SiteGate

AI-Powered Autonomous Web Development Platform

Architecture & Implementation Document

Current State — Claude Code SDK + Iterative Agentic Loop

Aaron Gonsior

April 2026

Status: Core platform fully operational

Contents

1 Executive Summary **2**

2 High-Level System Architecture **3**

3 AWS Backend Architecture **4**

4 Frontend Architecture **5**

5 Authentication & Credits **6**
- 5.1 Auth Flow 6
- 5.2 Credit System 6

6 Execution Flow **7**

7 Agentic Loop Design **8**
- 7.1 Overview 8
- 7.2 Context Files 10
- 7.3 SDK Call Signature 11
- 7.4 Termination & ECS Task Lifecycle 11

8 Asynchronous Handoff Pattern **12**

9 Docker Image Specification **13**

10 Cost Model **14**

11 Implementation Status **14**

12 Dynamic Agentic Workflow — Design Reference **16**
- 12.1 Design Philosophy 16
- 12.2 Model Assignments 16
- 12.3 Context Files 17
- 12.4 Resume Detection (Startup Logic) 18
- 12.5 Step Specifications 19
- 12.6 Execution Flow 21
- 12.7 State File Specifications 22
- 12.8 Cost Profile 24

13 Risks & Mitigations **25**

Executive Summary

SiteGate is an AI-powered web development platform where users describe websites in natural language and receive live, hosted results. Users interact through a dual-pane web interface: a chat panel on the left and a live `<iframe>` preview on the right.

The architecture is built around the **Claude Code SDK** (`@anthropic-ai/claude-code`), which exposes the full Claude Code agentic loop programmatically. Each user prompt launches an AWS Fargate task that runs an **iterative build-and-verify loop**: Claude builds the site, then self-assesses its work, optionally improves it, and signals completion via a `return.json` file. Files are persisted to S3 across sessions.

Layer	Original Plan	Implemented
Iteration	Single-pass	Build + self-assessment loop
Frontend	React / Next.js	Plain HTML + CSS + JS (no build step)
Auth	Not planned	JWT + bcrypt, credit system

Table 1: Planned vs. implemented architecture.

High-Level System Architecture

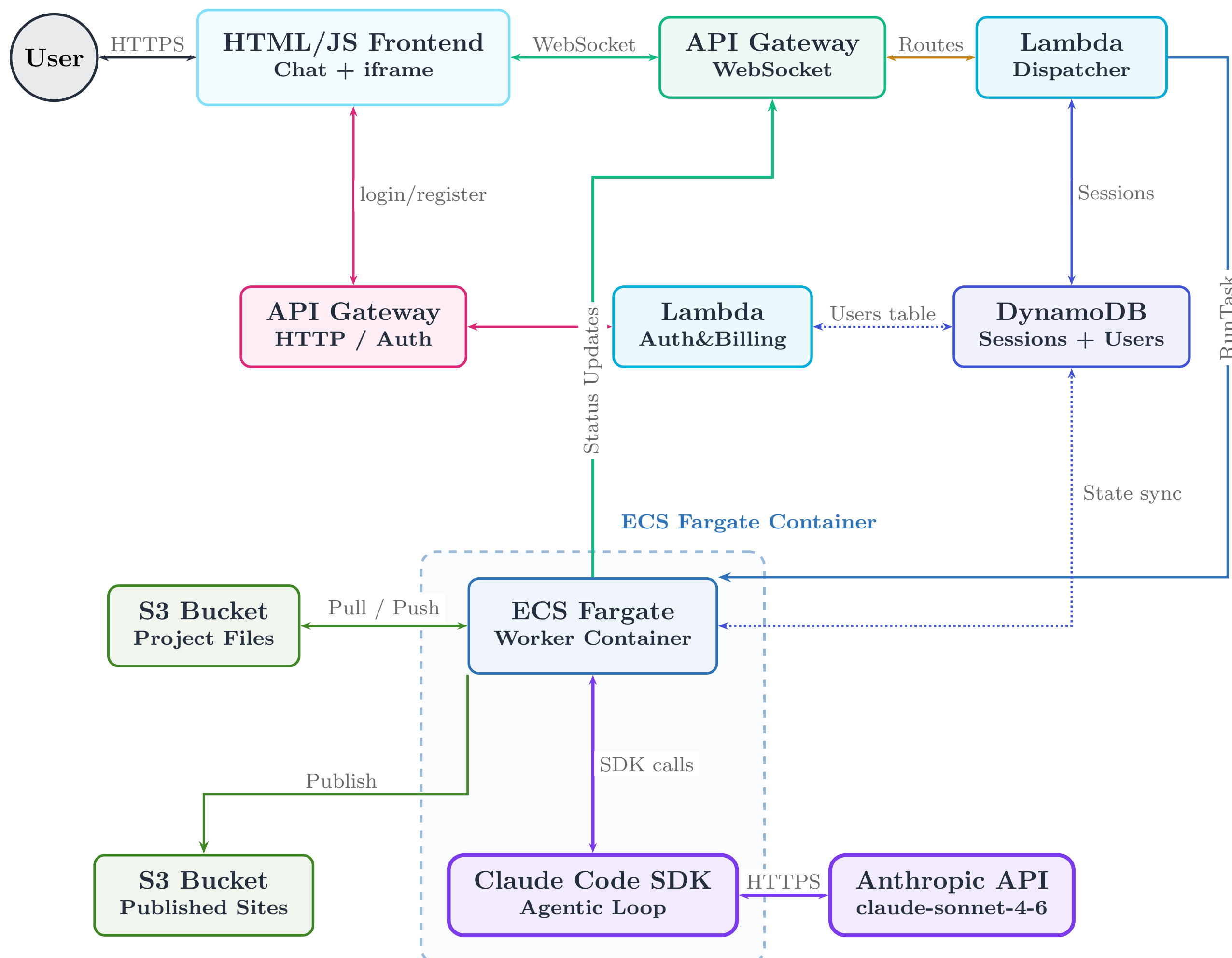

Figure 1: High-level system architecture. Purple = Claude SDK. Cyan = Go Lambda. The Fargate container hosts the SDK and runs the iterative build loop. Published sites are loaded in the frontend iframe via direct HTTPS S3 URL.

