aimatic.works

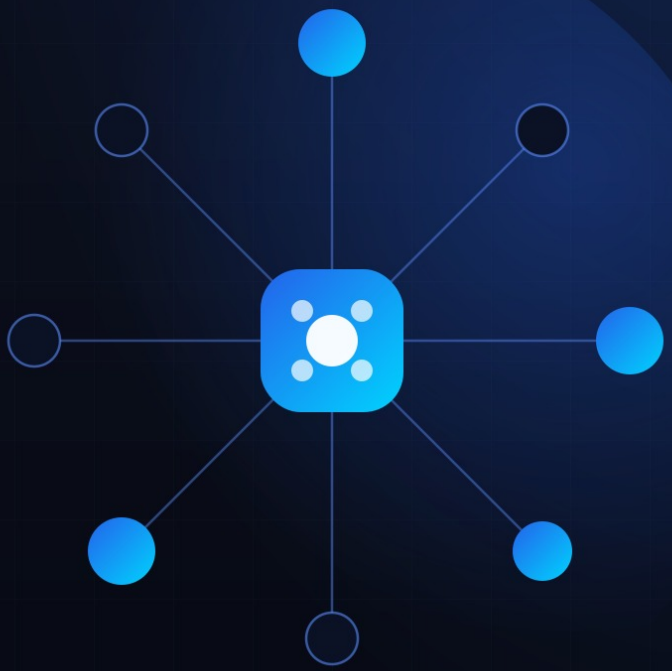
EBOOK · PRAKTYCZNY PRZEWODNIK

Automatyzacja biznesu z AI

15 narzędzi · 15 case studies

od kodu do produkcji — bez teorii

Realne wdrożenia | Kod i liczby | Co bym zrobił inaczej



aimatic.works

Automatyzacje i agenci AI · Projekt 180 Dni

Spis treści

Wstęp — Jak 60 zapisów do newslettera zmieniło mój sposób pracy

Był zwykły dzień roboczy w maju 2024 roku. Cashify zorganizowało jeden z pierwszych eventów — ja pomagałem przy organizacji, między innymi obsługiwałem formularz newslettera na naszej stronie cashify.eu. Strona działała na Webflow, formularzy mieliśmy wtedy trzy albo cztery: newsletter, kontakt, zapytania klientów, coś jeszcze.

Wszystkie lądowały na jednej skrzynce. Razem.

Tego dnia, w ciągu kilku godzin, zapisało się ponad sześćdziesiąt osób.

Nie miałem żadnego systemu. Wchodziłem na skrzynkę, otwierałem kolejnego maila, sprawdzałem czy to newsletter czy pytanie klienta, ręcznie kopiowałem adres email do Mailchimpa, odpisywałem, zamykałem, następny. Przez dwa dni. Wieczorami myślałem o tym stosie maili z poczuciem, że muszę coś z tym zrobić, bo tak nie można.

Wtedy pierwszy raz w życiu usiadłem do Make.com.

Kim jestem i dlaczego napisałem tę książkę

Nie jestem programistą. Nie studiowałem informatyki. Zanim zacząłem budować automatyzacje, moim narzędziem pracy był Excel, Webflow i telefon.

W Cashify dołączyłem gdy firma miała dziesięć, może dwanaście kantorów krypto. Kilka tygodni po moim przyjściu zostałem z dnia na dzień Project Managerem całego projektu kryptomatów — wtedy mieliśmy dwadzieścia ATM. Załadunek, rozładunek, serwis, obsługa klienta, pilnowanie żeby transakcje wychodziły, koordynacja między działami. Równolegle dostałem pod opiekę stronę cashify.eu — uczyłem się Webflow żeby nie angażować agencji przy każdej drobnej zmianie.

Byłem PM-em, który pilnował zbyt wielu rzeczy naraz. Tak jak wielu z was.

Te sześćdziesiąt zapisów było kroplą. Ale nauczyło mnie czegoś, czego żaden kurs mi nie dał: **ból jest lepszym nauczycielem niż teoria**.

Dwa dni przy skrzynce i jedno popołudnie z YouTubem

Obejrzałem kilka filmów o Make.com. Nie kurs, nie certyfikat — filmy na YouTube, jeden po drugim. Potem otworzyłem aplikację i zacząłem łączyć klocki.

Wynikowy scenariusz miał docelowo trzydzieści siedem modułów.

Jeden trigger Webflow obsługiwał wszystkie formularze naraz. Każde zgłoszenie trafiało automatycznie do właściwej skrzynki odpowiedzialnej osoby. Klient dostawał maila

potwierdzającego w ciągu sekund — treść zależna od tego, co wypełnił. Jeśli zaznaczył zgody marketingowe, automatycznie lądował w Mailchimpie. Żadnego ręcznego kopiowania.

Zajęło mi jedno popołudnie. Efekt działał przez następne miesiące bez mojej interwencji.

Nie byłem wtedy „człowiekiem od AI". Byłem PM-em który miał dość.

Jak to się rozwijało

Po Make przyszły kolejne problemy. Dla każdego znalazłem automatyzację.

Gdy Cashify zaczęło oferować rozliczenia podatkowe klientom, pracownik musiał ręcznie — przy każdym zgłoszeniu — wysłać potwierdzenie klientowi, przekazać dane do biura rachunkowego, wysłać umowę i zalogować leada. Cztery czynności, każdy klient, bez wyjątku. Zbudowałem scenariusz który robił to wszystko automatycznie po wypełnieniu formularza. Pracownik przestał być przekaźnikiem.

Gdy ruszyło Kryptopogotowie i zewnętrzna współpracowniczka nie zdążała odpowiadać na maile, stworzyłem dla niej kanał Telegram i bota który pingował ją natychmiast po każdym zgłoszeniu. Zmiana kanału powiadomień. Klient przestał czekać.

Gdy system wewnętrzny wysyłał automatyczne maile o transakcjach wymagających zgłoszenia do organu regulacyjnego, zrobiłem alert na Telegram — numer transakcji, data, bez danych osobowych. Zero przeoczonych zgłoszeń. To nie była już wygoda — to było zarządzanie ryzykiem prawnym firmy.

Każda automatyzacja rodziła się z bólu. Żadna nie była z góry zaplanowana.

Kiedy po raz pierwszy się nie udało

Równolegle próbowałem czegoś własnego. Kanał TikTok „Nietypowe Święta" — każdego dnia inne nieznane święto z całego świata. Miałem arkusz z promptami, Make wysyłał je do Gemini, Gemini generował wideo, wideo lądowało na Drive. Technologia działała.

Kanał umarł śmiercią naturalną po kilku filmach.

Nie dlatego że coś nie działało technicznie. Dlatego że nie miałem systemu który zmuszałby mnie do codziennej publikacji. Wąskim gardłem nie była automatyzacja — byłem nim ja.

Ta porażka nauczyła mnie czegoś ważnego: **automatyzacja rozwiązuje powtarzalne procesy. Nie zastąpi nawyku i konsekwencji.**

Projekt 180 Dni i ta książka

Na początku 2026 roku mój kolega Janek wpadł na pomysł aplikacji dla seniorów. Senior klika „żyję" raz na dobę — jeśli nie kliknie, rodzina dostaje alert. Prosta idea, duże znaczenie.

Usiadłem i zacząłem pisać kod z AI. Po raz pierwszy w życiu.

Wcześniej budowałem przepływy w Make, stawiałem n8n na VPS, eksperymentowałem z lokalnymi modelami przez Ollama — ale nigdy nie pisałem kodu od zera. LifeSignal był pierwszym razem.

Kilka tygodni później podjąłem decyzję: 180 dni na to żeby zostać ekspertem AI i automatyzacji. Zacząłem dokumentować każdy projekt, każdy błąd, każde narzędzie. Ta książka jest skondensowanym wynikiem tej drogi.

Dla kogo jest ta książka

Dla kogoś kto ma zbyt wiele powtarzalnych czynności i zbyt mało czasu. Niekoniecznie programisty — ja zacząłem od łączenia klocków w Make. Dla PM-a, właściciela małej firmy, freelancera który widzi że część jego pracy mogłaby się dziać bez niego.

Każdy rozdział to prawdziwy projekt: co nie działało, dlaczego, co zbudowałem, co bym zrobił inaczej. Bez owijania w bawełnę, z kodem i schematami tam gdzie potrzeba.

Nie musisz być programistą żeby zacząć. Musisz mieć dość ręcznego kopiowania adresów email do Mailchimpa.

To wystarczy.

Rozdział 1 — Myślenie automatyzacyjne

Zanim zaczniesz łączyć klocki w Make albo pisać pierwszy skrypt w Pythonie, musisz nauczyć się jednej rzeczy: patrzeć na swoją pracę jak na przepływ.

Większość ludzi patrzy na zadania. Ja muszę wysłać tego maila. Ja muszę skopiować te dane do arkusza. Ja muszę przypomnieć klientowi o płatności. Zadania są konkretne, zrozumiałe, zamknięte.

Problem w tym, że jeśli patrzysz na zadania, widzisz tylko to co robisz teraz. Nie widzisz że robisz to samo po raz dwudziesty. Nie widzisz wzorca.

Myślenie automatyzacyjne to przełączenie się z trybu „co teraz robię" na tryb „co powtarzam".

Trzy pytania które zadaję przed każdym projektem

Zanim zacznę budować cokolwiek, zadaję sobie trzy pytania. Nie w kolejności — często odpowiedź na trzecie zmienia odpowiedź na pierwsze.

Czy to się powtarza?

Jednorazowe zadanie nie ma sensu automatyzować. Jeśli raz w roku robisz zestawienie dla księgowej, zrób je ręcznie — czas na budowanie automatyzacji będzie dłuższy niż czas samego zestawienia.

Ale jeśli to samo robisz przy każdym kliencie, każdego dnia, każdego tygodnia — to jest kandydat.

Co jest wąskim gardłem — czas, uwaga, czy człowiek?

To ważne rozróżnienie. Automatyzacja formularzy Webflow zaoszczędziła mi czas. Alert na Telegram dla współpracowniczki z Kryptopogotowia nie zaoszczędził czasu — przyspieszył reakcję człowieka. Compliance alert nie zaoszczędził czasu — wyeliminował ryzyko przeoczenia.

Trzy różne problemy, trzy różne motywacje, ta sama technika.

Jaka jest cena błędu jeśli automatyzacja padnie?

Automatyzacja maila powitalnego do klienta — jak padnie, klient nie dostanie maila. Niedobrze, ale nie tragedia.

Automatyzacja zgłoszenia transakcji do organu regulacyjnego — jak padnie, firma ma problem prawny.

Nie buduję każdej automatyzacji tak samo. Im wyższa stawka, tym więcej sprawdzam, tym więcej buduję alertów i fallbacków.

Co automatyzować, a czego nie

Dobrzy kandydaci do automatyzacji mają kilka wspólnych cech:

- Powtarzalność — to samo, w kółko, bez kreatywnych decyzji

- Dane wejściowe — jasno zdefiniowane: formularz, email z systemu, nowy wiersz w arkuszu
- Dane wyjściowe — jasno zdefiniowane: wyślij maila, dodaj do Mailchimpa, wyślij Telegram
- Niska tolerancja na błąd w logice — robot nie interpretuje, wykonuje

Złe kandydaci do automatyzacji:

- Zadania wymagające oceny sytuacji, kontekstu, empatii
- Jednorazowe akcje
- Procesy które zmieniają się co tydzień
- Cokolwiek gdzie „to zależy” pojawia się częściej niż raz

Nie automatyzuję rozmów z klientami w trudnych sytuacjach. Automatyzuję potwierdzenie że zgłoszenie wpłynęło.

ROI automatyzacji — jak liczyć opłacalność

Ludzie często pytają: „Czy warto budować to przez trzy dni jeśli oszczędza mi godzinę tygodniowo?”

Zła metoda: podziel czas budowania przez czas oszczędności. Wyjdzie Ci za długo.

Lepsza metoda: policz ile razy ta czynność pojawi się w ciągu roku. Pomyśl co możesz zrobić z zaoszczędzonym czasem albo uwagą. Dodaj wartość wyeliminowanego ryzyka błędu.

Mój alert compliance działał przez kilkanaście miesięcy bez mojej interwencji. Przez ten czas system wewnętrzny wysyłał powiadomienia, Telegram pingował właściwą osobę, transakcje były zgłaszane. Ja nie myślałem o tym ani razu.

Wartość spokoju jest trudna do policzenia. Ale jest.

Błędy które robi każdy na początku

Zrobiłem je wszystkie. Możesz je pominąć.

Automatyzowanie zanim zrozumiesz proces

Zanim cokolwiek zautomatyzujesz, powinieneś zrobić to ręcznie kilka razy. Nie dlatego że lubisz ból — dlatego że nie wiesz co dokładnie się dzieje dopóki nie zrobisz tego sam. Każde „a co jeśli klient wpisze coś dziwnego” wychodzi właśnie wtedy.

Budowanie za duże za szybko

Mój pierwszy scenariusz Webflow mógłem zbudować w dwie godziny — jeden formularz, jeden mail, koniec. Zamiast tego budowałem całość od razu: router, dziesięć ścieżek, Mailchimp, Telegram, Sheets. Trwało dłużej, błędy były trudniejsze do znalezienia.

Teraz zaczynam od najmniejszej działającej wersji. Potem rozbudowuję.

Brak monitorowania

Scenariusz który nikt nie monitoruje to scenariusz który pada i nikt tego nie wie. Buduj alerty — przynajmniej jedno powiadomienie gdy coś się wykrzacza. Make ma wbudowane powiadomienia o

błędach — włącz je.

Zakochanie w narzędziu

Make.com jest świetny. n8n jest świetny. Python bywa lepszy od obu. Żadne narzędzie nie jest odpowiedzią na każde pytanie.

Kiedy zacząłem, robiłem wszystko w Make bo Make znałem. Potem nauczyłem się n8n i zobaczyłem gdzie jest lepszy (większa kontrola, self-hosted). Potem zacząłem pisać w Pythonie i zobaczyłem gdzie żadne narzędzie no-code nie dotrze.

Narzędzie jest środkiem. Problem jest celem.

Jak zacząć jeśli nie wiesz od czego

Nie szukaj „najlepszego narzędzia do automatyzacji”. Szukaj największego bólu.

Zadaj sobie pytanie: co robiłem w tym tygodniu więcej niż dwa razy i czego nie lubiłem robić?

Odpowiedź to Twój pierwszy projekt.

Nie musi być duży. Nie musi być perfekcyjny. Musi działać wystarczająco dobrze żebyś przestał robić to ręcznie.

Moja pierwsza automatyzacja działała od pierwszego dnia. Nie dlatego że byłem dobry — dlatego że problem był prosty i ból wystarczająco duży.

Zacznij od bólu. Resztę się nauczysz po drodze.

Rozdział 2 — Stack narzędzi 2026

1 · Orkiestracja (no-code)
Make.com · n8n · Zapier

2 · AI API
Claude (Haiku/Sonnet) · Gemini · Ollama

3 · Python
Playwright · boty Telegram · scrapery

4 · Infrastruktura
VPS (OVH) · Docker · Cloudflare

Kiedy zaczynałem, próbowałem znaleźć „najlepsze narzędzie do automatyzacji". Przeczytałem dziesiątki porównań, obejrzałem dziesiątki filmów. Każdy polecał coś innego.

Straciłem czas. Prawda jest prostsza: każde z popularnych narzędzi jest dobre do czegoś i słabe do czegoś innego. Twój stack zależy od Twoich problemów, nie od rankingów.

Poniżej to czego używam, z czego zrezygnowałem i dlaczego.

Warstwa 1 — Orkiestracja bez kodu: Make.com vs n8n vs Zapier

Jeśli nie piszesz kodu, to tu zaczyna się Twoja automatyzacja. Łączysz serwisy wizualnie — trigger tutaj, akcja tam, filtr w środku.

Make.com (dawniej Integromat)

Moje pierwsze narzędzie i nadal używane przy Cashify. Wizualny, intuicyjny, ma moduły do prawie wszystkiego — Webflow, Mailchimp, Telegram, Google Sheets, webhooks, HTTP requests.

Kiedy Make:

- Chcesz uruchomić coś szybko bez VPS
- Integracje z popularnymi serwisami (Webflow, Mailchimp, HubSpot, Notion)
- Potrzebujesz routera — jeden trigger, wiele ścieżek zależnie od danych

Słabości:

- Płacisz za operacje — przy dużym wolumenie robi się drogo
- Ograniczona kontrola nad błędami przy złożonych scenariuszach
- Wszystko w chmurze Make — zależność od ich dostępności

Cena: Free plan (1000 operacji/mc), płatne od ~$9/mc

---

n8n (self-hosted)

Postawiłem na VPS. Bardziej techniczne, więcej możliwości, pełna kontrola. Kod JavaScript w węzłach gdy logika jest zbyt złożona dla wizualnych filtrów.

Kiedy n8n:

- Chcesz self-hosted (dane nie wychodzą poza Twój serwer)
- Wolumen operacji który byłby drogi w Make
- Potrzebujesz kodu w środku workflow
- Budujesz agentów AI — n8n ma wbudowane węzły AI

Słabości:

- Wymaga VPS i podstawowej znajomości Linuxa
- Mniej gotowych integracji niż Make (ale HTTP Request zastępuje prawie wszystko)
- Debugowanie trudniejsze dla początkujących

Cena: Bezpłatny (self-hosted), cloud od $20/mc

Zapier

Najprostszy z trójki, największa biblioteka integracji. Drogi.

Kiedy Zapier:

- Potrzebujesz integracji z bardzo niszowym serwisem którego Make nie ma
- Firma ma już płatny plan i nie chcesz wdrażać czegoś nowego

Kiedy nie Zapier:

- Prawie zawsze — Make robi to samo taniej, n8n robi to samo z większą kontrolą

Warstwa 2 — AI API: Claude, GPT, Gemini, lokalne modele

Automatyzacja bez AI to przepływ danych. Automatyzacja z AI to przepływ danych plus rozumienie — klasyfikacja, generowanie, ekstrakcja, podejmowanie decyzji.

Anthropic Claude (claude-haiku-4-5, claude-sonnet-4-6)

Mój wybór do automatyzacji produkcyjnej. Haiku jest tani i szybki — do klasyfikacji, ekstrakcji danych z tekstu, prostych decyzji. Sonnet jest mocniejszy — do generowania treści, analizy, pisania kodu.

W Wysprzątani CRM Claude Haiku klasyfikuje każdy post z grupy Facebook: czy to zlecenie, jak pilne, jaka kategoria. Tysiące klasyfikacji dziennie za grosze.

Cena Haiku: ~$0.80 / 1M tokenów wejścia — w praktyce groszowe operacje przy automatyzacjach

---

OpenAI GPT-4o

Dobry, droższy. Sprawdza się przy zadaniach wymagających vision (analiza zdjęć) i gdy budujesz coś co musi działać z ekosystemem OpenAI. W moich projektach zastąpiony przez Claude prawie wszędzie.

