# The Complete Architecture Guide to Self-Improving AI Systems

*System Design, Database Schema & Feedback Loops*

By Mark Kashef

*Self-Improving AI Systems Package*

# Table of Contents

# Introduction

This guide walks you through building AI systems that evaluate their own performance and improve over time—without human intervention. By the end, you'll understand the architecture, database design, and feedback loops that make this possible.

**Related guides:**

- *Rubric Template* - How to structure evaluation criteria
- *Handoff Template* - Managing context across sessions
- *Quick Start* - Build your first system in 30 minutes

---

# 1. The Paradigm Shift: Linear Apps vs Self-Improvement Loops

## The Old Mental Model (Linear)

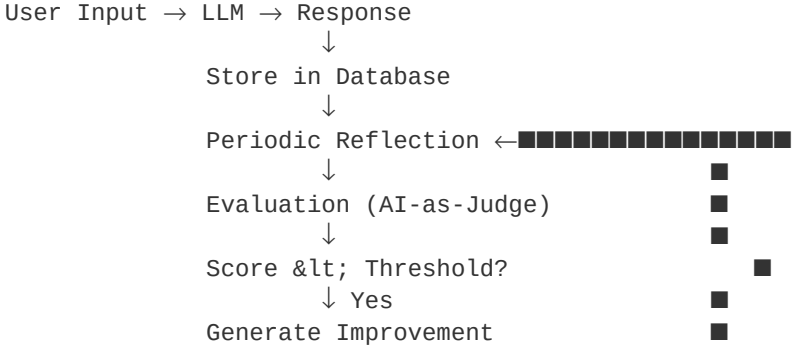
```
User Input → LLM → Response → Done
```

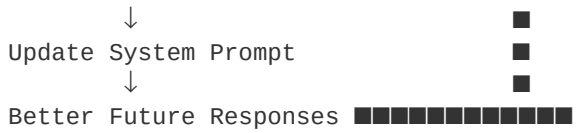
Traditional AI applications follow a request-response pattern. The system never learns from interactions. If responses are subpar, a human must manually edit prompts. This creates a bottleneck: the system is only as good as your last manual intervention.

**Problems with linear apps:**

- Quality degrades over time as edge cases accumulate
- Human bottleneck for improvements
- No institutional memory
- Same mistakes repeated indefinitely

## The New Mental Model (Self-Improving Loop)






**Key insight:** The AI evaluates its own outputs, identifies weaknesses, and rewrites its own instructions. The human sets the criteria; the system does the optimization.

### What Changes in Practice

| Aspect | Linear App | Self-Improving System |
|---|---|---|
| Prompt updates | Manual, reactive | Automatic, proactive |
| Quality trend | Stagnant or declining | Continuously improving |
| Edge case handling | Accumulates debt | Self-corrects |
| Developer role | Constant firefighting | Setting evaluation criteria |

---

## 2. The Two-Tool Stack: Claude Code + Supabase

### Why This Combination Works

**Claude Code** handles:

- Code generation and iteration
- Complex reasoning about architecture
- Multi-file edits in a single session
- Direct database manipulation via MCP

**Supabase** provides:

- Instant Postgres database
- Edge Functions for serverless logic
- Built-in auth (optional but useful)
- Real-time subscriptions
- MCP server for Claude Code integration