AWS Backend Architecture

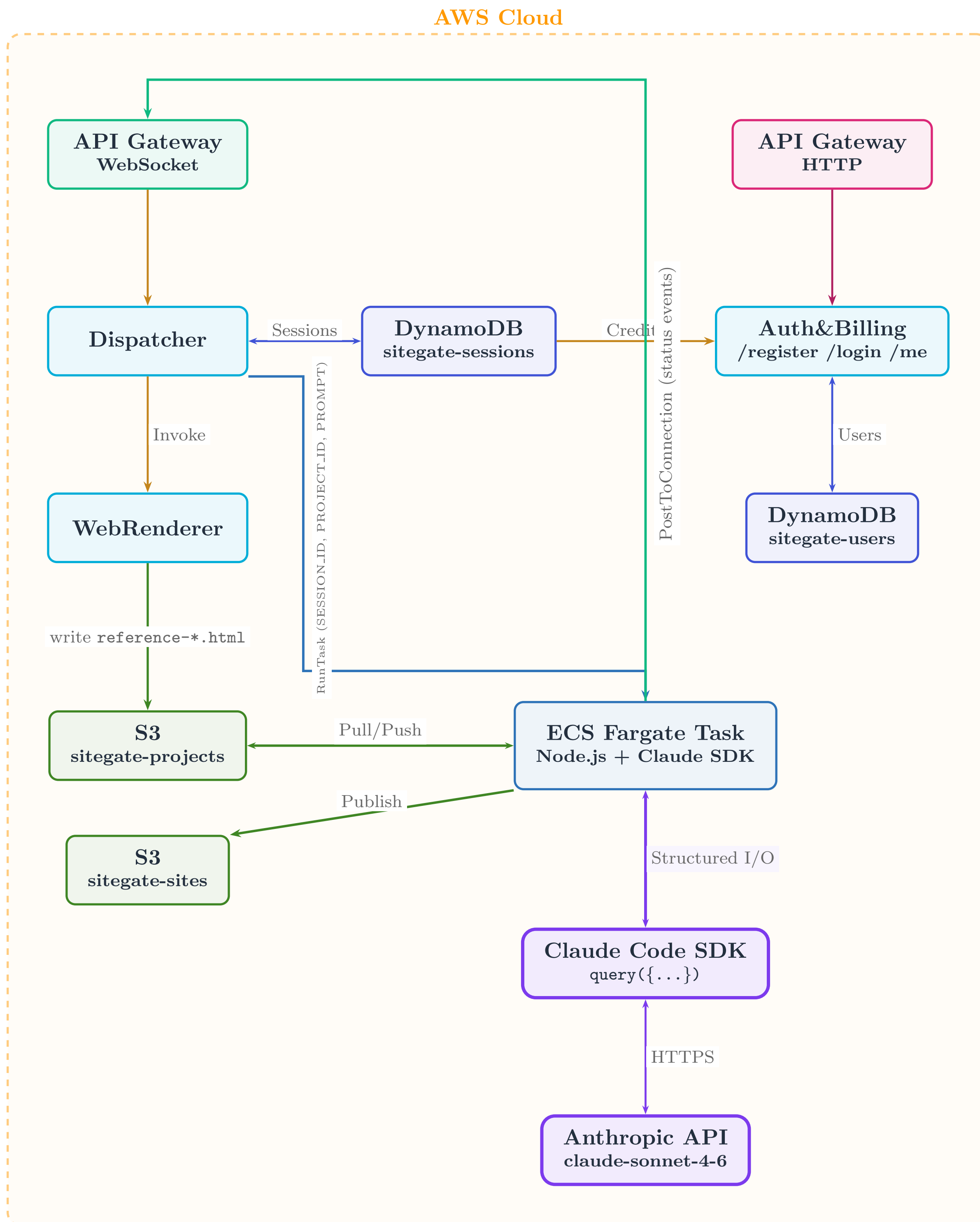

Figure 2: AWS backend. Dispatcher handles WebSocket; on each prompt it calls Haiku to extract reference URLs, invokes the WebRenderer Lambda (Playwright/Chromium) in reference mode if any are found, then launches Fargate. After publishing, the worker invokes WebRenderer in validate mode to capture console errors and auto-fix them (max 2 attempts). Auth-Billing handles HTTP auth. Fargate drives the Claude SDK agentic loop and streams status events back via the Management API.

Frontend Architecture

The frontend is plain HTML + CSS + JavaScript with no framework and no build step. Deployment is a direct S3 sync.

