

Análisis de Patrones Adaptables: Clawdbot → CourseForge

Resumen Ejecutivo

Clawdbot ofrece **patrones de arquitectura valiosos** que pueden adaptarse a CourseForge para construir un agente de seguimiento de hábitos, aunque su modelo de ejecución (daemon continuo) es incompatible con el stack serverless actual. Este documento extrae los patrones clave y propone cómo implementarlos sobre Netlify + Supabase + Gemini.

| Aspecto | Patrón de Clawdbot | Adaptación para CourseForge |
|-----------------|--------------------------------------|-------------------------------------|
| Memoria | Archivos Markdown + SQLite vectorial | Supabase PostgreSQL + pgvector |
| Canales | Multi-canal con routing unificado | Webhooks + Edge Functions |
| Proactividad | Cron jobs internos + heartbeats | Supabase pg_cron + Edge Functions |
| Skills/Tools | Plugins TypeScript modulares | Módulos TS en Netlify Functions |
| Personalización | SOUL.md + AGENTS.md + workspace | Tablas de configuración por usuario |

1. Arquitectura de Memoria (Patrón Más Valioso)

Cómo lo hace Clawdbot

Clawdbot implementa un sistema de memoria en **dos capas**:

```
~/clawd/
├── MEMORY.md      # Memoria a largo plazo (curada, durable)
└── memory/
    ├── 2026-01-24.md # Log diario (append-only)
    ├── 2026-01-25.md
    └── ...
└── .clawdbot/
    └── memory/
        └── <agentId>.sqlite # Índice vectorial para búsqueda semántica
```

Principios clave:

- Los archivos Markdown son la **fuente de verdad** (no el modelo)
- El modelo solo "recuerda" lo que se escribe a disco

- Búsqueda híbrida: **BM25** (keywords exactos) + **embeddings** (semántica)
- Auto-flush antes de compactación de contexto
- Sincronización lazy con debounce de 1.5s

Adaptación para CourseForge

```
sql

-- Tabla de memoria a largo plazo (MEMORY.md equivalente)
CREATE TABLE user_memories (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES auth.users(id),
    category TEXT, -- 'preference', 'fact', 'goal', 'habit'
    content TEXT NOT NULL,
    embedding VECTOR(768), -- Para pgvector con Gemini embeddings
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);

-- Tabla de logs diarios (memory/YYYY-MM-DD.md equivalente)
CREATE TABLE daily_logs (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES auth.users(id),
    date DATE NOT NULL,
    entries JSONB DEFAULT '[]', -- Array de entradas append-only
    created_at TIMESTAMPTZ DEFAULT NOW()
);

-- Índice para búsqueda híbrida
CREATE INDEX idx_memories_embedding ON user_memories
USING ivfflat (embedding vector_cosine_ops) WITH (lists = 100);

CREATE INDEX idx_memories_content_fts ON user_memories
USING gin (to_tsvector('spanish', content));
```

Función de búsqueda híbrida:

```
typescript
```

```

// lib/memory-search.ts

import { createClient } from '@supabase/supabase-js';
import { GoogleGenerativeAI } from '@google/genai';

interface SearchResult {
  id: string;
  content: string;
  score: number;
  source: 'semantic' | 'keyword';
}

export async function hybridMemorySearch(
  userId: string,
  query: string,
  limit = 10
): Promise<SearchResult[]> {
  const supabase = createClient(/* ... */);
  const genai = new GoogleGenerativeAI(process.env.GOOGLE_GENERATIVE_AI_API_KEY!);

  // 1. Generar embedding de la query
  const embeddingModel = genai.getGenerativeModel({ model: 'embedding-001' });
  const embeddingResult = await embeddingModel.embedContent(query);
  const queryEmbedding = embeddingResult.embedding.values;

  // 2. Búsqueda semántica (vector)
  const { data: semanticResults } = await supabase.rpc('match_memories', {
    query_embedding: queryEmbedding,
    match_threshold: 0.7,
    match_count: limit,
    p_user_id: userId
  });

  // 3. Búsqueda por keywords (BM25-like con FTS)
  const { data: keywordResults } = await supabase
    .from('user_memories')
    .select('id, content')
    .eq('user_id', userId)
    .textSearch('content', query, { type: 'websearch', config: 'spanish' })
    .limit(limit);

  // 4. Fusionar y rankear resultados (RRF - Reciprocal Rank Fusion)
}

