Mission Impossible self-destructing invoice system

A Cloudflare Worker with static assets, D1 database, and cron-triggered email expiry creates a delightful, brand-matched invoice experience that nudges prospects to act within 7 days. The system works by generating a static HTML invoice page at build time (styled to match the client's own brand via LLM-powered CSS generation), deploying it to Cloudflare's edge network, running a daily cron job to detect expired invoices and fire off emails, and presenting a spectacular "Thanos snap" disintegration animation when the countdown hits zero — followed by a one-click Calendly booking embed. The entire pipeline is triggered by a single API call (from a CRM, Zapier, or Make.com), which kicks off a GitHub Actions workflow that builds and deploys a unique invoice page in under 60 seconds.

Architecture overview and core design decisions

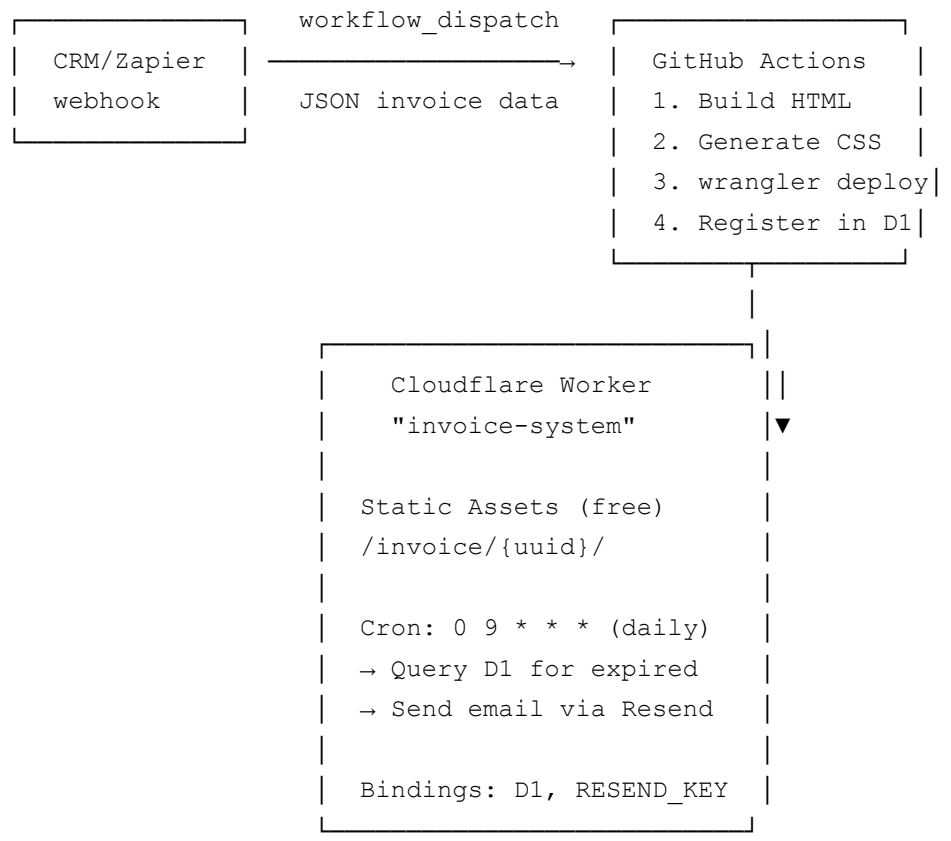
The system runs on a **single Cloudflare Worker project** that serves dual duty: delivering pre-generated static HTML invoice pages at the edge (free, globally cached) and running server-side logic for expiry checks and email sending. This is the modern Cloudflare-recommended approach Astro — Pages Functions are legacy and critically **do not support cron triggers**, which this system requires.

The architecture has three layers:

Static layer — Pre-built HTML/CSS/JS files in a `public/` directory, served automatically by Cloudflare's asset pipeline with zero Worker invocations. Each invoice lives at `/invoice/{uuid}/index.html` and contains a hardcoded UTC expiry timestamp, the countdown timer, self-destruct animation code, and Calendly embed configuration. These pages load in under **50ms globally** thanks to edge caching. Cloudflare

Worker layer — A TypeScript Worker handles two responsibilities: (1) an `/api/register` endpoint called during deployment to register new invoice metadata in D1, and (2) a `scheduled()` handler triggered by a daily cron that queries D1 for expired invoices, sends notification emails via Resend, and marks them as sent.