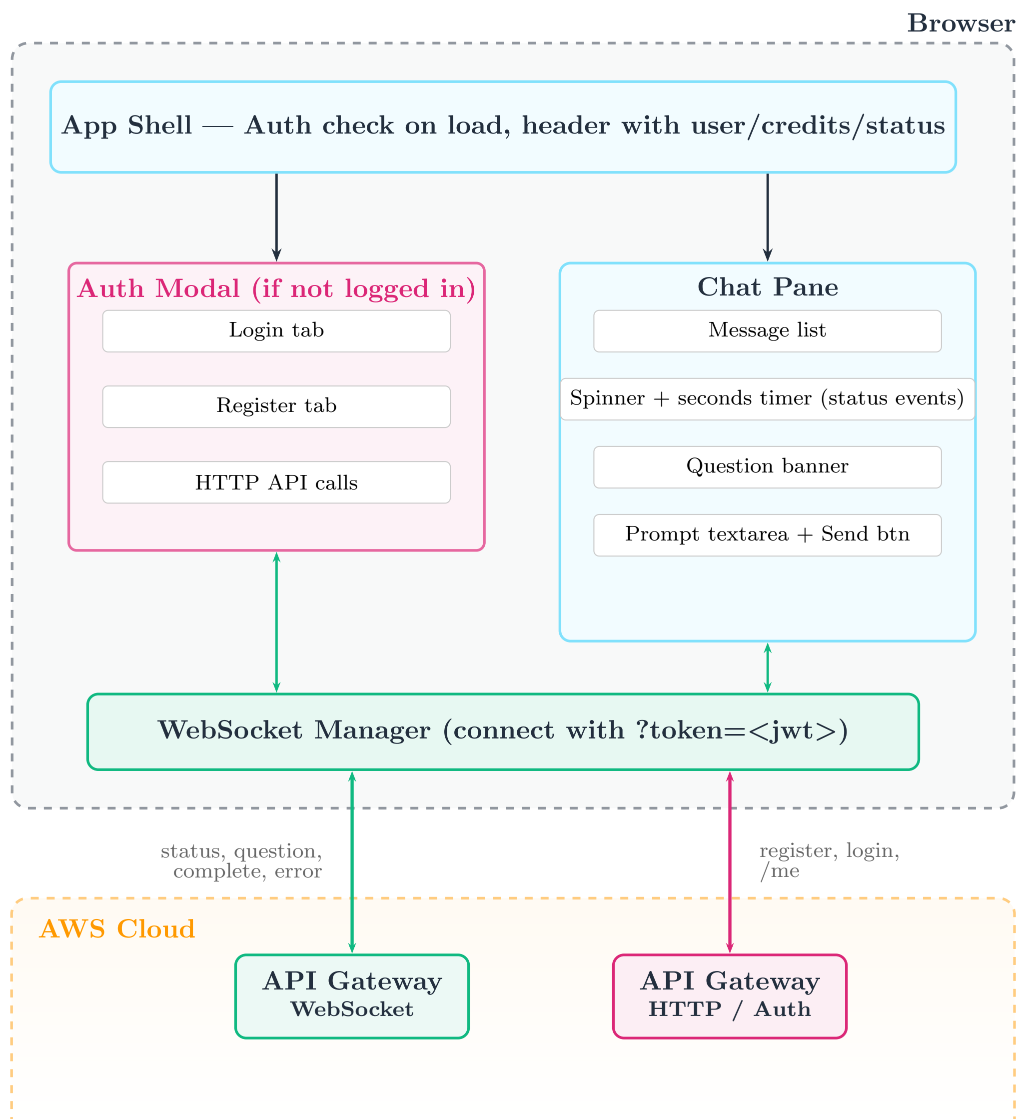

Figure 3: Frontend: plain HTML/JS. Auth modal shown on load if no valid JWT. After login, WebSocket connects with the JWT as a query parameter. Three-pane layout: collapsible Projects sidebar (rename via ...), Chat pane (project name header + spinner timer), iframe Preview. Build button disabled when credits are zero. The API Gateways sit inside the AWS Cloud boundary (faded at bottom to indicate additional backend infrastructure not shown here — see Figure 2).

Authentication & Credits

Auth Flow

1. User submits login or registration form (plain HTML form).
2. Frontend POSTs to sitegate-auth-billing via HTTP API.
3. Lambda hashes/verifies password with bcrypt, issues a JWT (HS256, 30-day expiry).
4. Frontend stores JWT in localStorage.
5. All WebSocket connections include ?token=<jwt>.
6. The dispatcher's $connect handler verifies the JWT using the same JWT_SECRET and stores the authenticated userId in the DynamoDB session record.

Credit System

Users have a credits integer field in the sitegate-users table. New accounts start at 0 credits. Credits are checked at two layers:

Layer	Mechanism	Can be bypassed?
Frontend	Build button disabled; submit() returns early if credits ≤ 0	Yes (JS)
Backend	Dispatcher reads DynamoDB before RunTask; sends WS error if ≤ 0	No

Table 2: Two-layer credit check. Backend check is authoritative.

Credit deduction is atomic: the dispatcher uses DynamoDB UpdateItem with ConditionExpression: "credits > :zero" and SET credits = credits - :one, returning the updated balance in the same call (ReturnValues: UPDATED_NEW). This prevents race conditions where two simultaneous prompts both see credits=1. A ConditionalCheckFailedException maps to a "no credits" error sent back via WebSocket.

Execution Flow

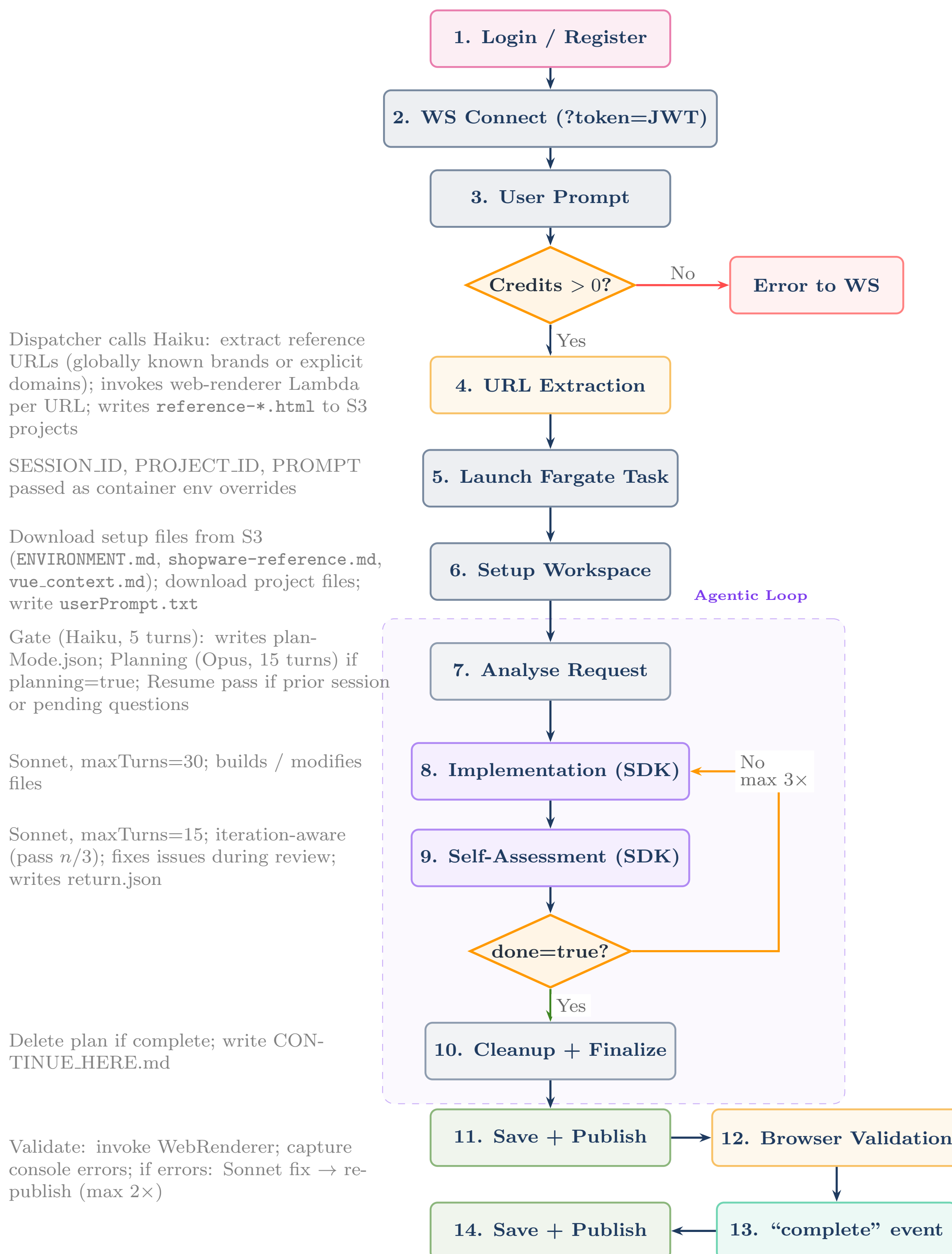

Figure 4: End-to-end execution flow. Credits are checked in the Lambda before any Fargate task is launched. Step 4 runs Haiku URL extraction and the web-renderer Lambda (reference mode, synchronously) before Fargate starts, so `reference-*.html` files are ready in the workspace on arrival. Step 7 runs a Gate (Haiku) and optionally a Planning (Opus) or Resume pass. The implementation + self-assessment loop (steps 8–9) repeats up to 3 times; the self-assessment is iteration-aware and treats in-pass fixes as done. Cleanup and Finalization run after the loop before publishing. Step 12 invokes the web-renderer in validate mode to capture console errors; if errors are found, a Sonnet fix pass runs and the site is re-published (max 2 attempts, best-effort).

