FleetBot (Staff): An AI Assistant Deployed Inside a 1,500-Student School

An AI assistant for school staff that answers questions from the school's own official documents, covering policies, forms, schedules, and live athletics calendars, running on a physical terminal in the main office.

Role: Sole builder, covering the data pipeline, knowledge base, AI logic, system prompt, and live integrations **Timeline:** Feb–Apr 2026 · deployed April 22, 2026 **Stack:** Botpress (Autonomous Node) · Make.com · Google Calendar API · Microsoft 365 · Gemini 2.5 Flash · GPT-5 Mini

- Cut a worst-case query from **$0.45** (185,000 tokens, 12 seconds) to **under half a cent per message** at **~2.5 seconds**, with no loss of accuracy, by re-architecting retrieval.
- Built a live integration across all **6** school athletics calendars, collapsing a fragile **20+ module** Make.com workflow to **5** with an Iterator loop; end-to-end answers in **~4 seconds**.
- As far as I know, the first student-built AI system deployed inside the school; handled **150+ real staff queries** in its first **10 days** on a physical office terminal.

The Problem

Fleetwood Park Secondary has over 1,500 students and around 80 staff, and its core information, including daily schedules, policies, contacts, forms and emergency procedures, lives in PDF files nobody reads. Staff repeatedly ask the front office the same questions because there is no fast way to get an accurate, school-specific answer on their own. Success meant a staff member could ask a plain-English question and get an answer pulled straight from the school's own documents, in seconds.

My Role

Built and owned the full system: the document pipeline, knowledge base, AI logic, system prompt, and the live Make.com and Google Calendar integration. Supporting contributions: a friend hand-coded the staff-facing website and connected the Botpress API; Ms. Stusiak (AI Club sponsor) made the paid subscriptions with club funds; Ms. Tran (athletics) specified the athletics requirements.

Key Decisions

1. Getting the school to yes, and letting requirements reset the architecture

The challenge: A school won't run a student's AI unless the data story is airtight, and the principal kept delaying the decision while club funding sat unused. On top of that, I was about to build integrations without confirming what infrastructure the school actually ran on.

Options considered: Push the already-built Voiceflow prototype through. It worked, but it sent data to third-party servers the school couldn't control, which was a dealbreaker for the principal. Wait for the principal to decide on her own timeline, which risked losing the funding window and the momentum. Start building on the assumption the school ran Google, which risked engineering the wrong integration entirely.

What I chose and why: Rebuilt on Botpress to control the data flow, which killed the privacy objection outright. Then sent the principal a direct message that forced a yes or no: three concrete things the bot would do, each tied to a problem she already had, plus a note that I'd otherwise redirect the funding. She agreed quickly. Before writing any integration code, I ran a requirements meeting with the vice-principal and the athletics lead, which surfaced that the school runs **Microsoft 365 (Outlook), not Google**. That one fact reset every downstream integration decision. It's why the athletics side used Google *public* calendars and why the later booking feature needed Microsoft OAuth. The tradeoff was weeks of non-engineering work up front, but it's what turned a demo into an approved, deployed system.

2. Autonomous Node over a deterministic flow

The challenge: Staff ask follow-ups that only make sense in context, like "Who manages the gym?" and then "What's her email?" My earlier student bot ran a deterministic flow with a fixed 5-message window and couldn't reliably resolve who "her" was across a longer exchange.

Options considered: Keep the cheap deterministic flow, which is predictable but breaks on open-ended context. Or use Botpress's Autonomous Node, which lets an LLM manage the whole conversation, handling context but processing more tokens and costing more.

What I chose and why: The Autonomous Node, then controlled the cost downside by keeping the system lean, with one tool and a tuned chunk limit. The tradeoff was paying more per message for reliable context, then clawing that cost back through retrieval tuning (below).

3. Retrieval tuning to bring the cost back down

The challenge: At a chunk limit of 50, one query for a deliberately buried name ran multiple searches, consumed **185,000 tokens**, took **12 seconds**, and cost **$0.45**. At that rate, a single day of staff use could blow the budget.

Options considered: A bigger budget, which wasn't real on club funds. A cheaper model, which meant an accuracy hit on hard questions. Or shrink how much text the bot pulls per question.

What I chose and why: Dropped the chunk limit to **5–7**, set a search threshold of 0.5–0.7, and forbade multiple searches for one question. Cost fell to **under half a cent per message** and response time to **~2.5 seconds**, with no accuracy loss, because the right answer was almost always in the top few chunks. The tradeoff was slightly lower recall on deeply buried facts, which never showed up in real accuracy.

4. Clean the data instead of uploading raw PDFs

The challenge: The 27 school PDFs were full of headers, footers, scrambled tables, and invisible images that a RAG system reads as real content. The 115-page staff handbook arrived as scanned images the AI couldn't read at all.