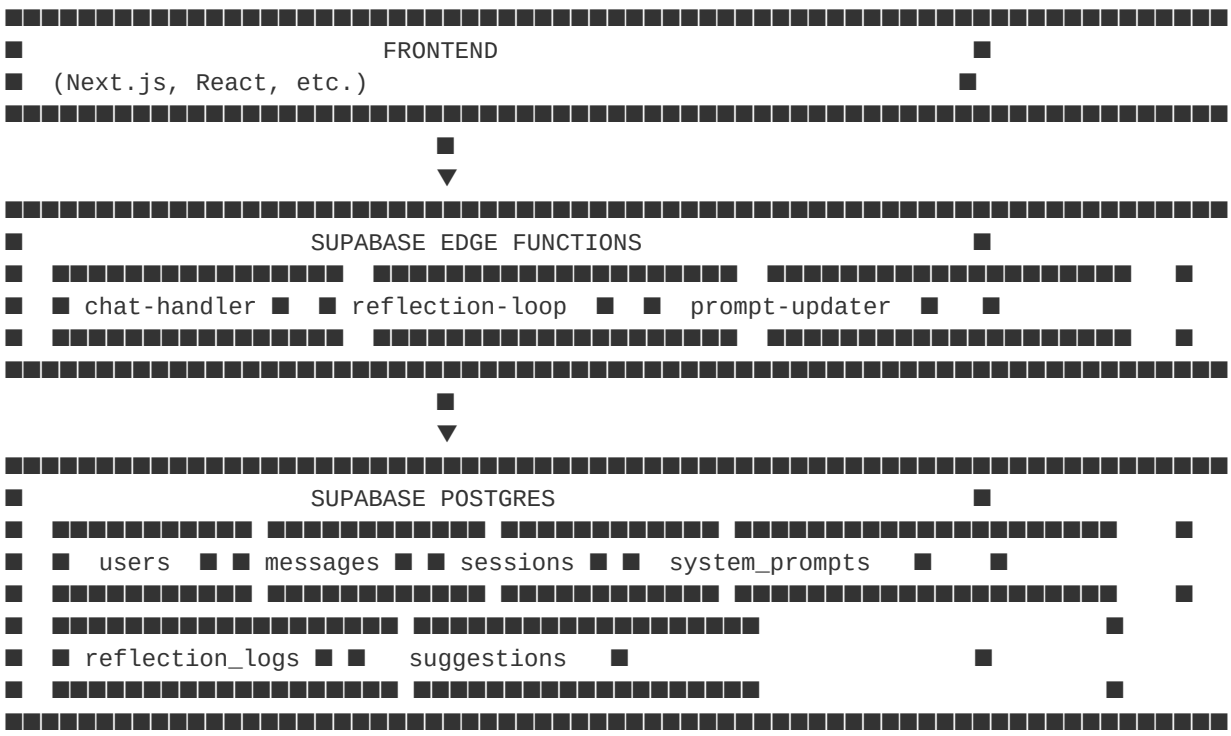
### The Magic: MCP Integration

The Model Context Protocol (MCP) lets Claude Code directly interact with your Supabase database. This means:

```
You: &amp;quot;Create a messages table with user_id, content, and timestamp&amp;quot;
Claude Code: *executes SQL directly via MCP*
```

No copy-pasting SQL, no switching contexts. Claude Code becomes a database-aware coding agent.

### Architecture Overview




---

## 3. Database Schema Design

### Users Table

```
CREATE TABLE users (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  email TEXT UNIQUE,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  metadata JSONB DEFAULT '{}'::jsonb
);

-- Index for email lookups
```

```
CREATE INDEX idx_users_email ON users(email);
```

**Purpose:** Track who's using the system. The `metadata` field stores preferences, usage tier, or any custom data.

## Sessions Table

```
CREATE TABLE sessions (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID REFERENCES users(id) ON DELETE CASCADE,
  title TEXT,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW(),
  is_active BOOLEAN DEFAULT true,
  metadata JSONB DEFAULT '{}'::jsonb
);

-- Index for user's sessions
CREATE INDEX idx_sessions_user_id ON sessions(user_id);
CREATE INDEX idx_sessions_updated_at ON sessions(updated_at DESC);
```

**Purpose:** Group messages into conversations. `is_active` lets you soft-delete sessions.

## Messages Table

```
CREATE TABLE messages (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  session_id UUID REFERENCES sessions(id) ON DELETE CASCADE,
  role TEXT NOT NULL CHECK (role IN ('user', 'assistant', 'system')),
  content TEXT NOT NULL,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  token_count INTEGER,
  model_used TEXT,
  prompt_version INTEGER,
  metadata JSONB DEFAULT '{}'::jsonb
);

-- Indexes for common queries
CREATE INDEX idx_messages_session_id ON messages(session_id);
CREATE INDEX idx_messages_created_at ON messages(created_at DESC);
CREATE INDEX idx_messages_prompt_version ON messages(prompt_version);
```

