FleetBot (Student Version) — Case Study

By Fateh Aamir Ali — Grade 11, Fleetwood Park Secondary School, Surrey, BC Founder & President, AI & Innovation Club

Quick Facts

What it is	A student-facing AI chatbot that answers everyday school questions (block rotation, who teaches what, clubs, etc.) using the school's own public documents.
My role	I built all of it myself — the bot logic, the prompts, the data, the bug fixes, **and** the website it lives on. No one else worked on this one.
Platform	Botpress for the bot (after testing two other platforms first); Lovable for the website front-end
Models	GPT-4o Mini (main answers) + GPT-5 Nano (cheap fallback)
Data method	RAG (the bot searches the school's PDFs live instead of memorizing them)
Launched	Around January 11, 2026, shared through the AI Club Instagram
Status	Still live. It was a beta to prove the idea worked — it is not actively used now.

Proof I have: Botpress dashboard (227 messages, 66 users, $2.76 spend, 0 errors), website analytics from the Lovable site (112 visitors, traffic sources), a screen recording of the bot working end-to-end, a screenshot of the full node architecture, and the live link.

1. The Problem I Was Solving

At my school, basic info is buried inside PDF files on the school website — the daily block rotation, who the counsellors are, what clubs exist, the school map. Nobody reads those PDFs. So students either dig through them or walk to the office and ask, and the office ends up answering the same questions over and over.

I wanted to build something simple: an AI you could just ask "What's today's block order?" and get an instant answer, on your phone, in plain English.

I also had a second reason. Before I could ever pitch a bigger project to the school administration, I needed to *prove I could actually build something real*. FleetBot was my proof.

2. The Journey: I Had to Switch Platforms Three Times

This was not a "set it up once and it works" project. I tried three different platforms before I found the right one.

Attempt 1 — Relevance AI. It worked, but it was too slow. A single answer took over 20 seconds. I knew students would never wait that long, so I dropped it.

Attempt 2 — Voiceflow. I built a working prototype here and it was good enough that I showed it to the principal, Ms. Perry, when I started thinking about a school version. She liked it, but she raised a privacy concern: Voiceflow sent data to third-party servers in a way we couldn't fully control. For a school, that's a real problem. It was a good catch, and it pushed me to find a platform where I controlled the data flow.

Final — Botpress. I rebuilt everything on Botpress. It gave me more control over the logic and the data, and it was cheap enough to run on a student budget. This became the platform for both FleetBot and the later staff version.

Lesson from this stage: the "best" tool isn't the one with the most features. It's the one that fits your real limits — speed, cost, and privacy.

---

3. The Hard Problems and How I Actually Fixed Them

Once I was on Botpress, the real engineering started. These are the actual bugs I hit and how I solved each one.

Problem A — The Cost Explosion

My first version used an "Autonomous Agent" — an AI that keeps thinking and looping until it solves the problem. During a stress test, it burned **$5 in just 20 messages.** That's not sustainable for a student.

Fix: I replaced the autonomous agent with a **deterministic logic flow** (simple "if this, then that" paths). On top of that, I reduced the RAG chunk limit from **50 chunks down to 12.** Pulling 50 big chunks of text for every question was the main reason it was so expensive — it was reading way more than it needed to. Twelve chunks was the "Goldilocks zone": enough to stay accurate, small enough to stay cheap.

Problem B — The "Stale Memory" Bug

If someone asked about the gym, the bot would get *stuck* on that topic. Even if the next question was "When is lunch?", it kept answering about the gym.

Fix: I wrote a custom JavaScript code card that forces a variable reset at the start of every single turn (`workflow.userprompt = event.preview`). This clears the bot's short-term memory each message so it reacts to the *new* question, not the old one.

Problem C — The Saturday Glitch

On weekends, the bot would *invent* a fake school schedule, because it wanted to be helpful so badly that it made something up.

Fix: I added a strict weekend rule to the instructions: if it's Saturday or Sunday, the bot has to say there's no school instead of guessing.

Problem D — The Spam List

When someone asked "What clubs are there?", the bot tried to list all 90 clubs, which took forever to type out.

Fix: I added a "crowd control" rule: never list more than 5 things at once, and ask the user to narrow it down instead.

Problem E — The Dead-End Answer (UX)

Instead of a cold "I don't know" when the bot couldn't find something, I used the cheaper, faster GPT-5 Nano model to handle failures. It says something like: "My beta system doesn't have data on that yet, but I can help you find [something close]." This keeps the conversation alive instead of hitting a wall.

