

How We Built Franchise Performance Intelligence

A record of the human-and-AI build: what was made, which tools made it, and the one thing the machine could not solve.

THE STACK THAT BUILT IT

Claude architecture + specs

Claude Code the build

HubSpot Developer MCP the platform

HubSpot APIs ~60 endpoints

Emerjent memory + ops

Product	Franchise Performance Intelligence (FPI)
What it is	An installable HubSpot public app: a multi-agent operating layer for franchise networks
Built by	One builder, working with Claude and Claude Code
Live in	Portal 51781063 (fpi-demo), installed since 24 July 2026
Document	Build record. Every figure verified against the repository, the AWS account, and the HubSpot build API

CONTENTS

0	Why this document exists	3
1	The team and the toolchain	4
2	The four things that got built	7
3	The wall: the bug the machine could not find	14
4	How the build stayed honest	16
5	What the platform taught us	18
A	Appendix · The build at a glance	20

0 Why this document exists

The contest asks for a working agent. This document is the record of how one got made: what was built, which tools built it, and where the automated build hit a wall that a person had to climb over by hand.

It is a build record rather than a product pitch. A companion dossier covers what Franchise Performance Intelligence is and why it matters. This one stays on the making of it, because the interesting story of a modern build is not only the finished system. It is the division of labor between a human and a set of AI tools, the places where that division worked, and the one place where it did not.

THE SHORT VERSION OF THE BUILD

A single builder, working with Claude for architecture and specification and Claude Code for the actual code, shipped a marketplace-grade HubSpot public app: four custom objects, nine published agent tools, a fleet of Breeze agents, a Content Hub member portal, and a 46-function AWS backend. The platform was driven through the HubSpot Developer MCP and roughly sixty HubSpot API endpoints. Emerjent held the build memory, so nothing hard-won had to be learned twice.

The app is installed and running in a live HubSpot portal. Two Breeze agents do real diagnostic work against live data. And exactly one defect in the whole build could not be found by the tools that built it. A human found it by reading a screen.

0.1 What was built, in one list

Four things make up the finished system. The rest of this document is largely the story of building them.

Component	What it is
HubSpot public app	HubSpot-approved app objects, app events, agent tools, CRM cards, and a Settings page that deploys and provisions the install
AWS Lambdas	46 functions behind one API Gateway, holding every piece of logic and every credential
Agent Hub agents and workflows	Breeze agents that run the FPI agent tools, plus native HubSpot workflows for orchestration
Content Hub member portal	Membership-gated portal pages where a franchisee submits data and gets something back

Each of the four fought back in its own way, and each taught something about building a vertical app on this platform. The account that follows is honest about all of it, including the parts that are not finished, because a build record that only reports success is not a build record.

1 The team and the toolchain

This was a small team by headcount and a large one by capability: one person, and a set of AI tools each doing the kind of work it is genuinely good at. Setting out who did what is the clearest way to show how a build this size came from a team this small.

1.1 Claude, for architecture and specification

Claude was the reasoning partner. It shaped the architecture, wrote and rewrote the technical specification through two full versions, and worked through the design decisions that had no obvious answer. Three of those decisions set the shape of everything downstream.

- **Influence, not control.** The obvious version of this product autonomously optimises franchisees. That version is commercially wrong, because franchisees are independent owners and a system that acts on their business without consent is a liability rather than a tool. The design constraint that came out of that reasoning is absolute: the system recommends and equips, and a human approves. It shaped every surface after it.
- **Four authorization planes.** The same question, may this caller see this record, is answered four different ways depending on who is asking and what proves their identity. Naming those planes explicitly, rather than letting them blur, is what made the security model tractable later.
- **The refusal contract.** A language model handed a denial will try to turn it into an oracle if you let it. Designing the refusal so that an out-of-reach record is byte-identical to a nonexistent one was a reasoning task before it was a coding task.

The specification that came out of this work was not a static document. It was revised against what the platform actually turned out to allow, and the revision itself became a source of build knowledge, captured as it happened.

1.2 Claude Code, for the build

Claude Code did the construction. Working agentially against the repository, it produced the great majority of the code in the finished system.

