotham, K., Vecerik, M., Hester, T., Paduraru, C., and Tassa, Y. Safe exploration in continuous action spaces. *arXiv preprint arXiv:1801.08757*, 2018.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., et al. The llama 3 herd of models. *arXiv e-prints*, pp. arXiv–2407, 2024.

Garcıa, J. and Fernández, F. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1):1437–1480, 2015.

Gräßler, I., Wiechel, D., Roesmann, D., and Thiele, H. V-model based development of cyber-physical systems and cyber-physical production systems. *Procedia Cirp*, 100: 253–258, 2021.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S. K. S., Lin, Z., et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*, 2023.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Kömmling, A., Jaunich, M., Goral, M., and Wolff, D. Insights for lifetime predictions of o-ring seals from five-year long-term aging tests. *Polymer Degradation and Stability*, 179:109278, 2020.

Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012. doi: 10.1109/TSE.2011.104.

Lee, S., Seo, Y., Lee, K., Abbeel, P., and Shin, J. Offline-to-online reinforcement learning via balanced replay and pessimistic q-ensemble. In *Conference on Robot Learning*, pp. 1702–1712. PMLR, 2022.

Levine, S., Kumar, A., Tucker, G., and Fu, J. Offline reinforcement learning: Tutorial, review, and perspectives on open problems, 2020. URL https://arxiv.org/abs/2005.01643.

Mechtaev, S., Yi, J., and Roychoudhury, A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 691–701, 2016. doi: 10.1145/2884781.2884807.

Nair, A., Dalal, M., Gupta, A., and Levine, S. Accelerating online reinforcement learning with offline datasets. *CoRR, vol. abs/2006.09359*, 2020.

Otto, S., Wlcek, M., and Wortmann, F. Towards conceptualizing software-defined vehicles: A systematic review and future research avenues. In *Proceedings of the Thirty-Third European Conference on Information Systems (ECIS 2025)*, 2025. URL https://aisel.aisnet.org/ecis2025/digitrans/digtrans/8. Paper 8.

Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, 2024.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

Smith, K., Shi, Y., and Santhanagopalan, S. Degradation mechanisms and lifetime prediction for lithium-ion batteries—a control perspective. In *2015 American Control Conference (ACC)*, pp. 728–730. IEEE, 2015.

Su, H., Diao, S., Lu, X., Liu, M., Xu, J., Dong, X., Fu, Y., Belcak, P., Ye, H., Yin, H., et al. Toolorchestra: Elevating intelligence via efficient model and tool orchestration. *arXiv preprint arXiv:2511.21689*, 2025.

Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D., and Kavukcuoglu, K. Feudal networks for hierarchical reinforcement learning. In *International conference on machine learning*, pp. 3540–3549. PMLR, 2017.

Wang, X., Li, B., Song, Y., Xu, F. F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., Singh, J., et al. Opendevin: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 3, 2024.

Xia, C. S., Wei, Y., and Zhang, L. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1482–1494. IEEE, 2023.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

A. Appendix

A.1. Reward Signal Decomposition and Regularization

The total reward R_t is a weighted sum of the immediate verification signal R_{fast} and the delayed alignment signal R_{slow}.

A.1.1. Immediate Reward (R_{fast})

The dense, immediate reward signal R_{fast} is calculated as:

$$R_{fast} = w_1 \cdot R_{\text{test}} + w_2 \cdot R_{\text{req}} - w_3 \cdot R_{\text{cost}} \tag{5}$$

Here, R_{test} measures functional correctness via automated tests, R_{req} imposes penalties for safety and functional requirement violations (linters/static analysis), and R_{cost} penalizes inference compute. The weights w_1, w_2, w_3 were determined via a grid search on a held-out validation set to optimally balance safety (critical) vs. feature velocity.

The immediate performance reward R_{test} is calculated strictly based on passing verification suites:

$$R_{test} = \frac{1}{N} \sum_{n=1}^{N} \mathbb{I}(\text{test}_n = \text{pass}) \tag{6}$$

where N is the number of test cases.

If any constraint (safety, functional, or non-functional) is violated during testing (e.g., static analysis failure, latency limits), the term R_{req} is set to a large negative constant (e.g., -100), effectively overriding any performance gains and terminating the episode.