**Purpose:** Store conversation history. Critical fields:

- `prompt_version`: Links message to which system prompt generated it (for A/B analysis)
- `token_count`: Track costs
- `model_used`: Important when mixing models

## System Prompts Table (with Versioning)

```
CREATE TABLE system_prompts (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  version INTEGER NOT NULL,
  content TEXT NOT NULL,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  created_by TEXT DEFAULT 'system', -- 'human' or 'system'
  is_active BOOLEAN DEFAULT false,
  performance_score NUMERIC(3,2),
  change_reason TEXT,
  previous_version_id UUID REFERENCES system_prompts(id),
  metadata JSONB DEFAULT '{}'::jsonb
);

-- Only one active prompt at a time
CREATE UNIQUE INDEX idx_system_prompts_active ON system_prompts(is_active) WHERE is_active = true;
CREATE INDEX idx_system_prompts_version ON system_prompts(version DESC);
```

**Purpose:** Full history of every system prompt. Key design decisions:

- `is_active`: Only one prompt active at a time (enforced by unique index)
- `created_by`: Track human vs AI-generated prompts
- `previous_version_id`: Chain for rollback capability
- `performance_score`: Aggregate score from reflections using this prompt

## Reflection Logs Table

```
CREATE TABLE reflection_logs (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  created_at TIMESTAMPTZ DEFAULT NOW(),
  messages_analyzed INTEGER NOT NULL,
  time_window_start TIMESTAMPTZ NOT NULL,
  time_window_end TIMESTAMPTZ NOT NULL,
  prompt_version_evaluated INTEGER NOT NULL,

  -- Scores (1-5 scale)
  completeness_score INTEGER CHECK (completeness_score BETWEEN 1 AND 5),
  depth_score INTEGER CHECK (depth_score BETWEEN 1 AND 5),
  tone_score INTEGER CHECK (tone_score BETWEEN 1 AND 5),
  scope_score INTEGER CHECK (scope_score BETWEEN 1 AND 5),
  missed_opportunities_score INTEGER CHECK (missed_opportunities_score BETWEEN 1 AND 5),
  overall_score NUMERIC(3,2),

  -- Analysis
  strengths TEXT[],
  weaknesses TEXT[],
  patterns_noticed TEXT[],

  -- Outcome
```

```
  action_taken TEXT CHECK (action_taken IN ('none', 'suggestion', 'prompt_update')),
  suggestion_id UUID,
  new_prompt_version INTEGER,

  raw_analysis JSONB
);

CREATE INDEX idx_reflection_logs_created_at ON reflection_logs(created_at DESC);
CREATE INDEX idx_reflection_logs_overall_score ON reflection_logs(overall_score);
```

**Purpose:** Complete record of every self-evaluation. This table is gold for debugging:

- See exactly why the system decided to update (or not)
- Track score trends over time
- Identify which criteria trigger the most updates

## Suggestions Table

```
CREATE TABLE suggestions (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  created_at TIMESTAMPTZ DEFAULT NOW(),
  reflection_log_id UUID REFERENCES reflection_logs(id),

  suggestion_type TEXT CHECK (suggestion_type IN ('prompt_change', 'behavior_note', 'escalation')),
  priority TEXT CHECK (priority IN ('low', 'medium', 'high', 'critical')),

  title TEXT NOT NULL,
  description TEXT NOT NULL,
  proposed_change TEXT,

  status TEXT DEFAULT 'pending' CHECK (status IN ('pending', 'approved', 'rejected', 'implemented',
  reviewed_by TEXT,
  reviewed_at TIMESTAMPTZ,
  review_notes TEXT,

  implemented_in_version INTEGER
);

CREATE INDEX idx_suggestions_status ON suggestions(status);
CREATE INDEX idx_suggestions_priority ON suggestions(priority);
```

**Purpose:** Queue of potential improvements. Not all reflections trigger immediate updates—some generate suggestions for human review. This table enables:

- Human-in-the-loop for critical changes
- Batching minor improvements
- Tracking suggestion quality over time

# 4. The Evaluation Layer: AI-as-Judge

## The Core Concept

Instead of humans reviewing AI outputs, another AI instance (or the same model with a different prompt) evaluates quality. This creates scalable quality assurance.