Surface	What Claude Code built
The AWS backend	46 Lambda functions, a manifest-driven deploy system, capability-generated IAM, token custody, and the queue workers
The HubSpot app	The hsmeta component definitions for four app objects, eight associations, nine agent tools, two workflow actions, two CRM cards, and the Settings page
The Content Hub theme	The franwise theme: page templates, 30 modules, the HubL macro library, and the browser-side progressive enhancement

Surface	What Claude Code built
The test suites	719 backend tests across 36 files, the theme inspection suite, and the live isolation and governance test plans

The pattern that worked was tight: Claude Code proposed and built, the build was exercised against a real portal, and the results were read back and captured. Most of what it built held up under that loop on the first or second pass. One thing did not, and Part 3 is about that thing.

1.3 The HubSpot Developer MCP, for driving the platform

The app was built and deployed through the HubSpot Developer MCP, which put the developer platform inside the build loop rather than beside it. Uploading the project, inspecting a build, validating a component, and reading back what the platform actually created all happened without leaving the workflow.

ONE PLACE THE MCP PAID FOR ITSELF OUTRIGHT

Before the agent-tools beta was granted, the local CLI printed a refusal for the agent-tool component type. That local refusal answered the entitlement question without uploading anything. Component ids are permanent after upload, so confirming the gate locally, rather than by uploading speculative components to see whether they took, saved spending something irreversible to learn a yes-or-no fact.

The same loop surfaced most of the platform findings in Part 5. A component that validated clean and failed at upload, a schema that advertised a field the builder refused, a scope string that passed local checks and failed server-side: each was found because the build and the platform were close enough together to compare what was asked against what happened.

1.4 HubSpot APIs, for everything the app actually does

The running system consumes roughly sixty distinct HubSpot API endpoints across six product areas. The complete list lives in the companion dossier; the shape of it is worth seeing here, because the breadth is part of the build story.

Area	Representative endpoints	What FPI does with them
CRM	objects, properties, associations v4, batch, lists, owners	The object model, the idempotent write path, and the association graph that is also the security model
CMS	source-code, pages, hubdb, domains, graphql	Installs the theme, publishes the portal pages, and reads config at render time

Area	Representative endpoints	What FPI does with them
Marketing	emails clone, campaigns, communication preferences	Clones and adapts the proven asset on deploy, under a real subscription type
Automation	flows v4, action callbacks	Imports the native workflows and runs the two custom actions
OAuth	token	Per-portal token custody with automatic refresh
Events	timeline v4	App events, authored and held back (Part 2.1)

1.5 Emerjent, for the build memory

A build this size, run by one person over weeks, generates more hard-won knowledge than any one person can hold. Emerjent held it. It was the operational backbone of the whole effort, and using it to run the build was also the point: the platform proves itself by running the work it was made for.

What Emerjent held	Why it mattered to the build
Session episodes and decision records	Every working session that changed the architecture left a durable record, so a decision made in July was still legible in August
The platform field report	Each place the platform disagreed with its documentation was captured the moment it was found, so no landmine was ever stepped on twice
Masterclass captures	The moments where a human judgment call redirected the build were recorded as they happened, including the ones in Part 3
Time and the engagement record	The build ran on tracked time against a live deal, the same way any client work does

THE DIVISION OF LABOR, STATED PLAINLY

Claude was strongest where the work was reasoning about a decision with no clean answer, and weakest where it needed a fact only the live platform could give.

Claude Code was strongest where the work was building against a clear contract and exercising it, and weakest exactly where the platform behaved differently from its own documentation, because there was no correct contract to build against.

The human was strongest where the failure was invisible to every automated signal, which is precisely the situation Part 3 describes.

2 The four things that got built

The finished system is four components. This section is the build story of each: what it is, how it was made, and what fought back during the making. The order is the order of dependence, because the app objects had to exist before the backend could write to them, the backend before the agents could read, and the portal on top of all of it.

2.1 The HubSpot public app

Everything the platform sees is one app. It carries 28 components on developer platform 2026.03, and it is what an installing franchise brand actually installs.

Component type	Count	What they are
Application	1	The app itself, OAuth 2.0
App objects	4	Unit, Monthly Financials, Play, Play Assignment
App-object associations	8	The graph that is also the security model
CRM record cards	2	React cards on Unit and Play Assignment
Settings page	1	Brand config, autonomy, and the setup runner
Webhooks	1	Four subscriptions into the backend
Workflow actions	11	Two automation actions plus nine agent tools
App events (held back)	8	Authored in full, deliberately not in the build

The data model ships as code inside the app

The four app objects, their 84 properties, and their eight labelled associations are declared in the app and created by HubSpot on upload rather than provisioned per portal. This is the single most important build decision in the app, because it is what makes the governing rule true: built once, installed many times. The second franchise brand gets the entire data model the moment they install, with no per-portal object creation and no drift between installs.