```

```
    return fuseResults(semanticResults, keywordResults);
}
```

2. Sistema de Canales Multi-Plataforma

Cómo lo hace Clawdbot

Clawdbot usa un **registry de canales** con adaptadores por plataforma:

```
typescript

// Estructura conceptual del channel registry
interface Channel {
  id: string;
  adapter: 'whatsapp' | 'telegram' | 'slack' | 'discord' | ...;
  config: ChannelConfig;

  // Métodos unificados
  send(to: string, message: Message): Promise<void>;
  onMessage(handler: MessageHandler): void;
}

// Routing unificado
interface DeliveryContext {
  channel: string;
  accountId?: string;
  peerId: string;
  sessionId: string;
}
```

Características clave:

- Allowlists por canal (`allowFrom`, `groups`)
- Pairing para DMs desconocidos (seguridad)
- Mention gating en grupos
- Reply-back automático al canal de origen

Adaptación para CourseForge

```
typescript
```

```
// types/channels.ts

export type ChannelType = 'whatsapp' | 'telegram' | 'web' | 'email';

export interface ChannelConfig {
    type: ChannelType;
    enabled: boolean;
    credentials: Record<string, string>;
    allowFrom?: string[]; // Whitelist de usuarios
}

export interface InboundMessage {
    channel: ChannelType;
    senderId: string;
    senderName?: string;
    content: string;
    mediaUrls?: string[];
    timestamp: Date;
    metadata?: Record<string, unknown>;
}

export interface OutboundMessage {
    channel: ChannelType;
    recipientId: string;
    content: string;
    buttons?: MessageButton[];
    mediaUrl?: string;
}
```

typescript

```

// lib/channels/registry.ts

import { TelegramAdapter } from './telegram';
import { WhatsAppAdapter } from './whatsapp';
import { WebAdapter } from './web';

export class ChannelRegistry {
  private adapters = new Map<ChannelType, ChannelAdapter>();

  register(type: ChannelType, adapter: ChannelAdapter) {
    this.adapters.set(type, adapter);
  }

  async send(message: OutboundMessage): Promise<void> {
    const adapter = this.adapters.get(message.channel);
    if (!adapter) throw new Error(`Channel ${message.channel} not registered`);
    await adapter.send(message);
  }
}

// Webhook handler unificado para Netlify Function
async handleInbound(
  channel: ChannelType,
  rawPayload: unknown
): Promise<InboundMessage> {
  const adapter = this.adapters.get(channel);
  return adapter.parseInbound(rawPayload);
}

```

Netlify Function para webhooks:

typescript

```

// netlify/functions/webhook-telegram.ts
import { Handler } from '@netlify/functions';
import { ChannelRegistry } from '../../lib/channels/registry';
import { processMessage } from '../../lib/agent/processor';

export const handler: Handler = async (event) => {
  const registry = new ChannelRegistry();

  // Parsear mensaje entrante
  const message = await registry.handleInbound('telegram', JSON.parse(event.body!));

  // Verificar allowlist
  const isAllowed = await verifyAllowlist(message.senderId, 'telegram');
  if (!isAllowed) {
    return { statusCode: 403, body: 'Unauthorized' };
  }

  // Procesar con el agente (async, no bloqueante)
  await processMessage(message);

  return { statusCode: 200, body: 'OK' };
};

```

3. Sistema de Proactividad (Cron + Heartbeats)

Cómo lo hace Clawdbot

Clawdbot tiene dos mecanismos de proactividad:

1. Cron Jobs - Tareas programadas

- Expresiones cron estándar con timezone
- Jobs one-shot (ISO timestamp + `deleteAfterRun`)
- Jobs recurrentes
- Delivery automático a canales

2. Heartbeats - Pulsos periódicos

- El agente puede "despertar" sin input del usuario
- Se combina con cron para triggers

```
// Ejemplo de job de Clawdbot
{
  name: "Morning briefing",
  cron: "0 7 * * *",    // Todos los días a las 7am
  tz: "America/Mexico_City",
  session: "isolated",   // Sesión separada
  message: "Dame un resumen del progreso del usuario en sus hábitos",
  deliver: true,
  channel: "telegram",
  to: "@user123"
}
```