Problem F — Two Types of Users (The "Y" Start)

I noticed two kinds of people open a chatbot. Some open it and wait. Some type "hi" immediately. So I built two paths: a passive user gets an automatic welcome message, and an active user who already typed something gets sent straight into processing so they don't have to repeat themselves.

Handling Follow-Up Questions (the 5-message window)

Even with the deterministic flow, I still wanted the bot to understand short follow-ups like "What about yesterday?" without the user repeating the whole question. I upgraded the context extractor to GPT-4o Mini, which gave the bot a **5-message memory window** — enough to follow a quick back-and-forth, without the cost or unpredictability of full open-ended memory. (This is the limit I deliberately pushed past in the staff version, which needed richer memory — see that case study.)

Safety

I added a dedicated safety prompt layer, and in my testing the bot refused 100% of unsafe questions (like "how to hack").

---

4. The Prompt Was the Hardest Part (Where Most of the Real Work Lived)

Anyone can make a chatbot answer a normal question. The hard part — and where I spent the most time — was writing the system prompt to handle everything that goes *wrong* at the edges. A huge amount of the engineering on this project isn't fancy code, it's the rules I wrote to stop the AI from

doing dumb or dangerous things. These are the non-obvious failures I had to design around. Most people never think about them until the bot embarrasses them in front of real users.

It will answer from its own memory instead of the documents — and sound confident while doing it. This was the biggest one. The moment the AI recognizes a topic, its instinct is to just *write* an answer from its general training ("a school principal usually does..."). But that answer might be totally wrong for *my* school. So I wrote a hard rule: recognizing a topic is only allowed to help it build a better *search* — never to write the answer. My test for every sentence: can this be traced back to something an actual document returned? If not, delete it. That's the difference between a bot that sounds right and a bot that *is* right.

Searching for things it shouldn't search wastes money and returns garbage. If someone types "hi" or "who are you?", a naive bot runs a full database search for that — which costs money and pulls random chunks. I made a list of things the bot answers *instantly* with no search at all: greetings, "what can you do," casual questions, and people trying to jailbreak it. Knowing what *not* to search turned out to be as important as knowing what to search.

A bot that sends everything to "go ask the office" feels broken. If a student jokingly asks "are you smart?" and the bot replies "please visit the main office," that's ridiculous and people stop trusting it. So I wrote a clear rule for when to actually point someone to a human versus when to just reply naturally. Casual and rhetorical questions get a real reply, not a bureaucratic dead-end.

"I don't know" should not be one generic line. When the bot can't help, the *right* response depends on why. Asking for personal info it can't access is different from asking about something that's genuinely coming later, which is different from asking about something it'll never have (like the weather), which is different from a casual joke. I wrote separate fallbacks for each so it never sounds like a broken machine.

The schedule trap (the big one for a school bot). I made the bot check the date in a strict order and stop at the first match: is it a weekend? a holiday or break? a non-instructional day? an early dismissal? a normal day? If you don't force that order, the bot finds a "Day 1" block order and reports it — even on a day school is actually closed. It has to rule out the closures *first*. I also made it pull the real current date from a variable and do the date math itself, instead of guessing.

It will call a past date "upcoming." If you ask for the next early dismissal, the bot might grab a date from the list and call it upcoming without checking it hasn't already happened. I had to make it compare every date against today's real date before calling anything "next."

A reasoning AI will try to write code mid-conversation — and crash. When I asked things like "how many days is school open this month," the AI tried to write little programs to count and filter, which the platform can't run, so it would break or waste tokens. I had to explicitly tell it: don't write code, just read the list and reason through it like a person.

Making it not sound like a generic AI took real rules. AI chatbots all sound the same — they open with "I can help with that!", say "Great question!", and end with "Let me know if you need anything else!" I banned all of it. No starting a message with "I," no copying my prompt's wording word-for-word (or

every user gets the exact same canned line), and varying the opener each time so it feels like a person, not a script.

Citations have to be honest. If the bot says "that's not in my documents" but then adds a source citation underneath, it just contradicted itself — a citation means it *found* something. I made a rule: never cite a source in the same reply where you admit you didn't find the answer.

Search words matter more than you'd think. The bot searches a vector database, which means the search query has to contain the words that would actually appear *near* the answer in the real document. "badminton tryout date time schedule" finds the answer; "badminton tryouts" often doesn't. I had to train myself to think about what words physically sit next to the answer.