It also forced a distribution choice. The specification originally wanted private distribution, specifically to stay out of the marketplace. That turned out to be impossible, because HubSpot marks app objects as available only for OAuth marketplace apps. The data model the product needs therefore forced marketplace distribution, unlisted, which permits 25 installs before a public listing is required. The capability matrix, not a preference, decided the architecture.

App events: authored in full, held out of the build on purpose

Eight app-event types are written, reviewed, and complete, and they are not in the build. The reason is precise enough to be worth stating, because it is a decision rather than a failure. The builder refuses a

field the schema advertises: calling for the app-event feature config returns a schema that defines `objectHandlingConfig` and does not mark it required, and yet the build fails on it, which is why validation passed and upload did not.

Component ids are permanent after upload. Uploading eight permanent ids configured for a resolution the platform cannot currently honour spends something irreversible to gain an event stream that would resolve its object by an unknown default. That trade is not worth making, so emission is suppressed at source behind a single flag, the call sites stay intact and reviewable, and a test asserts the flag and the held-back component set agree so they cannot drift. Restoring the feature is a two-line change on the day the builder supports the field.

Agent tools: nine, published

The agent-tools beta was granted on 4 August and all nine FPI tools were published: seven that read and two that take action. They are the contest's subject matter, and Part 2.3 covers what they let an agent do. What matters to the app build is that they are real, published components with live endpoints, reachable by an agent in the portal.

The Settings page is how the app deploys and provisions itself

The Settings page is a React UI extension with no storage of its own, which is the platform's intended shape: it calls its own backend through `hubspot.fetch`. It carries three panels, and the third is the one that does the heavy lifting on every install.

Panel	What it does
Brand configuration	Reads and writes the per-brand config row, with client-side validation mirroring the server
Autonomy	The autonomy dial. For the contest it is configuration rather than approval: the branch exists and always resolves to gated
Setup and health	The 12-item install checklist: a read-only verify, an async run, and per-item polled status

The setup runner behind that third panel is the deployment mechanism for everything the app cannot ship as code. It creates the standard-object properties app objects cannot carry, the config table, the access lists, the theme, the eight portal pages, and the seven workflows, in dependency order. It never deletes, never retypes a property, and never overwrites a value an administrator edited, so re-running it is how you repair a partial install rather than something to avoid.

2.2 The AWS Lambdas

Every piece of logic and every credential in the system lives in AWS: 46 Lambda functions behind one API Gateway HTTP API. The theme has no credentials, the cards have no database, and the agents write through the CRM, so if a behaviour is not in this service or in a native workflow, it does not exist. For a system with this many moving parts, that is a deliberately small surface.

Why the whole backend ended up in AWS, which was not the plan

The specification split the backend across three runtimes. All of it moved to AWS instead, for two reasons the platform imposed rather than a preference.

- 1. HubSpot excludes OAuth-authenticated apps from project serverless functions on 2026.03.** The app-function directory the specification described was never available to this app, so the cards and the Settings page call Lambda equivalents through `hubspot.fetch` instead.
- 2. HubL provably cannot compute an HMAC.** Its only hash filter is md5 and it cannot make outbound calls, so the specification's signed context token was not implementable in the theme. A small CMS serverless broker confirms the visitor is logged in and asks AWS to mint the token, so the signing key never leaves AWS.

The manifest is the source of truth and the shell scripts stay dumb

A single manifest file holds the declarative half of every function: name, route, handler, auth mode, timeout, memory, capabilities, and an optional trigger. Adding an endpoint is one entry there plus one handler file. The deploy scripts loop over it, and no function, role, or route is ever hand-edited.

Each function's IAM policy is generated from its declared capabilities, so a read-only card endpoint never receives a decrypt permission while the OAuth callback does. A capability string the library does not recognise warns and produces no policy, so the function deploys with a role that cannot do the thing and fails loudly at the right time rather than silently over-permissioned.

```
manifest entry → generated IAM role (least privilege)
                → Lambda function
                → API Gateway route
one entry, three artifacts, zero hand-editing
```

The deploy pipeline, where order is load-bearing

One root script is the only supported way to deploy, because four things must happen in one order and getting it wrong produces failures that look like something else entirely.