Adaptación para CourseForge

Opción 1: Supabase pg_cron (Plan Pro)

```
sql
```

```
-- Habilitar extensión
CREATE EXTENSION IF NOT EXISTS pg_cron;

-- Job diario de recordatorios de hábitos
SELECT cron.schedule(
    'habit-reminders-morning',
    '0 7 * * *', -- 7am UTC (ajustar para timezone)
    $$

    SELECT net.http_post(
        url := 'https://tu-app.netlify.app/.netlify/functions/habit-reminder-trigger',
        headers := '{"Authorization": "Bearer ' || current_setting('app.cron_secret') || '"}':jsonb,
        body := '{"type": "morning_reminder"}':jsonb
    )
    $$
);

-- Job de seguimiento nocturno
SELECT cron.schedule(
    'habit-progress-check',
    '0 21 * * *', -- 9pm UTC
    $$

    SELECT net.http_post(
        url := 'https://tu-app.netlify.app/.netlify/functions/habit-progress-trigger',
        headers := '{"Authorization": "Bearer ' || current_setting('app.cron_secret') || '"}':jsonb,
        body := '{"type": "evening_check"}':jsonb
    )
    $$
);
```

Opción 2: Netlify Scheduled Functions (más simple)

typescript

```
// netlify/functions/habit-reminder.ts
import { schedule } from '@netlify/functions';

export const handler = schedule('0 7 * * *', async (event) => {
    // Obtener usuarios con hábitos activos
    const { data: users } = await supabase
        .from('user_habits')
        .select('user_id, habits, channel_preferences')
        .eq('reminder_enabled', true)
        .eq('reminder_time', '07:00');

    // Generar y enviar recordatorios personalizados
    for (const user of users) {
        const reminderMessage = await generateHabitReminder(user);
        await sendToPreferredChannel(user, reminderMessage);
    }

    return { statusCode: 200 };
});
```

```
async function generateHabitReminder(user: UserHabits): Promise<string> {
    const model = genai.getGenerativeModel({ model: 'gemini-2.0-flash' });

    // Cargar contexto del usuario desde memoria
    const userMemories = await loadUserMemories(user.user_id, 5);
    const recentProgress = await getRecentHabitProgress(user.user_id, 7);
```

```
    const prompt = `

Eres un coach de hábitos amigable. Genera un recordatorio matutino personalizado.
```

Contexto del usuario:

```
 ${userMemories.map(m => `- ${m.content}`).join('\n')}
```

Hábitos activos:

```
 ${user.habits.map(h => `${h.name}: ${h.frequency}`).join('\n')}
```

Progreso últimos 7 días:

```
 ${JSON.stringify(recentProgress)}
```

Genera un mensaje corto, motivador y personalizado para WhatsApp/Telegram.

Máximo 280 caracteres.

```
`;
```

```
const result = await model.generateContent(prompt);
return result.response.text();
}
```

4. Sistema de Skills/Herramientas

Cómo lo hace Clawbot

Clawbot usa un sistema de **skills** basado en archivos Markdown:

```
~/clawd/skills/
├── habit-tracker/
│   ├── SKILL.md      # Descripción y reglas del skill
│   └── tools.ts      # Implementación de herramientas
└── meditation/
    └── SKILL.md
    ...
...
```

Estructura de SKILL.md:

markdown

```
---
```

name: habit-tracker
description: Track and manage user habits
triggers:

- "track habit"
- "habit progress"
- "complete habit"

tools:

- habit_log
- habit_status
- habit_streak

```
---
```

Habit Tracker Skill

This skill helps users track their daily habits...

Available Tools

habit_log

Log a completed habit for today.

habit_status

Check the current status of all habits.

habit_streak

Get streak information for a specific habit.