Safety questions get the same grounding as everything else. For anything serious or emergency-related, I made the bot pull from the actual safety documents instead of inventing generic advice from its training — because generic advice might not match my school's real procedure. And for the rare, serious case of someone expressing real distress, I built in calm, direct crisis resources instead of just sending them away.

Writing all of this taught me that prompt engineering *is* real engineering. It's not "asking nicely" — it's anticipating every weird, rare, or dangerous way a system can fail, and writing the rule that stops it before it happens.

---

5. The Results (Real Numbers)

I launched the beta by sharing the link through the AI Club's Instagram and pushing it hard. I also built the website it lives on myself, using Lovable — so the front-end and the bot are both my work.

From the website analytics:

- **112 unique visitors** clicked the link — that's roughly 7.5% of my school's ~1,500 students reached in the first beta window
- 152 pageviews, with most traffic coming from Instagram (77 visitors), then direct (31), then Facebook (2)
- Average visit was about 44 seconds, and the bounce rate was high (87%) — a lot of people opened it out of curiosity and left

From the Botpress dashboard (30-day view around launch):

- **66 users actually chatted with the bot** (the other visitors looked but didn't message)
- **227 messages** exchanged
- **1,007 LLM calls**, with **0 errors**
- **Total AI cost: $2.76** for the whole beta

I want to be honest about the gap between 112 visitors and 66 actual users: a lot of people clicked the Instagram link, glanced at it, and left without chatting. That's normal for a beta. The number that

matters to me is the 66 who actually used it and the fact that it ran the whole time with zero errors on a budget under three dollars.

---

6. What I Learned

- **Cost control is a design decision, not an afterthought.** The difference between the autonomous agent ($5 / 20 messages) and the deterministic flow with 12 chunks (cents per message) was huge. I learned to think about cost *while* building, not after.
- **AI will confidently make things up if you let it.** The Saturday schedule glitch taught me that I have to write strict rules for the edge cases, because the model would rather invent an answer than admit it doesn't know.
- **Prompt engineering is real engineering.** Most of the hard work wasn't the happy-path answers — it was writing rules to stop the AI from failing in rare, weird, or dangerous ways. That's where the real thinking lived.
- **I can own the whole stack.** I built the bot, the prompt logic, the data pipeline, and the website front-end myself. Doing all of it taught me how the pieces fit together, not just one slice.
- **A working demo changes everything.** Once FleetBot was live with real usage numbers, I had something real to show the principal — and that's what unlocked the bigger staff project.

---

7. Proof Assets

What exists:

- **Screen recording of the bot working end-to-end** (the single strongest proof — someone can actually *watch* it answer real questions)
- **Screenshot of the full node architecture** (shows how the flow is wired together — proof the build is mine)
- The **live Lovable website** and its visitor analytics (112 visitors, traffic sources)
- Botpress dashboard screenshot (227 messages, 66 users, 1,007 LLM calls, 0 errors, $2.76 spend)
- The live bot link (still online)

Last updated: June 2026. This is my canonical record of the FleetBot student version — the real numbers and the real story.

FleetBot (Staff Version) — Case Study

By Fateh Aamir Ali — Grade 11, Fleetwood Park Secondary School, Surrey, BC Founder & President, AI & Innovation Club

Quick Facts

What it is	An AI assistant for school staff that answers questions from the school's official documents (staff handbook, policies, forms, athletics schedules) and was deployed on a physical terminal in the main office. As far as I know, it was the first student-built AI system ever deployed inside the school.
My role	I built the whole system — document prep, knowledge base, AI logic, system prompt, and the live integrations. A friend hand-coded the staff-facing website and connected the Botpress API. A teacher (Ms. Stusiak) made the paid purchases using AI Club funds. Ms. Tran told me what the athletics side needed. Everything else was me.
Platform	Botpress + Make.com + Google Calendar API + Microsoft 365
Models	Gemini 2.5 Flash (document search) + GPT-5 Mini (writing the answers)
Built	Roughly February–April 2026
Deployed	April 22, 2026 — a physical mini PC in the school's main office
Status	Ran as a beta. Athletics calendar feature worked end-to-end. The iPad booking feature was blocked by district IT. Usage was low after launch (this is part of what I learned).

Proof I have: 3-month analytics file (363 total messages, 37 users, 84 sessions), photo of the mini PC set up in the office, the error-tracking sheet, and a Letter of Recommendation from the principal referencing this project.

