Nilla AI Agent Architecture

This document describes the AI agent system powering Nilla's intelligent features. These are task-specific agents designed to help open source contributors succeed, not general-purpose chatbots.

Why Agents, Not a Chatbot

Nilla uses **specialized agents** rather than a conversational chatbot for several important reasons:

1. Task-Oriented Design

Each agent is purpose-built for a specific job. They don't engage in open-ended conversation, they receive structured inputs, make decisions, and return structured outputs. This makes them:

- **Predictable**: Same inputs produce consistent, reliable outputs
- **Testable**: Structured I/O enables automated testing and validation
- **Focused**: No risk of going off-topic or hallucinating unrelated content

2. Structured Input/Output Contracts

Every agent has Zod-validated schemas for both input and output. This ensures:

- Type safety throughout the application
- Clear API contracts between frontend and AI
- Graceful fallbacks when LLM output is malformed

3. Domain-Specific Prompting

Each agent has carefully crafted system prompts with:

- Specific personas (e.g., "supportive commitment coach")
- Clear rules and constraints
- Output format specifications
- Context-appropriate behavior based on user state

4. Deterministic Logic + LLM Reasoning

Agents combine deterministic calculations (time remaining, progress percentage, risk levels) with LLM reasoning (personalized messages, explanations). This hybrid approach provides:

- Reliable metrics the UI can depend on
- Human-like coaching and explanation quality
- Auditability of the decision-making process

Agent Overview

Agent	Purpose	RAG-Enabled
Issue Explainer	Breaks down GitHub issues for contributors at their skill level	Yes
Commitment Coach	Provides personalized guidance to keep contributors on track	No

Issue Recommender	Suggests the best-fit issue from a list of candidates	No

---

Issue Explainer Agent

Purpose: Help contributors understand GitHub issues by providing plain-English explanations, expected outcomes, and actionable guidance tailored to their experience level.

Inputs

```
interface IssueExplainerInput {
  issue: {
    title: string;           // Issue title
    body?: string;           // Issue description (truncated to 3000 chars)
    labels: string[];        // Labels like "bug", "good-first-issue"
    repository: string;      // "owner/repo" format
    url: string;             // Link to the issue
  };
  user: {
    experienceLevel: "beginner" | "intermediate" | "advanced";
  };
  repoContext?: string;      // RAG-retrieved documentation (optional)
}
```

Outputs

```
interface IssueExplainerOutput {
  summary: string;              // Plain-English explanation of the issue
  expectedOutcome: string;      // What "done" looks like
  repoGuidelines: string[];     // Repo-specific rules from documentation
  beginnerPitfalls: string[];   // Common mistakes to avoid
  suggestedApproach: string;    // Step-by-step guidance
  keyTerms: Array<{             // Technical terms explained
    term: string;
    definition: string;
  }>;
  confidenceNote: string;       // What to verify with maintainers
}
```

Decisions Made

1. **Explanation Depth**: Adapts language complexity based on experience level
    - Beginners: Simple language, defines jargon, assumes no Git/PR knowledge
    - Intermediate: Clear but not basic, focuses on project-specific conventions
    - Advanced: Concise, skips basics, highlights nuances
2. **Guideline Extraction**: Only includes repo guidelines that actually appear in the provided RAG context—never invents rules

3. **Pitfall Identification**: Proactively warns about common mistakes relevant to the issue type and user level
4. **Confidence Assessment**: Distinguishes between what the agent is confident about vs. what the contributor should verify

RAG Integration

The Issue Explainer is the **only agent that uses RAG**. When invoked:

1. The API fetches the tracked repository's ID
2. Calls `retrieveRepoContext()` with the issue title and body
3. Performs vector similarity search against ingested repo documentation
4. Injects matching chunks into the prompt as `repoContext`

If RAG context is available, the agent extracts specific contribution guidelines. If not, `repoGuidelines` is explicitly set to an empty array to prevent hallucination.

Commitment Coach Agent

Purpose: Provide personalized, actionable coaching to help contributors stay on track with their 7-day commitments.

Inputs

```
interface CommitmentCoachInput {
  commitment: {
    id: string;
    issueTitle: string;
    issueUrl: string;
    repository: string;
    createdAt: string;           // ISO timestamp
    deadlineAt: string;          // ISO timestamp (7 days from creation)
    currentMilestone: Milestone;
    milestonesCompleted: Milestone[];
    lastActivityAt?: string;
  };
  user: {
    username: string;
    totalCommitments?: number;
    completedCommitments?: number;
    timezone?: string;
  };
  currentTime?: string;          // For testing/reproducibility
}

type Milestone =
  | "not_started"
  | "read_issue"
  | "ask_question"
  | "work_on_solution"
```

```
  | "open_pr"
  | "completed";
```