The cost penalty R_{cost} accumulates per-step penalties that incentivize efficient trajectories without exposing the agent to its own "progress" as a learnable signal:

$$R_{cost} = \alpha \cdot N_{steps} + \beta \cdot \sum_i \mathbb{I}(\text{repeat}_i) + \zeta \cdot \mathbb{I}(\text{no\_state\_change}) \tag{7}$$

where N_{steps} counts total agent invocations within an episode, the second term penalizes redundant tool calls (e.g., re-reading the same file or re-issuing an identical query), and the third is an indicator for steps whose post-conditions match the pre-conditions (no observable change in the codebase, telemetry summary, or requirements set). Crucially, the agent never receives an explicit "progress" signal – it observes only this aggregate cost penalty alongside R_{test} and R_{req}. This formulation prevents reward hacking via spurious progress signals while still discouraging stalled or repetitive trajectories. We use $\alpha = 0.01$, $\beta = 0.05$, and $\zeta = 0.1$ throughout.

A.1.2. Delayed Alignment and Regularization (R_{slow})

The delayed alignment reward R_{slow} is computed as:

$$R_{slow} = w_a \cdot \mathbb{I}(C' \equiv_{func} C^{human}) + w_b \cdot \text{ASTMatch}(C', C^{human}) + w_c \cdot \text{Sim}_{emb}(C', C^{human}) \tag{8}$$

where $\mathbb{I}(C' \equiv_{func} C^{human})$ is a binary indicator of functional equivalence (passing the same test subset), ASTMatch measures structural tree edit distance, and Sim_{emb} is the cosine similarity between CodeBERT embeddings.

We treat R_{slow} not as a primary learning objective, but as a **regularization term** for maintainability. In long-lived safety-critical firmware, code readability and adherence to architectural patterns are not merely stylistic preferences but functional requirements for future human debugging. R_{slow} ensures that the agent does not converge towards "obfuscated" solutions that pass tests but degrade the codebase's long-term evolvability. In scenarios where human ground truth is unavailable, R_{slow} is set to 0. This approach effectively leverages implicit expert supervision without requiring manual review of every agent action.

Mitigating Specification Gaming: We acknowledge the limitation of using embedding similarity (R_{slow}) as a reward signal; while useful for stylistic alignment, it is not a proxy for functional correctness. However, R_{slow} serves as a critical counter-measure to **specification gaming**. Automated tests (R_{fast}) are inherently imperfect proxies for system intent;

without regularization, RL agents often discover "shortcuts" that satisfy the verifier but violate unwritten safety or logic constraints. Grounding the policy in human demonstrations prevents this by anchoring the agent to the manifold of safe, human-verified solutions. Therefore, we strictly gate policy updates on R_{fast} (verification results) to prevent the agent from optimizing for semantic similarity at the expense of logic.

Handling Divergent Solutions: A challenge in alignment-based rewards is the "false negative" scenario where the agent generates a valid solution that differs structurally from the human ground truth, yielding a low R_{slow}. However, since $\lambda \ll 1$, functional verification (R_{fast}) dominates the reward signal, allowing the agent to pursue structurally novel solutions provided they satisfy rigorous correctness checks. To distinguish between "lucky" hacks and valid alternative approaches, we implement a **periodic manual audit** of "High-Reward, Low-Alignment" trajectories. Divergent solutions that pass all automated checks are flagged for human review. If validated, they are added to the expert dataset, effectively expanding the manifold of accepted solutions. Conversely, "false positives" - where an agent mimics human style but fails functional tests - are automatically rejected by the strict gating on R_{fast}.

A.1.3. HYPERPARAMETERS

The global reward function utilizes the following weights: $w_1 = 0.45$ (Tests), $w_2 = 0.45$ (Requirements), and $w_3 = 0.1$ (Compute). For the delayed alignment signal R_{slow}, we set the component weights to $w_a = 0.5$ (Functional Equivalence), $w_b = 0.3$ (AST Match), and $w_c = 0.2$ (Embedding Similarity). The alignment coefficient is set to $\lambda = 0.1$ to ensure functional verification dominates. We prioritize safety and correctness equally. The RL training uses a discount factor $\gamma = 0.99$, a learning rate of $1e-5$, and a PPO clip range $\epsilon = 0.2$. The GAE parameter λ_{GAE} is set to 0.95. For generation, we use a temperature of 0.2 for code and 0.7 for the Orchestrator plan. LoRA rank is set to $r = 16$ with alpha 32.