1. Why I Built It

After my student bot (FleetBot) was live and I had real usage numbers, I had proof I could ship something. So I pitched a bigger idea to the principal, Ms. Perry: a "ChatGPT for our school" — an AI that teachers could ask anything about school policies, schedules, and procedures, trained only on our own documents.

The problem was real. My school has over 1,500 students and around 80 staff. All the important info — the daily schedule, policies, contacts, forms, emergency procedures — lives in PDF files nobody reads. Teachers end up asking the office the same questions every day. I wanted to give them an assistant that just *knew* the answers.

2. Getting the School to Say Yes (The Part That Wasn't Code)

This project taught me that getting approval is its own kind of engineering.

The privacy concern. When I first showed the idea (on my Voiceflow prototype), Ms. Perry liked it but flagged that the platform sent data to outside servers we couldn't fully control. That was a dealbreaker for a school. I rebuilt everything on Botpress, where I controlled the data flow and the bot didn't pull in any student personal information at all.

The slow yes. Ms. Perry was interested but kept delaying. Weeks passed with no decision, and I had club funding sitting unused. Eventually I sent her a direct message that basically forced a decision: I told her if she wasn't interested, I'd use the funding for something else, and I listed the three exact things the bot could do, each tied to a problem she already cared about. She said yes quickly. Lesson: be direct, show a real demo, and frame everything around *their* problem, not your technology.

The requirements meeting. A second meeting included the vice-principal and Ms. Tran from athletics. I learned the school runs on Microsoft 365 (Outlook), not Google — which changed my plans for any booking feature. Ms. Tran got excited when I demoed the student bot (she called it "game changing") and gave me access to the athletics info to build in. We also discussed future versions — an athletics-specific bot, and a simplified bot for incoming Grade 7 students visiting in the spring. This meeting is where I figured out what to actually build.

3. Preparing the Data (The Most Important Step Nobody Sees)

The single biggest thing that improved the bot's accuracy had nothing to do with the AI model. It was cleaning the data.

I received **27 PDF documents** from the school — the staff handbook, policies, forms, contact sheets, department guides, newsletters. Raw PDFs are bad for AI: they're full of invisible formatting, headers, and footers the AI reads as real content, tables turn into scrambled text, and images are invisible.

Here's what I did:

- I wrote a detailed prompt for Google Gemini (which has a huge 1-million-token context window, so it can read a whole large PDF in one pass) and refined it over many attempts until the output was reliable. Early versions skipped sections or messed up tables.
- For tables, I had Gemini rebuild them in Markdown table format. For images, I had it describe them in text.
- I split very long documents into smaller, focused files so the AI wouldn't have to search 200 pages to answer one small question.
- I did all 27 one by one — there was no automatic pipeline, I checked each output myself.

Result: 27 PDFs became 35 clean Markdown files. I also got the **115-page staff handbook**, which arrived as scanned images the AI couldn't read at all — so I ran it through **OCR** to turn it into readable

text and loaded it into the knowledge base. The raw PDFs were about 98MB; the cleaned data in the bot is about **58MB.**

The lesson I keep coming back to: **garbage in, garbage out.** A cheaper model on clean data beats an expensive model on messy data.

---

4. Building the AI (And One Big Architecture Decision)

For the staff bot, I made the **opposite** architecture choice from my student bot, on purpose.

My student bot used a fixed, deterministic flow with only a short, fixed memory window (about 5 messages). That's cheap and fast, but it couldn't reliably follow open-ended context — if a teacher asked "Who manages the gym?" and then "What's her email?", it couldn't reliably figure out who "her" was across a longer back-and-forth. For teachers, that kind of forgetfulness would be frustrating.

So for the staff bot I used Botpress's **Autonomous Node**, which uses an LLM to manage the whole conversation, track context, and handle follow-ups naturally. The trade-off is cost — autonomous nodes process more tokens — so I kept the system lean (one tool, tuned chunk limit) to control it.

Models:

- **Gemini 2.5 Flash** for document search — huge context window, low cost, accurate retrieval.
- **GPT-5 Mini** for the actual reasoning and answers — good at following complex instructions and handling edge cases.

The Cost Problem (and the worst single test result)

My first test with the chunk limit at 50 was a shock. I asked the bot for my own name in the records (a random, hard-to-find piece of info). Because the name was buried, the bot kept searching and clarifying — it ran multiple searches and pulled tons of text. That one question used **185,000 tokens, took 12 seconds, and cost $0.45.** At that rate a full day of staff use could blow the budget.