## Setting Up the Evaluator

The evaluator receives:

- The original user message
- The assistant's response
- The system prompt that generated it
- Evaluation criteria (the rubric)

```
// Example evaluator prompt structure
const evaluatorPrompt = `
You are a ruthless quality evaluator. Your job is to score AI responses
on specific criteria. Be harsh—mediocre responses should fail.

## Evaluation Criteria
${rubricContent}

## System Prompt Being Evaluated
${systemPrompt}

## Conversation to Evaluate
User: ${userMessage}
Assistant: ${assistantResponse}

## Your Task
Score each criterion 1-5. Provide specific evidence for each score.
Identify patterns across multiple conversations if provided.
`;
```

## The Reflection Query

The reflection loop pulls recent conversations for batch evaluation:

```
-- Get last N assistant messages with context
SELECT
  m.id,
  m.content as assistant_response,
  m.prompt_version,
  prev.content as user_message,
  sp.content as system_prompt_used
FROM messages m
JOIN messages prev ON prev.session_id = m.session_id
```

```
  AND prev.created_at &lt; m.created_at
  AND prev.role = 'user'
JOIN system_prompts sp ON sp.version = m.prompt_version
WHERE m.role = 'assistant'
  AND m.created_at &gt; NOW() - INTERVAL '24 hours'
ORDER BY m.created_at DESC
LIMIT 20;
```

### Interpreting Scores

| Overall Score | Interpretation | Action |
|---|---|---|
| 4.5 - 5.0 | Excellent | None—system is performing well |
| 3.5 - 4.4 | Good | Log patterns, consider suggestions |
| 2.5 - 3.4 | Needs improvement | Generate suggestion for review |
| 1.0 - 2.4 | Poor | Auto-update prompt if enabled |

---

## 5. Edge Functions: What Each One Does

### Chat Handler (`/functions/chat-handler`)

**Responsibility:** Process user messages and generate responses.

```
// Simplified structure
import { serve } from &quot;https://deno.land/std@0.168.0/http/server.ts&quot;;
import Anthropic from &quot;npm:@anthropic-ai/sdk&quot;;

serve(async (req) =&gt; {
  const { message, session_id, user_id } = await req.json();

  // 1. Get active system prompt
  const { data: prompt } = await supabase
    .from('system_prompts')
    .select('content, version')
    .eq('is_active', true)
    .single();

  // 2. Get conversation history
```

```
  const { data: history } = await supabase
    .from('messages')
    .select('role, content')
    .eq('session_id', session_id)
    .order('created_at', { ascending: true });

  // 3. Call Claude
  const response = await anthropic.messages.create({
    model: &amp;quot;claude-sonnet-4-20250514&amp;quot;,
    max_tokens: 4096,
    system: prompt.content,
    messages: [...history, { role: &amp;quot;user&amp;quot;, content: message }]
  });

  // 4. Store both messages with prompt version
  await supabase.from('messages').insert([
    { session_id, role: 'user', content: message, prompt_version: prompt.version },
    { session_id, role: 'assistant', content: response.content[0].text, prompt_version: prompt.versio
  ]);

  return new Response(JSON.stringify({ response: response.content[0].text }));
});
```

**Key detail:** Every message is tagged with `prompt_version` so reflections can trace responses back to the prompt that generated them.