Options considered: Upload the raw PDFs, which is fast but low accuracy. Build an automatic conversion pipeline, which is faster but leaves the output unverified. Or convert each document by hand, which is slow but fully controlled.

What I chose and why: Manual conversion of all **27 PDFs into 35 clean Markdown files** using a refined Gemini prompt (1M-token context, tables rebuilt as Markdown, images described in text), splitting long documents into focused files, and running the scanned handbook through OCR. Raw PDFs were ~98MB; the cleaned knowledge base is ~58MB. This single step improved accuracy more than any model change would have.

5. One Iterator loop instead of a 6-path Router for the athletics calendars

The challenge: Teachers want live answers ("When's the next soccer game?"), but Botpress connects only one calendar and the school runs six (home, away, two gyms, two fields). The athletics account was only *subscribed* to those calendars, so it couldn't grant a service account access, and the reasoning model kept looping and re-firing paid calls.

Options considered: A Router with 4+ parallel paths, which is fragile, since every change had to be made in four places and vague questions had no keyword to route on. Or a separate manual path per calendar, which is unmaintainable.

What I chose and why: One Make.com **Iterator** that loops through all six Calendar IDs and aggregates the results, taking **20+ modules down to 5**, so adding a calendar is now one line. I read the *public* calendars by Calendar ID (no sharing needed), used a static service-account credential over OAuth (it won't silently break when someone changes a password), added a session cache so follow-ups don't re-hit Make.com, and ended the flow at a dedicated display node to stop the infinite loop. End-to-end answers land in ~4 seconds across all six calendars.

6. A system prompt that refuses to answer from memory

The challenge: The instant the model recognizes a topic, say "Principal Jodie Perry," it wants to answer from its training, which sounds confident and is often wrong for this specific school.

Options considered: Let it blend training knowledge with retrieved text, which is fluent but unreliable. Or force strict grounding, which is occasionally terse but always traceable.

What I chose and why: One mode only, where recognition is allowed to build a better search query but never to write the answer, and every sentence must trace back to a retrieved chunk or get deleted. Layered on top: typed fallbacks that respond differently to personal-data, coming-soon, never-coming, and casual questions; a rule against routing casual questions to the office; a source-of-truth hierarchy (master calendar wins for schedules, handbook for policy); a "never invent a name" placeholder rule for staff lists; grounded safety answers pulled from the school's own emergency documents; and a single crisis carve-out that returns crisis-line resources directly instead of a referral.

The Solution

A document-grounded assistant that answers policy, schedule, form, and contact questions from 35 cleaned Markdown files plus the OCR'd handbook, with a citation line on every document-based answer. A live athletics layer checks all six calendars and returns the next games with dates, times, and locations in readable Pacific Time (see: end-to-end screen recording).

Deployed as a physical terminal, a mini PC with monitor, keyboard, and mouse in the main office, with a capability sheet and an error-log sheet taped beside it. It was introduced to staff as a "new employee on its first day," so mistakes read as blind spots to report rather than a system that was broken (see: deployment photo, error-tracking sheet).

Results & Impact

Launch week (week of April 20, 2026) drew **269 messages, 9 users, 34 sessions**, including heavy self-testing. In its first 10 days the terminal handled **150+ real staff queries** at roughly **70% right / 30% wrong**, with the errors clustering on vague count-style questions the retrieval design can't answer, like "how many days is school open this month?", while specific questions such as "Who is the head of the math department?" answered cleanly. Over three months it logged **363 messages, 37 users, 84 sessions**, with usage dropping sharply after launch week: 43, then 9, then 2, then single digits (see: 3-month analytics file).

That decline was the most useful result. The product was a vitamin, not a painkiller: staff needed policy info only occasionally and had to leave their normal workflow to walk to a terminal, so usage flatlined. The broad-FAQ scope was a bet made before any usage data existed; the data corrected it, pointing the next build at a painful, high-frequency task that intercepts an existing workflow rather than adding a new one.

That next build, automated iPad-cart booking, was fully designed (same-tenant Outlook calendar sharing, Make.com scenarios for check, create and cancel, and a navigated Microsoft OAuth 2.0 admin-consent request routed through the principal to district IT) but blocked at the approval stage: a student couldn't be granted write access to confidential staff data. The athletics calendars had shipped because they were public and read-only; the booking feature needed write access to a private staff calendar, which was the line IT wouldn't cross. The pilot closed with a Letter of Recommendation from the principal referencing the project.

Two patterns held across the build. Both times I added privacy or cost late, in the Voiceflow rebuild over the data objection and in the 50-chunk cost blowup, I paid for it with a rebuild, so the lesson was to design both in from the start. And the moments that decided whether the system shipped at all weren't code: the privacy objection, the stalled approval, and the IT rejection were engineering problems in everything but name.

What Came Next

The usage data redirected the roadmap from broad FAQ coverage toward high-frequency workflows a bot can intercept, the through-line into everything I built next: go where the painful, repeated work already is.