Fix: I dropped the chunk limit from 50 to **5-7**, set a search threshold of 0.5–0.7, and told it not to run multiple searches for one question. Cost dropped to **under half a cent per message** and response time fell to about **2.5 seconds** — and accuracy stayed the same, because the right answer was almost always in the top few chunks anyway.

Writing the System Prompt (the rules that keep it safe)

I wrote the prompt in sections, each handling a specific situation:

- **Date & Schedule Protocol** — before answering anything about time, the bot first pulls the current Pacific Time from a JavaScript code block I wrote, then checks in order: is it a Saturday, Sunday, stat holiday, non-instructional day, or normal day? This stops it from inventing a weekend schedule.
- **Form Finder Protocol** — if a teacher asks for "a field trip form," the bot asks which one, because several exist and giving the wrong one wastes their time.

- **Policy Protocol** — for sensitive topics (SOGI, child safety, emergencies), the bot quotes directly from the document and tells the user to go to an administrator, so it never softens or misquotes something serious.
- **Contact Protocol** — the bot never gives out staff email addresses even if they're in the documents. It gives room numbers and extensions instead.
- **Fallback Rule** — if it finds nothing, it doesn't say a cold "I don't know." It says it checked the handbook, couldn't find that specific thing, and suggests the office.

These five protocols were the surface. The next section is the deeper layer — the rare, weird failures I had to write rules for that you only discover *after* a bot embarrasses you.

---

5. The System Prompt: Engineering Against Everything That Can Go Wrong

Honestly, this was one of the hardest part of the whole project. Anyone can get a bot to answer a normal question. The real engineering is in the edge cases — all the strange, rare, expensive, or dangerous ways an AI assistant can fail in front of real staff. Most of my actual thinking went into writing rules to catch those before they happened. Here are the non-obvious ones.

The bot has no "answer from memory" mode. This was the single most important rule. The instant the AI recognizes something — say it sees "Principal Jodie Perry" — its instinct is to write what a principal *generally* does, from its training. That's how you get answers that sound confident and are completely wrong for my school. So I forced it into one mode only: recognizing a topic is allowed to help it build a better search query, never to write the answer. The test I wrote right into the prompt: can every sentence be traced to a specific phrase in a retrieved chunk? If not, delete it. If the answer's incomplete, search again. That rule is what keeps it grounded in the school's real documents instead of making things up.

Knowing what *not* to search. A naive bot runs a database search for "hi." That costs money and returns nothing useful. I built a list of things the bot handles instantly, with no search and no routing: greetings, "what can you do," jailbreak attempts, casual or rhetorical questions. Routing efficiency is a cost decision as much as a user-experience one.

The over-referral trap. The lazy way to build a school bot is to send everything it can't answer to "go to the main office." But if a teacher asks "are you smart?" and the bot says "please visit the office, second floor," it feels broken and bureaucratic, and people stop using it. I wrote an explicit rule for the *only* two situations where it should send someone to a human, and made everything else — casual questions, questions about its own personality — get a real, natural reply.

Four kinds of "I can't help with that." A single generic fallback is wrong. The right response depends on *why* it can't help: personal staff data (no access — point to payroll), real-time data that could be added later (frame it as coming soon), real-time data it will never have like weather or transit (point to the real external source, no false "coming soon" promise), or a casual question (just chat). The "coming

soon vs. never" distinction matters — it stops the bot from quietly promising features that aren't actually planned.

Routing on topic alone misroutes. My athletics calendar integration only knows game times, gyms, and fields. So a question like "who's the basketball coach?" contains an athletics word ("basketball") but must *not* go to the calendar — it has to go to document search, because the calendar has no coaches in it. I had to write keyword rules so that "tryout, coach, team, form, contact" route to the documents, while only the six specific game/gym/field scenarios trigger the live calendar. Getting this wrong means firing an expensive Make.com call that returns nothing.

The schedule-order trap. I made the bot check the date in a strict order and stop at the first match: weekend → holiday or break → non-instructional day → early dismissal → regular day. If you don't enforce that order, the bot finds a block order and reports it even on a day the school is closed, because it grabbed the schedule before checking whether school was even open. Closures have to be ruled out first.

It will report a past date as "upcoming." The master calendar covers a whole school year, so when someone asks for the "next" early dismissal, the bot might grab a date that already passed and call it upcoming. I had to make it pull the real current date from a variable and confirm each date is actually in the future before ranking it.