A.1.4. END-TO-END WALKTHROUGH: A SINGLE EPISODE

To illustrate how the hierarchy executes a complete issue-to-PR lifecycle, we trace a single episode for the "Continuous Control Adaptation" scenario (subsection A.5). The goal $\mathcal{G}_t$ in this episode is the ticket: *"Window regulator on rear-left door fails closing-time spec (> 4.2 s observed; spec is 3.5 ± 0.2 s) on ∼2,400 vehicles past 18 k cycles. Investigate and patch."* We highlight how each agent contributes and how rewards are computed at the terminal step.

Step 1 – Orchestrator plan (π_{orch}): The Orchestrator observes $s_t = [\mathcal{D}_t, \mathcal{C}_t, \mathcal{K}_t, \mathcal{G}_t]$ and emits a three-step plan $z_t = \{(k_j, i_j)\}_{j=1}^{3}$:

- (DATA, "correlate closing-time with cycle-count and ambient temp");
- (REQ, "retrieve constraints binding window-regulator firmware");
- (CODE, "adjust PID gains in `window_motor.cpp` to restore spec without exceeding current limit").

The discrete action head selects the sub-agent k_j; the instruction tokens i_j are sampled from the high-level LLM at temperature $T = 0.7$.

Step 2 – Data Agent diagnosis (π_{data}): The Data Agent decomposes the instruction into computable metrics, generates a Spark query against the fleet telemetry lake, and reasons over the result. Output: *"Closing-time has drifted +0.7 s relative to factory baseline; the drift is uncorrelated with ambient temperature ($r = 0.04$) but strongly correlated with cumulative cycles ($r = 0.81$). Hypothesis: seal stiffening; recommend re-tuning K_p and K_i to compensate."* This diagnosis becomes part of the state passed to subsequent agents.

Step 3 – Requirements Agent retrieval (π_{req}): The Requirements Agent retrieves the active constraints binding `window_motor.cpp` via RAG: (i) closing-time spec 3.5 ± 0.2 s; (ii) peak motor current ≤ 4.5 A; (iii) anti-pinch reverse latency ≤ 100 ms. It returns these constraints to the Orchestrator as a structured set, ready for verification.

Step 4 – Code Agent generation, attempt 1 (π_{code}): The Code Agent samples a patch: `Kp = 2.5; Ki = 0.1;`. The candidate firmware C' is sent to the verification harness.

Step 5 – Layer 1 verification (deterministic): The CI/CD pipeline compiles C', runs the regression test suite, and executes static analyzers. *Closing-time test passes*, but the static current-overflow analyzer flags a potential violation of constraint (ii) under cold-start conditions.

Step 6 – Reward computation, attempt 1: Per Equation 7 and the R_{fast} definition, the layer-1 violation sets $R_{req} = -100$:

$$R_{fast} = 0.45(1) + 0.45(-100) - 0.1 \cdot R_{cost} \approx -44.6,$$

which terminates the episode. PPO computes a negative advantage and updates the Code Agent's LoRA weights to *decrease* the log-probability of $K_p = 2.5, K_i = 0.1$ in this state context. The Orchestrator is also updated, since its sub-agent selection led to a failed terminal step.

Step 7 – Subsequent episode, attempt 2: On the next presentation of a similar state, the Code Agent samples a more conservative patch: `Kp = 2.1; Ki = 0.05;`. All Layer 1 checks pass: $R_{test} = 1$, $R_{req} = 0$. The Layer 2 Requirements Agent then performs the semantic check; the patch satisfies all retrieved constraints. Final reward:

$$R_{fast} = 0.45(1) + 0.45(0) - 0.1 \cdot R_{cost} \approx 0.41.$$

PPO updates push the Code Agent's policy toward this action; the Orchestrator's plan is reinforced for similar future states.

Step 8 – Shadow PR and delayed alignment (R_{slow}): The agent emits a shadow PR on a detached branch. When the human engineer eventually merges C^{human} for the same ticket two days later, the system asynchronously computes R_{slow} – functional equivalence, AST match, and CodeBERT similarity – and folds it into the next periodic policy update with weight $\lambda = 0.1$. The dominant signal remains R_{fast}; R_{slow} acts as a regularizer toward maintainable, human-aligned style.