- 1. The HubSpot project uploads first,** because the app objects carry the properties the Lambda code writes. Deploy the service first and every write fails on a property that does not exist yet.
- 2. The schema cache is flushed,** because a property added in step one is invisible for up to an hour, and that window looks exactly like step one having failed.
- 3. The theme archive goes to S3,** because the function deploy bakes the theme version in and the setup worker fetches that exact key.

4. Then the service deploys: bootstrap, build, functions, API, verify.

It refuses to run on a dirty tree, runs the tests first, and requires a typed confirmation word before the irreversible half. Every individual deploy step is create-or-update, so re-running is the intended way to repair drift.

2.3 Agent Hub agents and workflows

This is where the contest's subject lives. The agent layer is nine published tools, a roster of Breeze agents that call them, and seven native workflows that orchestrate the record lifecycle around them.

A FINDING THAT RESHAPED THE ROSTER MID-BUILD

The original design called for one conversational agent that sequenced tools across a conversation. Building against the actual Agent Hub surface reversed that decision. **HubSpot Agent Hub agents are single-shot task runners, not conversational assistants.** A run is instructions plus typed inputs, executed once, with no user turn to read from and no second hop.

With no conversation to sequence across, agents are tasks rather than personas. The roster became one agent per job, which also happens to be better for the contest's per-agent activation math than one agent was.

Agent	Input	Tools it holds	Status
Book Scan	none	units-needing-attention	Live, verified
Location Briefing	unit_id	unit-summary, cohort-comparison, matching-plays	Live, verified
Draft Recommendation	unit_id, play_id	all four above plus draft-recommendation	In build

What the two live agents actually did

Book Scan returned 13 coached locations for a seeded consultant, correctly ranked, and quoted a two-day-old trajectory score as of its date rather than as a current figure. Location Briefing ran all three of its tools against one location, reported a suppressed benchmark as insufficient reporting rather than as poor performance, and reasoned from weak lead flow and steady conversion to a demand problem rather than a conversion problem, then picked the play whose target metric matched. That last step is real diagnostic work, and it is the thing the product is actually selling.

Instruction patterns that measurably worked

Each of these was written in response to an observed failure against real tool output, and each one fixed it. Agent instruction design is the least documented part of building on this platform, so they are worth recording.

Instruction	The failure it fixed
Run all three tools even if the first looks unremarkable	The agent short-circuited on a steady summary and skipped the cohort check, which is exactly where a comfortable-looking location turns out to be sliding

Instruction	The failure it fixed
Locations with no score are a separate state from a low score; never drop them	The agent silently dropped 9 of 13 units by reading "needs attention" as at-risk plus drifting, making the most neglected state invisible
Do not cite a bare figure as support for a play unless the data establishes it is a problem	The agent cited a 9.1% net margin as support for a margin play, with no benchmark to say whether 9.1% was good
Relay a refusal as given and stop; do not retry with different inputs	An agent that hits a permission refusal and rephrases is defeating the guard

THE WALL INSIDE THE AGENT LAYER: TOOLS YES, AGENTS NO

Tools can be created and published programmatically. **Agents cannot.** An agent's existence, its instructions, its inputs, and the binding of a tool to it are all manual, UI-only steps, verified adversarially: the agent feature config schema returns "unable to locate JSON schema", the published API catalogue contains the word agent zero times, and sixteen plausible endpoints all return 404 while a control returns 401.

The cost is per customer, per agent, per tool: roughly 10 to 15 discrete interface selections in every franchise client's account, plus pasting each agent's instructions. Under a "built once, installed many times" rule, that is the one part that cannot be engineered away. It becomes setup-guide work instead.

Seven native workflows carry the record lifecycle

Around the agents, seven native HubSpot workflows move records through their stages: financials intake routing, the Play Assignment lifecycle, drift alerting, membership provisioning, and three ownership-stamping flows that ship disabled because they restamp who can see what and a human enables them only after reading them back. Two custom workflow actions do the work no native construct expresses: deploying an approved play and stamping access across an association to an app object.