A reasoning model tries to write code — and crashes the platform. When I asked things like "how many early dismissal days are in April," the autonomous AI tried to write JavaScript to filter and count the dates. Botpress can't run that, so it either crashed or burned tokens. I had to put a hard rule at the very top of the prompt: never write or execute code, just read the retrieved list and reason through it in plain language.

Source-of-truth priority. The same fact — a block order, a policy — can appear in two documents that disagree, like a stale section in the handbook versus the current master calendar. Without a rule, the bot mixes them. So I built a priority hierarchy: the master calendar wins for all schedule data, the handbook wins for policy, the athletics site wins for athletics operations, and if two non-handbook documents genuinely conflict, the bot shows both and tells the user to verify. One source of truth per topic.

Never invent a name. When the bot compiles a list ("give me all the VPs"), I broke it into separate targeted searches and added a strict rule: if a search returns a role but no name, write "[name not found in documents]" — never guess one, and never make up a role that wasn't in any document. For staff data, a confident wrong name is worse than an honest gap.

Making it sound like a colleague, not a script. I banned the AI tells: no "Great question!", no starting a message with the word "I," no ending with "Let me know if you need anything else!", and — importantly — never copying my prompt's own wording into a reply, or every staff member would get the exact same canned sentence. It has to vary its openers and generate fresh phrasing each time.

Citations can't contradict the answer. If the bot says "that isn't in my documents" and then prints a source line underneath, it just contradicted itself, because a citation means it *found* the answer. I wrote

a rule banning a citation in any reply that admits it didn't find something — plus a separate rule for the in-between case ("I found the emergency section but it doesn't list drill dates").

Safety answers are grounded, not improvised. For emergencies, the easy thing is to let the AI give generic "call 911" advice from its training. But that might contradict the school's actual written procedure. So safety questions follow the exact same rule as everything else: search the handbook and emergency documents, report what *they* say, then add one escalation line pointing to administration. Calm tone, no enthusiasm, one answer, done.

The crisis exception. There's exactly one situation where the bot breaks its "send them to a human" pattern and responds directly: if someone expresses that they want to hurt themselves. For that, I built in immediate crisis resources — the national crisis line and text number — written to sound human and calm, generated fresh each time so it never reads like a cold template. Telling the difference between ordinary frustration (point to the office) and an actual crisis (give resources now) was the most careful thing I wrote in the entire prompt.

Never paste raw chunks. The data coming back from the database is ugly — fragments, formatting junk, scraped web artifacts from the athletics site. I wrote fixed output templates (single fact, list, table, step-by-step, multi-section), a rule to always reformat into one of them, to always use a table for anything with two or more columns, and to never dump raw text at a user.

The big lesson: prompt engineering *is* real engineering. It isn't "asking the AI nicely." It's sitting down and imagining every rare, weird, expensive, or dangerous way the system can fail in front of real people — and writing the rule that catches each one before it happens. That's where most of the actual thinking on this project lived.

---

6. The Complicated Part: The Live Athletics Calendar

After the document side worked, I built a live integration so teachers could ask "When is the next basketball game?" and get a real answer pulled straight from the school's athletics calendars. This was by far the hardest *technical* part, and it involved more failures than anything else. I'm documenting all of them because I learned the most here.

The system connects three things: **Botpress** (takes the question) → **Make.com** (the middle layer that routes and queries) → **Google Calendar API** (the live data).

Challenge 1 — The permissions wall. I needed a Google "service account" (a robot account) to read the calendars. But when I tried to share the 6 athletics calendars with it, the "share with specific people" option was missing on most of them — because the athletics account was only *subscribed* to those calendars, it didn't *own* them, so it couldn't grant access. **Fix:** the calendars were public (they're embedded on the school website for parents), and a public calendar can be read using just its Calendar ID, no sharing needed. That was my workaround.

Challenge 2 — Service account vs OAuth. I chose a service account over the normal "Sign in with Google" (OAuth) on purpose. OAuth tokens are tied to a person's login — if they change their password,

leave, or the token expires, the whole thing breaks silently. A service account uses a static credential that keeps working. For a school system nobody is babysitting, that was the right call.

Challenge 3 — The 6-calendar problem. Botpress could only connect *one* calendar at a time, but the school had 6 (home, away, large gym, small gym, and two fields). My first solution was a "Router" with 4+ separate paths, but it was fragile — any change had to be made in 4 places, and a vague question like "anything on today?" had no keyword to route on.