Data layer — **Cloudflare D1** (SQLite at the edge) stores invoice metadata: ID, client email, expiry timestamp, email-sent status, and page URL. D1 was chosen over KV because the cron job needs to query by expiry timestamp (`WHERE expiry_timestamp < ? AND email_sent = 0`), which KV cannot do without maintaining manual secondary indexes. D1 provides strong consistency for tracking send status, preventing duplicate emails.



Complete file and folder structure

```
invoice-system/
├── .github/
│   └── workflows/
│       └── generate-invoice.yml    # GitHub Actions: build + deploy
│
├── src/
│   └── index.ts                    # Cloudflare Worker (cron + API)
│
├── templates/
│   ├── invoice.ejs                 # Main invoice page template
│   └── expired.ejs                 # Expired state (embedded in invoice.ejs)
│
├── static/
│   ├── countdown.js                # Countdown timer module
│   ├── self-destruct.js            # Thanos snap disintegration effect
│   └── base.css                    # Base layout styles (non-brand)
│
├── scripts/
│   ├── build.js                    # Main build script (Node.js)
```

```
│   ├── extract-design-tokens.js    # Playwright → design token extraction
│   ├── generate-css.js             # LLM-powered CSS generation
│   └── register-invoice.js         # POST to /api/register after deploy
│
├── public/                         # Build output (gitignored)
│   ├── invoice/
│   │   ├── {uuid-1}/index.html
│   │   └── {uuid-2}/index.html
│   └── robots.txt
│
├── schema.sql                      # D1 database schema
├── wrangler.toml                   # Cloudflare Worker config
├── package.json
└── .gitignore
```

The `public/` directory is the **deployment artifact** — it's generated fresh by the build script and contains only static files. The `src/index.ts` Worker script handles server-side logic. Both are deployed together via `wrangler deploy`. Cloudflare

---

The Cloudflare Worker: cron triggers and email

The Worker configuration in `wrangler.toml` ties everything together:

```
name = "invoice-system"
main = "src/index.ts"
compatibility_date = "2025-10-19"

[assets]
directory = "./public"
binding = "ASSETS"
html_handling = "drop-trailing-slash"
not_found_handling = "404-page"
run_worker_first = ["/api/*"]

[[d1_databases]]
binding = "DB"
database_name = "invoice-db"
database_id = "your-database-id"

[triggers]
crons = ["0 9 * * *"]   # Daily at 9 AM UTC

[observability]
enabled = true
```

The `run_worker_first = ["/api/*"]` directive ensures that only API routes invoke the Worker — all static invoice pages are served directly from Cloudflare's CDN with zero compute cost. Cloudflare The cron trigger fires the `scheduled()` handler daily at 9 AM UTC. Cloudflare

The Worker itself is straightforward. It queries D1 for invoices whose expiry has passed but whose notification email hasn't been sent, sends each email via Resend's API, then marks them as sent:

```
import { Resend } from "resend";

interface Env {
  DB: D1Database;
  RESEND_API_KEY: string;
  ASSETS: Fetcher;
}

export default {
  async fetch(request: Request, env: Env): Promise<Response> {
    const url = new URL(request.url);
    if (url.pathname === "/api/register" && request.method === "POST") {
      const invoice = await request.json();
      await env.DB.prepare(
        `INSERT INTO invoices (id, client_name, client_email, amount,
         currency, expiry_timestamp, page_url, calendly_link)
         VALUES (?, ?, ?, ?, ?, ?, ?, ?)`
      ).bind(
        invoice.id, invoice.clientName, invoice.clientEmail,
        invoice.amount, invoice.currency, invoice.expiryTimestamp,
        invoice.pageUrl, invoice.calendlyLink
      ).run();
      return new Response(JSON.stringify({ success: true }), { status: 201 });
    }
    return new Response("Not Found", { status: 404 });
  },

  async scheduled(controller: ScheduledController, env: Env, ctx: ExecutionContex
    ctx.waitUntil(processExpiredInvoices(env));
  },
};

async function processExpiredInvoices(env: Env) {
  const resend = new Resend(env.RESEND_API_KEY);
  const now = Date.now();
```

```
  const { results } = await env.DB.prepare(
    "SELECT * FROM invoices WHERE expiry_timestamp < ? AND email_sent = 0"
  ).bind(now).all();

  for (const inv of results) {
    const { error } = await resend.emails.send({
      from: "invoices@yourdomain.com",
      to: inv.client_email as string,
      subject: `Your invoice has self-destructed 💥`,
      html: `<h1>Invoice Expired</h1>
        <p>Hi ${inv.client_name},</p>
        <p>The proposal we sent has passed its expiry window.
           Pricing and availability may have changed.</p>
        <p><a href="${inv.calendly_link}">Book a quick call</a>
           to get an updated proposal.</p>`,
    });
    if (!error) {
      await env.DB.prepare(
        "UPDATE invoices SET email_sent = 1, email_sent_at = ? WHERE id = ?"
      ).bind(new Date().toISOString(), inv.id).run();
    }
  }
}
```