## Reflection Loop (`/functions/reflection-loop`)

**Responsibility:** Periodically evaluate recent conversations and decide whether to update the prompt.

```
serve(async (req) =&gt; {
  // 1. Check cooldown (prevent rapid-fire updates)
  const lastReflection = await getLastReflection();
  if (lastReflection &amp;amp;&amp;amp; hoursSince(lastReflection) &amp;lt; COOLDOWN_HOURS) {
    return new Response(JSON.stringify({ skipped: true, reason: 'cooldown' }));
  }

  // 2. Gather recent conversations
  const conversations = await getRecentConversations(TIME_WINDOW_HOURS);
  if (conversations.length &amp;lt; MIN_CONVERSATIONS) {
    return new Response(JSON.stringify({ skipped: true, reason: 'insufficient_data' }));
  }

  // 3. Get current system prompt and rubric
  const systemPrompt = await getActiveSystemPrompt();
  const rubric = await getRubric();

  // 4. Run evaluation
  const evaluation = await anthropic.messages.create({
    model: &amp;quot;claude-sonnet-4-20250514&amp;quot;,
    max_tokens: 4096,
    system: EVALUATOR_SYSTEM_PROMPT,
```

```
    messages: [{
      role: &amp;quot;user&amp;quot;,
      content: formatEvaluationRequest(conversations, systemPrompt, rubric)
    }]
  });

  // 5. Parse scores and decide action
  const scores = parseEvaluationResponse(evaluation);
  const action = decideAction(scores);

  // 6. Log reflection
  await logReflection(scores, action);

  // 7. Take action if needed
  if (action === 'prompt_update' &amp;amp;&amp;amp; scores.overall &amp;lt; AUTO_UPDATE_THRESHOLD) {
    await triggerPromptUpdate(scores);
  } else if (action === 'suggestion') {
    await createSuggestion(scores);
  }

  return new Response(JSON.stringify({ scores, action }));
});
```

## Prompt Updater (`/functions/prompt-updater`)

**Responsibility:** Generate and apply improved system prompts.

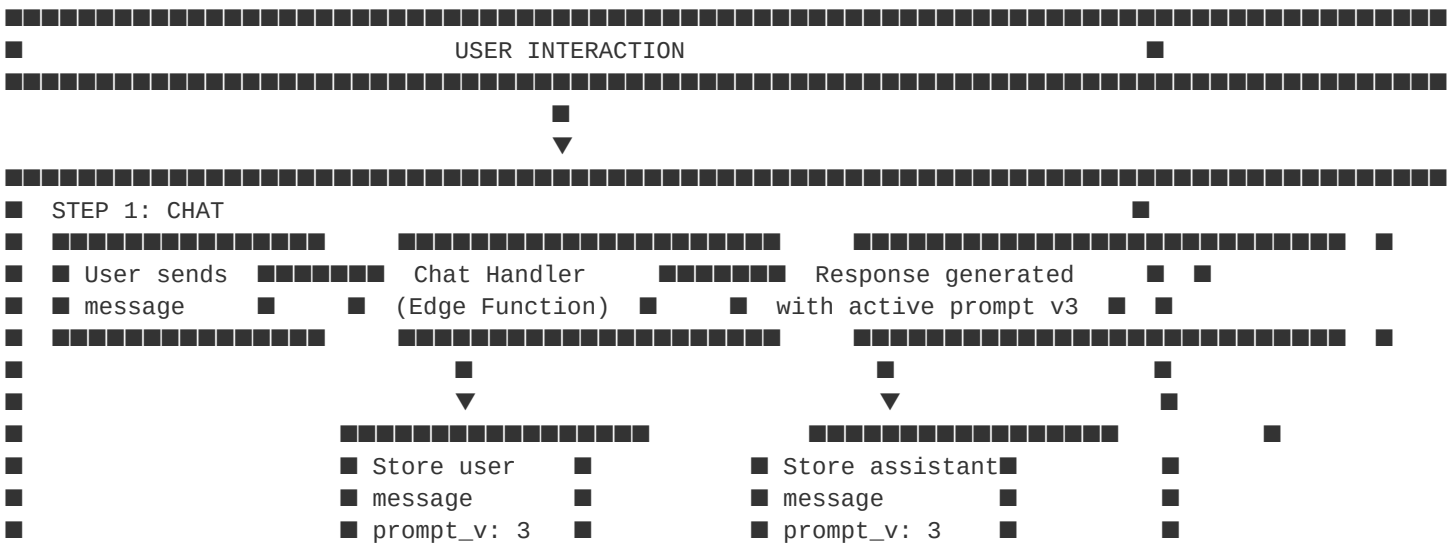
```
serve(async (req) =&amp;gt; {
  const { reflection_id, weaknesses, patterns } = await req.json();

  // 1. Get current prompt
  const currentPrompt = await getActiveSystemPrompt();

  // 2. Generate improved prompt
  const improvement = await anthropic.messages.create({
    model: &amp;quot;claude-sonnet-4-20250514&amp;quot;,
    max_tokens: 4096,
    system: PROMPT_IMPROVER_SYSTEM,
    messages: [{
      role: &amp;quot;user&amp;quot;,
      content: `
Current prompt:
${currentPrompt.content}

Identified weaknesses:
${weaknesses.join('\n')}

Patterns noticed:
${patterns.join('\n')}

Generate an improved version that addresses these issues while maintaining existing strengths.
      `
    }]
```