Adaptación para CourseForge

typescript

```
// lib/skills/types.ts
export interface Skill {
    id: string;
    name: string;
    description: string;
    triggers: string[];
    tools: Tool[];
    enabled: boolean;
}

export interface Tool {
    name: string;
    description: string;
    parameters: ToolParameter[];
    execute: (params: Record<string, unknown>, context: AgentContext) => Promise<ToolResult>;
}

export interface AgentContext {
    userId: string;
    sessionId: string;
    channel: ChannelType;
    memories: Memory[];
}
```

typescript

```
// lib/skills/habit-tracker/index.ts
import { Skill, Tool } from '../types';

export const habitTrackerSkill: Skill = {
  id: 'habit-tracker',
  name: 'Habit Tracker',
  description: 'Seguimiento y gestión de hábitos basados en talleres de CourseForge',
  triggers: [
    'registrar hábito',
    'progreso de hábitos',
    'racha de',
    'completé',
    'hice mi'
  ],
  tools: [
    logHabitTool,
    getHabitStatusTool,
    getHabitStreakTool,
    suggestNextActionTool
  ],
  enabled: true
};

const logHabitTool: Tool = {
  name: 'log_habit',
  description: 'Registra que el usuario completó un hábito',
  parameters: [
    { name: 'habit_id', type: 'string', required: true },
    { name: 'notes', type: 'string', required: false },
    { name: 'quality', type: 'number', required: false, min: 1, max: 5 }
  ],
  execute: async (params, context) => {
    const supabase = createClient(/* ... */);

    const { data, error } = await supabase
      .from('habit_logs')
      .insert({
        user_id: context.userId,
        habit_id: params.habit_id,
        completed_at: new Date().toISOString(),
        notes: params.notes,
        quality_rating: params.quality
      });
  }
};
```

```

if (error) throw error;

// Actualizar memoria del usuario
await updateUserMemory(context.userId, {
  category: 'habit',
  content: `Completó hábito ${params.habit_id} el ${new Date().toLocaleDateString()}`,
});

// Calcular racha actualizada
const streak = await calculateStreak(context.userId, params.habit_id as string);

return {
  success: true,
  message: `¡Hábito registrado! Llevas ${streak} días seguidos.`,
  data: { streak }
};
}
};


```

5. Personalización por Usuario (SOUL.md Pattern)

Cómo lo hace Clawbot

Clawbot usa archivos de configuración por workspace:

- **SOUL.md** - Personalidad y tono del agente
- **AGENTS.md** - Reglas de comportamiento
- **IDENTITY.md** - Información del usuario
- **USER.md** - Preferencias

Adaptación para CourseForge

sql

```

-- Configuración de personalización por usuario
CREATE TABLE user_agent_config (
    user_id UUID PRIMARY KEY REFERENCES auth.users(id),
    -- Equivalente a SOUL.md
    personality_traits JSONB DEFAULT '{
        "tone": "friendly",
        "formality": "informal",
        "encouragement_style": "supportive"
    }',
    -- Equivalente a IDENTITY.md
    user_context JSONB DEFAULT '{}',
    -- Ejemplo: { "name": "Juan", "goals": ["meditar diario"], "timezone": "America/Mexico_City" }

    -- Equivalente a USER.md (preferencias)
    preferences JSONB DEFAULT '{
        "reminder_frequency": "daily",
        "preferred_channel": "telegram",
        "language": "es",
        "notification_hours": {"start": "07:00", "end": "22:00"}
    }',
    -- Hábitos derivados de talleres
    active_habits JSONB DEFAULT '[]',
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);

-- Trigger para actualizar updated_at
CREATE TRIGGER update_user_agent_config_timestamp
BEFORE UPDATE ON user_agent_config
FOR EACH ROW
EXECUTE FUNCTION update_modified_column();