What this trace illustrates: (i) Each agent contributes a distinct, learnable subtask; the Orchestrator's high-level plan determines which subtask is invoked when. (ii) The deterministic Layer 1 gate is the fundamental safety mechanism; the Requirements Agent at Layer 2 is a semantic complement, not a substitute. (iii) The agent never observes the "correct" answer during the online phase; it discovers valid trajectories purely through reward feedback. (iv) The same offline-to-online curriculum is applied uniformly to the Orchestrator and all sub-agents – enabling the entire system to co-adapt to the verifier and to the human engineer's eventual solutions.

A.2. Agent Capabilities

To address the complexity of the automotive software domain, each agent in the AutoEvolve hierarchy is highly specialized with reasoning capabilities that extend beyond simple text generation.

1. **Data Sub-Agent:** The Data Sub-Agent is not merely a SQL generator; it functions as an analytical engine. Upon receiving a query (e.g., "Analyze the correlation between battery temperature and charging rate"), it performs a multi-step reasoning process:
   - **Metric Definition:** It first decomposes the high-level goal into specific, computable metrics (e.g., 'avg_temp_per_session', 'peak_charge_rate').
   - **Query Generation & Execution:** It generates optimized SQL queries to fetch these metrics from the data lake.
   - **Analysis & Reasoning:** Crucially, it does not just return raw rows. It analyzes the query results to identify trends, outliers, or anomalies, providing a synthesized insight back to the Orchestrator. This "reasoning over data" approach allows it to handle schema ambiguities by iteratively refining queries based on intermediate execution feedback.
2. **Code Sub-Agent:** This sub-agent acts as a fully capable software engineer with direct access to the repository.
   - **Environment:** It operates within a persistent shell environment with access to the full git history and file system.
   - **Tooling:** It utilizes standard Linux utilities ('grep', 'find', 'ls') and specialized code search tools to navigate the codebase, identify relevant files, and trace function definitions.
   - **Action Space:** It can read files, apply patches, run compilers, and execute unit tests. This allows it to not only generate code but also validate its own changes and fix syntax errors autonomously.

3. **Requirements Sub-Agent:** This sub-agent serves as the compliance guardrail.
    - **Knowledge Base:** It is backed by a RAG (Retrieval-Augmented Generation) system indexing the complete corpus of homologation standards, functional safety requirements, and internal system specifications.
    - **Verification:** It validates proposed changes against these retrieved constraints to ensure that performance optimizations do not compromise safety or regulatory compliance.

A.3. Training Implementation Details

We employ a two-stage training pipeline consisting of SFT followed by Online RL.

A.3.1. Phase 1: Supervised Fine-Tuning (SFT)

For the SFT phase, we treat historical developer traces as expert demonstrations. The objective is the standard autoregressive language modeling loss:

$$\mathcal{L}_{SFT} = -\sum_{t} \log P(x_t | x_{<t}; \theta) \tag{9}$$

where x represents the tokenized sequence of the issue description followed by the resolution code. We format the data using the standard instruction-tuning template compatible with the Llama-3 family. We train for 3 epochs with a learning rate of $2e-5$ and a cosine decay schedule, utilizing the same LoRA configuration as the subsequent RL phase to ensure smooth transfer.

A.3.2. Phase 2: Online Reinforcement Learning

We utilize PPO to train both the Orchestrator policy π_{orch} and the specialized sub-agent policies π_{sub}. The sub-agents can also be fine-tuned using a filtered rejection sampling approach to stabilize the initial online learning phase.

Episode and Horizon Specification: An RL episode begins when an incoming issue $\mathcal{G}_t$ is dispatched to the Orchestrator and ends with one of three terminal conditions: (i) a candidate firmware C' is submitted for verification, (ii) the verification harness returns a fatal R_{req} violation that terminates the episode, or (iii) the step budget $T_{max} = 25$ is exhausted. Across our internal benchmark, successful episodes consume 5–15 sub-agent invocations (median 9, 95th percentile 17). Each invocation is itself a multi-step rollout for the corresponding sub-agent (e.g., the Code Agent may execute several shell commands before returning), but credit assignment treats each Orchestrator-level invocation as one high-level decision step.