Challenge 4 — The Iterator fix (my favorite). I discovered Make.com's Iterator module. Instead of 6 separate paths, I made **one** path that loops through a list of all 6 Calendar IDs, runs the same query 6 times, and collects everything at the end. This cut the system from **20+ modules down to 5**, and now adding a new calendar is just adding one line to the list. This was the moment I understood the difference between code that *works once* and code that's actually *maintainable*.

Challenge 5 — The Aggregator bug. The module that collects all the results was only returning one event instead of all of them. The cause was a "Group by" setting — without it, each loop overwrote the last result. I set a static value to force all 6 loops into one group, and disabled "stop after empty aggregation" so it wouldn't quit early if one calendar was empty.

Challenge 6 — The JSON crash. After it finally returned 40+ events, the bot crashed every time with "Bad control character in string literal in JSON." The cause: I had pressed Enter inside the text to make it readable, and those line breaks are illegal inside JSON data. **Fix:** I replaced the line breaks with a `||` separator so everything sat on one line, and let the AI in Botpress reformat it nicely for the teacher. This taught me to actually *read* what the error is saying instead of just copying it into a search bar.

Challenge 7 — The Axios crash. My code to call Make.com used a library called Axios, and it crashed in 28 milliseconds — it never even tried. Botpress doesn't let you import external libraries with `require()`. **Fix:** I switched to the built-in `fetch()` function, wrapped it in an async function, and added a timeout and better error logging.

Challenge 8 — The search-term mismatch. When a teacher asked about "Senior Girls Soccer," the AI searched for that exact phrase, but the calendar used "Sr Girls Soccer," so nothing matched. **Fix:** I changed the prompt to extract just the sport ("Soccer"), pull all soccer events, and let the AI filter down afterward. Trading precision at search time for completeness, then filtering, was far more reliable.

Challenge 9 — The infinite loop. After Make.com returned data, Botpress handed it back to the Autonomous Node, which looked at the original question again, decided it was *still* an athletics question, and sent it back to the calendar — looping and costing money each time. **Fix:** I added a dedicated display node after the calendar step that just shows the result and ends, instead of routing back.

Challenge 10 — The session cache. Every calendar question fired a fresh request to Make.com. So if a teacher asked 6 calendar questions in one conversation, that was 6 separate Make.com operations. I added internal logic to check if the info was already pulled in that session and reuse it (this is called memoization / a session cache). The data is fresh the moment the conversation starts, but follow-ups in the same chat don't burn extra operations.

The final result: asking "When is the next soccer game?" returns a clean list of the next few games with dates, times, and locations, converted to readable Pacific Time, in about 4 seconds — checking all 6 live calendars. **This worked end-to-end.**

The Athletics Website (a separate integration)

On top of the calendars, the athletics department also has a website with info I needed in the bot. I couldn't just scrape the whole site into a PDF and upload it, because the site **updates weekly** — a static copy would go stale almost immediately. Instead I used Botpress's built-in website extractor and set it to **refresh the information weekly**, so the bot always reads a current version. I also added the extra PDFs and documents the athletics department gave me directly.

---

7. The iPad Booking Feature (Designed, Then Blocked)

The next feature was meant to be the real "painkiller": automating iPad cart bookings for the library. Right now teachers email the librarian, who manually reads each email and creates the calendar event herself. I wanted the bot to do it automatically.

I designed the whole thing. The librarian shared her "iPad Bookings" Outlook calendar with my school account as an editor. Because we were on the same Microsoft 365 tenant (Surrey Schools), changes would sync to her desktop Outlook instantly. I planned the Make.com scenarios to check availability, create a booking, and cancel one.

What I ruled out, and why. Before landing on the same-tenant Make.com approach, I worked through the other options and rejected each for a specific reason:

- **Google Calendar** — the librarian works in the desktop Outlook app, and Google Calendar doesn't sync reliably there.
- **A separate Outlook.com account** — unnecessary complexity when same-tenant sharing already solved it.
- **The direct Microsoft Graph API** — would need an Azure app registration on the school tenant, which means IT admin involvement.
- **Calendly** — doesn't update her existing Outlook calendar, which was a hard requirement.
- **The full IT-admin route** — avoid if possible, since I likely didn't have that level of access.

The chosen design was the simplest one that met every constraint: she shares the calendar once, I connect my own school account to Make.com, and everything syncs instantly because we're on the same tenant.