Why Resend over alternatives? The free MailChannels integration with Cloudflare **ended August 2024**. Resend is now the officially recommended provider in Cloudflare's documentation, offering **3,000 free emails/month** (more than enough for invoices). Cloudflare announced its own native email service in September 2025, but it remains in private beta cloudflare — Resend is the production-ready choice today.

The D1 schema is minimal:

```
CREATE TABLE invoices (
  id TEXT PRIMARY KEY,
  client_name TEXT NOT NULL,
  client_email TEXT NOT NULL,
  amount REAL NOT NULL,
  currency TEXT DEFAULT 'USD',
  expiry_timestamp INTEGER NOT NULL,
  email_sent INTEGER DEFAULT 0,
  email_sent_at TEXT,
  page_url TEXT,
  calendly_link TEXT,
  created_at TEXT DEFAULT (datetime('now'))
);
CREATE INDEX idx_expiry ON invoices(expiry_timestamp, email_sent);
```

Build-time invoice generation pipeline

The build script takes invoice data as input, extracts the client's brand design tokens, generates matching CSS via an LLM, renders an HTML page from an EJS template, and outputs everything to `public/`. **EJS** was chosen over Handlebars or Nunjucks because it has the lowest friction for single-page generation — familiar `<%= %>` syntax, CBT Nuggets works standalone without Express, and supports async rendering LogRocket for the LLM call.

The core build script orchestrates the pipeline:

```
// scripts/build.js
const ejs = require('ejs');
const fs = require('fs');
const path = require('path');
const { extractDesignTokens } = require('./extract-design-tokens');
const { generateBrandCSS } = require('./generate-css');
const { v4: uuidv4 } = require('uuid');

async function build() {
  // Load invoice data from file (written by GitHub Actions decode step)
  const raw = fs.readFileSync(process.env.INVOICE_DATA_FILE, 'utf8');
  const invoice = JSON.parse(raw);

  // Generate UUID if not provided
  invoice.id = invoice.id || uuidv4();

  // Calculate expiry: 7 days from now
  const expiryMs = Date.now() + (7 * 24 * 60 * 60 * 1000);
  invoice.expiryTimestamp = invoice.expiryTimestamp || expiryMs;

  // Step 1: Extract client's design tokens
  let brandCSS;
  try {
    const tokens = await extractDesignTokens(invoice.clientWebsite);
    brandCSS = await generateBrandCSS(tokens);
  } catch (err) {
    console.warn('Brand extraction failed, using fallback:', err.message);
    brandCSS = generateFallbackCSS(invoice.brandColor || '#2563eb');
  }

  // Step 2: Render invoice HTML
```

```
  const html = await ejs.renderFile(
    path.join(__dirname, '../templates/invoice.ejs'),
    { ...invoice, brandCSS, expiryTimestamp: invoice.expiryTimestamp },
    { async: true }
  );

  // Step 3: Write to public/invoice/{id}/
  const outDir = path.join(__dirname, '../public/invoice', invoice.id);
  fs.mkdirSync(outDir, { recursive: true });
  fs.writeFileSync(path.join(outDir, 'index.html'), html);

  // Step 4: Copy static assets
  const staticDir = path.join(__dirname, '../static');
  for (const file of fs.readdirSync(staticDir)) {
    fs.copyFileSync(path.join(staticDir, file), path.join(outDir, file));
  }

  // Step 5: Write robots.txt
  fs.mkdirSync(path.join(__dirname, '../public'), { recursive: true });
  fs.writeFileSync(
    path.join(__dirname, '../public/robots.txt'),
    'User-agent: *\nDisallow: /invoice/\n'
  );

  console.log(`✅ Invoice ${invoice.id} built → /invoice/${invoice.id}/`);
  // Output for GitHub Actions to use in subsequent steps
  fs.writeFileSync('/tmp/invoice-meta.json', JSON.stringify({
    id: invoice.id,
    pageUrl: `/invoice/${invoice.id}/`,
    expiryTimestamp: invoice.expiryTimestamp,
    clientName: invoice.clientName,
    clientEmail: invoice.clientEmail,
    amount: invoice.items.reduce((s, i) => s + i.hours * i.rate, 0),
    calendlyLink: invoice.calendlyLink,
  }));
}

build().catch(err => { console.error(err); process.exit(1); });
```