```

System prompt dinámico:

typescript

```
// lib/agent/system-prompt.ts
export async function buildSystemPrompt(userId: string): Promise<string> {
  const config = await getUserAgentConfig(userId);
  const memories = await getRecentMemories(userId, 5);
  const habits = await getActiveHabits(userId);

  return `

# Tu Rol
Eres un asistente de seguimiento de hábitos para CourseForge. Tu objetivo es ayudar al usuario a desarrollar y mantener los hábitos que establece en su perfil.

# Personalidad
- Tono: ${config.personality_traits.tone}
- Formalidad: ${config.personality_traits.formality}
- Estilo de ánimo: ${config.personality_traits.encouragement_style}

# Contexto del Usuario
Nombre: ${config.user_context.name || 'Usuario'}
Zona horaria: ${config.user_context.timezone || 'UTC'}
Objetivos: ${config.user_context.goals?.join(',') || 'No definidos'}

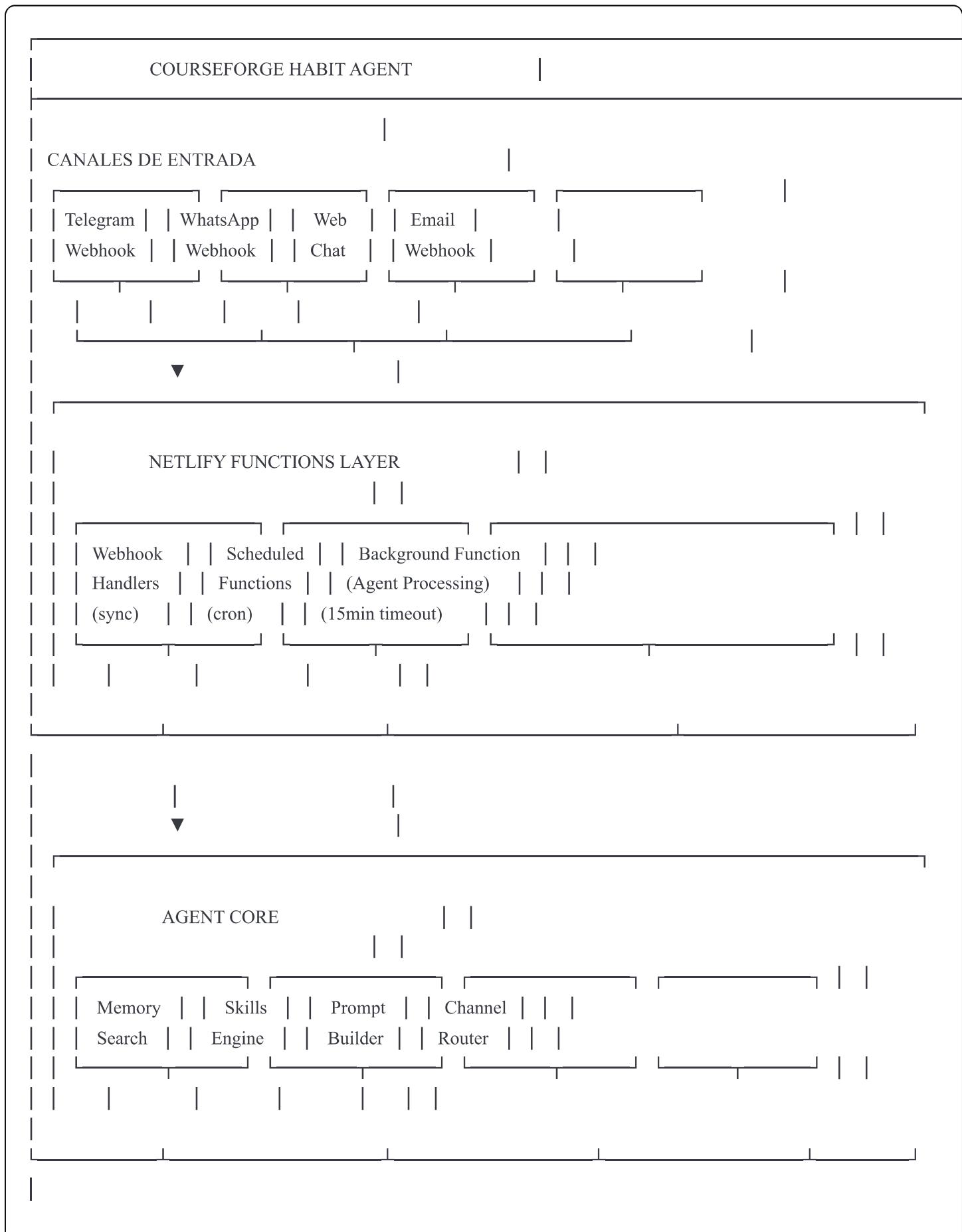
