Goose Swarm: A Leaderless, Peer-Based Swarm Intelligence Framework for Autonomous Software Development

A Technical Whitepaper on Distributed Agent Collaboration

Block, Inc.
Goose Open Source Project

September 11, 2025

Abstract

Goose Swarm is a distributed agent coordination framework for autonomous software development. Multiple agent instances collaborate through a shared environment that serves as a state machine for synchronization and work allocation. The architecture eliminates centralized control—agents independently discover work, claim tasks atomically, and coordinate exclusively through environmental modifications.

The system employs three dynamic roles: Planners decompose complex problems into parallel tasks, Drones execute implementation work, and Evaluators assess and integrate contributions. Agents switch roles based on available work, enabling efficient resource utilization without central scheduling. This peer-based approach achieves linear scalability and fault tolerance through emergent coordination patterns rather than explicit orchestration.

Initial deployments demonstrate effective parallel task execution, automatic failure recovery, and successful completion of real-world development tasks. The framework represents a practical application of swarm intelligence principles, showing that complex software engineering can be accomplished through simple local interactions and stigmergic coordination.

1 Introduction

Software development is getting complex fast. We need automation that can scale, but also adapt when things change. Traditional CI/CD pipelines and testing frameworks work fine for routine stuff, but they're built around central orchestration and predetermined workflows. When requirements shift, unexpected issues pop up, or you need creative problem-solving, these systems struggle.

Goose Swarm takes a different approach. Instead of central control, it uses swarm intelligence—the same principles you see in ant colonies, bird flocks, and fish schools. These natural systems achieve remarkable collective behaviors through simple local interactions, without any central authority telling them what to do. Goose Swarm agents work the same way: they make decisions based on what they can see locally, but together they accomplish system-wide goals that emerge from their collective behavior.

1.1 Motivation

The development of Goose Swarm addresses several critical limitations in existing automated development tools:

1. **Scalability Bottlenecks**: Centralized systems create single points of failure and performance constraints as the number of tasks or agents increases.

2. **Rigid Workflows**: Traditional automation requires predefined pipelines that struggle to adapt to novel situations or changing requirements.
3. **Coordination Overhead**: As teams and codebases grow, the complexity of coordinating development efforts increases exponentially.
4. **Resource Utilization**: Fixed allocation of resources fails to adapt to varying workload demands, leading to inefficient use of computational resources.

1.2 Contributions

This paper presents the following key contributions:

- A fully decentralized agent architecture that eliminates the need for central orchestration while maintaining coherent system behavior.
- A role-based agent model where agents dynamically assume responsibilities (Planner, Drone, Evaluator) based on available work and system state.
- A novel coordination mechanism using GitHub's issue tracking system as a distributed state machine for agent synchronization.
- Implementation of recipe-based task specifications that enable complex workflows while preserving agent autonomy.
- Empirical validation through real-world deployments demonstrating effective parallelization and task completion without explicit coordination.

2 System Architecture

The Goose Swarm architecture embodies principles of distributed systems design, swarm intelligence, and autonomous agent theory. The system operates without central control, relying instead on emergent behaviors arising from local agent interactions with a shared environment.

2.1 Core Design Principles

2.1.1 Leaderless Operation

Unlike master-slave or hierarchical architectures, Goose Swarm implements a truly peer-based system where no agent holds permanent authority over others. Leadership emerges temporarily and contextually when agents claim specific roles for particular tasks.

2.1.2 Stigmergic Coordination

Agents coordinate through stigmergy—indirect communication through environmental modifications. In Goose Swarm, GitHub issues serve as the environment, with labels, comments, and issue states providing the stigmergic signals that guide agent behavior.

2.1.3 Autonomous Decision Making

Each agent maintains complete autonomy in decision-making, selecting work based on local observations and internal heuristics. This autonomy ensures system resilience—the failure of any single agent does not compromise overall system operation.