Google Gemini

Używałem przez Make.com do generowania wideo (scenariusz Sheets -> Gemini -> Drive). Dobre do multimedialnych zadań — wideo, obrazy, audio. Do czystego tekstu i kodu — wolę Claude.

Lokalne modele przez Ollama

Postawiłem Ollama na VPS — możliwość uruchomienia Llama, Mistral, Qwen i innych lokalnie. Dane nie wychodzą do chmury, brak kosztów per token.

Kiedy lokalne modele:

- Dane wrażliwe których nie chcesz wysyłać do API
- Bardzo duży wolumen gdzie koszty API byłyby wysokie
- Eksperymenty i testowanie

Słabości:

- Słabsze od najlepszych modeli chmurowych (szczególnie w polskim języku i kodzie)
- Wymagają mocnego VPS lub własnego sprzętu z GPU
- Konfiguracja i utrzymanie na Tobie

Warstwa 3 — Python: kiedy no-code nie wystarczy

Make i n8n rozwiązują 80% problemów. Resztę rozwiązuje Python.

Piszę w Pythonie gdy:

- Potrzebuję Playwright do scrapowania strony (Make tego nie zrobi)
- Logika warunkowa jest zbyt złożona dla wizualnych filtrów
- Buduję coś co ma działać jako serwis 24/7 na VPS z własnym harmonogramem
- Potrzebuję bibliotek których nie ma w żadnym narzędziu no-code (fuzzy matching, przetwarzanie PDF, własny bot Telegram)

Nie piszę Pythona bo „tak profesjonalnie”. Piszę gdy Make lub n8n nie dają rady.

Warstwa 4 — Infrastruktura: VPS, Docker, Cloudflare

VPS (Virtual Private Server)

Mój główny VPS: OVH, Ubuntu. Koszt: ~60-80 zł miesięcznie.

Na VPS działają kontenery Docker z botami i automatyzacjami które muszą być aktywne 24/7. Skrypt uruchomiony lokalnie na laptopie padnie gdy zamkniesz komputer. VPS działa zawsze.

Dla każdego kto poważnie traktuje automatyzację — VPS to konieczność.

Docker

Każdy projekt jako osobny kontener. Izolacja środowisk, łatwy restart, łatwy deploy.

```
docker-compose up -d   # uruchamia w tle
docker logs -f nazwa   # podgląd logów
```

Cloudflare

DNS + ochrona przed botami + darmowy SSL. Każda domena Cashify przechodzi przez Cloudflare. Przy skrapowaniu odkryłem że Cloudflare blokuje ruch z VPS OVH (datacenter IP) — musiałem obejść to przez zmianę nagłówków i rotację.

Mój rzeczywisty stack w 2026

Warstwa	Narzędzie	Do czego
Orkiestracja	Make.com	Integracje Cashify (Webflow, Mailchimp, Telegram)
Orkiestracja	n8n (VPS)	Własne projekty, agenty AI
AI klasyfikacja	Claude Haiku	Wysprzątani CRM, QA Bot
AI generowanie	Claude Sonnet	Ebook, skrypty, analiza
AI multimedial	Gemini	Eksperymenty wideo
Lokalne LLM	Ollama + Open WebUI	Testy, dane wrażliwe
Scraping	Python + Playwright	Wysprzątani scraper, Cashify Gold monitor
Boty	Python (python-telegram-bot)	Wszystkie boty Telegram
Infrastruktura	VPS + Docker	Wszystko co działa 24/7
DNS/SSL	Cloudflare	Wszystkie domeny

Jak wybrać gdzie zacząć

Jeśli nie pisałeś kodu nigdy: Make.com. Darmowy plan wystarczy na pierwsze projekty. Naucz się routerów, filtrów, webhooków.

Jeśli chcesz więcej kontroli i masz podstawy Linuxa: n8n na VPS. Raz postawiony, działa

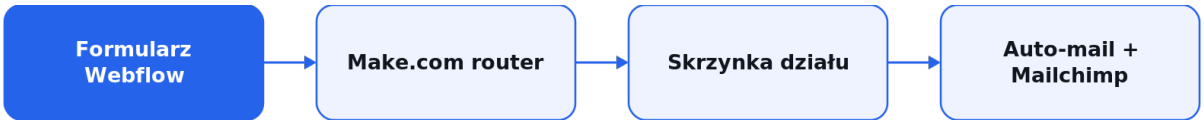
latami.

Jeśli chcesz AI w automatyzacjach: Claude API przez HTTP request w Make lub węzeł AI w n8n. Klucz API, jedno połączenie, gotowe.

Jeśli potrzebujesz scrapować strony lub budować coś niestandardowego: Python. Nie ma drogi na skróty.

Zacznij od jednej warstwy. Opanuj ją zanim dodasz kolejną. Stack który widzisz powyżej budowałem przez blisko dwa lata. Nie powstał w tydzień.

Case Study 1 — Od chaosu do routera: 60 zapisów i 37 modułów



Problem: Wszystkie formularze strony lądują na jednej skrzynce. Event przynosi sześćdziesiąt zapisów jednego dnia. Ręczne sortowanie przez dwa dni.

Rozwiązanie: Make.com, jeden trigger Webflow, trzydzieści siedem modułów, zero ręcznej pracy.

Wynik: Każde zgłoszenie trafia do właściwej osoby w sekundach. Klient dostaje potwierdzenie zanim zdąży zamknąć zakładkę.

Problem

Strona cashify.eu działa na Webflow. W maju 2024 mamy trzy albo cztery aktywne formularze: newsletter, kontakt ogólny, kryptoorganizacja, zapis na webinar. Webflow wysyła wszystkie na jedną skrzynkę. Razem, bez oznaczeń, w jednym strumieniu.

Przez większość czasu to znośne. Dziennie kilka zgłoszeń, łatwo ogarnąć ręcznie.

Potem przychodzi event. Kilka godzin, ponad sześćdziesiąt zapisów do newslettera. Plus normalne zgłoszenia kontaktowe, zapytania klientów. Wszystko na jednej skrzynce.

Przez dwa dni sortowałem ręcznie. Kopiowałem adresy email do Mailchimpa jeden po jednym. Odpisywałem na każde zgłoszenie. Przekazywałem zapytania klientów do właściwej osoby.

To był mój próg bólu. Wiedziałem że nie może tak zostać.

Rozwiązanie

Obejrzałem kilka filmów o Make.com na YouTube. Nie szukałem kursu — szukałem odpowiedzi na

konkretne pytanie: jak sprawić żeby jeden trigger obsługiwał różne typy formularzy i robił z nimi różne rzeczy.

Odpowiedź: Router.

Router w Make to węzeł który odbiera dane z triggera i wysyła je różnymi ścieżkami zależnie od warunków. Jeden wejściowy strumień, wiele wyjściowych.

Architektura:

```
Webflow Watch Events
        ↓
     Router
   /   |   |   \   \
  NL  Org  Wb  Tax  Kontakt
  ...  ...  ...  ...  ...
```

Każda ścieżka ma swój warunek: jeśli typ formularza to X, idź tą drogą. Jeśli to Y, idź inną.

Co robi każda ścieżka:

Newsletter główny:

- Kopia zgłoszenia na skrzynkę marketingu
- Email potwierdzający do klienta (z podziękowaniem za zapis)
- Powiadomienie Telegram dla teamleada (natychmiastowy sygnał)
- Dodanie do Mailchimp — ale tylko jeśli klient zaznaczył zgodę marketingową

Kryptoorganizacja:

- Kopia na właściwą skrzynkę
- Email potwierdzający z inną treścią (dopasowaną do kontekstu)
- Zapis do Google Sheets (dla trackowania leadów B2B)
- Mailchimp jeśli zgody

Kontakt:

- Kopia na skrzynkę obsługi klienta
- Email potwierdzający że zgłoszenie wpłynęło i ktoś odezwie się w ciągu X godzin

Tax (formularz podatkowy) — osobna ścieżka, wywołuje drugi scenariusz przez HTTP webhook.

Kluczowy element — warunek zgód marketingowych

Webflow przesyła dane formularza jako JSON. Pola checkboxów — w tym zgody marketingowe — mają wartość `true` lub `false`.

W Make ustawiłem filtr przed węzłem Mailchimp:

```
Warunek: zgoda_marketingowa = true
Jeśli tak -> Add/Update Subscriber w Mailchimp
Jeśli nie -> pomiń
```

Przed automatyzacją: kopiowałem wszystkie emaile do Mailchimpa bez sprawdzania zgód. Błąd prawny i proceduralny.

Po automatyzacji: do Mailchimpa trafiają tylko osoby które świadomie wyraziły zgodę. RODO przestaje być problemem operacyjnym.

Schemat przepływu krok po kroku

- Klient wypełnia dowolny formularz na cashify.eu
- Webflow wysyła webhook do Make (natychmiast po submicie)
- Make odbiera dane JSON z formularza
- Router sprawdza typ formularza (pole form_name lub URL strony)
- Dane trafiają na właściwą ścieżkę
- Równolegle (Make wykonuje gałęzie w sekwencji):
  - Email do odpowiedzialnej osoby w Cashify
  - Email do klienta
  - Mailchimp (jeśli zgody)
  - Telegram / Sheets (jeśli ścieżka tego wymaga)

Całość: kilka sekund od submita do wszystkich akcji.

Wynik

Scenariusz: 37 modułów, data utworzenia 10 maja 2024, nadal aktywny.

Przed: 2 dni ręcznego sortowania po jednym evencie. Błędy przy kopiowaniu emaili. Klienci czekający na potwierdzenie.

Po: Zero ręcznej pracy przy obsłudze formularzy. Klient dostaje potwierdzenie w sekundach. Właściwa osoba dostaje kopię natychmiast. Mailchimp rośnie automatycznie i zgodnie z RODO.

Czas budowania: jedno popołudnie.

Co bym zrobił inaczej

Zacząłbym od jednej ścieżki, nie od wszystkich naraz. Zbudowałem całość od razu — trzydzieści siedem modułów za pierwszym razem. Trwało dłużej niż powinno, błędy były trudniejsze do znalezienia.

Dziś zaczynam od minimalnej działającej wersji: jeden formularz, jeden email, jeden Mailchimp. Sprawdzam czy działa. Potem dokładam kolejne ścieżki.

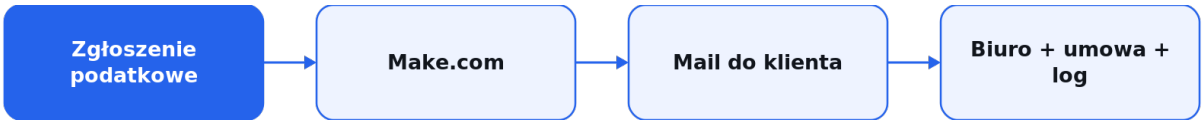
Mniejszy zakres, szybszy start, łatwiejszy debugging.

Lekcja

Router to jedno z najpotężniejszych pojęć w automatyzacji. Jeden punkt wejścia, wiele logik wyjściowych. Raz opanujesz myślenie routerowe, zaczniesz widzieć je wszędzie — nie tylko w Make.

I jeszcze jedno: nie potrzebujesz kursu żeby zacząć. Potrzebujesz konkretnego problemu i kilku filmów na YouTube. Reszta wychodzi w trakcie.

Case Study 2 — Pracownik jako przekaźnik: automatyzacja obsługi zgłoszeń podatkowych



Problem: Przy każdym zgłoszeniu klienta chcącego rozliczyć podatek od krypto, pracownik ręcznie wykonuje cztery czynności. Każdy klient. Bez wyjątku.

Rozwiązanie: Make.com, custom webhook, trzy ścieżki per typ klienta — cztery czynności dzieją się automatycznie po wypełnieniu formularza.

Wynik: Pracownik przestaje być przekaźnikiem. Klient dostaje umowę zanim pracownik w ogóle zobaczy zgłoszenie.

Problem

Cashify oferuje klientom usługę rozliczenia podatku od transakcji kryptowalutowych. Klient wypełnia formularz na stronie — podaje dane, wybiera typ (konsument, przedsiębiorca, przedsiębiorca uproszczony).

Po każdym zgłoszeniu pracownik musiał:

- Wysłać klientowi email potwierdzający zgłoszenie
- Przekazać dane klienta do biura rachunkowego
- Wysłać klientowi odpowiednią umowę
- Zalogować leada do systemu

Cztery czynności. Przy każdym kliencie. Ręcznie.

Przy małym wolumenie — do zniesienia. Przy większym — wąskie gardło. Pracownik spędza czas na mechanicznym przekazywaniu informacji między klientem a biurem. Zero wartości dodanej, sto procent powtarzalności.

To idealny kandydat do automatyzacji.

Rozwiązanie

Strona z formularzami Tax działa na WordPressie (osobna od głównej cashify.eu na Webflow). Kastomowe formularze — tylko dane, bez kalkulatora. Trzy typy klientów, trzy osobne formularze, trzy różne ścieżki obsługi.

Architektura:

```
WordPress Form Submit
        ↓
Custom Webhook -> Make.com
        ↓
     Router
   /    |    \
Konsument  Przedsiębiorca  Przed. Uproszczony
   ↓           ↓                ↓
Email    Email           Email
+ HTTP   + HTTP          + HTTP
+ HTTP   + HTTP          + HTTP
+ Email  + Email         + Email
(zgody)  (zgody)         (zgody)
+ Mailchimp + Mailchimp  + Mailchimp
+ Email  + Email         + Email
```

Każda ścieżka identyczna w strukturze — różni się treścią emaili, typem umowy i odbiorcą w biurze rachunkowym.

Jak działa HTTP webhook do biura:

Make wysyła dane klienta przez HTTP POST do systemu biura rachunkowego (lub na ich adres email przez API). Biuro dostaje ustrukturyzowane dane — nie przesłany dalej mail, tylko czyste pola: imię, nazwisko, email, typ rozliczenia, okres podatkowy.

Jak działa wysyłka umowy:

Drugi HTTP call generuje lub pobiera odpowiednią umowę (PDF per typ klienta) i wysyła do klienta. Konsument dostaje umowę konsumencką, przedsiębiorca — B2B.

Kluczowy element — trzy typy klientów, jedna logika

Router rozpoznaje typ klienta po polu w formularzu. Reszta jest identyczna — zmienia się tylko zawartość, nie struktura.

To ważna zasada którą stosuję zawsze: **izoluj zmienną, nie powielaj logiki**.

Zamiast trzech osobnych scenariuszy (jeden per typ klienta), mam jeden scenariusz z routerem. Gdy biuro rachunkowe zmienia adres email — zmieniam w jednym miejscu, nie w trzech.

Schemat przepływu

- Klient wypełnia formularz Tax na WordPress
- Formularz wysyła dane przez webhook do Make
- Router identyfikuje typ klienta
- Równolegle:
  - Email potwierdzający do klienta („Twoje zgłoszenie wpłynęło”)
  - Dane do biura rachunkowego przez HTTP
  - Umowa do klienta przez HTTP
  - Email z informacją o zgodach marketingowych
  - Mailchimp jeśli zgody
  - Email końcowy potwierdzający wysyłkę umowy

Klient dostaje potwierdzenie i umowę. Biuro dostaje dane. Pracownik nie robi nic.

Wynik

Przed: każde zgłoszenie = cztery ręczne kroki pracownika. Czas: kilkanaście minut per klient.

Po: zero ręcznej pracy przy standardowym zgłoszeniu. Pracownik wkracza tylko gdy klient ma niestandardowe pytanie lub problem.

Pracownik nie jest już przekaźnikiem — jest rozwiązującym problemy.

Co bym zrobił inaczej

Podpiąłbym monitoring. Przez pierwsze tygodnie sprawdzałem ręcznie czy wszystko działa poprawnie — czy umowy dochodzą, czy biuro dostaje dane. Strata czasu.

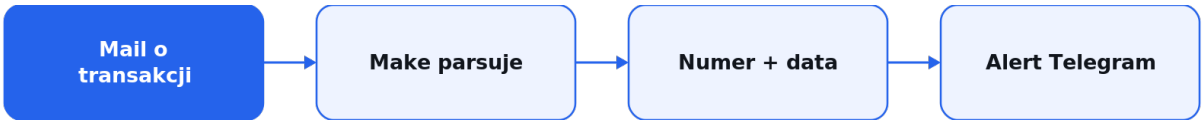
Teraz przy każdym scenariuszu produkcyjnym ustawiam alert na Telegram jeśli scenariusz zwróci błąd. Make ma to wbudowane — wystarczy włączyć w ustawieniach scenariusza.

Lekcja

Automatyzacja najlepiej sprawdza się tam gdzie człowiek jest tylko przekaźnikiem — przyjmuje dane z punktu A i przekazuje do punktu B. Jeśli nie dodajesz żadnej wartości do przepływu informacji, prawdopodobnie możesz to zautomatyzować.

Pytaj: „Co konkretnie robię przy każdym przypadku?” Jeśli odpowiedź jest taka sama przy każdym przypadku — masz kandydata.

Case Study 3 — Automatyzacja której nikt nie widzi: alert compliance dla kantoru krypto



Problem: System wewnętrzny wysyła emaile o transakcjach wymagających zgłoszenia do organu regulacyjnego. Główna skrzynka firmowa jest pełna innych wiadomości. Łatwo przeoczyć.

Rozwiązanie: Make monitoruje skrzynkę, wyłapuje emaile compliance i wysyła alert na Telegram z numerem i datą transakcji.

Wynik: Zero przeoczonych zgłoszeń. Zero ryzyka prawnego z powodu przeoczonego maila.

Problem

Kantor kryptowalutowy w Polsce podlega regulacjom AML (Anti-Money Laundering). Część transakcji wymaga zgłoszenia do odpowiedniego organu nadzoru. Obowiązek nie jest opcjonalny — niedopełnienie to ryzyko prawne i finansowe dla firmy.

Wewnętrzny system Cashify generuje automatyczne powiadomienia gdy pojawia się taka transakcja. Wysyła email na główną skrzynkę firmową.

Problem: główna skrzynka firmowa to nie jest skrzynka compliance. To skrzynka wszystkiego — klienci, dostawcy, faktury, zgłoszenia, newslettery, spam. Dziesiątki maili dziennie.

Email compliance wyglądał jak każdy inny. Nie miał specjalnego znacznika. Wymagał czytania i interpretacji.

Wyobraź sobie że go przeoczysz. Transakcja niezgłoszona w terminie. Konsekwencje regulacyjne.

Rozwiązanie

Make monitoruje skrzynkę (moduł Email Watch) pod kątem wiadomości spełniających określone kryteria — nadawca systemowy, konkretny temat lub słowa kluczowe w treści.

Gdy warunek jest spełniony, Make wyciąga z treści maila dwa pola:

- Numer transakcji
- Data transakcji

I wysyła alert na Telegram do właściwej osoby lub kanału.