# Hábitos Activos
${habits.map(h => `- ${h.name} (${h.frequency}): ${h.description}`).join('\n')}

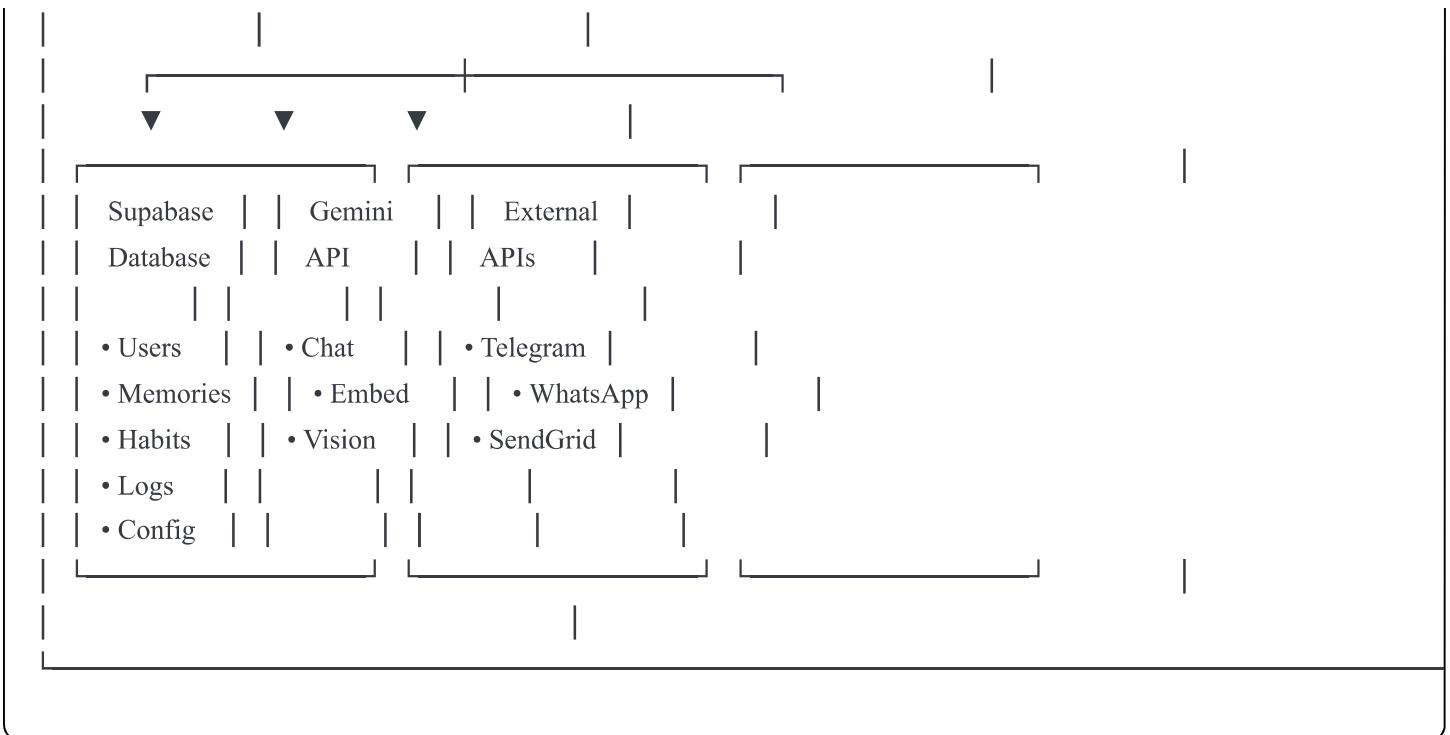
# Memorias Recientes
${memories.map(m => `- ${m.content}`).join('\n')}

# Instrucciones
1. Siempre saluda al usuario por su nombre si lo conoces
2. Celebra sus logros, por pequeños que sean
3. Si no ha registrado actividad en X días, pregunta amablemente cómo va
4. Conecta los hábitos con el contenido de los talleres que tomó
5. Mantén respuestas concisas (máx 200 palabras en mensajería)
6. Si el usuario expresa frustración, ofrece apoyo antes que soluciones

# Herramientas Disponibles
- log_habit: Registrar completación de hábito
- get_habit_status: Ver estado actual de hábitos
- get_habit_streak: Ver racha de un hábito
- suggest_next_action: Sugerir siguiente acción basada en contexto
- save_to_memory: Guardar información importante sobre el usuario
`;
```