Agentic Loop Design

Overview

The orchestrator (`worker/src/orchestrator.js`) runs a multi-step pipeline for every user prompt. Each Fargate task executes three phases:

1. **Pre-loop routing** — determines the session type. A Gate call (Haiku 4.5, cheap) decides whether full planning is needed; if prior session state exists (`OPEN_QUESTIONS.md` or `implementationPlan.md`), dedicated Resume passes (Opus 4.6) absorb context and skip the Gate entirely.

2. **Iteration loop** — up to 3 passes of Implementation (Sonnet 4.6, maxTurns=30) followed by Self-Assessment (Sonnet 4.6, maxTurns=15). The Self-Assessment is iteration-aware: it knows which pass it is (`pass` n`/3`) and treats in-pass fixes as `done:true`. If Claude writes `OPEN_QUESTIONS.md` during any pass, the grace-period handoff pattern activates (see § Asynchronous Handoff Pattern).

3. **Post-loop** — Cleanup (Sonnet, 5 turns) checks whether the implementation plan was fully executed and flags it for deletion. Finalization (Sonnet, 10 turns) writes `CONTINUE_HERE.md` with an accurate session summary. After publishing, a browser validation step invokes the WebRenderer Lambda to detect console errors; if errors are found, a Sonnet fix pass runs and the site is re-published (max 2 attempts, best-effort).

Each step uses the model best suited to its cost/quality tradeoff: Haiku for cheap routing, Opus for deep planning and context absorption, Sonnet for all implementation and review work. All SDK calls share the same `query()` interface but differ in model, `maxTurns`, and prompt.

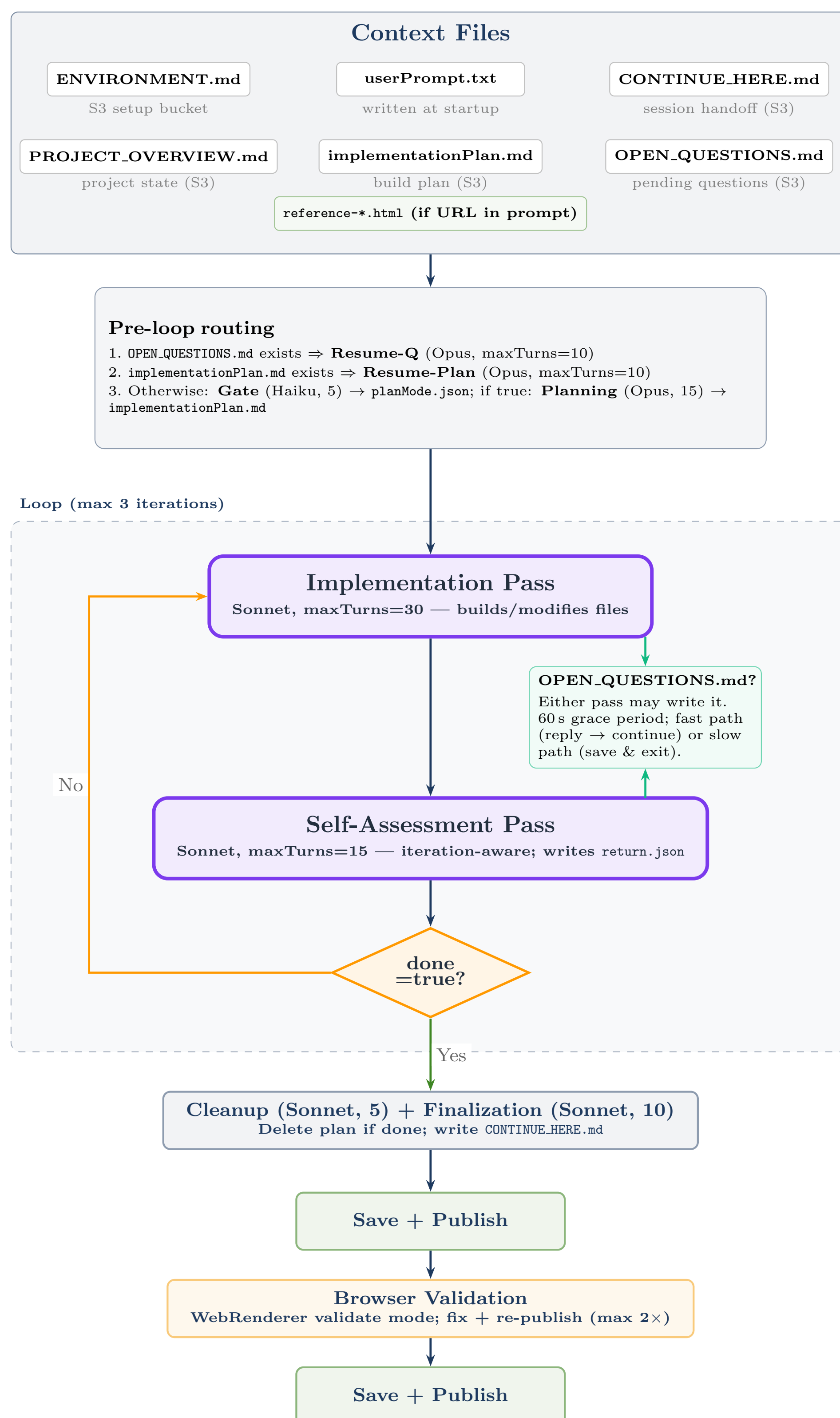

Figure 5: The agentic pipeline. Pre-loop routing selects Gate/Planning/Resume based on session state. The implementation + self-assessment loop runs up to 3 times. Either SDK pass may write OPEN_QUESTIONS.md to invoke the Async Handoff pattern (fast or slow path). Cleanup and Finalization always run after the loop. After publishing, browser validation invokes the web-renderer Lambda to detect console errors; fix passes run automatically (max 2 attempts).