Dlaczego tylko numer i data — bez danych klienta:

RODO. Telegram to zewnętrzna aplikacja — nie wysyłam przez nią danych osobowych klientów.

Wysyłam minimum potrzebne do identyfikacji transakcji w systemie wewnętrznym. Pracownik dostaje ping, wchodzi do systemu, odnajduje transakcję po numerze, realizuje zgłoszenie.

Automatyzacja transportuje sygnał, nie dane wrażliwe.

Schemat przepływu

- System wewnętrzny wysyła email compliance na skrzynkę firmową
- Make sprawdza skrzynkę co 15 minut (Email Watch, scheduled)
- Filtr: czy email pochodzi z systemu? Czy zawiera słowa kluczowe?
- Jeśli tak: wyciągnij numer transakcji + datę z treści (Text Parser lub regex)
- Wyślij wiadomość Telegram: „Transakcja [NUMER] z dnia [DATA] — wymagane zgłoszenie”
- Właściwa osoba dostaje ping -> wchodzi do systemu -> zgłasza

---

Kluczowy element — wyciąganie danych z treści emaila

Treść maila systemowego ma stały format. Make ma moduł Text Parser który wyciąga tekst między podanymi znacznikami.

Przykład: jeśli email zawsze ma linię „Numer transakcji: XXXXXXXX”, wyciągam to co jest po dwukropku. Prosto, bez AI, bez ML — zwykłe parsowanie tekstu.

Gdy format maila jest nieregularny — tu warto dodać Claude Haiku który rozumie kontekst i wyciąga właściwe pole nawet gdy struktura się zmienia.

Wynik

Ta automatyzacja jest niewidoczna gdy działa. Nie ma dashboardu, nie ma raportu, nie ma liczby zaoszczędzonych godzin.

Jest tylko spokój: żadna transakcja compliance nie zostanie przeoczona dlatego że ktoś był zajęty i nie przeczytał maila.

Wartość tej automatyzacji mierzy się w tym czego nie ma — w braku incydentu regulacyjnego, w braku kary, w braku stresu prezesa który dostaje pytanie od organu nadzoru.

Co bym zrobił inaczej

Dodałbym potwierdzenie zwrotne. Pracownik dostaje alert, realizuje zgłoszenie — ale jak wiem że zgłoszenie zostało zrealizowane? Nie wiem.

Kolejny krok (nie zbudowany): po realizacji zgłoszenia pracownik odpowiada na Telegramie komendą `/zrealizowano [numer]` — bot aktualizuje log w Google Sheets. Mamy historię

każdego zgłoszenia: kiedy alert, kiedy realizacja, kto zrealizował.

To jest różnica między automatyzacją reaktywną (alert gdy coś się dzieje) a automatyzacją procesową (pełny cykl życia zadania).

Lekcja

Nie każda automatyzacja oszczędza czas. Niektóre chronią przed ryzykiem.

Gdy rozmawiasz z właścicielem firmy o automatyzacji, czas zaoszczędzony to dobry argument. Ale ryzyko wyeliminowane to lepszy argument — szczególnie dla firm w regulowanych branżach.

„Nie przepuścimy żadnego zgłoszenia compliance" to coś za co każdy prezes kantoru zapłaci.

Case Study 4 — Bot który sam szuka klientów: Wysprzątani CRM

Problem: Zlecenia na sprzątanie pojawiają się w dziesiątkach grup na Facebooku. Ręczne monitorowanie trzynastu grup, filtrowanie szumu, odpowiadanie na oferty — to kilka godzin dziennie.

Rozwiązanie: Python + Playwright scraper monitoruje grupy 24/7, Claude Haiku klasyfikuje czy post to zlecenie, Telegram bot zarządza leadami — przyjmij lub odrzuć jednym kliknięciem.

Wynik: System autonomiczny od maja 2026. Zero ręcznego monitorowania grup. Fazy 1-5 wdrożone, działa w tle na VPS.

Problem

Wysprzątani to firma sprzątająca działająca pod marką Grupy Carkul. Zlecenia pojawiają się organicznie w lokalnych grupach Facebook — ktoś pisze „szukam kogoś do sprzątania po remoncie w Wilanowie", ktoś inny „kto może wyczyścić auto przed sprzedażą".

Przed automatyzacją: ktoś musiał codziennie przeglądać grupy i wyławiać takie posty. Problem: grup jest trzynaście. Każda ma dziesiątki lub setki postów dziennie. 90% to ogłoszenia, dyskusje, pytania — nie zlecenia.

Ręczne filtrowanie trzynastu grup to minimum godzina pracy dziennie, przy czym ta praca nie wymaga żadnych decyzji — tylko uwagi i czasu.

Idealne zadanie dla automatyzacji.

Rozwiązanie

System składa się z trzech warstw:

Warstwa 1 — Scraper (kod/scraper.py)

Python + Playwright. Scraper loguje się do Facebooka przez zapisane ciasteczka (żaden oficjalny API nie da dostępu do prywatnych grup), przegląda trzynaście grup i zbiera nowe posty.

Playwright symuluje przeglądarkę desktopową — identyczny fingerprint co normalny użytkownik. Bez tego Facebook blokuje boty.

Ciasteczka odświeżane lokalnie na MacBooku przez osobny skrypt (`refresh_cookies.py`) — sesja Facebooka musi być aktualna.

Scraper działa w Docker na VPS, uruchamiany przez cron co 30 minut.

Warstwa 2 — Klasyfikacja AI (Claude Haiku)

Każdy zebrany post przechodzi przez Claude Haiku z promptem:

```
Czy ten post to zlecenie na usługę sprzątania lub detailingu?
Odpowiedz: TAK/NIE
Jeśli TAK: kategoria (sprzątanie domu, po remoncie, auto, inne), pilność (dziś, ten tydzień, elastyczna)
```

Haiku jest tani i szybki — klasyfikacja setek postów dziennie kosztuje grosze.

Posty sklasyfikowane jako NIE — ignorowane.
Posty TAK — trafiają do bota Telegram.

Warstwa 3 — CRM Telegram (kod/handler.py)

Bot wysyła do operatora kartę leada: treść posta, autor, grupa, kategoria, pilność.

Dwa przyciski: **[OK] Przyjmij** / **[X] Odrzuć**

Kliknięcie Przyjmij -> operator dostaje dane kontaktowe + link do posta -> bot loguje lead do Google Sheets ze statusem „aktywny”.

Kliknięcie Odrzuć -> rozwijane menu przyczyn (za daleko, poza zakresem, za tanie) -> log z przyczyną.

Cały CRM w Telegramie. Bez osobnej aplikacji, bez logowania do panelu — operator obsługuje leady z telefonu, z dowolnego miejsca.

Kluczowy element — ciasteczka Facebooka

Facebook nie ma API dla grup prywatnych. Jedynym sposobem na automatyczny dostęp jest sesja zalogowanego użytkownika.

Scraper używa ciasteczek z prawdziwej sesji Facebook. Problem: ciasteczka wygasają.

Rozwiązanie: skrypt `refresh_cookies.py` uruchamiany lokalnie na MacBooku — otwiera przeglądarkę, loguje się, zapisuje świeże ciasteczka, przesyła na VPS.

To jedyna ręczna czynność w całym systemie — odświeżenie ciasteczek co kilka tygodni.

Dodatkowo: scraper używa przeglądarki z profilem „Comet" (desktopowy Chrome) z losowymi opóźnieniami między akcjami — symuluje ludzkie zachowanie.

Architektura na VPS

```
VPS (Twój serwer)
 Docker: scraper      # scraper.py — zbiera posty
 Docker: handler      # handler.py — bot Telegram CRM
 watchdog.py (cron co 20 min) # sprawdza czy kontenery żyją
 Google Sheets                # baza leadów + historia
```

Watchdog to osobny skrypt monitorujący — jeśli kontener padnie, restartuje go automatycznie i wysyła alert Telegram.

Schemat przepływu

- Cron uruchamia scraper co 30 minut
- Scraper przegląda 13 grup FB z ciasteczkami
- Nowe posty -> Claude Haiku klasyfikuje (TAK/NIE)
- TAK -> handler.py buduje kartę leada
- Bot wysyła kartę do operatora na Telegram
- Operator klika Przyjmij lub Odrzuć
- Log do Google Sheets
- Watchdog sprawdza co 20 minut czy wszystko żyje

Wynik

System w produkcji od 19 maja 2026. Autonomiczny — działa 24/7 bez ręcznej interwencji poza odświeżaniem ciasteczek.

Przed: 1–2 godziny dziennie ręcznego monitorowania grup, nieregularne — posty przeoczane.

Po: zero ręcznego monitorowania. Operator widzi tylko leady które spełniają kryteria, jednym przyciskiem decyduje.

Co bym zrobił inaczej

Zacząłbym od jednej grupy, nie trzynastu. Pierwsze testy na jednej grupie, weryfikacja jakości klasyfikacji AI, dopiero potem rozszerzenie.

Budowałem od razu na trzynaście — dłuższe debugowanie, trudniejsza weryfikacja czy prompt działa dobrze.

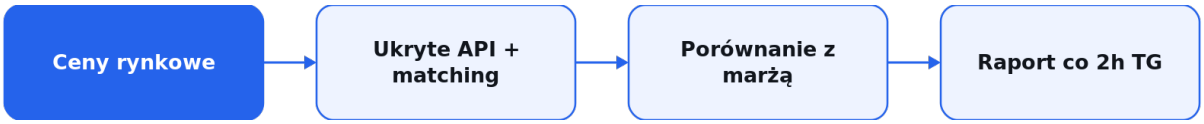
Lekcja

Facebook nie ma API. Wiele platform nie ma API. To nie znaczy że nie można pobierać danych — znaczy że trzeba użyć innego podejścia.

Playwright to narzędzie które robi w kodzie to co Ty robisz w przeglądarce. Otwiera stronę, klika, scrolluje, czyta. Jeśli Ty możesz to zrobić ręcznie, Playwright może to zrobić automatycznie.

Granica etyczna i prawna: nie używaj tego do masowego zbierania danych prywatnych. Tu używamy do monitorowania publicznych postów w grupach, w których konto jest aktywnym członkiem. Cel: odpowiadanie na oferty usług — dokładnie to, co robiłby człowiek.

Case Study 5 — Ukryte API i monitor marż: Cashify Gold Analityk



Problem: Cashify Gold sprzedaje monety i sztabki. Ceny rynkowe zmieniają się co godzinę. Ręczne porównywanie oferty z cenami konkurencji i cenami skupu to praca dla analityka — lub bota.

Rozwiązanie: Scraper WooCommerce + odkryty ukryty REST API NumiTracker + fuzzy matching produktów + raport Telegram co dwie godziny.

Wynik: System działa na produkcji. Zespół sklepu dostaje automatyczny raport konkurencyjności cen co pół godziny, ceny skupu odświeżane co 5 minut, a zgłoszenia klasyfikuje AI.

Problem

cashify.gold to sklep z monetami bulionowymi i sztabkami złota. Ceny produktów zależą od aktualnych cen kruszców — złoto, srebro, platyna zmieniają się w czasie rzeczywistym.

Pytania które pojawiały się regularnie:

- Czy nasza cena sprzedaży Krugerranda 1oz jest konkurencyjna?
- Jaka jest aktualna marża na monetach srebrnych?
- Kiedy cena skupu spadnie poniżej progu opłacalności?

Odpowiedź wymagała: pobrania aktualnych cen z cashify.gold + pobrania cen rynkowych z zewnętrznego źródła + ręcznego porównania pozycja po pozycji.

Przy katalogu liczącym kilkadziesiąt produktów i cenach zmieniających się co godzinę — nie da się tego robić ręcznie sensownie.

Odkrycie: ukryte API NumiTracker

Zewnętrzne źródło cen rynkowych monet to serwis NumiTracker. Nie ma oficjalnego publicznego API.

Ale ma ukryty REST API.

Jak to odkryłem: otworzyłem serwis w przeglądarce i włączyłem DevTools -> zakładka Network. Przeładowałem stronę i obserwowałem jakie requesty wysyła przeglądarka. Między requestami do statycznych plików pojawiły się wywołania do endpointu:

```
https://admin.metalapi.co.pl/numitracker/api/v1/products
```

Endpoint zwraca JSON z pełnym katalogiem produktów wraz z cenami. Bez klucza API, bez autoryzacji — dostępny przez zwykłe GET.

To dane których nie ma nikt kto nie poszuka. Dla scrapera to idealne źródło — ustrukturyzowane, szybkie, niezawodne.

Uwaga: odkrycie ukrytego API to technika, nie hack. Przeglądarka robi te requesty za Ciebie każdego dnia — Ty tylko patrzysz co ona robi i powtarzasz to w kodzie.

Problem dopasowania: fuzzy matching

Produkty na cashify.gold i w NumiTracker nazywają się inaczej.

cashify.gold: „Krugerrand 1 oz Złoto 2024 RPA”
NumiTracker: „Krugerrand 1oz Au RPA 2024”

Dokładne dopasowanie tekstu nie zadziała. Potrzebny fuzzy matching — algorytm który mierzy podobieństwo ciągów znaków i dopasowuje produkty mimo różnic w nazewnictwie.

Dodatkowy problem: wagi ułamkowe. Monety sprzedawane są w wagach 1/10, 1/4, 1/2, 1 oz. Algorytm musi traktować „1/10” i „1/4” jako różne tokeny — nie jako ułamki matematyczne.

```
# Złe — Python interpretuje 1/10 jako 0.1 i 1/4 jako 0.25
# Oba stają się podobne numerycznie

# Dobre — traktuj ułamki jak stringi
def normalize_fraction(text):
    return re.sub(r'(\d+)/(\d+)', r'\1_\2', text)
# "1/10" -> "1_10", "1/4" -> "1_4" — różne tokeny
```

Architektura

```
Scraper WooCommerce (Playwright)
   cashify.gold — pobiera katalog z cenami
          ↓
NumiTracker API (HTTP GET)
   admin.metalapi.co.pl — pobiera ceny rynkowe
          ↓
Fuzzy Matching
   dopasowanie produktów między źródłami
          ↓
Obliczenie marż
```

```
   cena sprzedaży − cena rynkowa = marża
          ↓
Google Sheets (Panel + Archiwum)
   zapis wyników, historia
          ↓
Telegram Bot
   raport cykliczny + alert gdy marża < próg + komendy na żądanie
```

Paginacja WooCommerce:

WooCommerce domyślnie zwraca produkty stronami (np. 20 per strona). Wersja z sesji Gemini pobierała tylko pierwszą stronę — brakło części katalogu.

Poprawka: pętla pobierająca kolejne strony dopóki API zwraca produkty:

```
page = 1
all_products = []
while True:
    response = requests.get(f"{url}?per_page=100&page={page}")
    products = response.json()
    if not products:
        break
    all_products.extend(products)
    page += 1
```

Ekstrakcja ceny po rabacie:

WooCommerce zwraca kilka cen per produkt — regularna, promocyjna, po rabacie. Scraper zawsze bierze ostatnią aktualną cenę z bloku, nie pierwszą.

Schemat przepływu

- Monitor uruchamia cykl co 30 minut (ceny skupu — osobny cron co 5 minut)
- Playwright otwiera cashify.gold, pobiera katalog (wszystkie strony)
- HTTP GET do NumiTracker API, pobiera ceny rynkowe
- Fuzzy matching: dopasowanie produktów między źródłami
- Obliczenie marży per produkt
- Zapis do Google Sheets (Panel: aktualne, Archiwum: historia)
- Telegram: raport z tabelą marż
- Jeśli marża < próg -> dodatkowy alert

Od MVP do produkcji — co dołożyło życie

Pierwsza wersja to był prosty monitor marż. Produkcja wymusiła kilka rozszerzeń, które warto

znać, bo powtarzają się w każdym projekcie scrapującym:

Cloudflare zamiast Playwright. Sklep chronił część katalogu Cloudflare. Ciężka przeglądarka Playwright okazała się zbędna — wystarczyła biblioteka `curl-cffi`, która podszywa się pod odcisk palca prawdziwej przeglądarki (Chrome). Lżej, szybciej, stabilniej. Osobny cron odświeża ceny skupu co 5 minut, uderzając w wewnętrzny endpoint sklepu tą samą techniką (zwykły `curl` dostawał 403).

Dopasowanie produktów przez AI. Fuzzy matching po nazwach mylił się na nietypowych tytułach. Doszło wsparcie modelu Claude Haiku: dostaje listę tytułów i wybiera właściwy produkt — plus „sanity-check" wagi, żeby nie dopasować sztabki 1 g do uncjowej (kiedyś dawało absurdalne -88% marży).

Panel w arkuszu — 12 kolumn. Zamiast surowej tabeli: cena Cashify, cena rynkowa, marża w procentach, głębokość rynku (ile ofert), czas ostatniej aktualizacji w czasie polskim. Kolory odwrócone i przemianowane na „konkurencyjność": czerwony = jesteśmy drożsi od rynku, zielony = najbardziej konkurencyjni.

Komenda /spot — ceny kruszców na żywo. Bot na żądanie podaje aktualne ceny złota, srebra, platyny i palladu, łącząc dwa źródła (NBP dla złota, Yahoo Finance dla reszty) z kursem USD/PLN i stopką źródeł.

ZGŁOSZENIA z klasyfikacją AI. Zespół zgłasza błędy i sugestie przez bota. Każde zgłoszenie Claude Haiku klasyfikuje: kategoria, priorytet, status — i ląduje w osobnej zakładce arkusza z dziennym raportem posortowanym po pilności.

Wynik

System działa na produkcji na VPS w Dockerze. Raport konkurencyjności co 30 minut, ceny skupu co 5 minut, komendy na żądanie (/spot, /panel, /top5, /skup).

Przed: brak regularnego monitorowania, decyzje cenowe oparte na intuicji albo raz dziennie ręcznym sprawdzeniu.

Po: zespół sklepu widzi w arkuszu i na Telegramie, gdzie jesteśmy drożsi od rynku, bez jednej ręcznej czynności — plus kanał zgłoszeń, który sam się porządkuje.

Co bym zrobił inaczej

Pilnowałbym jednej rzeczy od początku: filtra kraju przy cenach rynkowych. Zewnętrzne API domyślnie zwracało też zagranicznych dealerów z niższymi cenami niż te widoczne na polskiej stronie — przez chwilę raporty pokazywały marże liczone względem złych odniesień. Lekcja: przy zewnętrznych źródłach cen zawsze sprawdź, jakiego rynku dotyczą, zanim zbudujesz na nich alerty.