LLM-powered CSS matching: how it actually works

The CSS matching pipeline has three stages: **extract → prompt → validate**. At build time, a headless browser visits the client's website, captures computed styles from every DOM element, and distills them into structured design tokens. These tokens feed an LLM prompt

that generates scoped CSS for the invoice layout. A PostCSS validation step ensures the output is clean.

Stage 1: Extraction uses Playwright to render the client's site and extract the actual computed colors, fonts, border radii, and spacing — not just what's in the CSS files, but what the browser actually renders after cascade resolution, media queries, and JavaScript execution.

```
// scripts/extract-design-tokens.js
const { chromium } = require('playwright');

async function extractDesignTokens(url) {
  const browser = await chromium.launch();
  const page = await browser.newPage();
  await page.goto(url, { waitUntil: 'networkidle' });

  const tokens = await page.evaluate(() => {
    const colors = new Map();
    const fonts = new Set();
    document.querySelectorAll('*').forEach(el => {
      const s = getComputedStyle(el);
      [s.color, s.backgroundColor, s.borderColor].forEach(c => {
        if (c && c !== 'rgba(0, 0, 0, 0)' && c !== 'transparent')
          colors.set(c, (colors.get(c) || 0) + 1);
      });
      if (s.fontFamily) fonts.add(s.fontFamily);
    });
    const sorted = [...colors.entries()].sort((a, b) => b[1] - a[1]);
    return {
      colors: sorted.slice(0, 12).map(([c]) => c),
      fonts: [...fonts],
    };
  });

  // Also extract CSS custom properties from :root
  const cssVars = await page.evaluate(() => {
    const vars = {};
    for (const sheet of document.styleSheets) {
      try {
        for (const rule of sheet.cssRules) {
          if (rule.selectorText === ':root') {
            for (const prop of rule.style) {
              if (prop.startsWith('--'))
                vars[prop] = rule.style.getPropertyValue(prop).trim();
            }
          }
```

```
        }
      } catch (e) {}
    }
    return vars;
  });

  // Detect Google Fonts
  const googleFonts = await page.evaluate(() =>
    [...document.querySelectorAll('link[href*="fonts.googleapis"]')]
      .map(l => l.href)
  );

  await page.screenshot({ path: '/tmp/client-screenshot.png', fullPage: true });
  await browser.close();

  return { ...tokens, cssVars, googleFonts };
}
```

An alternative to writing custom extraction is the **`dembrandt`** npm package, which wraps Playwright and outputs design tokens in the DTCG (Design Token Community Group) standard format with a single CLI command: `dembrandt https://acme.com > tokens.json`. It extracts colors, typography, spacing, borders, logos, and shadows automatically. github Open-source Projects

Stage 2: LLM generation sends the extracted tokens to Claude's API with a carefully constrained system prompt that enforces CSS scoping under `.invoice-container`, uses CSS custom properties for all brand values, and outputs only valid CSS without markdown fences.

```
// scripts/generate-css.js
const Anthropic = require('@anthropic-ai/sdk');
const fs = require('fs');

async function generateBrandCSS(tokens) {
  const anthropic = new Anthropic();
  const hasScreenshot = fs.existsSync('/tmp/client-screenshot.png');

  const content = [
    ...(hasScreenshot ? [{
      type: 'image',
      source: {
        type: 'base64',
        media_type: 'image/png',
        data: fs.readFileSync('/tmp/client-screenshot.png').toString('base64'),
      },
```

```
    }] : []),
    {
      type: 'text',
      text: `Generate CSS for a professional invoice page matching this brand.

EXTRACTED TOKENS:
- Top colors by frequency: ${tokens.colors.slice(0, 6).join(', ')}
- Font families: ${tokens.fonts.slice(0, 3).join(', ')}
- CSS variables from site: ${JSON.stringify(tokens.cssVars)}
- Google Fonts: ${tokens.googleFonts.join(', ') || 'none detected'}

REQUIREMENTS:
1. Scope ALL selectors under .invoice-container
2. Define CSS custom properties on .invoice-container
3. Style: .invoice-header, .invoice-meta, .invoice-table (th/td),
   .invoice-totals, .invoice-footer, .countdown-display, .cta-button
4. Include @media print styles
5. Output ONLY valid CSS — no markdown, no backticks, no explanation`,
    },
  ];

  const response = await anthropic.messages.create({
    model: 'claude-sonnet-4-20250514',
    max_tokens: 4096,
    temperature: 0.2,
    system: 'You are a CSS expert. Output only valid, scoped CSS. Never use !impo
    messages: [{ role: 'user', content }],
  });

  return response.content[0].text.replace(/```css\n?/g, '').replace(/```\n?/g, ''
}
```