Context Files

File	Content	Lifecycle
`ENVIRONMENT.md`	Environment description, tools, goals, validation awareness	Downloaded from `sitegate-setup` bucket at startup (with `vue_context.md`, `shopware-reference.md`); fallback bundled in Docker image; never modified
`userPrompt.txt`	Current user request or answer to open questions	Written at startup from `PROMPT` env var; overwritten on grace-period fast path
`CONTINUE_HERE.md`	Session handoff: what was done, current state, next steps	Written by Finalization every session; persisted to S3
`PROJECT_OVERVIEW.md`	What was built, file structure, key decisions	Created on first run; persisted to S3; updated each session
`implementationPlan.md`	Detailed build plan	Written by Planning (Opus); persisted; deleted when `planExecuted=true`
`OPEN_QUESTIONS.md`	Blocking questions for the user	Written by any pass; triggers Async Handoff; deleted when answered
`return.json`	`{done: true|false}`	Written by Self-Assessment; read by orchestrator to loop or exit
`planMode.json`	`{planning: true|false}`	Written by Gate; read once; not persisted
`reference-*.html`	Pre-rendered reference site HTML	Written by web-renderer Lambda before Fargate launch; read for style reference

Table 3: Context files used by the agentic pipeline.

SDK Call Signature

All SDK passes use the same `query()` call shape but differ in model, `maxTurns`, and prompt content:

```
const { query } = require("@anthropic-ai/claude-code");

for await (const event of query({
  prompt: builtPrompt,   // constructed per-pass from context files
  options: {
    cwd:          `/workspace/${PROJECT_ID}`,
    maxTurns,              // varies: Gate=5, Planning=15, Resume=10,
                           //         Implementation=30, Self-Assessment=15,
                           //         Cleanup=5, Finalization=10
    allowedTools: ["Bash", "Read", "Write", "Edit", "Glob", "Grep", "LS"],
    model,                 // Haiku for Gate; Opus for Planning/Resume;
                           // Sonnet for Implementation/Self-Assessment/Cleanup/Finalization
  },
})) {
  const msg = translateEvent(event);   // tool use → friendly status string
  if (msg) await ws.send("status", msg);
}
```

The `query()` generator yields typed events (`assistant`, `user`, `result`). Tool calls and text blocks are extracted from `assistant` events to produce user-friendly status strings streamed to the frontend via WebSocket.

Termination & ECS Task Lifecycle

The ECS task terminates naturally — there is no explicit API call to stop it. When the Node.js `main()` function resolves (after publishing), the process exits and ECS marks the task as `STOPPED`.

1. Self-Assessment writes `return.json{done:true|false}`.
2. Node.js reads `return.json`; loops or breaks accordingly (max 3 iterations).
3. After loop exit: Cleanup pass runs (deletes plan if complete).
4. Finalization pass runs (writes `CONTINUE_HERE.md`).
5. Workspace uploaded to `sitegate-projects`; site published to `sitegate-sites`.
6. Post-publish validation: web-renderer Lambda invoked in validate mode; if console errors found, Sonnet fix pass runs, site is re-published (max 2 attempts, best-effort).
7. `complete` WebSocket event sent with the site URL.
8. `process.exit(0)` — Node.js exits cleanly.
9. ECS detects the container exited and transitions the task to `STOPPED`.

The error path (`catch` block) sends a WebSocket `error` event, sets DynamoDB status to `ERROR`, then calls `process.exit(1)`.

Asynchronous Handoff Pattern

The handoff pattern handles the case where any SDK pass writes `OPEN_QUESTIONS.md` during a build. The Finalization pass runs first to save session state, then the container sends a `question` WebSocket event and waits up to 60 seconds (grace period), polling DynamoDB every 3 s. If the user responds in time (**fast path**), the reply is written to `userPrompt.txt`, `OPEN_QUESTIONS.md` is deleted, and the iteration loop continues in the *same container* with no restart. If the timer expires (**slow path**), the workspace is pushed to S3, DynamoDB is set to `WAITING_ON_USER`, and the container exits; the user's eventual reply launches a new Fargate task that detects `OPEN_QUESTIONS.md` and runs the Resume-Q pass. A race condition — reply arriving during shutdown — is resolved by launching a new Fargate task.

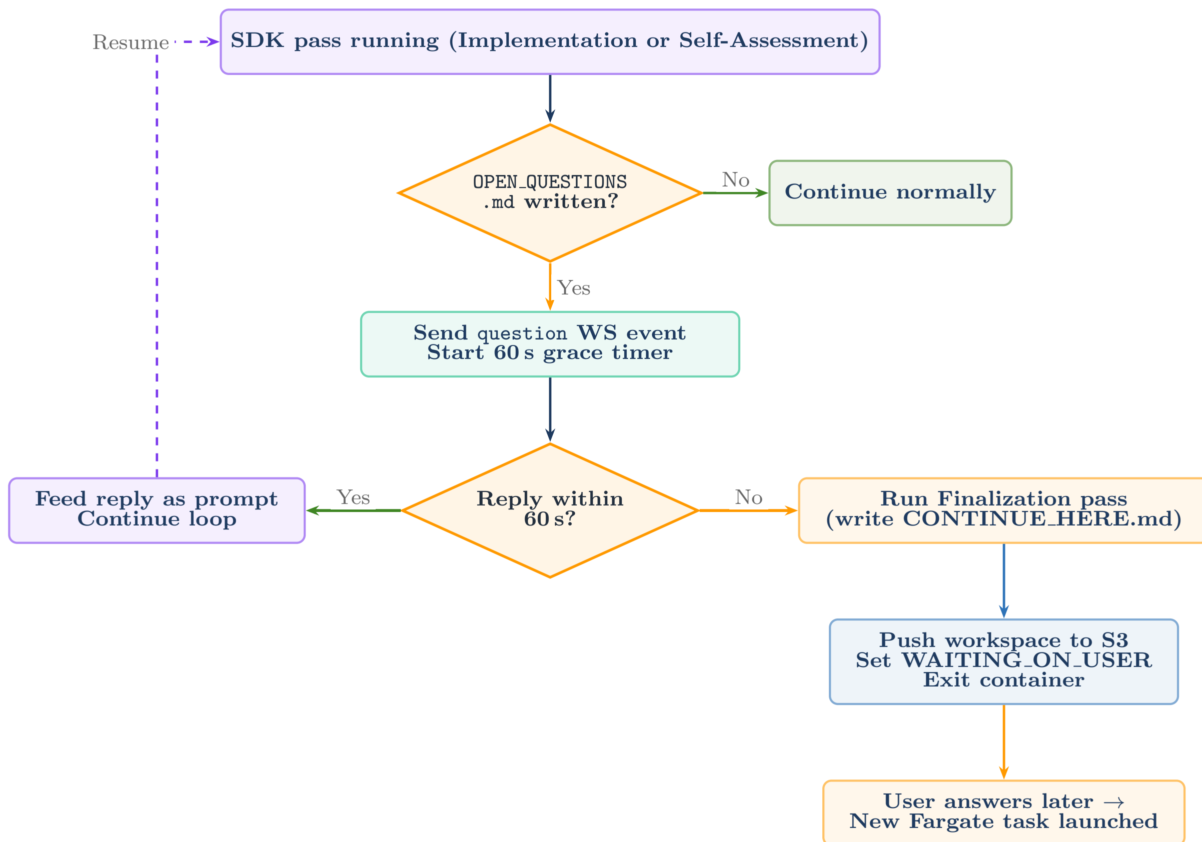

Figure 6: Grace-period handoff pattern. Question detection is file-based: any SDK pass may write `OPEN_QUESTIONS.md`; the orchestrator checks for its existence after each pass. Finalization runs before the grace timer starts to ensure state is saved. Fast-path cost: ∼$0 (container stays alive, no restart). Slow-path restart cost: ∼30–45 s cold boot + S3 pull. Race condition: reply arriving during shutdown triggers a new Fargate task.