Outputs

```
interface CommitmentCoachOutput {
  nextAction: {
    action: string;              // Specific next step to take
    why: string;                 // Why this matters now
    estimatedMinutes?: number;   // Time estimate
  };
  nudge: {
    message: string;             // Encouraging coach message
    tone: "encouraging" | "motivating" | "celebratory" | "urgent" | "supportive";
  };
  riskAssessment: {
    level: "on_track" | "needs_attention" | "at_risk" | "critical";
    reason: string;
    daysRemaining: number;
    hoursRemaining: number;
  };
  warning?: {                    // Only present if at_risk or critical
    message: string;
    suggestion: string;
  };
  progress: {
    currentMilestone: Milestone;
    milestonesRemaining: number;
    percentComplete: number;     // 0-100
  };
  meta: {
    generatedAt: string;
    commitmentId: string;
  };
}
```

Decisions Made

1. **Risk Level Calculation** (Deterministic):
   - `critical` : Overdue OR <24 hours remaining and not working on solution yet
   - `at_risk` : <2 days remaining and haven't started working
   - `needs_attention` : <4 days remaining and still in early stages
   - `on_track` : Everything else
2. **Tone Selection**: LLM chooses based on situation:
   - `encouraging` : Normal progress
   - `motivating` : Slow progress, needs a push
   - `celebratory` : Good progress, milestones achieved
   - `urgent` : Critical deadline situations

- `supportive`: Struggling users who need empathy

3. **Next Action Specificity**: Provides concrete actions based on current milestone, not vague encouragement
4. **Warning Generation**: Only surfaces warnings when risk is elevated, avoiding alarm fatigue

Why No RAG

The Commitment Coach doesn't use RAG because:

- It focuses on time management and accountability, not technical details
- All necessary context (deadlines, milestones, progress) is passed directly
- Repo-specific guidance is already provided by the Issue Explainer

---

Issue Recommender Agent

Purpose: Analyze a list of candidate issues and recommend the best fit for a specific user based on their skills, interests, and experience.

Inputs

```
interface RecommendIssueInput {
  user: {
    id: string;
    username: string;
    skillLevel: "beginner" | "intermediate" | "advanced";
    preferredLanguages: string[];
    interests?: string[];              // e.g., ["testing", "docs", "frontend"]
    pastContributions?: number;
    availableHoursPerWeek?: number;
  };
  issues: Array<{                      // 1–20 candidate issues
    id: string;
    title: string;
    body?: string;
    labels: string[];
    repository: string;
    language?: string;                 // Primary repo language
    openedAt?: string;
    commentCount?: number;
    url: string;
  }>;
}
```

Outputs

```
interface RecommendIssueOutput {
  recommendedIssue: {
    id: string;
    title: string;
```

```
    repository: string;
    url: string;
  };
  explanation: string;                // Why this is the best fit
  riskLevel: "low" | "medium" | "high";
  riskFactors: string[];              // Specific concerns
  alternativeIssues: Array<{          // Up to 2 alternatives
    id: string;
    title: string;
    reason: string;
  }>;
  rankedIssues: Array<{               // All issues scored
    issueId: string;
    difficultyScore: number;          // 1-10
    fitScore: number;                 // 1-10
    reasoning: string;
  }>;
}
```