Algorithm 1 Goose Swarm Agent Lifecycle

```
1: Initialize agent with unique identifier
2: Register presence in swarm tracking issue
3: while active do
4:     Poll for available work
5:     if work detected then
6:         Determine required role
7:         Attempt to claim work atomically
8:         if claim successful then
9:             Execute role-specific behavior
10:            Update environmental state
11:            Release claim if transitioning phases
12:        end if
13:    else
14:        Wait for polling interval
15:    end if
16: end while
```

2.2 Agent Lifecycle

The agent lifecycle consists of several distinct phases:

2.3 Communication Protocol

Agents communicate exclusively through modifications to the GitHub repository state. This includes:

- **Issue Creation**: Generating new work items or planning issues
- **Label Management**: Adding/removing labels to signal work status
- **Body Modifications**: Claiming work through issue body updates
- **Pull Request Creation**: Submitting completed work for integration
- **Comment Posting**: Providing status updates and additional context

This approach ensures all communication is persistent, auditable, and accessible to both human observers and other agents.

3 Agent Roles and Behaviors

The Goose Swarm system defines three primary agent roles, each with distinct responsibilities and behavioral patterns. Agents dynamically assume these roles based on available work, without permanent assignment or specialization.

3.1 Planner Agents

Planner agents serve as the system's strategic layer, decomposing complex issues into manageable, parallelizable tasks. When a planner claims an issue, it:

1. **Context Analysis**: Loads issue content, comments, and repository state to understand requirements comprehensively.

2. **Node Assessment**: Counts available drone agents to optimize task distribution for parallel execution.

3. **Task Decomposition**: Breaks down the issue into independent work items, creating task files that specify clear objectives and implementation guidance.

4. **Issue Generation**: Creates new planning issues for complex sub-components requiring additional analysis.

5. **State Transition**: Marks issue as "in progress" and releases claim, allowing any peer to evaluate when ready.

The planner's decomposition strategy adapts to the number of available workers, balancing granularity with coordination overhead. This dynamic adjustment ensures efficient resource utilization regardless of swarm size.

After creating tasks, the planner releases its claim on the issue, enabling any available peer to assume the evaluator role once all tasks are complete.

3.2 Drone Agents

Drone agents represent the system's execution layer, implementing specific tasks identified by planners. Their operational cycle includes:

1. **Task Discovery**: Identifying unclaimed tasks marked with appropriate labels.

2. **Workspace Preparation**: Cloning or updating repository copies in isolated environments to prevent conflicts.

3. **Implementation**: Executing task requirements using recipe-based instructions and contextual information.

4. **Contribution Submission**: Creating pull requests that link back to original issues, maintaining traceability.

5. **Failure Handling**: Releasing claims and restoring labels if unable to complete tasks, allowing other agents to attempt the work.

Drones operate with complete autonomy within their claimed tasks, making implementation decisions based on local context and specified requirements.

3.3 Evaluator Agents

Evaluator agents represent the system's quality assurance and integration layer. Unlike the initial design where evaluation was tied to the planner role, the current implementation allows any peer in the swarm to assume the evaluator role when an issue is ready for evaluation. This occurs when:

- An issue is marked as "in progress" (indicating planning is complete)
- All associated task issues have submitted pull requests
- No other agent has claimed the evaluation role

When an evaluator claims an issue, it performs:

- **PR Assessment**: Analyzing submitted pull requests for correctness, completeness, and adherence to requirements.

- **Integration Decisions**: Determining whether to merge, request changes, or close pull requests based on quality criteria.
- **Follow-up Generation**: Creating new issues for identified problems or improvements discovered during evaluation.
- **Cleanup Operations**: Closing completed task issues and updating parent issue status.

This decoupled evaluation approach improves system efficiency by allowing any available agent to perform evaluation, rather than waiting for the original planner to become available again.