Docker Image Specification

Layer	Contents
Base	`node:20-slim`
System deps	`git`, `curl`, `python3`, `python3-pip`
Node deps	`@anthropic-ai/claude-code`, `@aws-sdk/client-dynamodb`, `@aws-sdk/client-s3`, `@aws-sdk/client-apigatewaymanagementapi`
Worker source	`src/orchestrator.js`, `src/wsClient.js`, `src/s3Client.js`
Context file	`ENVIRONMENT.md` (at `/app/ENVIRONMENT.md`; bundled as fallback only — normally all root-level files are downloaded from the `sitegate-setup` S3 bucket at startup)
Platform	`linux/amd64` (matches Fargate default X86_64)

Table 4: Docker image layers. Build command: `docker build --platform linux/amd64`.

Build and push:

```
# PowerShell (Windows)
$token = aws ecr get-login-password --region us-east-1
docker login --username AWS --password $token \
  137692571755.dkr.ecr.us-east-1.amazonaws.com

docker build --platform linux/amd64 -t sitegate-worker ./worker
docker tag sitegate-worker:latest \
  137692571755.dkr.ecr.us-east-1.amazonaws.com/sitegate-worker:latest
docker push \
  137692571755.dkr.ecr.us-east-1.amazonaws.com/sitegate-worker:latest
```

Cost Model

Component	Pricing Model	Est. per interaction
ECS Fargate	vCPU-hour + GB-hour (pay while running)	\$0.02–0.10
Grace period	Fargate idle wait for user reply (worst case 60 s)	\$0.00–0.04
Claude Sonnet 4.6	Input/output tokens × up to 6 SDK calls per interaction	\$0.05–0.80
DynamoDB	Read/write capacity units	\$0.001
S3	Storage + requests	\$0.001
API Gateway	WebSocket messages	\$0.001
Total		**\$0.07–0.92**

Table 5: Estimated cost per interaction. Up to 6 SDK calls (3 iterations × work + check) is the worst case. Most interactions complete in 1–2 iterations. Dominant cost: Claude API tokens.

Implementation Status

1. **Phase 1 – Core Engine** [COMPLETE]
   - Fargate task with Claude Code SDK (`query()`)
   - S3 project sync (pull on boot, push on completion)
   - Iterative build-and-verify loop with `return.json` termination
   - ENVIRONMENT.md context injection; PROJECT_OVERVIEW.md maintenance
   - Template-based status formatting; WebSocket streaming
2. **Phase 2 – API & Frontend** [COMPLETE]
   - WebSocket API (API Gateway + `sitegate-dispatcher` Lambda, Go)
   - HTTP Auth API (`sitegate-auth-billing` Lambda, Go)
   - Plain HTML/JS frontend deployed via S3 + CloudFront
   - JWT auth, bcrypt passwords, credits system
   - Two-layer credit check (frontend + backend dispatcher)
3. **Phase 3 – Handoff Pattern** [COMPLETE]
   - File-based question detection (`OPEN_QUESTIONS.md` presence; no `?` heuristic)
   - 60-second grace period; finalization runs before wait
   - Fast path (reply in time) and slow path (suspend + `WAITING_ON_USER`)
   - `CONTINUE_HERE.md` handoff document; `OPEN_QUESTIONS.md` lifecycle
   - DynamoDB resume logic; race condition handled by dispatcher
4. **Phase 4 – Dynamic Agentic Workflow & Frontend Polish** [COMPLETE]
   - Five-step workflow: Gate (Haiku 4.5) → Planning (Opus 4.6) → Implementation loop (Sonnet 4.6) → Cleanup → Finalization

- Resume detection: `OPEN_QUESTIONS.md` and `implementationPlan.md`
- `cleanup.json` flags + orchestrator cleanup scripts
- `ENVIRONMENT.md` served from dedicated `sitegate-setup` S3 bucket
- Projects sidebar; per-user project list in `localStorage`
- Collapsible projects sidebar with toggle button (</>)
- Project name header above chat pane; inline rename via ... button
- Seconds timer displayed next to spinner during build status messages
- New project default name "New Project" (not derived from prompt)

5. **Phase 5 – Polish & Production** [IN PROGRESS]

   - Credit deduction on task launch (not yet implemented)
   - CloudFront CDN for hosted sites (currently direct S3 HTTPS URLs)
   - Payment integration for credit top-up
   - Error handling improvements, monitoring, alerting

Dynamic Agentic Workflow — Design Reference

Implemented. This section is the full design reference for the five-step workflow (plus post-publish browser validation) now running in production. It documents decisions, rationale, and the complete execution sequence.

Design Philosophy

- **Right model per task** — Haiku 4.5 for cheap yes/no routing; Opus 4.6 for deep planning (rare); Sonnet 4.6 for all implementation and finalization work.
- **Thorough over fast** — the loop is designed for quality output, not minimal latency. Sessions taking several minutes are expected and acceptable; the user is as unbothered as possible.
- **Persistent, trustworthy state** — `CONTINUE_HERE.md` is always kept accurate so any future Fargate task can resume with full context, regardless of how the previous session ended.
- **Clean question lifecycle** — open questions surface in `OPEN_QUESTIONS.md`; the finalization step clears them once resolved and re-populates them only if new ones arise.

Model Assignments

Step	Model	Rationale	maxTurns
Planning Gate	`claude-haiku-4-5`	Cheapest available; task is only yes/no	5
Deep Planning	`claude-opus-4-6`	Strongest reasoning; rare invocation	15
Implementation	`claude-sonnet-4-6`	Best balance of quality and cost	30
Self-Assessment	`claude-sonnet-4-6`	Same capability as implementation	15
Finalization	`claude-sonnet-4-6`	Short task; needs full context understanding	10

Table 6: Model assignment by step. Haiku runs every session; Opus only when planning is triggered.