6. Arquitectura Propuesta Completa





7. Estimación de Esfuerzo

| Componente | Complejidad | Estimación | Prioridad |
|---------------------------|-------------|------------|-----------|
| Sistema de Memoria | Media | 3-4 días | P0 |
| - Schema PostgreSQL | Baja | 0.5 día | - |
| - Búsqueda híbrida | Media | 1.5 días | - |
| - Integración embeddings | Media | 1 día | - |
| Canal Telegram | Baja | 2 días | P0 |
| - Webhook handler | Baja | 0.5 día | - |
| - Bot setup | Baja | 0.5 día | - |
| - Respuestas | Baja | 1 día | - |
| Skills de Hábitos | Media | 3 días | P1 |
| - Tool definitions | Baja | 1 día | - |
| - Lógica de streaks | Media | 1 día | - |
| - Integración talleres | Media | 1 día | - |

| Componente | Complejidad | Estimación | Prioridad |
|--------------------------------|-------------|------------|-----------|
| Sistema de Proactividad | Media | 2-3 días | P1 |
| - Scheduled functions | Baja | 1 día | - |
| - Lógica de nudges | Media | 1-2 días | - |
| Canal WhatsApp | Media | 3-4 días | P2 |
| - Meta Business setup | Media | 1-2 días | - |
| - Webhook + templates | Media | 2 días | - |
| Personalización | Baja | 2 días | P2 |
| - Config schema | Baja | 0.5 día | - |
| - Prompt dinámico | Media | 1.5 días | - |

Total MVP (Telegram + Memoria + Skills básicos): ~8-10 días Total Completo (con WhatsApp y proactividad avanzada): ~15-18 días

8. Próximos Pasos Recomendados

Fase 1: MVP (Semana 1-2)

1. **Día 1-2:** Schema de base de datos (memorias, hábitos, logs, config)
2. **Día 3-4:** Bot de Telegram con webhook básico
3. **Día 5-6:** Sistema de memoria con búsqueda básica
4. **Día 7-8:** Skills de seguimiento de hábitos (log, status, streak)
5. **Día 9-10:** Integración con contenido de talleres existente

Fase 2: Proactividad (Semana 3)

1. Scheduled functions para recordatorios
2. Lógica de nudges inteligentes
3. Personalización de mensajes según contexto

Fase 3: Expansión (Semana 4+)

1. Canal WhatsApp (requiere Meta Business verification)

2. Búsqueda semántica con pgvector
 3. Dashboard de progreso para usuarios
 4. Reportes automáticos semanales
-

9. Riesgos y Mitigaciones

| Riesgo | Impacto | Probabilidad | Mitigación |
|------------------------------|---------|--------------|--|
| Rate limits de Gemini | Alto | Media | Usar flash para búsquedas, pro para generación; cache agresivo |
| Timeout de Netlify Functions | Medio | Baja | Procesar async, responder rápido al webhook |
| Complejidad de WhatsApp API | Medio | Media | Empezar con Telegram; WhatsApp como fase 2 |
| Costos de embeddings | Bajo | Media | Batch processing, cache de embeddings comunes |
| Seguridad de webhooks | Alto | Baja | Validar signatures, allowlists estrictas |

Conclusión

Los patrones de clawdbot son **altamente adaptables** a CourseForge con las siguientes modificaciones:

1. **Memoria:** De archivos Markdown a PostgreSQL + pgvector (mejor para serverless)
2. **Canales:** De adapters en runtime a webhooks + functions (compatible con Netlify)
3. **Proactividad:** De cron interno a pg_cron/scheduled functions (sin daemon)
4. **Skills:** De plugins dinámicos a módulos estáticos (más predecible)

El resultado será un agente que mantiene la filosofía de clawdbot (proactivo, personalizado, multi-canal) pero adaptado a la arquitectura serverless de CourseForge.