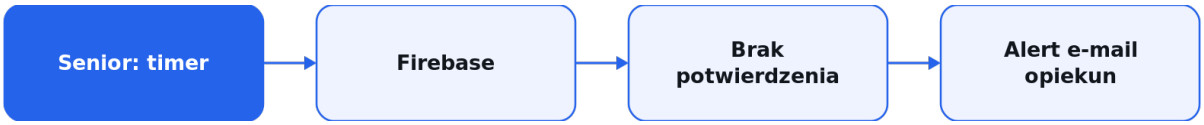
Lekcja

Zanim zaczniesz scraper który parsuje HTML strony — sprawdź co robi przeglądarka. DevTools -> Network -> przeładuj stronę -> szukaj wywołań do JSON lub API.

Strony internetowe często pobierają dane z wewnętrznych API zanim je wyświetlą. Jeśli znajdziesz ten endpoint, masz dostęp do ustrukturyzowanych danych bez parsowania HTML.

Oszczędza to dni pracy i daje znacznie bardziej stabilne źródło danych — HTML zmienia się przy każdym redesignie, API zmienia się rzadziej.

Case Study 6 — Pierwsza aplikacja: LifeSignal dla seniorów



Problem: Rodzina nie wie czy senior żyje i ma się dobrze. Telefon może być za daleko, senior może nie zadzwonić. Brak prostego systemu „żyję” który działa bez nauki nowych technologii.

Rozwiązanie: Aplikacja Flutter — senior klika serce raz dziennie. Jeśli nie kliknie w ciągu 24/48/72 godzin — rodzina dostaje alert email.

Wynik: Działająca aplikacja iOS/Android. Napisana z AI przez kogoś kto nigdy wcześniej nie pisał kodu aplikacji mobilnej.

Problem

Pomysł przyszedł od kolegi Janka na początku 2026 roku. Prosta obserwacja: dzwonimy do rodziców i dziadków żeby upewnić się że wszystko dobrze, ale nie robimy tego wystarczająco regularnie. A senior który ma wypadek w domu może leżeć godzinami zanim ktoś zadzwoni.

Istniejące rozwiązania: drogie opaski SOS, skomplikowane systemy telecaringu, czujniki ruchu wymagające instalacji. Żadne nie jest proste jak kliknięcie przycisku.

Pomysł LifeSignal: senior otwiera aplikację raz dziennie i klika serce. To jego sygnał „żyję”. Jeśli nie kliknie przez zdefiniowany czas — aplikacja wysyła alert do opiekuna.

Zero nauki. Zero subskrypcji. Jedno kliknięcie.

Dlaczego to ważny case study

LifeSignal to pierwszy projekt gdzie pisałem kod od zera.

Wcześniej: Make.com (klocki wizualne), Webflow (drag and drop), skrypty Python do konkretnych zadań kopiowane i modyfikowane. Nigdy architektury aplikacji mobilnej od początku.

Janek wpadł na pomysł. Usiadłem z Claude i zacząłem pytać. Nie „napisz mi aplikację" — „wytłumacz mi jak działa Flutter", „co to jest Firebase Firestore", „jak zbudować timer w Dart".

Kilka tygodni później mieliśmy działającą aplikację.

Stack techniczny

Frontend: Flutter/Dart

Flutter to framework Google do budowania aplikacji mobilnych (iOS + Android) z jednej bazy kodu. Dart to język programowania — mocno typowany, podobny do Javy ale bardziej nowoczesny.

Wybrałem Flutter bo:

- Jedna baza kodu = aplikacja na oba systemy
- Duża społeczność, dużo materiałów
- Claude dobrze zna Flutter — odpowiedzi na pytania były konkretne i działające

Backend: Firebase

Firebase to platforma Google — baza danych (Firestore), autentykacja, Cloud Functions, hosting. Bezserwerowa — nie zarządzam żadnym serwerem, płacę za użycie.

- Firestore: przechowuje konta użytkowników, dane timerów, historię kliknięć
- Cloud Functions: logika serwerowa — sprawdza czy senior kliknął, wysyła alerty
- Brevo: email alertów (przez Cloud Functions, EU — RODO)

Jak działa timer

Senior ustawia interwał: 24, 48 lub 72 godziny. Klika serce — timer się resetuje. Firebase zapisuje timestamp ostatniego kliknięcia.

Cloud Function uruchamiana co godzinę sprawdza: czy od ostatniego kliknięcia minął zdefiniowany interwał? Jeśli tak — wysyła email ostrzeżenia (jeszcze X godzin do alertu) i potem email alertu (senior nie kliknął, sprawdź czy wszystko dobrze).

Dwa poziomy powiadomień:

- Warning email: „Senior nie kliknął od X godzin, [Y] godzin do alertu"
- Alert email: „UWAGA: [Imię seniora] nie potwierdzał obecności przez [czas]. Prosimy o kontakt."

Naprawione błędy (K1, K2)

K1 — `_sendAlertEmail()` i `_sendWarningEmail()`:
Funkcje emailowe w `timer_cubit.dart` nie były wywołane we właściwych miejscach. Timer kończył się bez wysyłki. Naprawione przez dodanie wywołań we właściwych momentach cyklu życia timera.

K2 — Baner braku połączenia:
Aplikacja nie informowała użytkownika gdy nie ma internetu. Senior klikał serce, myślał że wszystko dobrze — a timer nie był zresetowany bo brak połączenia. Dodany baner w `home_screen.dart` informujący o braku sieci.

K3 — Dostarczanie maili:
Migracja dostawcy maili na Brevo (EU, RODO-compliant). Konfiguracja domain-auth + weryfikacja nadawcy. Maile przez Firebase Cloud Functions.

Schemat działania

```
Senior: otwiera aplikację -> klika serce
        ↓
Flutter: zapisuje timestamp do Firestore
        ↓
Cloud Function (co godzinę):
  - pobiera ostatni timestamp z Firestore
  - porównuje z interwałem
  - jeśli przekroczony: Brevo -> email ostrzeżenia
  - jeśli bardzo przekroczony: Brevo -> email alertu
        ↓
Opiekun: dostaje email -> dzwoni do seniora
```

Co znaczy „pisanie z AI”

Nie dyktowałem Claude co ma napisać i kopiowałem kod. To nie działa — kod który nie jest rozumiany nie może być debugowany ani rozwijany.

Pytałem o koncepty. „Co to jest stan w Flutter?” „Jak działa BLoC pattern?” „Dlaczego ten błąd kompilacji?” Claude wyjaśniał, pokazywał przykłady, sugerował jak to zastosować w moim konkretnym przypadku.

Pisałem kod sam, Claude weryfikował i poprawiał. Gdy coś nie działało — wklejałem błąd, dostawałem diagnozę.

To jak programowanie w parze z kimś kto zna odpowiedź na każde pytanie, jest dostępny o każdej godzinie i nigdy się nie denerwuje gdy pytasz po raz trzeci o to samo.

Wynik

Działająca aplikacja Flutter + Firebase. 47/47 testów. Build 1.1.3+26 gotowy na TestFlight (z SDK płatności Adapty i pre-alarmem budzikowym). Maile przez Brevo (EU, RODO).

Czas od pierwszej linii kodu do działającego prototypu: kilka tygodni.

Koszt infrastruktury Firebase: darmowy plan na start. Brevo free tier wystarczy na beta testy (limit 300 maili/dzień).

Co bym zrobił inaczej

Zacząłbym od backendowego MVP zanim budowałem UI. Najpierw Cloud Function + email działa, potem aplikacja mobilna. Zamiast tego budowałem równolegle — gdy UI był gotowy, backend nie działał i trudno było oddzielić gdzie jest błąd.

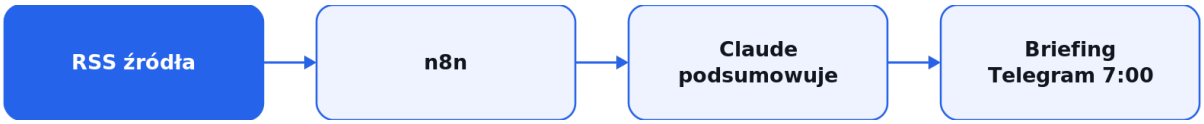
Lekcja

Nie potrzebujesz być programistą żeby zbudować aplikację mobilną w 2026.

Potrzebujesz umieć zadawać precyzyjne pytania, rozumieć co dostałeś, i nie poddawać się gdy pojawi się błąd którego nie rozumiesz.

AI jako partner programistyczny to nie sciągawka. To akcelerator dla kogoś kto myśli — i bariera dla kogoś kto nie myśli wcale.

Case Study 7 — Codzienny briefing z RSS, Claude i Telegram: projekt SIGNAL



Problem: Za dużo źródeł informacji (krypto, złoto, makro), za mało czasu na ich śledzenie. Ważne informacje gubią się w szumie.

Rozwiązanie: n8n czyta RSS ze skonfigurowanych źródeł, Claude analizuje i wyciąga to co istotne, gotowy briefing leci na Telegram o 7:00.

Wynik: Codziennie o poranku — pięć punktów z tego co ważne. Zero przeglądania portali.

Problem

Pracuję w branży krypto i śledzę rynki złota i srebra. Powinienem wiedzieć co się dzieje — nowe regulacje, ruchy cenowe, ważne wydarzenia makroekonomiczne.

Problem: źródeł jest kilkanaście. Każde trzeba odwiedzić osobno. Większość treści to clickbait, powtórki i szum. Wyodrębnienie naprawdę ważnych informacji zajmuje pół godziny — codziennie.

Pół godziny rano które mogłoby być na coś innego.

Rozwiązanie

n8n jako orkiestrator:

n8n (self-hosted na VPS) ma wbudowany węzeł RSS Read. Podpinam do niego kilkanaście kanałów RSS: portale kryptowalutowe, agregatory cen złota, serwisy makroekonomiczne, polskie portale finansowe.

n8n uruchamia flow przez cron o 6:30. Zbiera wszystkie artykuły z ostatnich 24 godzin.

Claude jako filtr i analityk:

Zebrane artykuły (tytuły + leady) trafiają do Claude Sonnet z promptem:

```
Jesteś analitykiem rynków krypto i metali szlachetnych.
Z poniższych artykułów wybierz 5 najważniejszych dla kogoś kto:
- handluje BTC, ETH, złotem i srebrem
- śledzi regulacje krypto w Polsce i UE
- interesuje się makroekonomią (stopy procentowe, inflacja, USD)
```

```
Dla każdego: jednozdaniowe podsumowanie + dlaczego ważne.
Format: lista punktowana, bez bullshitu.
```

Claude nie streszcza wszystkiego — selekcjonuje to co naprawdę istotne dla konkretnego profilu czytelnika.

Telegram jako dostarczanie:

Gotowy briefing leci przez n8n Telegram node na wybrany kanał lub czat o 7:00.

Format który działa:

```
 Briefing [data]

– BTC przekroczył kluczowy opór — Fed sygnalizuje pauzę w podwyżkach stóp
– Złoto utrzymuje się przy oporze — dane z rynku futures wskazują na presję wzrostową
– [3 kolejne punkty]

 Generowano: 6:52 | Źródła: [lista]
```

Dlaczego n8n a nie Make

Make ma moduł RSS, ale przy kilkunastu źródłach + wywołaniu Claude API + Telegram — liczba operacji rośnie szybko i robi się drogie przy codziennym uruchamianiu.

n8n self-hosted: zero kosztów per operacja, pełna kontrola, łatwe modyfikowanie flow bez limitów planu.

Jedyny koszt: Claude API (Sonnet) per wywołanie — przy ~10 000 tokenów dziennie to kilkanaście groszy.

Schemat przepływu

```
Cron 6:30 -> n8n start
        ↓
RSS Read × N źródeł (równolegle)
        ↓
Merge — zebranie wszystkich artykułów
        ↓
Filtr — tylko artykuły z ostatnich 24h
        ↓
Claude Sonnet API — selekcja 5 najważniejszych
        ↓
Format wiadomości Telegram
```

```
        ↓
Telegram Send — 7:00
```

Personalizacja

Briefing jest skrojony pod konkretny profil — nie generyczny „top 5 krypto news".

Twój profil determinuje co Claude uzna za ważne. PM kantoru krypto z pozycjami na złocie i BTC dostaje inny briefing niż startup founder śledzący AI.

Zmiana profilu = zmiana jednego promptu w n8n. Nowa osoba, nowa perspektywa.

Wynik

Workflow gotowy (`signal_workflow.json`), do wdrożenia na VPS.

Czas na poranny przegląd informacji: z ~30 minut do ~2 minut (czas czytania briefingu).

Jakość: nie wszystkie ważne newsy zostają złapane — Claude nie ma dostępu do wszystkiego co publikowane. Ale 80% ważnych rzeczy trafia. Na codzienną orientację wystarczy.

Co bym zrobił inaczej

Dodałbym mechanizm uczenia się — gdy klikam w link z briefingu, system wie że to był trafiony news. Gdy ignoruję — sygnał że to nie był ważny. Po tygodniu prompt jest lepiej skalibrowany pod moje zainteresowania.

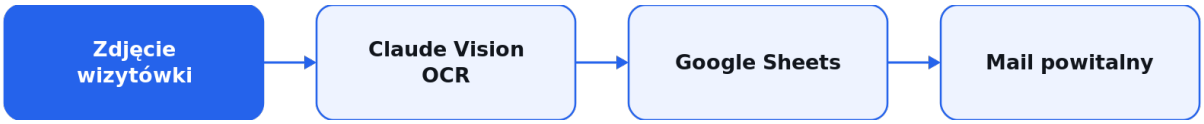
To nie jest trudne do zbudowania — Google Sheets log kliknięć + cotygodniowy prompt update.

Lekcja

Personalizacja informacji to jeden z najsilniejszych use case'ów dla AI. Nie „napisz mi artykuł" — „z tych trzydziestu artykułów wybierz pięć ważnych dla mnie konkretnie".

AI jako filtr jest często bardziej wartościowy niż AI jako generator. Mamy za dużo informacji, nie za mało.

Case Study 8 — Wizytówka do bazy kontaktów w 30 sekund: CardFlow AI



Problem: Stosy wizytówek po wydarzeniach networkingowych. Przepisywanie danych ręcznie, brak follow-up, kontakty które przepadają.

Rozwiązanie: Zdjęcie wizytówki -> Claude Vision OCR -> kontakt w Google Sheets -> automatyczny powitalny email.

Wynik: Zero ręcznego przepisywania. Kontakt w bazie i follow-up wysłany zanim wyjdziesz z parkingu.

Problem

Networking to inwestycja. Dostajesz wizytówkę, ściskasz dłoń, obiecujesz się odezwać. Wracasz do biura z piętnastoma wizytówkami w kieszeni i stosem bieżących zadań.

Tydzień później wizytówki są gdzieś na biurku. Follow-up nie poszedł. Kontakt przepadł.

Ręczne przepisywanie wizytówki do CRM lub arkusza to 2-3 minuty per kontakt. Piętnaście wizytówek = 30-45 minut. Tej pracy najczęściej nie ma kiedy zrobić — więc nie jest robiona.

Rozwiązanie: trzy warianty

Projekt ma trzy warianty implementacji w kolejności rosnącej złożoności. Zaczynam od najprostszego.

Wariant C — Make.com MVP (rekomendowany start)

Czas budowania: 4-6 godzin. Zero VPS, zero kodu.

Flow:

- Wyślij email ze zdjęciem wizytówki na dedykowany adres (np. cards@cashify.eu)
- Make: Email Watch -> gdy nowy email z załącznikiem zdjęcia
- Make: HTTP request do Claude API z obrazem (Vision) + prompt ekstrakcji
- Claude zwraca JSON: imię, nazwisko, firma, stanowisko, email, telefon, strona
- Make: Google Sheets Add Row — nowy kontakt w bazie
- Make: Gmail Send — email powitalny do kontaktu

Konfigurowalny per workspace: inne konto email, inny arkusz, inna treść maila. Każdy dział Cashify

może mieć swój kanał wizytówek.

Wariant A — Telegram Bot

Zdjęcie wysłane do bota Telegram -> Claude Vision OCR -> Sheets + Brevo.

Wygodniejszy niż email — Telegram jest zawsze pod ręką. Bot odpowiada podglądem wyciągniętych danych przed zapisem (potwierdzenie że OCR zadziałał poprawnie).

Wariant B — Flutter App

Dedykowana aplikacja mobilna: aparat -> rozpoznanie -> baza kontaktów + email. Najlepsza UX, najdłuższy czas budowania. Do zbudowania po walidacji rynku przez wariant C lub A.

Kluczowy element — Claude Vision do OCR

Klasyczne OCR (Tesseract, Google Vision) dobrze radzi sobie z drukowanym tekstem w standardowym układzie. Wizytówki bywają kreatywne — tekst pod kątem, grafiki w tle, kilka języków, ręczne dopiski.

Claude Vision rozumie kontekst. Wie że „CEO” to stanowisko a nie imię, nawet jeśli jest napisane większą czcionką. Radzi sobie z częściowo zasłoniętym tekstem, niestandardowymi układami, polskimi znakami.

Prompt który działa:

```
Z załączonego zdjęcia wizytówki wyciągnij dane kontaktowe.
Zwróć JSON z polami: first_name, last_name, company, title, email, phone, website.
Jeśli pole nie istnieje na wizytówce — null.
Nie zgaduj danych których nie ma na zdjęciu.
```

JSON jest czysty i bezpośrednio zapisywalny do Sheets lub CRM.

Email powitalny — automatyczny ale nieautomatyczny

Kluczowa decyzja projektowa: czy email powitalny wysyłać automatycznie czy pokazywać podgląd do zatwierdzenia?

Automatyczny: szybciej, zero uwagi wymaganej, ryzyko że wyślesz komuś kiepski email gdy OCR się pomyli.

Z podglądem: sekundę uwagi, pełna kontrola, naturalny moment do personalizacji („super rozmawiać o X”).

Rekomendacja dla Wariantu C (Make): automatyczny z prostym szablonem. Dla Wariantu A (Telegram bot): podgląd przed wysłaniem — Telegram to interfejs gdzie naturalnie klikasz zatwierdzenie.

Schemat przepływu (Wariant C)

```
Telefon: zrób zdjęcie wizytówki
        ↓
Email ze zdjęciem -> cards@cashify.eu
        ↓
Make: Email Watch (trigger)
        ↓
Make: HTTP -> Claude Vision API
  prompt: wyciągnij dane kontaktowe, zwróć JSON
        ↓
Make: Parse JSON response
        ↓
Make: Google Sheets -> Add Row
  (imię, nazwisko, firma, email, telefon, data, źródło)
        ↓
Make: Gmail -> Send Email
  "Miło było poznać, [imię]. Chętnie porozmawiam o [temat]."
```

Potencjał biznesowy

CardFlow jako produkt SaaS: 49-99 zł/mc per workspace.

Target: handlowcy, networkingowcy, właściciele firm którzy aktywnie chodzą na eventy. W Polsce kilka tysięcy takich osób — wystarczy na rentowny mikro-SaaS.

Cashify: gotowy klient wewnętrzny. Każdy handlowiec i PM ma problem z wizytówkami.