Credit Assignment Across the Hierarchy: Sparse terminal rewards R_{fast} are propagated to earlier planning steps via Generalized Advantage Estimation (GAE) with $\gamma = 0.99$ and $\lambda_{GAE} = 0.95$. The Orchestrator and each sub-agent maintain their own value head; the high-level policy is updated against returns computed at the Orchestrator timescale (one step per sub-agent invocation), while sub-agent policies are updated against returns computed at the token timescale within their own rollouts. The dense R_{cost} signal (Equation 7) shapes the trajectory at every step and therefore does not require credit propagation. The delayed signal R_{slow} is applied asynchronously in periodic batch updates rather than per episode, since human ground-truth commits arrive on a much longer timescale than agent execution.

A.3.3. Orchestrator Optimization

The Orchestrator's action space is defined as the selection of a sub-agent k and a generated instruction i. To make this tractable for PPO, we treat the sub-agent selection as a discrete action head and the instruction generation as a continuous token generation process. The objective function is:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{10}$$

where $r_t(\theta)$ is the probability ratio and $\hat{A}_t$ is the estimated advantage. The advantage is computed using the Generalized Advantage Estimation (GAE) based on the reward signals returned by the verification environment.

A.4. Algorithmic Pseudocode

To support reproducibility despite proprietary data constraints, we provide algorithmic pseudocode for the core control loops of AutoEvolve. Algorithm 1 specifies the outer shadow-mode loop that processes incoming issues and produces both immediate (R_{fast}) and delayed (R_{slow}) reward signals. Algorithm 2 specifies the inner Orchestrator/sub-agent dispatch within a single episode.

Algorithm 1 Shadow-Mode Online Learning Loop

Require: Initialized policies $\pi_{orch}, \pi_{sub} = \{\pi_{data}, \pi_{req}, \pi_{code}\}$ from SFT phase; live issue stream $\mathcal{I}$; verification harness $\mathcal{V}$; replay buffer $\mathcal{B}$.

while shadow mode active do
  Sample issue $\mathcal{G}_t \sim \mathcal{I}$ (parallel to human engineer).
  Construct state $s_t = [\mathcal{D}_t, \mathcal{C}_t, \mathcal{K}_t, \mathcal{G}_t]$.
  $(C', \tau) \leftarrow \textsc{RunEpisode}(s_t, \pi_{orch}, \pi_{sub})$ // Algorithm 2
  $R_{fast} \leftarrow \mathcal{V}(C', s_t)$ // Layer 1 + Layer 2 verification
  Append (τ, R_{fast}) to $\mathcal{B}$.
  if $R_{fast} > 0$ **and** candidate passes Layer 1 **then**
    Emit shadow PR; tag for asynchronous R_{slow} alignment.
  end if
  if human commit C^{human} arrives for ticket $\mathcal{G}_t$ **then**
    $R_{slow} \leftarrow w_a \mathbb{I}_{func}(C', C^{human}) + w_b \text{ASTMatch}(C', C^{human}) + w_c \text{Sim}_{emb}(C', C^{human})$
    Update buffer entry with $R_t \leftarrow R_{fast} + \lambda R_{slow}$.
  end if
  if $|\mathcal{B}| \geq B_{batch}$ **then**
    Run PPO update on π_{orch}, π_{sub} using $\mathcal{B}$ with GAE(γ, λ_{GAE}).
    $\mathcal{B} \leftarrow \emptyset$.
  end if
end while

Algorithm 2 Single-Episode Orchestrator/Sub-Agent Dispatch

Require: State s_t, Orchestrator policy π_{orch}, sub-agent policies π_{sub}, max steps T_{max}.

Initialize trajectory $\tau \leftarrow [\,]$, context $\mathcal{X}_0 \leftarrow s_t$.
for $j = 1, \ldots, T_{max}$ **do**
  Sample high-level action $(k_j, i_j) \sim \pi_{orch}(\cdot \mid \mathcal{X}_{j-1})$.
  if $k_j =$ TERMINATE **then**
    Submit candidate firmware $C' \leftarrow \mathcal{X}_{j-1}$.patch.
    break.
  end if
  Execute sub-agent: $o_{k_j} \sim \pi_{k_j}(\cdot \mid s_t, i_j)$.
  Update context: $\mathcal{X}_j \leftarrow \mathcal{X}_{j-1} \cup \{o_{k_j}\}$.
  Append (s_t, k_j, i_j, o_{k_j}) to τ.
  if o_{k_j} contains a Layer 1 hard violation **then**
    break // early termination on irrecoverable failure
  end if
end for
return C' (or NULL if no candidate produced), trajectory τ.