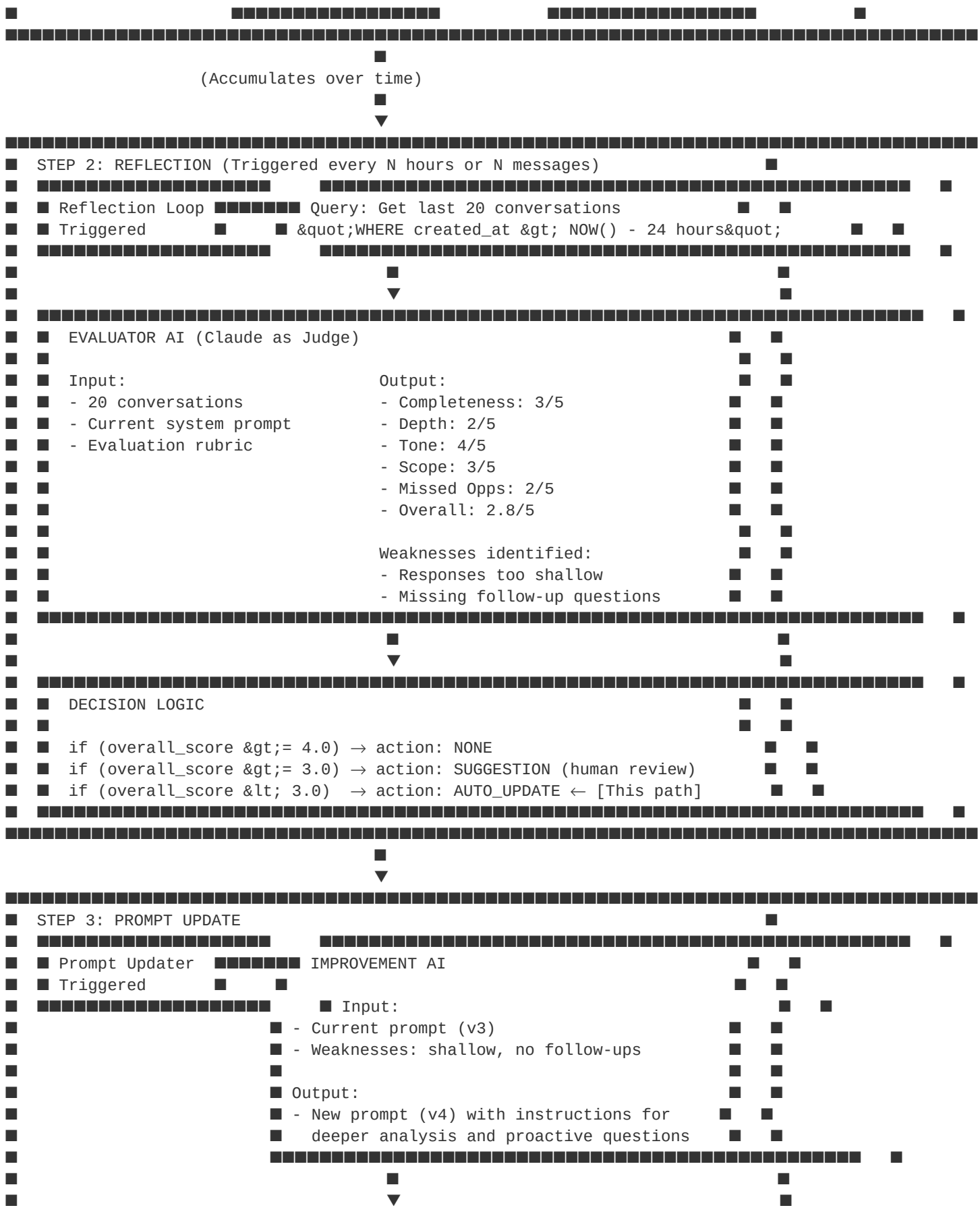
```
  });

  // 3. Deactivate current prompt
  await supabase
    .from('system_prompts')
    .update({ is_active: false })
    .eq('id', currentPrompt.id);

  // 4. Insert new prompt
  const { data: newPrompt } = await supabase
    .from('system_prompts')
    .insert({
      version: currentPrompt.version + 1,
      content: improvement.content[0].text,
      is_active: true,
      created_by: 'system',
      previous_version_id: currentPrompt.id,
      change_reason: `Reflection ${reflection_id}: ${weaknesses[0]}`
    })
    .select()
    .single();

  return new Response(JSON.stringify({
    new_version: newPrompt.version,
    change_reason: newPrompt.change_reason
  }));
});
```