Zasada: nie buduj platformy zanim masz trzech płacących klientów. Najpierw wariant C w Make, sprzedaj ręcznie, potem automatyzuj sprzedaż i onboarding.

Co bym zrobił inaczej

Zacząłbym sprzedawać zanim skończyłem budować. Wariant C można zbudować w jeden dzień — zamiast planować warianty A i B przez tygodnie, lepiej wdrożyć C dla jednego klienta i zobaczyć czy chce płacić.

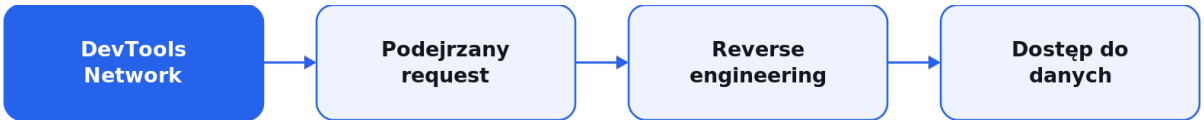
Walidacja rynku ważniejsza niż perfekcja produktu.

Lekcja

Claude Vision to nie tylko rozpoznawanie tekstu. To rozumienie kontekstu wizualnego — co jest ważne, co jest dekoracją, jaka struktura ma sens.

Jeśli masz zadanie które wymaga patrzenia na zdjęcie i wyciągania danych — zanim budujesz specjalizowane ML — sprawdź czy Claude Vision nie robi tego wystarczająco dobrze. Często robi.

Case Study 9 — Detektyw w DevTools: jak znaleźć API którego oficjalnie nie ma



Problem: Strona internetowa ma dane które chcę — ale nie ma publicznego API i scraping HTML jest kruchy.

Rozwiązanie: DevTools Network tab, obserwacja requestów przeglądarki, znalezienie ukrytego REST API.

Wynik: Dostęp do ustrukturyzowanych danych bez parsowania HTML. Szybszy, stabilniejszy, prostszy kod.

Skąd pomysł

Przy budowaniu Cashify Gold Analityk potrzebowałem cen rynkowych monet z serwisu NumiTracker.

NumiTracker nie ma publicznego API. Dokumentacja: nie istnieje. Support: brak odpowiedzi.

Opcje:

- Scraping HTML — parsuj strukturę strony, wyciągaj dane z divów i tabel
- Ręczne przepisywanie — wykluczone
- Znaleźć jak strona sama pobiera swoje dane

Wybrałem opcję 3. Zajęło 15 minut.

Jak działa każda nowoczesna strona internetowa

Strona internetowa to nie jednolity dokument. To aplikacja która:

- Ładuje szkielet HTML (pusty lub z podstawową strukturą)
- Wykonuje JavaScript
- JavaScript wysyła requesty do API po dane
- Dane wracają jako JSON
- JavaScript wstawia dane do DOM (tego co widzisz)

Gdy otwierasz stronę z cenami — ceny nie są „w HTML". Są pobierane przez JavaScript z API. Widoczne w przeglądarce, niewidoczne w źródle strony.

Ale wszystkie requesty które robi JavaScript są rejestrowane w DevTools.

Technika krok po kroku

Krok 1: Otwórz DevTools

Chrome/Firefox: F12 lub prawy klik -> Zbadaj.

Przejdź na zakładkę **Network**.

Krok 2: Odśwież stronę

Odśwież stronę z otwartymi DevTools. Network rejestruje KAŻDY request który strona wysyła — pliki JS, CSS, obrazki, i co nas interesuje: wywołania API.

Krok 3: Filtruj po typie

W filtrze Network wybierz **XHR** lub **Fetch** — to requesty do API, nie pliki statyczne.

Krok 4: Szukaj JSON

Klikaj po kolei requesty. W zakładce Response szukaj odpowiedzi w formacie JSON — listy obiektów z polami takimi jak `name`, `price`, `id`. Gdy znajdziesz — to Twoje API.

Krok 5: Skopiuj URL

Prawy klik na request -> Copy -> Copy link address. Wklej do nowej zakładki przeglądarki. Jeśli widzisz JSON — masz endpoint.

Krok 6: Sprawdź headers

Czasem endpoint wymaga nagłówków autoryzacyjnych (tokeny, klucze API). Sprawdź zakładkę Headers w DevTools — czy request wysyłany przez stronę zawiera Authorization lub inne specjalne nagłówki. Jeśli tak — musisz je odtworzyć w swoim kodzie.

NumiTracker: brak autoryzacji. Endpoint dostępny bez żadnych nagłówków. Idealny przypadek.

Kod w Pythonie

```
import requests

url = "https://admin.metalapi.co.pl/numitracker/api/v1/products"

response = requests.get(url)
products = response.json()

for product in products:
    print(f"{product['name']}: {product['price']} PLN")
```

Sześć linii. Zamiast setek linii parsowania HTML.

Kiedy ta technika działa i kiedy nie

Działa gdy:

- Strona ładuje dane dynamicznie przez JavaScript (99% nowoczesnych stron)
- API nie wymaga autoryzacji lub wymaga tokenu który możesz wyciągnąć z requestu
- Endpoint jest stabilny (nie zmienia się przy każdym deployu)

Nie działa gdy:

- Strona renderuje dane server-side (stary HTML, bez JS) — tu parsuj HTML
- API wymaga autoryzacji której nie możesz odtworzyć (OAuth z refresh tokenami, certyfikaty)
- Cloudflare lub inna ochrona blokuje requesty z datacenter IP

Kwestia etyczna i prawna

Korzystasz z API które istnieje na serwerze właściciela strony, ale nie jest publiczne. To szara strefa — nie włamanie, ale też nie oficjalny dostęp.

Zasady których się trzymam:

- Nie przeciążaj serwera (rate limiting — przerwy między requestami)
- Używaj danych do własnych potrzeb, nie do konkurencyjnego produktu
- Jeśli właściciel serwisu wyraźnie zabrania — nie rób tego (sprawdź ToS i robots.txt)
- Nie odsprzedawaj danych

W przypadku NumiTracker: używam danych do wewnętrznych raportów Cashify. Bez rate limitingu na ich stronie, bez zakazu w ToS.

Inne miejsca gdzie ta technika jest użyteczna

- Aplikacje mobilne: DevTools w Chrome nie pokazuje ruchu aplikacji mobilnych, ale Charles Proxy lub mitmproxy tak
- Webowe panele bez API: systemy wewnętrzne firm często mają nieudokumentowane endpointy
- Agregatory cen: porównywarki cenowe, giełdy — wszystkie pobierają dane z API

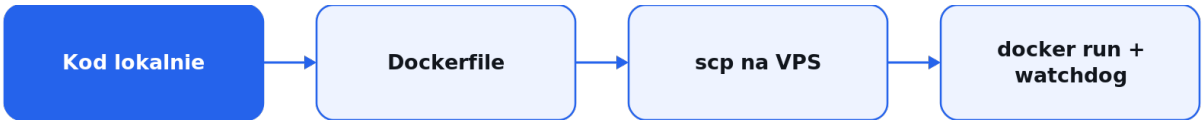
Każda strona która pokazuje Ci dane musi je skądś pobrać. Twoje zadanie: znaleźć skąd.

Lekcja

Zanim zaczniesz parsować HTML — otwórz DevTools. Pięć minut obserwacji może zaoszczędzić dni pracy przy scraping HTML który pęka przy każdej zmianie layoutu.

Myśl jak przeglądarka, nie jak użytkownik.

Case Study 10 — Bot który nie śpi: deploy na VPS z Dockerem



Problem: Bot działa lokalnie na laptopie. Wyłączasz komputer — bot pada. Chcesz żeby działał 24/7 bez Twojej interwencji.

Rozwiązanie: VPS za ~60 zł miesięcznie + Docker + docker-compose. Raz ustawiony, działa sam.

Wynik: Wszystkie boty i automatyzacje które wymagają ciągłości działają na VPS. Laptop wyłączony — system działa.

Problem

Pierwszy Wysprzątani scraper uruchomiłem lokalnie na MacBooku. Działał. Zbierał posty, klasyfikował, wysyłał na Telegram.

Problem ujawnił się w nocy. Zamknąłem laptop — scraper padł. Posty z grup nocnych przepadły. Rano musiałem uruchamiać ręcznie.

Każda automatyzacja która ma działać „zawsze” potrzebuje środowiska które działa zawsze. Laptop nie jest tym środowiskiem.

VPS (Virtual Private Server) to.

Co to jest VPS

Wynajmujesz kawałek serwera w centrum danych. Dostajesz: dedykowany CPU, RAM, dysk, stały IP, system operacyjny (Linux). Serwer działa 24/7, nawet gdy Twój komputer jest wyłączony.

Mój VPS: OVH, Ubuntu 22.04, 4 GB RAM, 2 vCPU, 60-80 zł miesięcznie. Wystarcza na kilka kontenerów Docker działających równolegle.

SSH: łączę się przez terminal na MacBooku — `ssh ubuntu@TWOJ_VPS_IP`. Pełna kontrola nad serwerem z dowolnego miejsca.

Co to jest Docker

Docker to technologia konteneryzacji. Kontener to izolowane środowisko z wszystkim czego

potrzebuje Twoja aplikacja: systemem operacyjnym, bibliotekami Python, zmiennymi środowiskowymi.

Dlaczego kontenery zamiast instalowania bezpośrednio na VPS:

- Izolacja — bot A nie zakłóca bota B, nawet jeśli używają różnych wersji Pythona
- Przenośność — ten sam kontener działa lokalnie i na VPS bez zmian
- Łatwy restart — docker restart moj-bot zamiast debugowania procesu
- Łatwy deploy — docker pull + docker-compose up = nowa wersja w 30 sekund

---

Minimalna struktura projektu

```
projekt/
 Dockerfile          # jak zbudować kontener
 docker-compose.yml  # jak uruchomić
 main.py             # Twój kod
 requirements.txt    # biblioteki Python
 .env               # sekrety (nie commituj!)
```

Dockerfile:

```
FROM python:3.11-slim

WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .

CMD ["python", "main.py"]
```

docker-compose.yml:

```
version: '3.8'
services:
  bot:
    build: .
    restart: always
    env_file: .env
    volumes:
      - ./logs:/app/logs
```

`restart: always` — jeśli kontener padnie (błąd, restart VPS), Docker uruchamia go automatycznie.

Deploy krok po kroku

Na laptopie:

```
# Zbuduj i przetestuj lokalnie
docker-compose up --build

# Skopiuj pliki na VPS (bez sekretów w git)
scp .env ubuntu@TWOJ_VPS_IP:/home/ubuntu/projekt/
```

Na VPS (przez SSH):

```
# Sklonuj repo lub skopiuj pliki
git clone [REPO_URL] && cd projekt

# Lub scp jeśli bez git

# Uruchom w tle
docker-compose up -d --build

# Sprawdź czy działa
docker-compose ps
docker-compose logs -f
```

Od tego momentu bot działa. Możesz wylogować się z SSH — kontener działa dalej.

Watchdog — monitoring kontenerów

Kontener może paść z powodów zewnętrznych — OOM (brak pamięci), błąd sieci, timeout. `restart: always` restartuje go automatycznie, ale nie informuje Cię o problemie.

W Wysprzątani zbudowałem `watchdog.py` uruchamiany przez cron co 20 minut na hoście VPS (poza kontenerami):

```
import subprocess
import requests

containers = ['wysprzatani-scraper', 'wysprzatani-handler']

for container in containers:
    result = subprocess.run(
        ['docker', 'inspect', '--format', '{{.State.Status}}', container],
        capture_output=True, text=True
    )
    status = result.stdout.strip()
```

```
    if status != 'running':
        # Restart kontenera
        subprocess.run(['docker', 'restart', container])
        # Alert Telegram
        send_telegram_alert(f"(!) {container} był {status} — zrestartowano")
```

Cron na hoście (nie w kontenerze):

```
# crontab -e
*/20 * * * * python3 /home/ubuntu/watchdog.py
```

Sekrety i bezpieczeństwo

Nigdy nie commituj sekretów do git. Klucze API, tokeny Telegram, hasła do baz danych — wszystko w `.env`.

`.gitignore`:

```
.env
credentials.json
*.key
```

Na VPS: plik `.env` kopiowany przez `scp`, nie przez git. Każdy serwer ma własny `.env`.

Dostęp SSH: wyłącz logowanie hasłem, używaj tylko kluczy SSH.

```
# Na VPS, w /etc/ssh/sshd_config:
PasswordAuthentication no
PubkeyAuthentication yes
```

Koszty

Element	Koszt
VPS OVH (4GB RAM)	~60-80 zł/mc
Domena (opcjonalna)	~50 zł/rok
Cloudflare (DNS + SSL)	0 zł
Razem	~60-80 zł/mc

Za tę cenę możesz uruchomić kilkanaście kontenerów — skraperów, botów, API, serwisów.

Co bym zrobił inaczej

Dodałbym logi do zewnętrznego systemu od początku. Teraz logi są w plikach na VPS — gdy chcę sprawdzić co się działo tydzień temu, loguję się przez SSH i grep. Prymitywne.

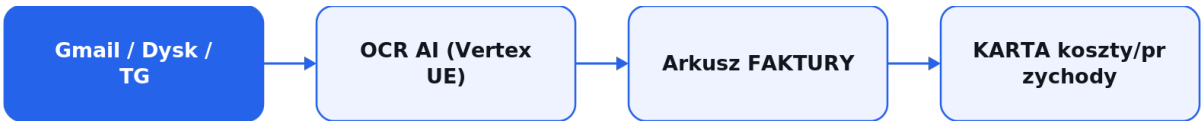
Lepsza opcja: logi do Google Sheets lub Logtail — przeglądalne z przeglądarki, bez SSH.

Lekcja

VPS + Docker to fundament każdej poważnej automatyzacji. Nie „programistyczna” decyzja — operacyjna.

Jeśli Twoja automatyzacja ma działać gdy śpisz, gdy jesteś na urlopie, gdy laptop jest wyłączony — potrzebujesz VPS. Koszt: kawa dziennie. Wartość: spokój że system działa.

Case Study 11 — Księgowa, która nie przepisuje faktur: Faktury Auto



Problem: Każda faktura kosztowa i przychodowa pięciu spółek musi trafić do arkusza księgowego. Ręcznie: otwórz PDF, przepisz NIP, numer, daty, kwoty, przypisz kategorię. Dziesiątki pozycji miesięcznie, mnóstwo błędów.

Rozwiązanie: Bot OCR — zdjęcie/PDF/e-mail faktury -> Claude Vision wyciąga wszystkie pola -> wpis do Google Sheets gotowy do akceptacji. Dwa tory: koszty i przychody. Na końcu dashboard liczy rentowność każdej spółki.

Wynik: Koniec ręcznego przepisywania. Księgowa tylko sprawdza i akceptuje jednym kliknięciem. Test na żywo: OCR sam rozróżnił, kto wystawił fakturę, a kto za nią płaci.

Problem

Cashify to grupa pięciu spółek (Interior Capital, Mennica Cashify, Cashify Estate, Keszuś, Wymień Bitcoina). Każda generuje faktury kosztowe (co kupujemy) i przychodowe (co sprzedajemy). Wszystko musi wylądować w arkuszu księgowym z poprawną kategorią, spółką i kwotami.

Przed automatyzacją wyglądało to tak: ktoś otwierał PDF albo zdjęcie faktury, przepisywał ręcznie nazwę kontrahenta, NIP, numer dokumentu, datę wystawienia, termin płatności, kwoty netto/VAT/brutto, dobierał kategorię ze słownika i wklejał wiersz do tabeli.

Przy kilkudziesięciu fakturach miesięcznie to godziny mozolnej, bezmyślnej pracy. A każde ręczne przepisanie to okazja do literówki w NIP-ie albo pomyłki o przecinek w kwocie.

To klasyczne zadanie pod automatyzację: dużo powtarzalności, zero kreatywności, wysoki koszt błędu.

Rozwiązanie

System ma trzy kanały wejścia i dwa tory przetwarzania.

Trzy kanały — wrzuć fakturę jak Ci wygodnie:

- Telegram — wyślij zdjęcie lub PDF do bota
- Google Drive — wrzuć plik do folderu INBOX
- E-mail — prześlij na dedykowany adres (filtr Gmail -> Apps Script -> folder INBOX)

Serce systemu — Gemini Vision OCR (Vertex AI, region UE):

Każdy plik trafia do Gemini 2.5 Flash przez Vertex AI — serwery w Europie (Frankfurt), dane nie opuszczają UE. Prompt każe zwrócić czysty JSON:

```
{
  "czy_faktura": true/false,
  "sprzedawca": "...", "nip": "...", "nr_faktury": "...",
  "data_wystawienia": "RRRR-MM-DD", "termin_platnosci": "...",
  "netto": 0.0, "vat": 0.0, "brutto": 0.0, "waluta": "PLN",
  "kategoria": "...", "podkategoria": "...", "opis": "..."
}
```

Model czyta zdjęcie tak, jak człowiek — nie obchodzi go, że każdy dostawca ma inny układ faktury. To przewaga AI nad starym podejściem z szablonami i wyrażeniami regularnymi: nie trzeba pisać osobnej reguły na każdego kontrahenta.

Prompt zawiera też instrukcję minimalizacji danych: model ma wyciągać tylko pola potrzebne do księgowania. Numer konta bankowego (IBAN) jest jawnie wykluczony — niepotrzebny do wpisania kosztu, więc nie ma sensu go przetwarzać i przechowywać.

Mądry filtr i deduplikacja:

Najpierw pytanie `czy_faktura?` — jeśli ktoś wrzuci raport albo wniosek urlopowy zamiast faktury, system to pomija (bez śmieciowego wpisu). Potem dedup po parze NIP + numer faktury — ta sama faktura wrzucona dwa razy nie zdubluje się w arkuszu.

Człowiek w pętli — KARTA akceptacji:

Bot nigdy nie księguje „na ślepo”. Wpisuje fakturę ze statusem „do akceptacji”. Księgowa otwiera zakładkę KARTA, wybiera numer faktury, widzi wszystkie pola w czytelnym układzie pionowym, poprawia ewentualne błędy OCR i klika jeden checkbox — „ZAAKCEPTUJ”. Wiersz robi się zielony i znika z kolejki.

To kluczowa zasada przy finansach: AI przygotowuje, człowiek zatwierdza.

Dwa tory: koszty i przychody

Faktura kosztowa i przychodowa to dwie różne historie, więc system traktuje je osobno.

Tor kosztów — faktury, które ktoś nam wystawił. Interesuje nas sprzedawca, kategoria kosztu, do której spółki należy.

Tor przychodów — faktury, które MY wystawiliśmy klientowi. Tu uwaga: model musi wyciągnąć **nabywcę** (kto płaci), a nie sprzedawcę (czyli nas). To było główne ryzyko przy prompcie przychodowym.