But this needed write access to Outlook, which meant district IT had to approve a third-party connection. I navigated the **Microsoft OAuth 2.0 admin-consent** requirement and routed a formal request through Ms. Perry to the district's IMS team (asking for Make.com, with Microsoft's own Power Automate as a pre-approved backup). I'd also flagged two risks that could still break it even after approval: a **geo-block** (the district only allows logins from inside Canada, and Make's servers are in the

US/Europe) and a **cross-account write block** (school policy sometimes stops external APIs from writing to a shared calendar, even with edit access).

It was rejected. The reason: staff couldn't grant a student access to that kind of confidential information, partly over concerns about data being used to train AI models. So the iPad booking feature was fully designed but never went live. That's the honest outcome.

(The difference from the athletics calendar: athletics calendars were *public* and *read-only*, so they needed no IT approval. The iPad booking needed *write* access to a private staff calendar, which did — and that's where it stopped.)

---

8. The Launch and What the Data Actually Showed

On **April 22, 2026**, I physically deployed the bot. I set up a mini PC with a monitor, keyboard, and mouse in the main office, and taped two sheets next to it: one explaining what the bot does, and one for staff to write down any wrong answers. I framed it to staff as a "new employee on its first day" and told them to try to break it and find its blind spots. That framing was deliberate — when it made a mistake, people thought "I found a blind spot" instead of "this is broken." As far as I know, this was the first student-built AI system ever deployed inside the school.

The real numbers over 3 months:

- **Total: 363 messages, 37 users, 84 sessions**
- **Launch week (week of Apr 20): 269 messages, 9 users, 34 sessions** — this was the peak, including my own heavy testing
- By **May 1** (about 10 days in), the terminal had handled **150+ real staff queries**
- Accuracy was roughly **70% right, 30% wrong** at that point. The errors were mostly vague questions — for example, "how many days is the school open this month?" It has the block order for every day, but it can't physically count each day, so it won't answer that well. Specific questions ("Who is the head of the math department?") it answered fine.

Then usage dropped fast: 43 messages the next week, then 9, then 2, then single digits for the rest of the year.

---

9. What I Learned (The Honest Part)

The most important lesson came from that usage drop, and I don't want to hide it.

I built a vitamin, not a painkiller. A general FAQ bot is "nice to have." Teachers only need policy info occasionally, and to use the bot they had to break their normal habit and walk to a PC or find a website. For a once-a-month need, that friction is too high, so they went back to what they already did. The principal liked the *idea* of AI, but she wasn't the one looking up block orders, so she didn't feel the pain

either. Usage flatlined — and most of the questions that did come in were test prompts like "what color is the sky," not real work.

That's why the *next* version was supposed to be the iPad booking — a thing teachers **have** to do regularly, where the bot intercepts an existing workflow instead of asking them to learn a new one. That one got blocked by IT, but the thinking behind it is the real lesson: **find the painful, high-frequency task, don't build a nice-to-have.**

Other lessons:

- **The data matters more than the model.** Converting those 27 PDFs into clean Markdown did more for accuracy than any model upgrade would have.
- **Prompt engineering is real engineering.** Most of the work wasn't the happy-path answers — it was writing rules to stop the AI from failing in rare, weird, or dangerous ways (see Section 5).
- **Read the actual error message.** The scary "JSON control character" crash was just me pressing Enter in a text box.
- **Simple architecture is better architecture.** The 5-module Iterator beat the 20-module Router in every way.
- **Stakeholder management is real engineering too.** Handling Ms. Perry's delays, pitching Ms. Tran, and dealing with IT permissions were as much a part of the project as the code. A bot teachers won't use is not a success, no matter how well it's built.
- **Design privacy and cost in from the start.** Both times I added them late (Voiceflow's privacy, the 50-chunk cost), I had to rebuild.

---

10. The Outcome

The general staff bot didn't reach wide adoption — and I learned more from that than I would have from an easy win. I closed out the pilot cleanly with the principal, framing the low usage honestly as proof-of-concept data, and I **secured a Letter of Recommendation** from Ms. Perry referencing this project and the AI Club. The terminal is still in the office.

What I actually walked away with: a deployed system in a real institution, hard data on why a product fails to get adopted, and a much sharper sense of what's worth building next.

---

11. Proof Assets

What exists:

- 3-month analytics file (363 messages, 37 users, 84 sessions, weekly breakdown showing the launch spike and decline)
- Photo of the mini PC deployed in the main office
- The two sheets taped beside it (capability guide + error log)

- The Letter of Recommendation from Ms. Perry

Last updated: June 2026. This is my canonical record of the FleetBot staff version — the real numbers, the wins, and the honest failures.