2.4 The Content Hub member portal

The portal is where a franchisee does the only work the system asks of them, which is submit a profit and loss statement, and get back a benchmark and a shortlist of interventions in the same interaction. That exchange is the whole product, so the portal carries more design weight than any other surface.

Layer	What was built
Theme	<code>franwise</code> 1.8.0, a flat self-contained fork with no inheritance key
Portal pages	8 templates: home, consultant cockpit, unit detail, my business, P&L submission, play review, executive dashboard, access denied
System templates	11 membership and system pages: login, register, reset, password prompt, subscription preferences, and the rest
Custom modules	30, of which 21 carry their own JavaScript
HubL macros	56 across 6 files, covering all 31 design-system components
Memberships	The whole franchisee experience. Contacts with membership passwords and no HubSpot seat, which is what makes opt-in possible at network scale
CMS serverless	3 endpoints: the member-auth broker, an allowlisted route proxy, and an identity echo
HubDB	<code>fpi_config</code> , one row per brand, chosen because it is natively readable by CMS templates at render time

The portal is also where the build hit its one genuinely unsolvable-by-machine wall. It is common for a theme to compile clean, upload clean, validate green, and still be wrong, because nothing in a HubSpot theme is exercised by a test and the validator compiles without executing. The logged-in render is the only thing that finds a whole class of defect, and it is the one branch neither tool covers. The next section is the story of the worst of them.

3 The wall: the bug the machine could not find

Every build has a moment where the tools that are doing the work run out of road. This build had exactly one, and it is worth telling in full, because it is the clearest illustration of where a human is still decisive and why.

It compiled, it uploaded, it validated, and it locked every single member out. No error, anywhere.

The shape of the problem

3.1 The symptom

Every membership-gated page in the portal denied every logged-in member. A franchisee who logged in correctly, with a valid session, reached nothing. There was no error, no log line, and nothing in the build, the upload, or the validation to catch it. By every signal the automated build could see, the portal was fine.

That combination is the trap. A defect that throws an error gets found, because the error is a thread to pull. A defect that denies access silently, while every tool reports green, has no thread. The build loop that had caught everything else, propose, build, exercise, read back, had nothing to read back, because the failure did not announce itself anywhere the loop was looking.

3.2 The cause: HubL does not compile in the order you would assume

The membership gate has to resolve at the base of the render, before any page renders its protected content. Claude Code built it the way the structure reads on the page: the check was placed on the individual page templates, where the protected content lives. That looks correct. It is not, because HubL does not compile a page in the order the template is written.

THE MECHANISM

HubL resolves a template in an order of its own, and blocks, includes, and page-level constants do not evaluate where a reading eye expects them to. A gate written into an individual page sits **too far down the render tree** to do its job: by the time it is reached, the render has already moved past the point where a clean denial was possible, so the gate does not deny the right members. It denies everyone.

Claude Code modelled the render order as top-to-bottom, the way the source reads. The platform does not work that way, and nothing in the toolchain says so. The template compiled, uploaded, and validated exactly as if the order it assumed were the order the platform used.

This is the heart of why the tools could not find it. Claude Code reasons from the contract it is given, and the contract, the template source plus the platform's documented behaviour, said the gate was in a valid place. The validator agreed, because the validator compiles without executing. The one authority that disagreed was the running system with a real member logged in, and that authority speaks to nobody unless a person goes and looks.

3.3 The fix was a human reading a live screen

The break in the loop came from logging in as a member and watching the actual render, rather than reading the template or trusting a green check. Seeing every valid session denied, against source that looked correct, pointed at the one thing source cannot show: the order the platform evaluates it in. Once the real compilation order was the thing in question rather than the template's correctness, the answer followed. The gate did not belong on the pages. It belonged at the base, where it resolves before page content, and it had to be structured so that HubL's resolution order could not strand it.

The gate now runs once, at the right level, in a fixed and known order. And because the arrangement that failed was so quiet, a structural check was added to make sure it cannot come back: an assertion that every guarded page routes through the base gate exactly once and that no page carries its own copy. The old placement cannot be silently reintroduced, because the test now fails if it is.

THE GENERAL LESSON, WHICH OUTLIVES THIS ONE BUG

The automated build loop has a blind spot, and it is a specific one: it sits exactly where the platform's runtime behaviour diverges from its compile-time behaviour. A tool that builds from a contract cannot see a place where the contract is wrong, and a validator that compiles without executing cannot see a defect that only appears at run time. HubL theming is dense with both.