Test na żywo rozwiał wątpliwości. Wygenerowaliśmy przykładową fakturę sprzedażową: sprzedawca „Mennica Cashify", nabywca „Złoty Inwestor", kwoty 24 390,24 / 5 609,76 / 30 000 zł. OCR wpisał poprawnie nabywcę jako płatnika (nie pomylił go z naszą spółką), rozpoznał spółkę wystawiającą i trafnie dobrał źródło przychodu. Pełny przepływ — od pliku do gotowego wiersza — zajął kilkadziesiąt sekund.

Architektura wielo-plikowa

Zamiast jednego molocha — trzy osobne arkusze Google Sheets:

Plik	Rola
Faktury Auto	koszty (tabela + kolejka + KARTA + słownik kategorii)
Przychody Auto	przychody (osobny tor, osobna KARTA)
Dashboard Finansowy	rentowność 5 spółek przez IMPORTRANGE (koszty + przychody)

Rozdzielenie ma sens praktyczny: tor przychodów jest całkowicie odizolowany od działających kosztów, więc nowe funkcje można wdrażać bez ryzyka, że coś zepsuje sprawdzony przepływ kosztów. A na końcu dashboard sumuje jedno i drugie i pokazuje dochód oraz marżę każdej spółki.

RODO jako wymaganie projektowe, nie formalność

Faktury zawierają dane osobowe — imię, nazwisko, adres, NIP osób fizycznych prowadzących działalność (JDG). Kantor krypto to firma regulowana, więc podejście do prywatności musiało być przemyślane.

Dwie decyzje, które zrobiły różnicę:

Po pierwsze — dane zostają w UE. OCR działa przez Vertex AI Google z wymuszonym regionem `europe-west3` (Frankfurt). Treść faktur trafia na serwery europejskie, nie do USA. To konkretna różnica prawna: Google podpisuje z firmami umowę powierzenia pod prawo UE (CDPA), a region UE gwarantuje, że dane nie wyjadą za granicę EOG — nawet tranzytem.

Po drugie — minimalizacja od początku. Numer konta bankowego (IBAN) widoczny na fakturze nie jest potrzebny do księgowania. Zamiast zbierać go „na wszelki wypadek", model dostał jawną

instrukcję: IBAN pomiń. Nawet gdyby model go wyciągnął, kod usuwa go z wyniku OCR zanim cokolwiek trafi do arkusza.

Efekt: w Sheets ląduje tylko to, co rzeczywiście potrzebne. Mniej danych = mniejsze ryzyko naruszenia, mniej do chronienia.

Element, który zaskakuje — autoryzacja bez klucza

Standardowo bot łączy się z Google przez Service Account (konto-robot z własnym kluczem). Tyle że polityka organizacji Workspace blokowała tworzenie kluczy Service Account ze względów bezpieczeństwa.

Rozwiązanie: OAuth 2.0 w trybie użytkownika. Bot działa „jako Hubert" — ma dostęp do tych samych plików co właściciel konta, bez generowania osobnego klucza. Token odświeżający trzymany jest poza repozytorium, a ekran zgody ustawiony na „Wewnętrzny", żeby token nie wygasał.

Lekcja na przyszłość: gdy polityka firmy blokuje jedną drogę uwierzytelnienia, prawie zawsze jest druga — trzeba ją znać.

System, który się uczy

Bot ma prostą pamięć kategorii per kontrahent. Jeśli dany NIP był już raz zaksięgowany i zweryfikowany przez księgową, przy kolejnej fakturze od tego samego dostawcy system od razu podpowiada tę samą, zatwierdzoną wcześniej kategorię — zamiast zgadywać od nowa.

To nie jest uczenie maszynowe w sensie trenowania modelu. To prosta, skuteczna pętla: poprawka człowieka staje się domyślną podpowiedzią na przyszłość. Im dłużej system działa, tym mniej poprawek wymaga.

Schemat przepływu

- Faktura wpada kanałem: Telegram / Drive / e-mail
- Claude Vision czyta plik i zwraca JSON z polami faktury
- Filtr czy_faktura? odrzuca dokumenty, które fakturą nie są
- Dedup po NIP + numer — duplikat pomijany
- Wpis do arkusza ze statusem „do akceptacji" + link do pliku
- Księgowa sprawdza w KARCIE, poprawia, klika ZAAKCEPTUJ
- Wiersz zielony, znika z kolejki
- Dashboard przelicza rentowność spółek

Wynik

Koniec ręcznego przepisywania faktur. Człowiek przeszedł z roli „przepisywacza” do roli „kontrolera jakości” — sprawdza i zatwierdza, zamiast wklepywać.

Przed: dziesiątki faktur miesięcznie przepisywanych ręcznie, godziny pracy, ryzyko błędu w każdej pozycji.

Po: faktura wpada dowolnym kanałem, w kilkadziesiąt sekund jest w arkuszu z wypełnionymi polami, czeka tylko na jedno kliknięcie akceptacji. Dwa tory (koszty + przychody) domykają obraz finansowy, a dashboard liczy rentowność z realnych, zatwierdzonych danych.

Co bym zrobił inaczej

Od początku rozdzieliłbym pliki na koszty i przychody. Pierwsza wersja trzymała wszystko w jednym arkuszu i każda zmiana groziła zepsuciem działającego toru. Rozbicie na trzy pliki przyszło później — szybciej byłoby zaprojektować tak od razu.

Lekcja

Przy automatyzacji finansów najważniejsza zasada to **człowiek w pętli**. AI świetnie czyta fakturę i przygotowuje wpis, ale ostatnie kliknięcie „akceptuję” musi należeć do człowieka. Nie chodzi o brak zaufania do modelu — chodzi o to, że w księgowości koszt błędu jest realny, a jeden checkbox akceptacji to tania polisa.

Druga lekcja: AI do odczytu dokumentów bije stare podejścia, gdy dokumenty są różnorodne. Nie piszesz szablonu na każdego dostawcę — model czyta każdą fakturę jak człowiek. To zmienia ekonomię całej klasy zadań typu „przepisz dane z dokumentu do tabeli”.

Trzecia lekcja: **RODO to wymaganie projektowe, nie papierologia.** Wybór regionu serwera i lista pól do pominięcia to decyzje techniczne podjęte raz na początku. Koszt — kilka linii konfiguracji. Korzyść — zgodność z prawem bez audytorskich niespodzianek.

Case Study 12 — Kasa w 23 oddziałach bez Excela: Gotowka_Oddzialy

Problem: Cashify Gold prowadzi 23 oddziały rozlokowane w całej Polsce. Każdy przechowuje gotówkę i metale szlachetne. Dotychczas stan kasy każdego oddziału był wpisywany ręcznie do arkusza raz w tygodniu — bez historii operacji, bez kontroli transferów między oddziałami, bez możliwości weryfikacji czy pieniądze w drodze faktycznie dotarły.

Rozwiązanie: Google Apps Script panel webowy osadzony w Google Sheets. Każdy oddział loguje się na konto @cashify.eu i widzi tylko swój oddział. Centrala widzi wszystko. Transfery gotówki mają status „w drodze" dopóki drugi oddział nie potwierdzi odbioru. Wpłaty depozytów z centrali automatycznie wchodzą do DZIENNIKA i MAGAZYNU — stan kasy zgadza się bez ręcznego wpisu.

Wynik: Koniec z jednorazowym tygodniowym odczytem stanu. Każda operacja jest append-only w DZIENNIKU — historia jest niezniszczalna. Transfer 25 000 zł nie znika ze stanu dopóki Bydgoszcz nie kliknie „Potwierdź odbiór". Depozyty metali od razu widać w stanach magazynowych.

Problem

Cashify Gold to sieć 23 oddziałów skupu krypto i złota w Polsce. Każdy oddział codziennie przyjmuje gotówkę, wydaje gotówkę i czasem transferuje środki między lokalizacjami — np. gdy jeden oddział ma nadwyżkę, a drugi pilnie potrzebuje zasilenia.

Przed automatyzacją wyglądało to tak:

- Stan gotówki wpisywany ręcznie do arkusza, raz w tygodniu lub rzadziej
- Transfer: telefonicznie — „wysyłamy Ci 25 tys." — bez żadnego śladu w systemie
- Inwentaryzacja metali: osobna tabela, bez powiązania ze stanem gotówki
- Depozyty z centrali: osobna kolumna, bez związku z historią operacji

Efekt: nikt nie wiedział w czasie rzeczywistym ile jest w kasie w Białymstoku. Nie wiadomo było czy transfer z Gdańska do Krakowa doszedł. Różnica między stanem w arkuszu a fizycznym liczeniem gotówki potrafiła przekraczać kilkadziesiąt tysięcy złotych.

Zasada projektowa: stan nigdy nie jest wpisywany — jest liczony

To najważniejsza decyzja całego projektu.

W poprzednim systemie ktoś wpisywał „38 500 zł" — i nikt nie wiedział skąd ta liczba.

W nowym systemie: STANY to zakładka tylko do odczytu. Wartość w kolumnie „Stan bieżący" to wynik formuły:

```
= saldo_otwarcia + SUMIF(DZIENNIK; oddział; wpływy) − SUMIF(DZIENNIK; oddział; wypływy)
  + SUMIF(DZIENNIK; oddział_docelowy; transfery_przychodzące)
```

Nie ma ręcznego wpisywania stanów. Jest tylko append-only DZIENNIK — każda operacja to nowy wiersz. Pomyłka nie jest edycją, tylko wierszem „Korekta" z kwotą odwrotną.

Dzięki temu:

- Historia jest niezniszczalna
- Stan można policzyć na dowolny dzień w przeszłości
- Ewentualną manipulację widać w historii wierszy

Architektura

```
Google Sheets (SHEET_ID: 1D5FeAEg...)
   DZIENNIK       <- append-only, 14 kolumn, 1500 wierszy
   STANY          <- tylko odczyt, formuły SUMIF
   ODDZIALY       <- lista oddziałów + saldo otwarcia
   MAGAZYN        <- stany metali per oddział (append-only)
   DEPOZYTY       <- depozyty z centrali (osobna ewidencja)
   OCR_BUFOR      <- skany paragonów czekające na akceptację

Apps Script WebApp (doGet -> HtmlService)
   Kod.gs         <- menu, formularze Google, sync list
   WebApp.gs      <- logika panelu, autoryzacja emailem
   Index.html     <- SPA panel webowy (vanilla JS)

OCR Pipeline (Tor B)
   VPS kasa-ocr -> n8n -> Vertex AI (gemini-2.5-flash, europe-west1, RODO)
      -> OCR_BUFOR -> akceptacja w panelu -> DZIENNIK
```

Panel webowy działa przez `google.script.run` — wywołania asynchroniczne z przeglądarki do Apps Script. Autoryzacja: `Session.getActiveUser().getEmail()` sprawdzone wobec tabeli MAPA_KONT. Konto @cashify.eu widzi tylko swój oddział. Konto admina widzi wszystkie.

Transfer dwustronny bez podwójnych wierszy

Klasyczny problem przy transferach między oddziałami: jeden wiersz czy dwa?

Podejście z dwoma wierszami (jeden „wypływ" u źródła, jeden „wpływ" u celu) brzmi naturalnie — ale prowadzi do niespójności. Co jeśli jeden wiersz zostanie omyłkowo skasowany? Transfer będzie „połówkowy" — pieniądze znikną albo się podwoją.

Decyzja projektowa: **jeden wiersz na transfer**.

```
Oddział: Białystok | Typ: Transfer | Cel: Bydgoszcz | Wypływ: 25 000
```

STANY u Białegostoku liczy ten wiersz jako wypływ. STANY u Bydgoszczy ma oddzielną formułę, która szuka wierszy gdzie CEL = Bydgoszcz i Typ = Transfer — i liczy je jako wpływ.

Ale tylko jeśli Status = „odebrane". Dopóki status to „w drodze" — pieniądze nie są u nikogo. Panel Bydgoszczy pokazuje baner „Do potwierdzenia odbioru TR-0001 · od Białystok · 25 000 zł" z przyciskiem „Potwierdź". Kliknięcie zmienia status i wpływ pojawia się w STANACH Bydgoszczy.

Pułapka sandboxa Apps Script

Apps Script osadza panel HTML przez `HtmlService`. To środowisko z restrykcjami:

- alert() — nie działa. Wywołanie jest ignorowane bez błędu.
- window.location.reload() — nie działa bezpośrednio. Trzeba użyć google.script.run po stronie serwera lub nawigacji przez top.location.
- google.script.run — asynchroniczne, bez await. Obsługa błędów przez .withFailureHandler(), nie przez try/catch.

Pierwszy symptom był mylący: użytkownik klikał „Potwierdź odbiór" i nic się nie działo. Baner zostawał. Nie było błędu w konsoli.

Przyczyna: funkcja serwerowa zwracała `ok: false` dla transferu już oznaczonego „odebrane" — ale `withSuccessHandler` wołał `alert(r.msg)` zamiast zreloadować stronę. Alert był ignorowany. Strona stała nieruchomo.

Naprawa: dla transferu już potwierdzonego zwracać `ok: true` z komunikatem — i zawsze na `ok: true` reloadować stronę.

```
// WebApp.gs
if (status === 'odebrane')
    return { ok: true, msg: 'Transfer już był potwierdzony — odświeżam panel.' };

// Index.html
.withSuccessHandler(function(r) {
    if (r.ok) reloadTop();   // zawsze reload, nie alert
    else showError(r.msg);
})
```

Depozyty i spójność stanów

Centrala wrzuca do oddziału 50 000 zł w gotówce i 2 Krugerrandy. To „depozyt" — nie jest to operacja kasowa oddziału, tylko zasób który centrala przekazuje.

Pierwsze podejście: depozyt ląduje w zakładce DEPOZYTY z kwotą i statusem „Aktywny". Panel

pokazuje tabelę „Depozyt Cashify Gold” z kolumnami: Typ | Kwota | Stan bieżący | Różnica.

Problem: „Stan bieżący” pochodzi z STANY, które czyta DZIENNIK. Ale depozyt nie tworzył wpisu w DZIENNIKU. Więc „Stan bieżący” wynosił −25 000 (bo oddział wysłał transfer bez żadnego wpływu), a „Różnica” wychodziła −75 000.

```
Różnica = Stan bieżący (−25 000) − Kwota depozytu (50 000) = −75 000
```

Lekcja: dwie ewidencje (DEPOZYTY i DZIENNIK) mogą pokazywać sprzeczne dane jeśli nie są zsynchronizowane.

Rozwiązanie: `dodajDepozytPanel` przy zapisie do DEPOZYTY automatycznie tworzy wpis w DZIENNIKU:

```
// Gotówka -> wpis w DZIENNIKU
var dzRow = new Array(14).fill('');
dzRow[D_DATA-1]   = now;
dzRow[D_ODDZ-1]   = oddzial;
dzRow[D_TYP-1]    = 'Wpłata gotówki';
dzRow[D_WPL-1]    = iloscN;
dzRow[D_OPIS-1]   = 'Depozyt CG · ' + id;
dzRow[D_STATUS-1] = 'zaakceptowana';
dz.appendRow(dzRow);

// Metal -> wpis w MAGAZYNIE
mag.appendRow([now, oddzial, produkt, iloscN, 'szt', '', '', 'wpływ', 'Depozyt CG · ' +
id, email]);
```

Po tej zmianie: Różnica dla nowych depozytów = 0. STANY i DEPOZYTY są spójne.

OCR paragonów z RODO

Oddziały wrzucają skany paragonów do podfolderów Drive (`KASA_INBOX/O01_Białystok/`). n8n monitoruje foldery co kilka minut i wysyła każdy plik do Gemini 2.5 Flash przez Vertex AI — region europe-west1, dane w UE.

Prompt OCR jest kasowy: wyciąga datę, kwotę, rodzaj operacji, numer dokumentu. IBAN i dane klientów są pomijane w promptcie (zasada minimalizacji danych).

Wynik ląduje w OCR_BUFOR ze statusem „do akceptacji”. Oddział widzi bufor w panelu i zatwierdza lub odrzuca jeden wiersz. Dopiero po zatwierdzeniu wpis trafia do DZIENNIKA.

Bufor ma dedup po nazwie pliku — ten sam paragon nie pojawi się dwa razy nawet jeśli n8n przetworzy go ponownie.

Wynik

23 oddziały mogą wpisywać operacje przez panel webowy — każde na swoim koncie Google Workspace. Centrala widzi realny stan gotówki w całej sieci w arkuszu STANY.

Transfer pieniędzy między oddziałami ma pełen ślad: kto wysłał, kiedy, ile, kiedy odebrano. Depozyt od centrali od razu pojawia się w STANACH oddziału.

OCR eliminuje ręczne przepisywanie paragonów. Kasjer robi zdjęcie — AI czyta — kasjer tylko zatwierdza.

Co bym zrobił inaczej

Zacząłbym od salda otwarcia przed pierwszą operacją, a nie po. Wpisanie pierwszej operacji przy saldzie = 0 i dopiero potem uzupełnienie salda otwarcia daje przez chwilę fałszywy obraz w STANACH. Oddziały widziały ujemne salda i pytały „czy coś jest nie tak” — a były to tylko brakujące salda otwarcia.

Lekcja

Nigdy nie pytaj „jaki jest stan?” — pytaj „jaka jest historia operacji?”. Stan to wynik obliczeń na historii. Gdy historia jest append-only, stan jest zawsze policzalny. Gdy stan jest wpisywany ręcznie, historia nie istnieje.

Append-only log to fundament każdego systemu finansowego — nie tylko w bankowości. Nawet w arkuszu Google.

Case Study 13 — Formularz MiCA bez backendu: kantor.cashify.eu

Problem: MiCA — nowe europejskie regulacje krypto — wymaga od kantorów zebrania danych klientów przed wymianą. Cashify potrzebował strony do zbierania leadów z wysyłką potwierdzenia e-mail. Proste zadanie. Ale firma działa na Google Workspace z włączonymi politykami organizacyjnymi, które blokują Service Accounts i Apps Script przed komunikacją z zewnętrznymi API. Google Forms nie wysyła maili z własnego adresu. Wysyłka przez Gmail API z konta organizacji wymagała zgody admina Workspace.

Rozwiązanie: Cloudflare Worker jako backend. Zero Google. Formularz HTML wysyła POST do workera, worker woła Brevo API, Brevo wysyła mail z szablonu. Strona hostowana przez Cloudflare Pages, domena kantor.cashify.eu skierowana przez Workers Routes.

Wynik: Landing page działa na produkcji. Klient wypełnia formularz, dostaje potwierdzenie e-mail z odpowiednim szablonem (zależnym od metody płatności). Cały backend to 150 linii JavaScript bez serwera, bez bazy danych, bez kosztów.

Problem

Cashify Gold to kantor krypto. MiCA (Markets in Crypto-Assets Regulation) to europejska regulacja wchodząca w życie w 2024–2025, która nakłada na kantory obowiązek zebrania danych klientów i potwierdzenia transakcji.