Context Files

File	Written by	S3	Purpose
`ENVIRONMENT.md`	Setup bucket	No	Env description, tools, startup checklist, validation awareness
`userPrompt.txt`	Orchestrator	No	Current user request, uniform across all steps
`PROJECT_OVERVIEW.md`	Claude (1st run)	Yes	What was built, file structure, key decisions
`implementationPlan.md`	Claude (planning)	Yes	Detailed plan; deleted when `planExecuted=true`
`CONTINUE_HERE.md`	Claude (finalization)	Yes	Session summary; read at next startup
`OPEN_QUESTIONS.md`	Claude (any step)	Yes	Blocking questions; deleted when resolved
`return.json`	Claude (self-assess.)	No	`{done: true|false}` termination signal
`planMode.json`	Claude (gate)	No	`{planning: true|false}` routing signal
`cleanup.json`	Claude (cleanup)	No	`{planExecuted: true|false}` cleanup flag

Table 7: Context files. "Yes" = persisted to S3 projects bucket between sessions.

Resume Detection (Startup Logic)

Before running the gate, the orchestrator checks for two conditions that indicate a session should skip the gate and planning entirely:

1. **`OPEN_QUESTIONS.md` exists in workspace** — The previous session ended with unresolved questions. The current user prompt is the user's answers. The orchestrator runs an Opus 4.6 continuation pass reading `ENVIRONMENT.md`, `PROJECT_OVERVIEW.md`, `OPEN_QUESTIONS.md`, and `userPrompt.txt` (answers). Then deletes `OPEN_QUESTIONS.md` from workspace and S3, and proceeds directly to the Implementation loop.

2. **`implementationPlan.md` exists in workspace** — A plan from a prior session is in progress. The orchestrator runs an Opus 4.6 continuation pass to review and refine the existing plan, then proceeds directly to the Implementation loop. The file is deleted from workspace and S3 once the plan is fully implemented.

If neither condition is met, the orchestrator runs the normal gate → optional planning → implementation sequence.

Step Specifications

Step	Description	Runs
1 — Gate	Reads context files. Decides if thorough planning is needed. Writes `planMode.json`: simple follow-ups → `false`; new builds or major changes → `true`. Default: `false`. Always runs on first prompt (no resume files exist).	Unless resume path
2 — Planning	Writes `implementationPlan.md` with a detailed, file-level plan. May write `OPEN_QUESTIONS.md` → orchestrator suspends.	If `planning=true`
3 — Implementation	Builds/modifies site; updates `PROJECT_OVERVIEW.md`. Question detection is file-based: orchestrator checks for `OPEN_QUESTIONS.md` after the SDK call. If present: Finalization → 60 s grace timer → fast path (reply: continue) or slow path (timeout: save, suspend).	Each iteration
4 — Self-Assessment	Reviews work against `userPrompt.txt`. Writes `return.json`: `done=true` → break loop. May also trigger question handoff via `OPEN_QUESTIONS.md`.	After each pass
5 — Cleanup	Was the plan fully executed? Writes `cleanup.json`. Orchestrator deletes `implementationPlan.md` if `planExecuted=true`.	After loop exits
6 — Finalization	Writes `CONTINUE_HERE.md`. Clears or refills `OPEN_QUESTIONS.md`. Quality gate — runs before every exit.	Always

Table 8: Step specifications. Steps 3–4 repeat up to `MAX_ITERATIONS` (3) times. Steps 5–6 always run after the loop.