Stage 3: Validation parses the output with PostCSS and auto-prefixes any selectors that aren't properly scoped:

```
const postcss = require('postcss');

async function validateCSS(css) {
  const root = postcss.parse(css);
  root.walkRules(rule => {
    if (!rule.selector.includes('.invoice-container')) {
      rule.selector = `.invoice-container ${rule.selector}`;
    }
  });
  return root.toString();
}
```

Graceful fallback — If the LLM API is unavailable or the client website can't be scraped, the build script falls back to a template that populates CSS custom properties with whatever colors were extracted (or sensible defaults). This ensures the build never fails:

```
function generateFallbackCSS(primaryColor) {
  return `.invoice-container {
    --brand-primary: ${primaryColor};
    --brand-text: #1f2937;
    --brand-bg: #ffffff;
    font-family: 'Inter', system-ui, sans-serif;
    /* ... standard invoice layout rules ... */
  }`;
}
```

Countdown timer and the self-destruct moment

The countdown uses `requestAnimationFrame` rather than `setInterval` — it's more battery-efficient (automatically pauses in background tabs) MDN Web Docs and syncs with the browser's repaint cycle. Medium Artmann The expiry timestamp is embedded as a `data-` attribute at build time, and all calculations use `Date.now()` in UTC, eliminating timezone bugs entirely.

The display follows a **graduated urgency** pattern that keeps the Mission Impossible theme fun rather than aggressive:

- **> 3 days remaining**: "⏳ This invoice will self-destruct in 5d 12h 30m 15s"
- **< 24 hours**: "🔥 Self-destruct sequence initiated — 23h 14m 02s"
- **< 1 hour**: "💣 Final countdown! 47m 12s"
- **Zero**: Triggers the disintegration animation

The self-destruct animation uses a **pixel disintegration effect** (commonly called the "Thanos snap"). It works by capturing the invoice DOM as a canvas via **html2canvas** (~40KB, the only dependency), distributing the pixels across 32 separate canvas layers with weighted randomness, then animating each layer with staggered rotation, translation, blur, and fade-out. Red Stapler The visual result is the invoice literally dissolving into particles from left to right over about 2 seconds.

```
async function disintegrate(element) {
  const sourceCanvas = await html2canvas(element);
```

```
  const { width, height } = sourceCanvas;
  const ctx = sourceCanvas.getContext('2d');
  const imageData = ctx.getImageData(0, 0, width, height).data;
  const LAYERS = 32;

  // Distribute pixels across canvas layers with left-to-right sweep
  const layers = Array.from({ length: LAYERS }, () =>
    ctx.createImageData(width, height)
  );

  for (let i = 0; i < imageData.length; i += 4) {
    const x = (i / 4) % width;
    const base = Math.floor((x / width) * LAYERS);
    const jitter = Math.floor((Math.random() - 0.5) * 8);
    const target = Math.max(0, Math.min(LAYERS - 1, base + jitter));
    for (let c = 0; c < 4; c++) layers[target].data[i + c] = imageData[i + c];
  }

  element.style.opacity = '0';
  const rect = element.getBoundingClientRect();

  layers.forEach((imgData, idx) => {
    const canvas = document.createElement('canvas');
    canvas.width = width;
    canvas.height = height;
    Object.assign(canvas.style, {
      position: 'absolute', left: `${rect.left}px`,
      top: `${rect.top + scrollY}px`, pointerEvents: 'none',
      transition: `transform ${800 + idx * 50}ms ease-out,
                   opacity ${600 + idx * 50}ms ease-out`,
    });
    canvas.getContext('2d').putImageData(imgData, 0, 0);
    document.body.appendChild(canvas);

    requestAnimationFrame(() => setTimeout(() => {
      canvas.style.transform = `rotate(${(Math.random()-.5)*30}deg)
        translate(${(Math.random()-.3)*120}px, ${(Math.random()-.7)*80}px)`;
      canvas.style.opacity = '0';
      canvas.style.filter = 'blur(2px)';
      setTimeout(() => canvas.remove(), 1500);
    }, idx * 70));
  });
}
```