Konkretna potrzeba: strona `kantor.cashify.eu` z formularzem — klient podaje imię, e-mail, kwotę i metodę płatności (gotówka / bitomat / przelew / mix), klika „Wyślij” i dostaje na maila potwierdzenie rezerwacji z instrukcją co dalej.

Na papierze: 2 godziny pracy.

W praktyce: org policy Google Workspace zablokował każde podejście przez Google:

- Apps Script + MailApp — MailApp wysyła jako osobiste konto użytkownika, nie jako noreply@cashify.eu
- Apps Script + Gmail API — wymaga OAuth z kontem organizacji, admin Workspace musi wyrazić zgodę
- Service Account — org policy constraints/iam.disableCrossProjectServiceAccountUsage blokuje SA z projektów poza org
- Google Cloud Functions — projekt na koncie prywatnym (poza org cashify.eu), dostęp do Workspace niemożliwy przez org policy

Każde rozwiązanie przez Google rozbiło się o jeden z tych blokerów.

Odkrycie: Cloudflare Worker jako backend

Cloudflare Workers to środowisko uruchomieniowe JavaScript działające na edge serwerach Cloudflare. Bez serwera, bez kontenerów, bez VPS. Piszesz funkcję, wdrażasz przez dashboard lub CLI, działa.

Kluczowe cechy:

- Bezpłatny plan: 100 000 requestów dziennie
- Czas zimnego startu: ~0 ms (kod żyje w pamięci edge)
- Dostęp do fetch() — może wołać dowolne zewnętrzne API
- Bez Google, bez org policy, bez admina Workspace

```
// Cloudflare Worker — cashify-mica-api
export default {
  async fetch(request, env) {
    if (request.method === 'OPTIONS') return corsResponse();
    const data = await request.json();
    const result = await sendBrevoEmail(data, env.BREVO_API_KEY);
    return Response.json({ ok: result.ok });
  }
}
```

Worker dostaje dane formularza przez POST, wywołuje Brevo API, zwraca wynik. Formularz na stronie woła worker przez `fetch()`. Klient dostaje mail. Koniec.

Architektura

```
Przeglądarka klienta
   kantor.cashify.eu (Cloudflare Pages)
         index.html — formularz (HTML+CSS+vanilla JS)
         fetch POST -> cashify-mica-api.workers.dev

Cloudflare Worker (cashify-mica-api)
   Parsuje JSON z formularza
   Wybiera szablon Brevo (wg pola "dotychczas" + "priorytet")
   POST -> api.brevo.com/v3/smtp/email

Brevo
   Wysyła mail z szablonu -> klient
      (noreply@cashify.eu, DKIM/SPF skonfigurowane)
```

Routing: Workers Routes w Cloudflare przekierowują `kantor.cashify.eu/api/*` do workera, a `kantor.cashify.eu/*` do statycznej strony na Cloudflare Pages.

Szablony maili zależne od kontekstu

Klient może chcieć wymienić krypto za gotówkę w okienku, przez bitomat, przelewem bankowym lub miksem tych metod. Każda metoda ma inne instrukcje co dalej.

Brevo pozwala tworzyć edytowalne szablony (drag-and-drop, bez kodu). Wdrożone 4 szablony przez Brevo API:

- gotowka — instrukcja wizyty w oddziale, lista adresów
- bitomat — instrukcja obsługi bankomatu Bitcoin
- przelew — dane bankowe i instrukcja przelewu
- mix — instrukcja łączona

Worker wybiera szablon na podstawie pola formularza:

```
function selectTemplate(dotychczas, priorytet) {
  if (dotychczas === 'nigdy' || priorytet === 'szybkosc') return TEMPLATES.gotowka;
  if (priorytet === 'wygoda') return TEMPLATES.przelew;
  if (dotychczas === 'bitomat') return TEMPLATES.bitomat;
  return TEMPLATES.mix;
}
```

Fallback: jeśli żaden warunek nie pasuje, worker wysyła inline HTML zamiast szablonu — żeby użytkownik zawsze dostał maila, nawet gdyby Brevo miało problem z szablonami.

Problem z IP whitelist Brevo

Brevo ma opcję bezpieczeństwa: whitelist IP. Gdy włączona, API akceptuje tylko requesty z określonych adresów IP.

Problem: Cloudflare Workers nie mają stałego IP. Edge serwery Cloudflare to tysiące węzłów na całym świecie — każdy request może wyjść z innego adresu. Whitelist IP jest niemożliwa do skonfigurowania.

Rozwiązanie: wyłączyć whitelist IP w Brevo na stałe. To decyzja projektowa — bezpieczeństwo zapewnia unikalny klucz API w zmiennych środowiskowych Workera (`env.BREVO_API_KEY`), a nie filtrowanie IP.

```
# wrangler.toml
[vars]
BREVO_API_KEY = ""  # ustawiony w dashboard Cloudflare, nie w pliku

[[routes]]
pattern = "kantor.cashify.eu/api/*"
zone_name = "cashify.eu"
```

Klucz API nigdy nie trafia do repozytorium — żyje wyłącznie w Cloudflare Secrets.

Logo i typografia bez zewnętrznych zależności

Strona musi działać bez CDN — Cloudflare CSP blokuje requesty do zewnętrznych hostów. Logo, fonty, ikony — wszystko musi być inline lub base64.

Logo Cashify Gold: zakodowane jako base64 w pliku HTML.

```
<img src="..." alt="Cashify Gold" style="height:48px">
```

Fonty systemowe zamiast Google Fonts:

```
font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', sans-serif;
```

Strona jest self-contained — jeden plik HTML, zero zewnętrznych zależności, działa offline.

Wynik

Landing page `kantor.cashify.eu` działa na produkcji. Formularz zbiera dane lead (imię, telefon, email, kwota, metoda), klient dostaje potwierdzenie z odpowiednim szablonem w ciągu kilku sekund.

Koszty: $0/mc (Cloudflare Workers bezpłatny plan, Brevo bezpłatny plan do 300 maili dziennie).

Czas budowy: 1 dzień — głównie na rozgryzienie dlaczego Google nie działa i przejście na Cloudflare.

Co bym zrobił inaczej

Od razu sprawdziłbym org policy przed projektowaniem rozwiązania. 4 podejścia przez Google, 4 blokery — każde kosztowało czas. Jeden `gcloud organizations list-policies` na początku pokazałby ograniczenia i oszczędził 3 godziny debugowania OAuth.

Zasada: w środowisku korporacyjnym zawsze najpierw sprawdź co org policy blokuje, zanim zaprojektujesz architekturę.

Lekcja

Org policy w Google Workspace to niewidoczna ściana. Technicznie masz konto Google, masz projekt GCP, masz Apps Script — ale administrator może zablokować dowolną kombinację funkcji na poziomie organizacji.

Gdy Google nie działa przez polityki organizacyjne — nie walcz z Google. Użyj czegoś co nie zależy od Google. Cloudflare Workers, Vercel Edge Functions, AWS Lambda — każde z nich to 150 linii kodu i zero zależności od firmowego SSO.

Wielkie platformy mają świetne narzędzia, ale też świetne mechanizmy blokowania. Zawsze miej plan B poza ekosystemem korporacyjnym.

Case Study 14 — Własny ekspert od zdrowia: RAG na 190 tysiącach fragmentów + dane z WHOOP

Problem: Najlepsza wiedza o zdrowiu, śnie i regeneracji jest rozsiana po setkach godzin podcastów (Attia, Huberman, Walker). Nie da się tego przeszukać. A dane z opaski WHOOP — sen, HRV, obciążenie — żyją osobno w aplikacji producenta i nie rozmawiają z tą wiedzą.

Rozwiązanie: Baza wektorowa (RAG) zbudowana z transkryptów podcastów zdrowotnych + integracja WHOOP przez OAuth. Bot Telegram odpowiada na pytania cytując konkretne źródła, a na życzenie łączy odpowiedź z Twoimi realnymi danymi ze snu i regeneracji.

Wynik: Prywatny asystent zdrowotny działający jak native AI. Pyta się go po ludzku, dostaje odpowiedź z cytatami z ekspertów i — jeśli chce — analizę własnego snu z ostatniej nocy. Cała inferencja i embeddingi w UE (RODO).

Problem

Słucham podcastów o zdrowiu od lat. Peter Attia o długowieczności, Andrew Huberman o neurobiologii, Matthew Walker o śnie. To setki godzin konkretnej, popartej badaniami wiedzy.

Problem: ta wiedza jest niewyszukiwalna. Gdy chcę wiedzieć „co Attia mówił o treningu strefy 2?” — muszę pamiętać w którym z 359 odcinków to było. Notatki nie pomagają, bo nie sposób zanotować wszystkiego.

Drugi problem: noszę opaskę WHOOP, która mierzy sen, HRV (zmienność rytmu serca) i dobowe obciążenie. Te dane są w aplikacji WHOOP — ale aplikacja pokazuje wykresy, nie odpowiada na pytania. I nie ma pojęcia o tym, co mówią eksperci.

Chciałem jedno miejsce: zadaję pytanie po polsku, dostaję odpowiedź opartą na wiedzy ekspertów **i** na moich własnych danych.

Skąd wiedza: transkrypty jako źródło

Zebrałem transkrypty publicznych odcinków podcastów zdrowotnych. Rozkład materiału:

```
Peter Attia        × 359 odcinków
Andrew Huberman    × 344
```

```
Matthew Walker      × 183
Rhonda Patrick      ×  50
Andy Galpin         ×  20

Razem: 956 plików tekstowych / ~108 MB surowego transkryptu
```

Surowy transkrypt jest bezużyteczny dla wyszukiwania — to ściana tekstu. Trzeba go pociąć na fragmenty (chunki) i zamienić każdy na wektor liczb, który opisuje jego znaczenie.

Po chunkowaniu: **193 158 fragmentów**. Każdy zamieniony na embedding (wektor) i zapisany w bazie.

Architektura

```
Transkrypty (956 TXT)
    chunking -> 193 158 fragmentów
           embedding (Vertex AI, region UE — RODO)
                  PostgreSQL + pgvector  (kontener rag-postgres, :5433)

Pytanie użytkownika
    API /ask (:5010)
           embedding pytania (Vertex UE)
           wyszukanie najbliższych fragmentów w pgvector
           Gemini (UE) generuje odpowiedź + CYTATY ze źródeł

WHOOP
    whoop_service.py (:5011) — OAuth2 v2, multi-user
           mapowanie użytkownika po tg{id}

Bot zespołu Telegram  @healthagent_ai_matic_works_bot
    /polacz    -> OAuth WHOOP (podpięcie konta)
    /dzis      -> dzisiejsze dane (sen, HRV, recovery)
    /analiza   -> dane WHOOP + kontekst z RAG
    /zdrowie   -> pytanie do bazy wiedzy

Publiczny front: https://zdrowie.aimatic.works
    OVH (rekord A) + nginx + certbot (Let's Encrypt)
```

Sercem jest **pgvector** — rozszerzenie PostgreSQL, które pozwala trzymać wektory i szukać „najbliższych znaczeniowo” jednym zapytaniem SQL. Nie potrzeba osobnej, płatnej bazy wektorowej. Zwykły Postgres w kontenerze Docker udźwignął 193 tysiące fragmentów.

Dlaczego RAG, a nie „zapytaj ChatGPT”

Model językowy sam z siebie halucynuje przy szczegółach — potrafi przypisać cytat niewłaściwej osobie albo zmyślić badanie. Dla wiedzy zdrowotnej to niedopuszczalne.

RAG (Retrieval-Augmented Generation) odwraca kolejność: **najpierw** wyszukaj prawdziwe fragmenty z bazy, **potem** każ modelowi odpowiedzieć wyłącznie na ich podstawie i podać źródło.

Efekt: odpowiedź kończy się listą „na podstawie: Huberman #12, Attia #211”. Można kliknąć i sprawdzić. Model nie wymyśla — streszcza to, co realnie padło w podcaście.

To jest różnica między „AI, które brzmi mądrze” a „AI, któremu można zaufać w temacie zdrowia”.

Dane w UE — bo to dane wrażliwe

Dane zdrowotne to najwyższa kategoria wrażliwości w RODO. Dlatego cały pipeline jest w Unii:

- Embeddingi transkryptów i pytań: Vertex AI, region europejski
- Generowanie odpowiedzi: Gemini przez Vertex, UE
- Baza wektorowa: własny kontener na VPS w OVH (Francja), nie zewnętrzna usługa
- Dane WHOOP: pobierane po OAuth, mapowane po ID Telegrama, nie przez modele trenowane na danych

Żaden fragment transkryptu ani żaden pomiar snu nie opuszcza infrastruktury UE.

Pułapka: Gemini ucinał odpowiedzi

Pierwsza wersja działała, ale odpowiedzi urywały się w połowie zdania. Wyglądało to jak limit tokenów — ale limit był ustawiony wysoko.

Przyczyna okazała się nieoczywista: parametr `thinkingBudget`. Gemini w nowych wersjach rezerwuje część budżetu na „myślenie” przed odpowiedzią. Przy krótkim limicie całość szła na rozumowanie, a na samą odpowiedź nie zostawało nic — model kończył w połowie.

Naprawa: `thinkingBudget: 0` dla tego zastosowania. Nie potrzebuję widocznego łańcucha rozumowania — potrzebuję pełnej odpowiedzi z cytatami.

Druga pułapka była po stronie Telegrama: limit 4096 znaków na wiadomość. Długie odpowiedzi z cytatami przekraczały go i API zwracało błąd. Rozwiązanie: funkcja `send_long`, która dzieli odpowiedź na kawałki na granicy akapitu.

Wynik

Piszę do bota `/zdrowie jak długi powinien być sen głęboki?` — dostaję odpowiedź opartą na Walkerze i Hubermanie, z numerami odcinków.

Piszę `/analiza` — bot bierze moje dane WHOOP z nocy (recovery, HRV, czas snu) i zestawia je z wiedzą z bazy: „Twoje HRV dziś niżej niż średnia — w kontekście tego, co mówi Attia o regeneracji, rozważ lżejszy trening”.

Publiczny adres `https://zdrowie.aimatic.works` działa na certyfikacie Let's Encrypt. System jest multi-user — każdy podpina swój WHOOP przez `/polacz`, dane są rozdzielone po ID Telegrama.

Koszt: jeden kontener Postgres i wywołania Vertex — grosze przy tej skali, bo embeddingi liczone są raz, a zapytania są tanie.

Co bym zrobił inaczej

Chunkowałbym mądrzej od początku. Pierwsze cięcie było „co N znaków” — przez co fragment potrafił się urwać w środku myśli i tracił sens jako wektor. Lepsze jest cięcie po zdaniach z zakładką (overlap), żeby każdy chunk był zamkniętą całością. Przy 193 tysiącach fragmentów przeliczenie embeddingów od nowa to nie jest pięciominutowa poprawka.

Lekcja

RAG demokratyzuje ekspertyzę. Nie musisz pamiętać 900 odcinków podcastów — musisz umieć je przeszukać znaczeniowo. Baza wektorowa w zwykłym Postgresie wystarcza do skali, o której jeszcze rok temu myślałbym „to wymaga dedykowanej platformy”.

A prawdziwa wartość pojawia się na styku dwóch światów: gdy sucha wiedza ekspertów spotyka Twoje własne, żywe dane. Wtedy AI przestaje być encyklopedią, a staje się doradcą.

Case Study 15 — Life OS: osobisty system operacyjny na dwie pory dnia

Problem: Poranek zaczynał się od rytuału klikania — ceny spot złota i krypto, ceny detaliczne w polskich kantorach, kilka portali z newsami. Piętnaście minut przeklikiwania, zanim w ogóle zacząłem myśleć. Wieczorem żadnego domknięcia dnia.

Rozwiązanie: Dwa zaplanowane przebiegi n8n. **7:30 — Poranny Raport Rynkowy:** ceny spot + detal PL + newsy z RSS, streszczone przez Claude Haiku, wysłane na Telegram. **21:00 — Wieczorny Check-in:** podsumowanie i pytania zamykające dzień. Wszystko przez jednego bota.

Wynik: Codzienny rytm bez klikania. O 7:30 mam gotowy briefing rynkowy w telefonie, o 21:00 domknięcie dnia. Przy okazji skonsolidowałem starszy projekt (SIGNAL) — zamiast dwóch osobnych botów mam jeden system.

Problem

Prowadzę tradingową część głowy na kilku rynkach — złoto, srebro, BTC, ETH. Do tego jestem PM w sieci kantorów, więc ceny detaliczne w polskich kantorach też są dla mnie danymi operacyjnymi, nie ciekawostką.

Poranny rytuał wyglądał tak: otwórz stronę ze spotem, otwórz drugą z cenami detalicznymi, przejrzyj Bankier, zajrzyj na portal krypto. Piętnaście minut, zanim zacząłem cokolwiek robić. Co gorsza — robiłem to nieregularnie, więc czasem przegapiałem ruch, który miał znaczenie.

Wieczorem nie miałem nic. Dzień się po prostu kończył, bez podsumowania, bez refleksji co zadziałało.

Chciałem osobisty system operacyjny: coś, co samo przynosi mi to, co rano potrzebuję, i samo pyta wieczorem o to, co warto domknąć.

Historia wcześniejsza: projekt SIGNAL

To nie był mój pierwszy briefing. Wcześniej (Case Study 7) zbudowałem SIGNAL — prosty digest newsów: RSS -> Claude -> Telegram, jeden raz dziennie.

SIGNAL działał, ale był wyspą. Osobny bot, osobny cron, tylko newsy — bez cen, bez wieczornej pory, bez związku z resztą mojego dnia.

Zamiast utrzymywać dwa osobne narzędzia, które robią „prawie to samo”, **wchłonąłem SIGNAL do Life OS jako sekcję NEWSY** porannego raportu. Jeden bot, jeden przepływ, dwie pory dnia. Ta konsolidacja to ważniejsza decyzja niż jakikolwiek pojedynczy fragment kodu — o tym jest lekcja na końcu.

Architektura

```
n8n (self-hosted, kontener n8n-lifeos, :5678)

  Cron 7:30   Poranny Raport Rynkowy
      ceny SPOT (złoto, srebro, BTC, ETH)
      ceny DETAL PL (kantory)
      NEWSY: RSS Bankier + bitcoin.pl   <- dawny SIGNAL
      wszystko -> Claude Haiku (streszczenie) -> Telegram

  Cron 21:00  Wieczorny Check-in
      podsumowanie dnia + pytania domykające -> Telegram

Wyjście: bot @hq_hubert_bot  (jeden bot na oba przebiegi)
```

Dwa niezależne Crony w tym samym workflow. Każdy zbiera swoje dane, przepuszcza przez Claude Haiku (tani, szybki model — do streszczania idealny) i wysyła gotowy tekst na Telegram.