Execution Flow

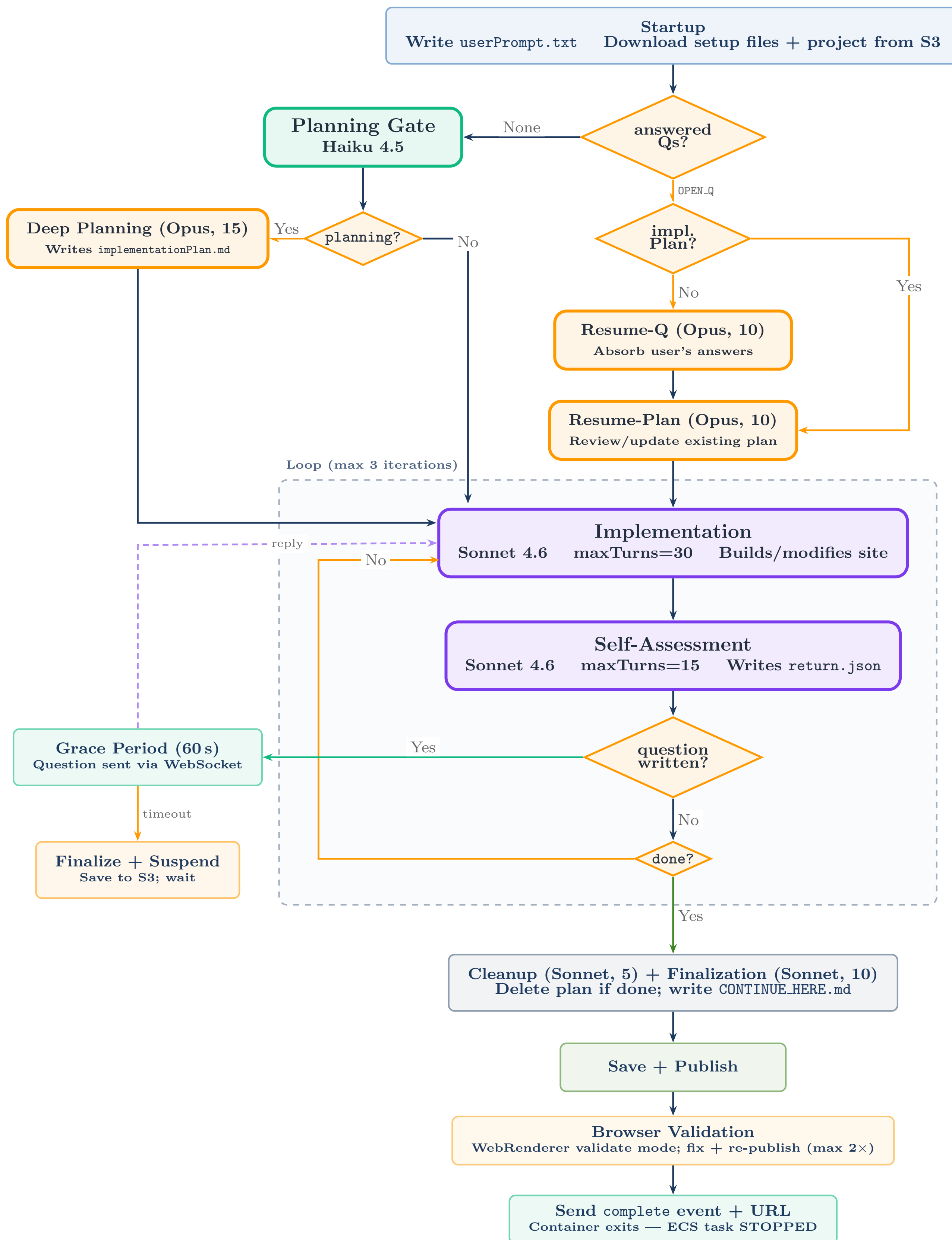

Figure 7: Detailed execution flow matching the orchestrator implementation. After startup, *answered Qs?* checks for `OPEN_QUESTIONS.md`. If present, a second diamond checks for `implementationPlan.md`: without a plan, Resume-Q absorbs answers then Resume-Plan runs; with a plan, Resume-Q is skipped and Resume-Plan handles both. If no questions exist, the normal Gate → optional Planning path runs. All paths converge at the implementation loop. After Self-Assessment, a question diamond triggers the grace-period handoff if `OPEN_QUESTIONS.md` was written. Browser validation runs after publishing.

State File Specifications

CONTINUE_HERE.md

Written (or overwritten) by the finalization step every session. Replaces the ad-hoc `CLAUDE_CONTINUE.md` used in the current implementation.

```
# Session Summary

## What was done this session
- [bullet list of changes made]

## Current project state
[brief description of what exists and works]

## What to pick up / resume from
[specific guidance for the next session]

## Open questions for the user
[present only if OPEN_QUESTIONS.md has content]
See OPEN_QUESTIONS.md for details.
```

OPEN_QUESTIONS.md — Lifecycle

Multiple questions may be present simultaneously.

1. May be written by any Claude step (gate, planning, implementation, self-assessment) whenever questions require user input.
2. The orchestrator checks for `OPEN_QUESTIONS.md` presence after *every* step. If present and non-empty, it sends a `question` WebSocket event to the frontend.
3. When a question is detected: the finalization step runs *first* (before waiting), then the 60 s grace timer starts.
4. **Fast path** (reply within 60 s): the reply feeds back as the current prompt; the loop continues in the same container. Finalization at session end clears `OPEN_QUESTIONS.md`.
5. **Slow path** (timeout): the workspace is saved to S3, container exits with status `WAITING_ON_USER`. Race condition: if a reply arrives during shutdown, a new Fargate task is started to resume.
6. When a new Fargate task starts and `OPEN_QUESTIONS.md` exists in the workspace: this triggers the *resume path* — Gate and Planning are skipped; an Opus 4.6 continuation pass reads `OPEN_QUESTIONS.md` together with the user's answer in `userPrompt.txt`. `OPEN_QUESTIONS.md` is then deleted from workspace and S3.

cleanup.json and Cleanup Scripts

After the implementation loop exits, a short Cleanup SDK call (Sonnet 4.6, `maxTurns=5`) asks Claude to assess completion state and write `cleanup.json`:

```
{ "planExecuted": true }
```

The orchestrator then runs cleanup shell scripts that read this file and act:

- If `planExecuted=true`: delete `implementationPlan.md` from workspace and from S3.

- Additional cleanup checks can be added to this phase in the future by extending `cleanup.json` with new flags.

Cleanup scripts run *before* the Finalization step so that the finalization step's `CONTINUE_HERE.md` accurately reflects which plan files still exist.

ENVIRONMENT.md — Delivery and Required Content

`ENVIRONMENT.md` is stored in the dedicated `sitegate-setup` bucket (key: `ENVIRONMENT.md`) and downloaded by the worker at container startup. Workers have read-only access (`s3:GetObject` only) to this bucket — they cannot write back to it. The file is not bundled in the Docker image. One-time upload after Terraform creates the bucket:

```
aws s3 cp worker/ENVIRONMENT.md \
  s3://sitegate-setup-137692571755/ENVIRONMENT.md
```

Updates require only re-uploading to that key — no Docker rebuild or ECR push.

The file must contain a **Startup Checklist** section and a **cleanup.json instructions** section:

> **At startup, before anything else:**
>
> 1. If `CONTINUE_HERE.md` exists, read it.
> 2. If `OPEN_QUESTIONS.md` exists, the user's prompt in `userPrompt.txt` answers those questions.
> 3. If `implementationPlan.md` exists, a planning continuation pass will run first.
> 4. Read `userPrompt.txt`.
>
> **When writing cleanup.json:** If `implementationPlan.md` has been fully implemented this session, set `planExecuted: true`. Otherwise set `planExecuted: false`.

Cost Profile

Step	Model	Notes	Est. add-on cost
Gate	Haiku 4.5	Every session; 5 turns max	< $0.01
Resume-Q / Resume-Plan	Opus 4.6	10 turns each; only on resume path	$0.10–0.50 each
Planning	Opus 4.6	Only when `planning=true`	$0.20–1.00
Implementation	Sonnet 4.6	$\leq N$ passes of 30 turns	$0.10–0.50 each
Self-Assessment	Sonnet 4.6	$\leq N$ passes of 15 turns	$0.05–0.20 each
Cleanup	Sonnet 4.6	5 turns; post-loop	$0.01–0.05
Finalization	Sonnet 4.6	10 turns; always	$0.02–0.10
Validation Fix	Sonnet 4.6	15 turns; only if browser errors found	$0.05–0.20 each
WebRenderer	—	Lambda invocation (Playwright); $\leq$2 calls	< $0.01
Total (typical)		1–2 iterations, no planning	**$0.20–1.00**
Total (complex)		3 iterations + Opus planning + validation fix	**$0.80–3.00**

Table 9: Estimated per-interaction cost for the dynamic workflow. Dominant cost: Sonnet 4.6 implementation passes and Opus 4.6 planning. Haiku gate and Lambda invocation costs are negligible. WebRenderer validation adds minimal cost (<$0.01 per Lambda call).

Risks & Mitigations

Risk	Impact	Mitigation
SDK API changes	Worker code breaks on update	Pin SDK version in `package.json`; test before deploy
Fargate cold start (30–45s)	UX latency on first prompt	Loading animation; consider warm pool in V2
Claude API cost spikes	Per-interaction cost exceeds budget	`maxTurns` cap; credit system limits total spend
SDK hang / infinite loop	Fargate runs indefinitely, high cost	`MAX_ITERATIONS=3` hard cap; Fargate task timeout
Claude writes invalid return.json	Loop doesn't terminate on schedule	Malformed JSON is caught; loop continues to max iterations then publishes
S3 sync race condition	Two tasks overwrite same project	`DynamoDB` session status prevents concurrent tasks
Mixed content in iframe	Preview broken on HTTPS frontend	Sites published as direct HTTPS S3 URLs (not HTTP website endpoint)

Table 10: Key risks and their mitigations.