After the animation completes (~2.5 seconds), the page fades to the **expired state**: a centered "💥 This invoice has self-destructed!" message, a subtitle ("Your mission, should

you choose to accept it, is to book a new call"), and a prominently styled CTA button that opens a Calendly inline embed pre-filled with the client's name and email. UTM parameters track the booking source as `utm_source=expired_invoice` Calendly with the invoice ID in `utm_content`, Calendly so you can attribute rebookings directly to the self-destruct flow.

GitHub Actions deployment pipeline

The pipeline is triggered via the GitHub API's `workflow_dispatch` endpoint, Graphite which can be called by a CRM webhook, Zapier, Make.com, or a manual curl command. Invoice data is passed as a **base64-encoded JSON string** to work around GitHub's 10-input limit for workflow_dispatch. Medium KodeKloud Notes

```
# .github/workflows/generate-invoice.yml
name: Generate and Deploy Invoice

on:
  workflow_dispatch:
    inputs:
      invoice_data:
        description: 'Base64-encoded JSON invoice data'
        required: true
        type: string

jobs:
  build-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with: { node-version: '20', cache: 'npm' }
      - run: npm ci

      - name: Decode invoice data
        run: echo '${{ inputs.invoice_data }}' | base64 -d > invoice-data.json

      - name: Install Playwright browsers
        run: npx playwright install chromium

      - name: Build invoice page
        run: node scripts/build.js
        env:
          INVOICE_DATA_FILE: invoice-data.json
          ANTHROPIC_API_KEY: ${{ secrets.ANTHROPIC_API_KEY }}
```

```
      - name: Deploy to Cloudflare
        id: deploy
        uses: cloudflare/wrangler-action@v3
        with:
          apiToken: ${{ secrets.CLOUDFLARE_API_TOKEN }}
          accountId: ${{ secrets.CLOUDFLARE_ACCOUNT_ID }}
          command: deploy

      - name: Register invoice in D1
        run: node scripts/register-invoice.js
        env:
          WORKER_URL: https://invoice-system.yourdomain.com
          REGISTER_SECRET: ${{ secrets.REGISTER_SECRET }}
```

The `cloudflare/wrangler-action@v3` is the current recommended action GitHub — the older `cloudflare/pages-action` is **deprecated**. GitHub The `wrangler deploy` command deploys both the static assets in `public/` and the Worker script in a single operation. Cloudflare

Triggering from external systems requires a POST to the GitHub API:

```
ENCODED=$(echo '{"clientName":"Acme","clientEmail":"cto@acme.com",
  "clientWebsite":"https://acme.com",
  "calendlyLink":"https://calendly.com/you/30min",
  "items":[{"description":"Web Dev","hours":40,"rate":150}]}' | base64 -w0)

curl -X POST \
  -H "Authorization: Bearer $GITHUB_PAT" \
  -H "Accept: application/vnd.github+json" \
  https://api.github.com/repos/you/invoice-system/actions/workflows/generate-invo
  -d "{\"ref\":\"main\",\"inputs\":{\"invoice_data\":\"$ENCODED\"}}"
```

For Zapier or Make.com, this is a single HTTP POST action with the base64 encoding done in a preceding code step. A **204 No Content** response confirms the workflow was queued. Medium

---

Security: making invoice URLs unguessable and data safe

Invoice URLs use **UUID v4** identifiers, which provide **122 bits of randomness** — approximately 5.3×10^{36} possible values. Zoer Brute-forcing is computationally infeasible. Never use sequential IDs or UUID v1 (which leaks MAC addresses and timestamps).

VerSprite The resulting URL pattern is `https://invoices.yourdomain.com/invoice/a1b2c3d4-e5f6-7890-abcd-ef1234567890/`.