Closing that blind spot did not take a better tool. It took a person looking at the running system with the right question in mind. That is not a limitation to apologise for. It is the correct division of labor: let the machine build at speed, and put the human exactly where the machine is blind.

4 How the build stayed honest

A solo build moving at the speed an AI toolchain allows can accumulate confidence faster than it accumulates correctness. A handful of disciplines kept the two in step, and each of them earned its place by catching something real.

4.1 Tests that were watched to fail

A guard test that has never been seen to fail proves nothing, because it may be asserting against the same wrong assumption the code makes. So the guard tests were run against deliberately broken builds and watched to fail alone, as they should, before being trusted. Two examples carry the point.

- **A drift test reads the live deployment contract at test time**, so a divergence between the service constants and what is actually deployed fails in continuous integration. It exists because a constant had already drifted to a property no portal has, silently breaking every read and write of a record's stage.
- **A pending-components test asserts both builder rules against everything in the app directory**, so re-adding the held-back app events before the platform supports them fails in CI rather than ten minutes into a deploy.

4.2 Live tests that fail closed, not open

The isolation and governance test plans run against a real portal and, in the governance case, actually write. Two properties make them trustworthy rather than reassuring.

1. **They report inconclusive rather than pass** for any check they could not evaluate. A test plan that fails open is the same class of defect as the guard it is testing, and the first drafts of both made exactly that mistake before it was caught.
2. **The writing tests refuse to run without an explicit confirm flag**, so a governance test can never fire a real approval by accident.

On the document date those plans stood at 24 isolation checks passed and 0 failed, and 18 governance-gate checks passed and 0 failed, including the hard case of a person who is both a consultant and an owner, reached by one walk first and still correctly gated as a review.

4.3 One rule did most of the filtering

Where a shortcut would work for one portal and not for two, it is treated as a defect rather than a shortcut.

The governing architectural rule

Almost every place the build could have taken a fast path that hardcoded something, that path was rejected by a single test: would this still be true for the second installed portal. Runtime schema resolution instead of hardcoded object ids, per-portal token custody instead of one shared credential, configuration in HubDB instead of in code, event occurrence ids composed from the portal id outward so two installs cannot collide. None of these was needed to make the demo work. All of them were needed to make the rule true, and the rule is what makes the product a product rather than a one-off.

4.4 Nothing was learned twice

The most quietly valuable discipline was capture. Every platform behaviour that disagreed with its documentation went into Emerjent the moment it was found, with the consequence and the workaround attached. Over the build that became a field report of more than forty entries. Its value is not the list. Its value is that no landmine was ever stepped on a second time, because the first person to hit it was also the last, and the record was there the next time the same corner of the platform came up.

THE HONESTY EXTENDS TO WHAT IS NOT DONE

A build record that reports only success is not a record. This one is paired with a companion dossier that lists, by name and severity, ten known-open items, five things not yet exercised against a live portal, and the full field report. Naming them is the point: a system this size with nothing open would be a system nobody had looked at closely, and the credibility of everything that is finished rests on being equally plain about what is not.

5 What the platform taught us

Building a vertical app on a platform this deep means discovering, repeatedly, that the platform and its documentation do not always agree. Each discovery below was found at build time and is handled in code. Collected here, they are the lessons most useful to anyone attempting a build of this shape, which is arguably the most transferable thing in this document, because it is the cost that appears in no estimate.

THE PRECEDENCE RULE THE BUILD RAN ON

The specifications outrank the prose. The build-time records outrank the specifications where they disagree. And where a document and the running code disagree, the code is right. Every lesson below is a place that rule earned its keep.

5.1 The app and object model

What we learned	What it meant for the build
Private distribution plus OAuth cannot carry app objects	The app ships marketplace-distributed and unlisted, because the data model required it
OAuth apps cannot ship project serverless functions on 2026.03	The entire backend moved to Lambda; cards and settings call it through <code>hubspot.fetch</code>
App-object properties are auto-prefixed with the app id at runtime	Property names are written un-prefixed in the repo and resolved at runtime, never hardcoded
A required agent-tool input field cannot be changed once uploaded	Every tool ships its inputs as not-required and enforces required-ness in the Lambda instead
<code>supportedValueTypes</code> is typed as an array but the builder demands exactly one	A permanent, one-time choice on all nine tools, made on incomplete information
<code>objectHandlingConfig</code> is defined in the app-event schema and refused by the builder	All eight app events held out of the build until the field is supported

5.2 HubL and the theme: the most expensive category by far

The single most costly lesson of the build is that a HubSpot theme is validated by compilation, not by execution, and nothing in a theme is exercised by a test. Every defect below compiled clean, uploaded clean, and validated green. The logged-in render is the only thing that finds them.