The constants B_{batch}, T_{max}, and the GAE hyperparameters are reported in subsubsection A.1.3. The shadow-mode loop is implemented in LangGraph; sub-agent rollouts use the TRL library's PPO trainer with LoRA adapters at $r = 16$. The same loop is applied uniformly during both Phase 1 (SFT, with the reward replaced by the behavior-cloning loss $\mathcal{L}_{BC}$ and policy updates restricted to LoRA weights) and Phase 2 (online RL, as written above), enabling smooth transition between phases without architectural changes.

A.5. Evaluation Scenarios Detail

We evaluate the system on the following specific scenarios. These are illustrative examples only; due to proprietary competitive data, we cannot disclose the exact task specifics or datasets.

1. **Continuous Control Adaptation:** *Scenario:* A window regulator motor's performance degrades due to seal stiffening. *Goal:* The agent must detect the deviation in 'closing_time' and adjust the PID controller's K_p and K_i gains to restore the target $3.5\text{s} \pm 0.2\text{s}$ without exceeding the motor current limit.

2. **Feature Enhancement:** *Scenario:* A new requirement mandates a "Comfort Mode" for the door latch, where the unlatch duration should be extended to reduce noise. *Goal:* The agent must modify the state machine logic in 'door_manager.cpp' to introduce a configurable delay parameter and expose it to the calibration interface.

3. **Safety-Critical Bug Fix:** *Scenario:* A race condition in the sensor fusion layer causes intermittent object dropouts when the radar and camera timestamps drift by $> 50ms$. *Goal:* The agent must diagnose the thread synchronization issue in 'fusion_core.cpp' and implement a mutex lock or a timestamp alignment buffer to ensure data consistency.

4. **Resource Optimization:** *Scenario:* The infotainment background service is consuming excessive CPU cycles ($> 15\%$) during idle mode, causing battery drain. *Goal:* The agent needs to profile the 'media_service.cpp', identify the busy-wait loop, and refactor it to use an interrupt-driven or event-based architecture.

5. **Security Hardening:** *Scenario:* A vulnerability scan identifies an outdated OpenSSL library in the telematics unit with a known CVE. *Goal:* The agent must identify the dependency in 'CMakeLists.txt', upgrade the library version, and refactor any deprecated API calls in the networking stack to ensure secure connectivity.

6. **Compliance and Localization:** *Scenario:* New EU regulations require a specific sequential turn signal pattern for vehicles sold in the region. *Goal:* The agent must modify the lighting control logic in 'exterior_lights.cpp' to implement the required pulse width modulation (PWM) sequence, verified against the regulatory timing constraints.

A.6. Time Measurement Methodology

Note that while these estimates are derived from our historical trajectories dataset, the details mentioned are not representative of the company's actual operational workflows or proprietary metrics.

To ensure a fair comparison between human and agent performance in Table 1, we employed distinct methodologies for estimating active effort.

Human Active Time Estimation: Directly measuring human cognitive effort is challenging due to the interleaving of tasks, meetings, and interruptions. We derived the *Active Time* estimates from a retrospective analysis of historical Jira tickets corresponding to the task categories in subsection A.5. For each ticket, we calculated the duration between the 'In Progress' and 'Review' state transitions. To isolate *active* effort from *elapsed* time, we:

1. Filtered out non-working hours (nights and weekends).

2. Applied a conservative *Focus Factor* of $\alpha = 0.65$, a metric accounting for administrative overhead and context switching.

3. Cross-validated these estimates via a survey of senior firmware engineers, who provided lower-bound estimates for the "ideal" time required to diagnose and implement these fixes without interruption.

The resulting estimate (e.g., $\sim$1 day for diagnosis) represents a highly optimized human baseline, assuming deep familiarity with the codebase. We explicitly acknowledge that deriving active time via a focus factor is a heuristic approximation; however, we validate this proxy against expert survey data to ensure it reflects a realistic lower bound for skilled human performance.