Host to `n8n-lifeos` — dedykowany kontener n8n, oddzielony od pozostałych automatyzacji, żeby awaria jednego projektu nie kładła briefingów.

Dlaczego Haiku, a nie większy model

Streszczanie to zadanie, w którym mały model błyszczy. Nie potrzebuję rozumowania na poziomie doktoratu — potrzebuję: weź pięć źródeł, wyciągnij to, co istotne, napisz zwięźle po polsku.

Haiku robi to za ułamek kosztu i w ułamku czasu większego modelu. Przy dwóch przebiegach dziennie przez cały miesiąc rachunek jest symboliczny. Dobór modelu do zadania to jedna z najważniejszych decyzji kosztowych w automatyzacji z AI — nie każdy krok potrzebuje najmocniejszego modelu.

Pułapka: martwy bot zwracał 401

Pierwszy przebieg po migracji nie dotarł. n8n pokazywał sukces węzła, ale wiadomość nie przychodziła.

Przyczyna: briefing wołał starego bota Life OS, którego token był już nieważny — Telegram zwracał `401 Unauthorized`. n8n traktował odpowiedź HTTP jako „zadanie wykonane” (bo request poszedł), nie sprawdzając kodu statusu.

To klasyczna pułapka: **węzeł HTTP kończy się sukcesem, gdy request wyszedł — nie gdy druga strona go zaakceptowała.** Sukces w n8n ≠ dostarczona wiadomość.

Naprawa dwuczęściowa: przełączenie na żywego bota `@hq_hubert_bot` (ten sam, przez który idzie panel HQ) i dodanie sprawdzania kodu odpowiedzi, żeby `401` był realnym błędem, a nie cichym „sukcesem”.

Wynik

O 7:30 telefon wibruje: spot złota i krypto, ceny detaliczne w kantorach, sekcja NEWSY z Bankiera i bitcoin.pl — wszystko streszczone w kilku akapitach. Zero klikania.

O 21:00 przychodzi check-in domykający dzień.

SIGNAL przestał istnieć jako osobny byt — jego funkcja żyje dalej jako sekcja NEWSY porannego raportu. Zamiast dwóch botów utrzymuję jeden system na jednym hoście.

Co bym zrobił inaczej

Zamiast budować SIGNAL i Life OS osobno, a potem je scalać, od początku zaprojektowałbym „system dnia” i traktował newsy jako jedną z jego sekcji. Konsolidacja po fakcie kosztowała mnie przepisanie przepływu i migrację bota. Gdybym od razu myślał produktem, a nie pojedynczą funkcją, oszczędziłbym ten krok.

Lekcja

Automatyzacje mają tendencję do rozmnażania się w wyspy — każdy pomysł to nowy bot, nowy cron, nowy token do pilnowania. Po kilku miesiącach masz zoo narzędzi, które robią „prawie to samo”.

Prawdziwa dojrzałość automatyzacji to nie budowanie kolejnego narzędzia — to odwaga, żeby scalić trzy istniejące w jedno. Mniej ruchomych części, mniej martwych tokenów, mniej rzeczy, które mogą po cichu paść. System bije zbiór skryptów.

Rozdział 11 — Jak wybrać swój pierwszy projekt

Najczęstszy błąd przy starcie z automatyzacją: wybór projektu który jest „fajny” zamiast projektu który rozwiązuje realny ból.

Fajny projekt: bot który generuje haiku po kliknięciu przycisku.
Realny ból: czterdzieści minut tygodniowo na kopiowanie danych między dwoma systemami.

Zrób to drugie.

Matryca: czas × trudność × wartość

Przed każdym projektem oceniam trzy wymiary:

Czas oszczędzony (lub ryzyko wyeliminowane)

- Ile czasu zajmuje to teraz ręcznie, tygodniowo?
- Lub: jaka jest wartość ryzyka które eliminuję?

Trudność budowania

- Czy mam dostęp do danych źródłowych (API, formularz, email)?
- Czy logika jest prosta (zawsze to samo) czy złożona (zależy od kontekstu)?
- Jak długo zajmie zbudowanie MVP?

Wartość po wdrożeniu

- Czy to działa przez miesiące bez utrzymania?
- Czy problem który rozwiązuje jest stały?

Szukam projektów z wysokim czasem oszczędzonym, niską trudnością budowania i wysoką wartością po wdrożeniu.

---

5 pytań które zadaję sobie przed każdym projektem

1. Co konkretnie robię i jak często?

Nie „zarządzam komunikacją” — „co poniedziałek kopiuję raporty sprzedaży z Excela do arkusza Google i wysyłam email do pięciu osób”. Konkret.

2. Czy każdy przypadek jest identyczny?

Jeśli „to zależy” pojawia się rzadziej niż raz na dwadzieścia przypadków — do automatyzacji. Jeśli częściej — zacznij od tej 80% reguły i obsługuj wyjątki ręcznie.

3. Kto korzysta z wyniku?

Automatyzuję dla siebie, dla współpracownika, dla klienta? Każdy ma inne oczekiwania co do niezawodności i interfejsu.

4. Co się stanie gdy automatyzacja padnie?

Alert compliance który pada = nieodkryta transakcja = problem prawny. Email powitalny który

pada = klient nie dostał maila = niedobrze ale nie tragedia.

Skala odpowiedzialności musi pasować do skali konsekwencji.

5. Jak zmierzę sukces?

„Oszczędza czas" to za mało. „Skraca tygodniowy raport z 45 do 5 minut" — to mierzalne. Bez miary nie wiesz czy zadziałało.

Gdzie szukać pierwszego projektu

Obserwuj co robisz przez tydzień

Notatka po każdej czynności którą robisz drugi raz. Na koniec tygodnia masz listę kandydatów.

Zapytaj kolegów

„Co robisz codziennie czego nie lubisz robić?" Najczęstsze odpowiedzi: kopiowanie danych, przypominanie, formatowanie raportów, sortowanie emaili.

Szukaj „przekaźników"

Ktoś dostaje informację i przekazuje ją dalej bez żadnej modyfikacji. Pracownik który bierze email klienta i przesyła do biura rachunkowego. To automatyzacja.

Szukaj „ręcznych synchronizacji"

Dane w systemie A i systemie B które ktoś ręcznie utrzymuje spójne. Każda ręczna synchronizacja to automatyzacja.

Pierwsze trzy projekty Huberta

Nie planowałem żadnego z nich. Każdy wyrósł z konkretnego bólu:

- Formularze Webflow — event + 60 zapisów + dwa dni przy skrzynce. Zbudowany w popołudnie.
- Compliance alert — jeden email który o mało nie przeszedł niezauważony. Zbudowany następnego dnia.
- Kryptopogotowie Telegram — współpracowniczka nie odpowiada na maile dość szybko. Zbudowany gdy problem stał się widoczny.

Żaden z nich nie był na liście „projektów AI do zbudowania". Wszystkie były odpowiedzią na ból który już istniał.

Jak nie wybierać projektu

Nie zaczynaj od największego bólu

Największy ból często ma największą złożoność. Jeśli twój największy ból to „chaotyczna komunikacja w całej firmie" — to projekt na kilka miesięcy, nie na tydzień.

Zacznij od bólu który możesz rozwiązać w dwa-trzy dni. Zdobądź doświadczenie, pewność siebie, jeden działający system — potem idź po większy problem.

Nie zaczynaj od projektu dla kogoś innego jako pierwsze

Pierwsze projekty mają błędy. Lepiej żebyś Ty był pierwszym klientem — uczysz się na własnym systemie, nie na czyimś.

Nie czekaj na idealny pomysł

Idealny projekt nie istnieje. Dobry projekt istnieje wszędzie wokół Ciebie — w każdej powtarzalnej czynności którą robisz w tym tygodniu.

Szablon oceny projektu

```
Projekt: [nazwa]
Problem: [co konkretnie, jak często]
Dane wejściowe: [skąd, w jakim formacie]
Dane wyjściowe: [dokąd, w jakim formacie]
Logika: [zawsze to samo / zależy od X]
Trudność: [1–5]
Wartość: [czas/tydzień lub ryzyko]
Narzędzie: [Make / n8n / Python / inne]
MVP w: [X godzin/dni]
Miernik sukcesu: [konkretna liczba]
```

Wypełnij dla trzech kandydatów. Wybierz ten z najlepszym stosunkiem wartości do trudności. Potem zacznij. Nie planuj dalej.

Rozdział 12 — Jak wyceniać automatyzacje

Zbudujesz coś co działa. Ktoś zapyta ile kosztuje. I staniesz przed pytaniem na które nikt Cię nie przygotował.

Automatyzacje są trudne do wyceny bo ich wartość jest asymetryczna. Budujesz raz, oszczędzasz przez lata. Klient płaci raz, a potem nie ma rachunku — automatyzacja po prostu działa.

To brzmi jak problem wyceny. W rzeczywistości to przewaga.

Trzy modele rozliczenia

Model projektowy (jednorazowy)

Wyceniasz i dostarczasz. Klient płaci raz za zbudowany system.

Dobre gdy:

- Projekt ma jasno zdefiniowany zakres
- Klient jest techniczny i weźmie system „pod opiekę”
- To pierwsza współpraca, budujesz zaufanie

Złe gdy:

- Klient będzie chciał zmian — a będzie chciał zawsze
- Utrzymanie jest krytyczne (monitoring, odnawianie tokenów API, aktualizacje)

Typowe stawki w Polsce (2026):

- Prosta automatyzacja Make (1-3 scenariusze): 1 000 – 3 000 zł
- Średnia automatyzacja (routing, AI, Sheets): 3 000 – 8 000 zł
- Bot Telegram + scraper + deploy VPS: 5 000 – 15 000 zł
- Kompleksowy system (wiele modułów, integracje): 15 000 zł+

Liczby orientacyjne — realnie zależy od złożoności i klienta.

Model retainerowy (miesięczny)

Płatność miesięczna za dostępność, utrzymanie i rozwój.

Dobre gdy:

- System wymaga monitorowania i interwencji
- Klient będzie chciał nowych funkcji
- Budujesz długoterminową relację

Typowe stawki:

- Utrzymanie prostego systemu: 500 – 1 500 zł/mc
- Utrzymanie + rozwój: 1 500 – 4 000 zł/mc

- Pełna obsługa automatyzacji firmy: 4 000 zł+/mc

Wysprzątani CRM: 600 zł/mc tech fee + 10% od leadów. Model hybrydowy — retainer za utrzymanie + udział w wartości którą system generuje.

Model SaaS (subskrypcja produktu)

Budujesz produkt, klient subskrybuje dostęp.

Dobre gdy:

- Ten sam problem ma wielu klientów
- Produkt wymaga minimalnej customizacji
- Możesz obsługiwać dziesiątki klientów bez proporcjonalnego wzrostu pracy

Przykład: CardFlow AI @ 79 zł/mc per workspace. Zbudujesz raz, sprzedajesz wielu.

Trudność: wymaga więcej pracy na start (onboarding, płatności, support), ale skala liniowa.

Zasada: nie buduj SaaS zanim nie masz trzech płacących klientów na modelu projektowym lub retainerowym. Najpierw waliduj że ktoś chce płacić, potem automatyzuj sprzedaż.

Jak liczyć wycenę projektową

Metoda 1: Godziny × stawka

Oszacuj godziny, pomnóż przez stawkę. Problem: zawsze za mało. Automatyzacje mają dużo „niewidocznej pracy" — research, testy, debugowanie, dokumentacja.

Multiply by 1.5–2x swojego pierwszego oszacowania.

Metoda 2: Wartość dla klienta

Ile oszczędza automatyzacja w skali roku?

Przykład: pracownik za 5 000 zł/mc spędza 2h dziennie na ręcznych czynnościach -> to 25% etatu -> 1 250 zł/mc -> 15 000 zł/rok.

Automatyzacja która to eliminuje jest warta przynajmniej 50% rocznej oszczędności = 7 500 zł.

Budujesz ją w tydzień? Twoja stawka wychodzi dobrze. Budujesz ją w dwie godziny? Wychodzi świetnie.

Metoda 3: Precedens rynkowy

Sprawdź co podobne firmy biorą za podobne projekty. Upwork, LinkedIn, polskie grupy automatyzacji.

Nie zaniżaj żeby „zdobyć klienta" — zaniżona cena to sygnał że nie wierzysz w wartość swojej pracy.

Jak rozmawiać z klientem który „chce AI"

Klienci często przychodzą z „chcę AI” bez wiedzy co konkretnie to znaczy. Twoje zadanie: zamienić „chcę AI” w „rozwiązuję ten konkretny problem”.

Pytania które zadaję:

„Co konkretnie robiecie ręcznie?”
Nie „co chcecie zautomatyzować” — to za abstrakcyjne. „Co konkretnie, krok po kroku, robicie ręcznie.” Lista procesów.

„Kto to robi i ile czasu zajmuje?”
Imię i stanowisko + godziny tygodniowo. To pozwala policzyć wartość.

„Co się stanie jeśli to nie zadziała?”
Czy to wygoda czy biznesowa konieczność? Od tego zależy poziom niezawodności który musisz zapewnić — i cena.

„Jakie systemy już używacie?”
CRM, ERP, arkusze, email — lista. Każda integracja to dodatkowy scope.

„Jak wygląda sukces za 3 miesiące?”
Mierzalna odpowiedź. Jeśli klient nie potrafi odpowiedzieć — projekt nie jest gotowy na wycenę.

Błędy przy wycenach których nauczyłem się na błędach

Wyceniaj zakres, nie czas

„Zbuduję Ci automatyzację za X złotych” jest lepsze niż „biorę Y złotych za godzinę”. Klient płaci za wynik, nie za Twój czas.

Pisz co jest w zakresie i co nie jest

„Automatyzacja formularzy na Webflow” — tak. „Integracja z Twoim CRM który poznałem dzisiaj” — osobna wycena.

Scope creep to zjawisko universalne. Jedyny sposób żeby go powstrzymać: jasny zakres na piśmie przed rozpoczęciem.

Pobieraj zaliczkę

50% przed startem, 50% po dostarczeniu. Chroni Cię przed klientem który znika po skończonej pracy.

Utrzymanie to osobny kontrakt

„Zbuduję i zostawię” to zawsze kłopoty — coś padnie, klient zadzwoni, Ty poprawiasz za darmo. Jeśli chcesz utrzymywać — retainer. Jeśli nie — jasno napisz że po odbiorze to klient odpowiada za system.

Realne stawki w Polsce 2026

Trudno podać jedno miejsce bo rynek jest rozproszony. Z obserwacji i rozmów:

- Freelancer „no-code” (Make, Zapier): 50-120 zł/h lub 1 000-5 000 zł za projekt
- Freelancer „low-code + Python”: 80-150 zł/h lub 3 000-15 000 zł za projekt

- Agencja automatyzacji: 150-300 zł/h
- Konsultant AI dla korporacji: 250-500 zł/h

Jeśli zaczynasz: zacznij od dolnych widełek dla pierwszych 2-3 klientów żeby zbudować portfolio i referencje. Potem podnoś.

Nie pracuj za darmo „dla referencji". Zrób jeden projekt za 30% rabatu w zamian za case study i rekomendację — to uczciwa wymiana.

Zakończenie — Co dalej

Zacząłem od sześćdziesięciu zapisów do newslettera i dwóch dni przy skrzynce mailowej.

Skończyłem na autonomicznych systemach działających 24/7, pierwszej napisanej aplikacji mobilnej i 180 dniach nauki której nikt mi nie zlecił.

Droga między tymi dwoma punktami nie była prosta. Były projekty które umarły śmiercią naturalną (Nietypowe Święta). Były VPS-y które postawiłem i skasowałem. Były automatyzacje które działały przez tydzień i padły bez wyjaśnienia. Były godziny debugowania błędu który okazał się literówką.

Wszystko to jest normalne. Niczego z tego nie znajdziesz w tutorialach YouTube bo nikt nie nagrywa filmów o swoich porażkach.

Co naprawdę zmieniło mój sposób pracy

Nie narzędzie. Nie konkretna automatyzacja. Nie certyfikat Google.

Zmieniło się to jak patrzę na powtarzalne czynności.

Teraz gdy robię coś po raz drugi — automatycznie myślę: „czy muszę to robić trzeci raz, czy mogę to zautomatyzować?"

To nie jest myślenie programisty. To myślenie kogoś kto ma za mało czasu na robienie tych samych rzeczy w kółko. I kogoś kto wie że istnieją narzędzia które mogą mu pomóc.

Projekt 180 Dni

W dniu gdy piszę te słowa jestem w dniu 51. Projektu 180 Dni. Cel był prosty: przez 180 dni budować, uczyć się i dokumentować wszystko związane z AI i automatyzacją.

Nie wiem czy do dnia 180 zarobię na tym pieniądze. Wiem że wiem więcej niż na początku. Wiem że mam projekty które działają. Wiem że każdy następny projekt buduję szybciej niż poprzedni.

Kompetencja rośnie przez robienie, nie przez planowanie robienia.

Twój pierwszy krok

Jeśli jesteś na początku tej drogi, daj sobie jedno zadanie na ten tydzień:

Znajdź jedną czynność którą robisz regularnie i która jest w 100% powtarzalna. Wypisz ją na kartce: co robisz, skąd przychodzą dane, dokąd trafiają.

Potem otwórz Make.com (jest darmowy) i spróbuj to połączyć.

Nie musisz tego kończyć. Nie musisz tego wdrażać. Musisz tylko zacząć — i zobaczyć że to jest do ogarnięcia.

Pierwszy raz trwa najdłużej. Drugi raz jest dwa razy szybszy. Dziesiąty raz jest naturalny.

Bonus — Mój stack i zasoby

Narzędzia których używam:

- Make.com (start, integracje)
- n8n self-hosted (zaawansowane flows, AI agenci)
- Claude API (Haiku do klasyfikacji, Sonnet do generowania)
- Python + Playwright (scraping)
- Python + python-telegram-bot (boty Telegram)
- Firebase + Flutter (aplikacje mobilne)
- Docker + VPS OVH (infrastruktura 24/7)
- Google Sheets (baza danych dla mniejszych projektów)
- Cloudflare (DNS, ochrona, SSL)

Gdzie się uczyć:

- YouTube: kanały Make.com Official, Liam Ottley (n8n + AI), David Ondrej
- Dokumentacja Claude API: docs.anthropic.com
- n8n docs: docs.n8n.io
- Grupy Telegram/Facebook: „Automatyzacja biznesu PL”, „No-Code Polska”

Gdzie znajdziesz mnie:
aimatic.works/#contact · kontakt@aimatic.works

Jedno zdanie na koniec

Automatyzacja to nie wiedza dla programistów.

To wiedza dla każdego kto ma dość robienia tych samych rzeczy w kółko.

Zacząłem od kopiowania adresów email do Mailchimpa. Ty możesz zacząć od czegokolwiek co robisz teraz ręcznie.

Powodzenia.

Hubert Adamiak
Projekt 180 Dni — 2026
aimatic.works