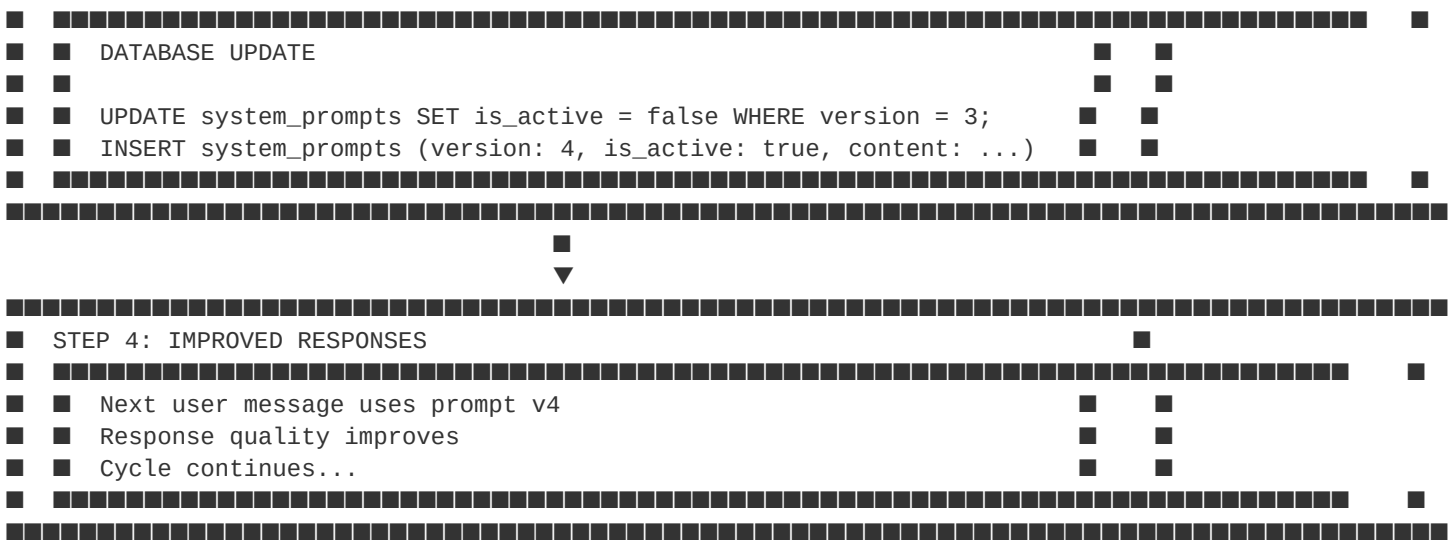
---

# 6. The Feedback Loop Flow Diagram

## Visual Representation




(Accumulates over time)
STEP 2: REFLECTION (Triggered every N hours or N messages)
Reflection Loop Triggered
Query: Get last 20 conversations
"WHERE created_at > NOW() - 24 hours"
EVALUATOR AI (Claude as Judge)
Input:
- 20 conversations
- Current system prompt
- Evaluation rubric
Output:
- Completeness: 3/5
- Depth: 2/5
- Tone: 4/5
- Scope: 3/5
- Missed Opps: 2/5
- Overall: 2.8/5
Weaknesses identified:
- Responses too shallow
- Missing follow-up questions
DECISION LOGIC
if (overall_score >= 4.0) → action: NONE
if (overall_score >= 3.0) → action: SUGGESTION (human review)
if (overall_score < 3.0) → action: AUTO_UPDATE ← [This path]
STEP 3: PROMPT UPDATE
Prompt Updater Triggered
IMPROVEMENT AI
Input:
- Current prompt (v3)
- Weaknesses: shallow, no follow-ups
Output:
- New prompt (v4) with instructions for
deeper analysis and proactive questions