Sensitivity Analysis: To verify that our efficiency conclusions are robust to the choice of α, we recomputed the human *Active Time* estimates in Table 1 across the range $\alpha \in \{0.50, 0.65, 0.80\}$. Recall Active Time $= \alpha \times$ (filtered working-hours elapsed), so a lower α implies more administrative overhead and therefore *less* productive time per elapsed working hour – yielding a smaller human active-time baseline and a smaller apparent speedup. The bookend values correspond to (i) a heavily interrupted workflow ($\alpha = 0.50$, lower bound from Jira analysis of fragmented tickets) and (ii) an idealized deep-focus engineer with minimal overhead ($\alpha = 0.80$, upper bound from senior-engineer survey).

Table 3. Focus Factor Sensitivity. AutoEvolve's order-of-magnitude DCT speedup is robust across plausible values of α: the qualitative conclusion holds even under the most conservative assumption ($\alpha = 0.50$, $\sim$15$\times$).

α	HUMAN ACTIVE DCT	AUTOEVOLVE ACTIVE DCT	DCT SPEEDUP
0.50	$\sim$ 3.9 DAYS	$<$ 6 HOURS	$\sim$15$\times$
0.65 (DEFAULT)	$\sim$ 5.0 DAYS	$<$ 6 HOURS	$\sim$20$\times$
0.80	$\sim$ 6.2 DAYS	$<$ 6 HOURS	$\sim$25$\times$

The DCT speedup varies proportionally with α, but the order-of-magnitude headline holds even under the most conservative setting. The structural advantage in *Organizational Cycle Time* (1–2 months vs. <6 hours) is independent of α, since it reflects elimination of meeting/sleep/weekend latencies rather than active-effort accounting.

AutoEvolve Active Time Measurement: For the agentic system, time was measured deterministically as the wall-clock duration of the execution pipeline: $T_{active} = T_{inference} + T_{retrieval} + T_{verification}$

- $T_{inference}$: The cumulative latency of token generation for the Orchestrator and Sub-agents.
- $T_{retrieval}$: Time spent executing SQL queries and RAG lookups.
- $T_{verification}$: The execution time of the CI/CD pipeline (compilation, static analysis, and regression testing).

Notably, $T_{verification}$ dominates the agent's active time ($\sim$80%). Since agents do not suffer from context switching or fatigue, their "Active" and "Elapsed" times are effectively identical during the execution window.

A.7. Detailed Comparison to Related Multi-Agent Frameworks

We expand on the three-axis comparison summarized in §2.

Coordination paradigm: ChatDev (Qian et al., 2024) and MetaGPT (Hong et al., 2023) use *collaborative* multi-agent dynamics with fixed Standard Operating Procedures encoded as prompt templates; agents do not learn or adapt from environment feedback. AutoEvolve's H-MARL formulation jointly learns the Orchestrator's high-level policy π_{orch} and each sub-agent's low-level policy π_{sub} via SFT initialization followed by PPO refinement (Section 4), so coordination is a learned competence rather than a hand-crafted protocol.

Safety enforcement: SWE-agent's (Yang et al., 2024) Agent-Computer Interface and OpenDevin's (Wang et al., 2024) tool layer are designed for general code-repair tasks where post-hoc test failures are the dominant safety signal. AutoEvolve replaces this open-loop assumption with an *adversarial* Requirements Agent (subsubsection 4.1.3) that can reject candidates against formal safety, homologation, and timing constraints *before* they enter the verification harness, complemented by a deterministic Layer-1 hard gate.

Online adaptation: Reflexion (Shinn et al., 2023) achieves test-time learning via verbal reinforcement against textual feedback. AutoEvolve achieves *policy-level* learning via shadow-mode RL with PPO updates driven by deterministic verification outcomes plus delayed human-engineer commits, which avoids the brittleness of verbal-feedback test-time learning while enabling continuous post-deployment improvement.

A.8. Layered Safety Architecture: Full Specification

We expand on the three-layer safety design summarized in subsubsection 4.1.3. A natural concern with LLM-based agents is that the system might rely on a learned model to enforce hard safety constraints, which would amount to circular reasoning

if the same model also generated the candidate. To preempt this, AutoEvolve enforces safety through three independent layers of increasing semantic richness, where every higher layer is *strictly subordinate* to the lower layers:

- **Layer 1 – Deterministic Hard Gate:** Static linters (MISRA-C, AUTOSAR), compiler checks, regression test suites, memory-safety analyzers, and timing-constraint checkers. These tools are external to the agent and produce binary pass/fail outcomes. Any failure at this layer immediately sets R_{req} to a large negative constant and terminates the episode. *No LLM output can override or bypass this layer.*
- **Layer 2 – Semantic Critic (Requirements Agent):** A learned LLM critic that reasons over natural-language safety and homologation requirements via RAG. It catches semantic violations that pass syntactic checks but contradict requirement intent (e.g., a patch that compiles but bypasses thermal limits described in plain text). Crucially, Layer 2 *cannot approve* a candidate that fails Layer 1 – it can only reject, never override.
- **Layer 3 – Human-in-the-Loop Gate:** Before any candidate firmware reaches a physical vehicle, a senior engineer reviews the shadow PR. This is a hard policy gate, not part of training; the agent's reward signal does not depend on whether a human ultimately merges the PR.

Avoiding circularity in training and evaluation: The training signal R_{req} comes *exclusively* from Layer 1 deterministic tools (Equation 7, subsection A.1); the Code Agent never sees Layer 2's judgments during PPO updates. Layer 2 is used at evaluation time to compute the reported *Requirement Violation Rate* (Table 2), measuring semantic violations that pass Layer 1 but contradict requirement intent. Because the training signal and the evaluation signal come from independent mechanisms, there is no pathway for the Code Agent to overfit to Layer 2's preferences.

A.9. Replication Recipe for Analogous Proprietary Domains

A practitioner with access to analogous proprietary data in another safety-critical CPS domain can instantiate the AutoEvolve framework end-to-end by following the recipe below. The architecture, the offline-to-online curriculum, and the dual-reward decomposition are model- and domain-agnostic; only the Layer-1 verifier and the SFT corpus are domain-specific.

Step 1 – Construct the expert trajectory dataset $\mathcal{T}_{expert}$: From the organization's issue tracker, version-control history, and CI pipeline logs, extract tuples $(s_0, a_0^*, \ldots, r_{final})$ where the human engineer's eventual merge serves as a^*. Pair each ticket with the relevant proprietary telemetry slice that motivated the change. This corpus drives the Behavior Cloning loss in subsubsection 4.2.1.

Step 2 – Replace the Layer-1 deterministic gate with the domain's verification harness: Avionics teams should substitute MISRA-C/AUTOSAR with DO-178C-compliant unit-test runners, MC/DC coverage analyzers, and WCET tools. Medical-device teams should substitute IEC 62304-aligned static analyzers and IEC 60601 hardware-in-the-loop fixtures. Industrial-control vendors should plug in their PLC validators and IEC 61131-3 compliance checkers. The interface contract is unchanged: a binary pass/fail verdict feeds R_{req}.

Step 3 – Configure the Requirements Agent's RAG corpus over domain certification standards: Index the homologation/certification documents that govern the target system (e.g., FAA Part 25 for transport-category aircraft; FDA 510(k) submissions for medical devices; UN/ECE regulations for vehicles). The Layer-2 critic uses this corpus to flag semantic violations that pass Layer 1.

Step 4 – Run the offline-to-online curriculum: Execute SFT on $\mathcal{T}_{expert}$ following subsubsection 4.2.1 with the hyperparameters in subsubsection A.1.3, then transition to shadow-mode PPO using Algorithm 1. The reward weights $w_1 = w_2 = 0.45$, $w_3 = 0.1$, $\lambda = 0.1$ are reasonable starting points; tune to the local cost-of-error profile.

Step 5 – Compare against domain-specific baselines using Table 2 as a template: At minimum, run a Monolithic ReAct baseline on the same backbone with equivalent tool access, and at least one task-agnostic multi-agent framework (MetaGPT, SWE-Agent, OpenDevin). The architecture-vs-training ablation – isolating the SFT-Only configuration of AutoEvolve – enables clean attribution of safety gains to the adversarial Requirements Agent vs. to online RL.

Step 6 – Maintain the Layer-3 HITL gate before any production deployment: The shadow-PR-then-human-merge pattern decouples agent exploration from physical actuation and is the load-bearing safety guarantee for any in-the-wild rollout.

The total engineering cost of this recipe is dominated by Step 2 (verifier integration) and Step 3 (RAG corpus curation); Steps 1, 4, and 5 follow templated code in subsection A.4. Open-weights backbones (Llama-3-70B-Instruct in our case) keep the recipe air-gap-compatible; substituting closed-source APIs is feasible only where data-residency policy permits.