Every invoice page includes these protective measures:

- `<meta name="robots" content="noindex, nofollow, noarchive, nosnippet">` prevents search engine indexing Google
- A `robots.txt` file with `Disallow: /invoice/` blocks crawlers at the directory level
- **Cache-Control: no-store** headers prevent browsers and CDNs from caching invoice content after expiry Medium
- A **Content Security Policy** whitelists only `self`, Calendly's asset domain, and Calendly's iframe origin — blocking all other script and frame sources
- `X-Frame-Options: DENY` and `Referrer-Policy: no-referrer` prevent clickjacking and URL leakage through referrer headers

These headers should be set in the Worker's fetch handler for `/invoice/*` routes, or via Cloudflare's `_headers` file in the `public/` directory:

```
/invoice/*
  X-Robots-Tag: noindex, nofollow, noarchive
  Cache-Control: no-store, no-cache, must-revalidate, max-age=0
  X-Content-Type-Options: nosniff
  X-Frame-Options: DENY
  Referrer-Policy: no-referrer
  Content-Security-Policy: default-src 'self'; script-src 'self' https://assets.c
```

For additional protection on high-value invoices, consider adding a **simple email verification gate**: before showing the invoice, the page prompts for the client's email address. A Worker endpoint validates it against the stored `client_email` in D1. This prevents casual link-sharing exposure with minimal friction.

Step-by-step implementation roadmap

Phase 1 — Foundation (Day 1–2): Set up the Cloudflare Worker project. Run `wrangler init invoice-system`, configure `wrangler.toml` with the D1 binding and cron trigger, create the D1 database with `wrangler d1 create invoice-db`, apply the schema with `wrangler d1 execute invoice-db --file=schema.sql`, and deploy a hello-world Worker to confirm the infrastructure works. Set up the GitHub repo with the folder structure above and

configure repository secrets for `CLOUDFLARE_API_TOKEN`, `CLOUDFLARE_ACCOUNT_ID`, `ANTHROPIC_API_KEY`, and `RESEND_API_KEY`.

Phase 2 — Static invoice page (Day 2–4): Build the EJS template with the invoice layout, countdown timer JavaScript, and self-destruct animation. Start with hardcoded test data and a default color scheme. Get the countdown working in the browser, verify the Thanos snap animation triggers correctly at zero, and confirm the Calendly embed loads in the expired state. This is purely frontend work — no backend needed yet.

Phase 3 — Build pipeline (Day 4–5): Create the `build.js` script that reads JSON input and generates `public/invoice/{uuid}/index.html`. Write the GitHub Actions workflow with `workflow_dispatch`, test it by triggering via the GitHub API, and confirm the built page deploys correctly to Cloudflare via `wrangler deploy`.

Phase 4 — LLM CSS matching (Day 5–7): Implement the Playwright design token extraction script. Integrate the Claude API call for CSS generation. Add PostCSS validation. Test against 5–10 different client websites to tune the prompt and ensure reliable output. Build the fallback CSS path for when extraction fails.

Phase 5 — Expiry email system (Day 7–8): Implement the Worker's `scheduled()` handler and the `/api/register` endpoint. Set up the Resend account and configure DNS (SPF/DKIM records for your sending domain). Write the `register-invoice.js` script that runs post-deploy to insert invoice metadata into D1. Test the full cron flow locally with `wrangler dev --test-scheduled`.

Phase 6 — Integration and polish (Day 8–10): Connect the full pipeline end-to-end: CRM webhook → GitHub API → build → deploy → register → cron → email. Add error handling, logging, and monitoring. Polish the expired page design, test on mobile devices, and verify all security headers are in place. Set up a Zapier/Make.com integration template for non-technical team members to trigger invoices.

Conclusion

This architecture achieves the rare combination of being both technically simple and experientially delightful. The entire system runs on Cloudflare's free tier (Workers free plan includes **100,000 daily requests**, D1 includes **5 million reads/month**, and Resend provides **3,000 emails/month**) — making it effectively zero-cost for a typical consulting or agency invoice volume. The key architectural insight is using a **single Cloudflare Worker project** with static assets rather than separate Pages + Workers, which unifies cron triggers, D1 access, and static serving under one deploy command. The LLM CSS matching — while the most complex component — degrades gracefully to a CSS-variables-based fallback, ensuring the build never fails even when the client's website is unreachable or the LLM API

is down. The pixel disintegration effect, requiring only html2canvas as a dependency, delivers a genuinely memorable “self-destruct” moment that turns an invoice follow-up from a chore into a conversation starter.