What we learned	What it meant for the build
HubL cannot compute an HMAC and cannot make outbound calls	A CMS serverless broker replaces the specification's in-template signed token

What we learned	What it meant for the build
HubL does not compile a page in written order	The membership gate had to move to the base and run before page content, or it denies everyone (Part 3)
<code>crm_object()</code> returns an enumeration's label, not its value	A role check comparing against the value passed server-side and silently failed template-side; every such check had to compare against the label
A child block whose name no parent declares is silently dropped	No error at compile, upload, or validate time; only the rendered page shows the gap
CSS comments do not nest and HubL does not strip them	A single stray comment terminator once swallowed the next rule and cost the portal its entire token file
<code>publicAccessRules</code> is write-only and reads back empty	Page privacy cannot be verified by reading it back, so the check fails closed rather than reporting a false green

5.3 CRM writes and automation

What we learned	What it meant for the build
A v4 association write validates the category and type id as a pair	A wrong category made every labelled association unwritable, with an error that named a type id that plainly exists, so it read as a bad label
A labels lookup made in the wrong direction returns undefined, not an error	The batch reader treated undefined as no filter, so a cascade had been reading every association a record has; found by tracing directions, not by a failing test
A HubDB draft PATCH deletes any column omitted from the request	Column edits must read the draft, carry every existing column forward, then append and publish
The v4 flows API publishes no contract for an action's fields	A workflow wired to a field name the action does not recognise is accepted, runs, and does nothing, silently; field names were learned by building in the UI and reading the flow back
An app's action type id belongs to the app, not the portal	Every portal-scoped discovery route dead-ended, so action-type resolution moved to deploy time with a developer credential

THE ONE SENTENCE TO TAKE FROM THIS SECTION

On a platform this capable, the gap between what the documentation says and what the runtime does is real, it is wide in a few specific places, and it is invisible to every tool that compiles rather than executes. Building here well means keeping a human pointed at the running system, and writing down every divergence the first time so it is never paid for twice.

A Appendix · The build at a glance

Every figure verified against the repository, AWS account 713881813219, and HubSpot build #15 on the document date.

Metric	Value
HubSpot app id	46836361
Developer platform version	2026.03
Developer test portal	51781063 (fpi-demo), installed since 24 July 2026
HubSpot project components in the build	28
Components held back (app events)	8, with a documented restore procedure
App objects	4 (Unit, Monthly Financials, Play, Play Assignment)
App-object properties	84
App-object associations	8, all integrator-defined
OAuth scopes	44
Agent tools	9 (7 read, 2 action), all published
Breeze agents	2 built and verified end to end, 1 in build
Native workflows	7
Custom workflow actions	2
Settings-page setup checklist items	12
AWS Lambda functions	46
API routes	42
DynamoDB tables / SQS queues incl. DLQs / secrets / alarms	3 / 4 / 3 / 5
Backend tests	719 across 36 files, all passing
HubSpot APIs consumed	~60 endpoints across 6 product areas
Theme	franwise 1.8.0
Portal page templates / system templates	8 / 11
Custom modules	30 (21 with their own JavaScript)
HubL macros	56 across 6 files
CMS serverless endpoints	3

Metric	Value
Theme inspection checks	52 passed, 0 failed
Live isolation tests	24 passed, 0 failed, 0 inconclusive
Live governance-gate tests	18 passed, 0 failed, 0 inconclusive
Platform divergences captured	more than 40, each with workaround
Bugs the toolchain could not find	1, found by a human reading a live render

A CLOSING NOTE

The finished parts are finished: an app installed and running, nine published agent tools, two agents doing real diagnostic work against live data, and more than 700 passing tests. The parts that are not are named and owned. And the one place the machine could not go, a person went. That last line is not a footnote to the build. It is the shape of it.