Decisions Made

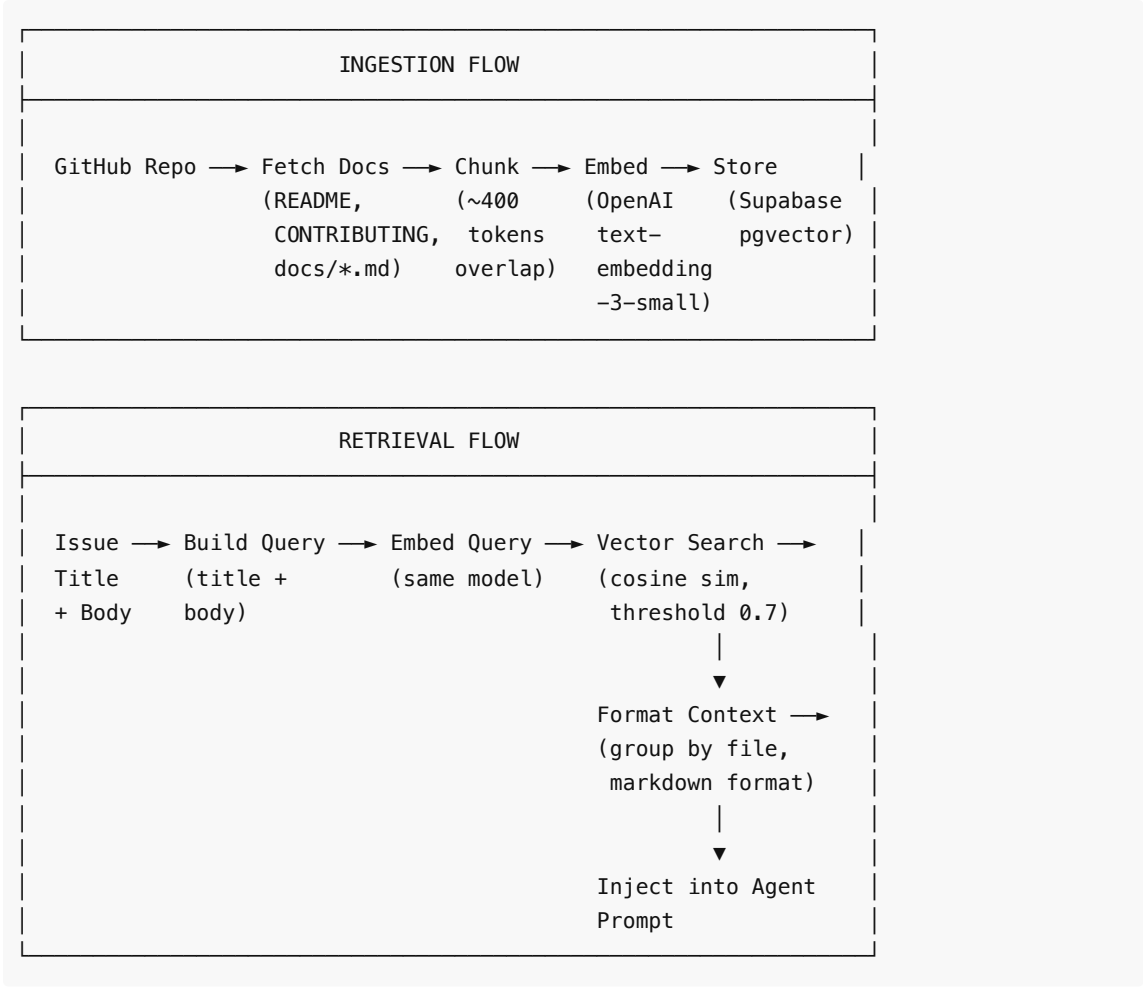
1. **Difficulty Assessment**: Scores each issue 1-10 based on:
   - Labels (good-first-issue = easier)
   - Description complexity
   - Comment volume (more comments may indicate complexity)
   - Repository language match
2. **User Fit Analysis**: Scores each issue 1-10 based on:
   - Skill level alignment
   - Language proficiency match
   - Interest alignment
   - Time commitment vs. availability
3. **Risk Evaluation**:
   - `low` : Clear scope, matches skills, maintainer responsive
   - `medium` : Some ambiguity, may require learning
   - `high` : Unclear requirements, skill gap, potential scope creep
4. **Ranking**: All issues ranked by overall suitability, not just difficulty
5. **Alternative Suggestions**: Provides up to 2 alternatives with reasoning

---

RAG System Architecture

The RAG (Retrieval-Augmented Generation) system provides repository-specific context to the Issue Explainer agent.

Pipeline Overview



Ingestion Details

Triggered: When a repository is added/tracked

Documents Fetched:

- `README.md`
- `CONTRIBUTING.md`
- `docs/*.md` files

Chunking Strategy:

- Target size: ~~400 tokens (~~1600 characters)
- Overlap: 50 tokens between chunks
- Split on paragraph boundaries for coherence
- Each chunk retains file path and index metadata

Embedding Model: `text-embedding-3-small` (1536 dimensions)

Storage: Supabase `repo_document_chunks` table with pgvector extension

Retrieval Details

Query Construction: Issue title + body (truncated to 8000 chars)

Similarity Search:

- Cosine similarity via `match_repo_documents` RPC
- Default threshold: 0.7 (configurable)
- Default match count: 5 chunks

Context Formatting:

- Chunks grouped by source file
- Sorted by original position within each file
- Formatted as markdown for prompt injection

Observability with Opik

All AI agent calls are traced using Opik for observability:

```
const trackedAI = createTrackedAI("issue-explainer-completion");

const completion = await trackedAI.chat.completions.create({
  model: "gpt-4o",
  messages: [...],
});

await flushTraces();
```

This enables:

- Cost tracking per agent type
- Latency monitoring
- Prompt/response logging for debugging
- Quality analysis over time

File Structure

```
lib/ai/
├── openai.ts                    # OpenAI client setup + Opik tracking
├── index.ts                     # Type exports
└── agents/
    ├── index.ts                 # Re-exports all agents
    ├── issue-explainer.ts       # Issue Explainer agent
    ├── commitment-coach.ts      # Commitment Coach agent
    └── recommend-issue.ts       # Issue Recommender agent

lib/rag/
├── ingest.ts                    # Document chunking + embedding pipeline
├── retrieve.ts                  # Vector search + context formatting
└── fetch-repo-docs.ts           # GitHub documentation fetching
```

Design Principles

1. **Fail Gracefully**: Every agent has fallback responses when LLM parsing fails
2. **Never Hallucinate Rules**: Agents only reference guidelines present in provided context
3. **Deterministic Where Possible**: Calculations (risk, progress, time) are done in code, not by the LLM
4. **Single Responsibility**: Each agent does one job well
5. **Structured Everything**: Zod schemas for inputs, outputs, and intermediate types
6. **Trace Everything**: All LLM calls are observable for debugging and improvement