### Timing and Triggers

The reflection can be triggered by:

- **Time-based:** Cron job every 6-24 hours
- **Volume-based:** After every N messages
- **Manual:** Admin endpoint for testing

Recommended starting point: Every 12 hours with minimum 10 conversations.

---

## 7. Safety Nets: Cooldown Periods, Thresholds, and Version Control

### Cooldown Period

**Problem:** Without limits, the system might update prompts too frequently, causing instability.

**Solution:** Enforce minimum time between updates.

```
const COOLDOWN_HOURS = 6;

async function canRunReflection(): Promise&lt;boolean&gt; {
  const { data: lastLog } = await supabase
    .from('reflection_logs')
    .select('created_at')
    .order('created_at', { ascending: false })
    .limit(1)
    .single();

  if (!lastLog) return true;
```

```
  const hoursSinceLastReflection =
    (Date.now() - new Date(lastLog.created_at).getTime()) / (1000 * 60 * 60);

  return hoursSinceLastReflection &gt;= COOLDOWN_HOURS;
}
```

## Update Thresholds

**Problem:** Minor fluctuations shouldn't trigger prompt changes.

**Solution:** Require significant score drops before updating.

```
const THRESHOLDS = {
  autoUpdate: 2.5,      // Below this: auto-update
  suggestion: 3.5,      // Below this: create suggestion
  acceptableMin: 4.0,   // Below this: log for monitoring
};

function decideAction(scores: EvaluationScores): Action {
  if (scores.overall &lt; THRESHOLDS.autoUpdate) {
    return 'prompt_update';
  }
  if (scores.overall &lt; THRESHOLDS.suggestion) {
    return 'suggestion';
  }
  if (scores.overall &lt; THRESHOLDS.acceptableMin) {
    return 'log_only';
  }
  return 'none';
}
```

## Version Control and Rollback

**Problem:** A bad prompt update could degrade all future responses.

**Solution:** Maintain full history and enable quick rollback.

```
// Rollback to previous version
async function rollbackPrompt(targetVersion: number): Promise&lt;void&gt; {
  // Deactivate current
  await supabase
    .from('system_prompts')
    .update({ is_active: false })
    .eq('is_active', true);

  // Activate target version
  await supabase
    .from('system_prompts')
    .update({ is_active: true })
    .eq('version', targetVersion);
```

```
  // Log the rollback
  await supabase
    .from('reflection_logs')
    .insert({
      action_taken: 'rollback',
      new_prompt_version: targetVersion,
      // ... other fields
    });
}
```

### Maximum Update Frequency

**Problem:** Rapid consecutive updates indicate instability.

**Solution:** Cap updates per time period.

```
const MAX_UPDATES_PER_DAY = 3;

async function hasReachedUpdateLimit(): Promise&lt;boolean&gt; {
  const { count } = await supabase
    .from('system_prompts')
    .select('*', { count: 'exact' })
    .eq('created_by', 'system')
    .gte('created_at', new Date(Date.now() - 24 * 60 * 60 * 1000).toISOString());

  return count &gt;= MAX_UPDATES_PER_DAY;
}
```

### Human Override

Always maintain the ability for humans to:

- Disable auto-updates temporarily
- Manually approve all updates
- Force a specific prompt version
- Review pending suggestions

```
-- Add admin controls
ALTER TABLE system_prompts ADD COLUMN locked BOOLEAN DEFAULT false;

-- Locked prompts can't be auto-deactivated
-- Reflection loop checks: WHERE is_active = true AND locked = false
```

---

## Next Steps

**Start building:** Follow the *Quick Start Guide*

**Set up evaluation:** Use the *Rubric Template*

**Manage sessions:** Implement the *Handoff Template*

---

*This guide is part of the Self-Improving AI Systems package.*