4 Coordination Mechanisms

The absence of central coordination requires sophisticated mechanisms for maintaining system coherence and preventing conflicts. Goose Swarm implements several key coordination strategies:

4.1 Work Claiming Protocol

To prevent multiple agents from working on the same task, the system implements an atomic claiming mechanism:

Algorithm 2 Atomic Work Claiming Protocol

```
1: Remove "help wanted" label from issue
2: Wait random interval (1-30 seconds)
3: Check if issue already claimed
4: if not claimed then
5:     Update issue body with agent identifier
6:     Proceed with work execution
7: else
8:     Restore "help wanted" label
9:     Return to work discovery
10: end if
```

The random wait interval reduces collision probability when multiple agents simultaneously discover the same work item.

4.2 State Synchronization

Agents maintain synchronization through continuous polling of repository state. This pull-based approach ensures:

- **Eventual Consistency**: All agents eventually observe the same system state
- **Fault Tolerance**: No dependency on message delivery or agent availability
- **Scalability**: Polling frequency can be adjusted based on system load

4.3 Conflict Resolution

When conflicts arise (e.g., multiple agents attempting to claim the same work), the system relies on GitHub's atomic operations and timestamp-based ordering to resolve disputes. The first agent to successfully modify the issue state proceeds, while others gracefully retreat and seek alternative work.

5 Recipe-Based Task Execution

Goose Swarm employs a recipe system that encapsulates complex workflows into reusable, parameterizable specifications. This approach balances structure with flexibility, enabling agents to execute sophisticated tasks while maintaining autonomy.

5.1 Recipe Structure

Recipes are defined in YAML format with the following components:

```yaml
version: 1.0.0
title: Task Execution Recipe
description: Template for autonomous task execution
instructions: |
  Detailed instructions for agent behavior
  including context variables and objectives
prompt: |
  Parameterized prompt template with
  {{ variable }} placeholders
settings:
  model: gpt-4
  temperature: 0.7
extensions:
  - developer
  - github
```

Listing 1: Recipe Structure Example

5.2 Parameter Substitution

Recipes support dynamic parameter substitution, allowing agents to customize execution based on context:

- `{{repo}}`: Repository identifier
- `{{issue_number}}`: Current issue being processed
- `{{worker_id}}`: Agent's unique identifier
- `{{context}}`: Issue content and comments
- `{{workspace}}`: Local working directory

5.3 Execution Flow

Recipe execution follows a structured flow:

1. **Parameter Resolution**: Substitute template variables with actual values
2. **Environment Setup**: Prepare working directories and load extensions
3. **Prompt Execution**: Process the parameterized prompt through the configured model
4. **Action Implementation**: Execute resulting commands and file operations
5. **Result Validation**: Verify successful completion and handle failures

6 Implementation Details

The Goose Swarm system is implemented in Rust, leveraging the language's performance characteristics and memory safety guarantees. Key implementation aspects include:

6.1 Technology Stack

- **Core Language**: Rust for system implementation
- **Async Runtime**: Tokio for concurrent operations
- **GitHub Integration**: GitHub CLI for repository interactions
- **AI Models**: Multiple LLM providers (GPT-4, Claude, etc.)
- **Serialization**: Serde for data structure handling

6.2 Agent Identity Management

Agents generate unique identifiers combining:

- Descriptive adjectives (e.g., "brave", "clever")
- Timestamp components for uniqueness
- Animal nouns for memorability (e.g., "fox", "owl")

Example: `clever-0910-fox`

This human-readable format aids in debugging and monitoring while ensuring uniqueness across the swarm.

6.3 Workspace Isolation

Each agent maintains isolated workspaces to prevent conflicts:

```
~/.local/share/goose-swarm/
  |-- repo-name-worker-id/
  |    \-- (cloned repository)
  |-- planner-work-issue-number/
  |    |-- tasks/
  |    \-- issues/
  \-- eval-work-issue-number/
       \-- issues/
```

6.4 Error Handling and Recovery

The system implements comprehensive error handling:

- **Transient Failures**: Automatic retry with exponential backoff
- **Claim Failures**: Graceful retreat and alternative work selection
- **Execution Failures**: Work unclaiming and label restoration
- **Network Failures**: Continued local operation with periodic retry