> INSERT_COIN

WHAT RETRO GAMES TAUGHT ME ABOUT MICROSOFT FABRIC

▶ PRESS START TO CONTINUE

ŠTĚPÁN REŠL • Lead Technical Consultant @ **DataBrothers**

DATA GRILLEN 2026

IT STARTED WITH A MEME.

"Can it run DOOM?"

#01

PREGNANCY TEST

#02

SMART FRIDGE

#03

POTATO

#04

MICROSOFT FABRIC?

THIS SESSION

> a 30-year-old race to run DOOM on the most absurd hardware imaginable.

so...

CAN FABRIC RUN DOOM?

> YES.

(SPOILER ALERT: also Pokémon… and probably your toaster, if it had a Lakehouse.)

PLAYER 01



ŠTĚPÁN REŠL

(Steve)

LEAD TECHNICAL CONSULTANT

DATABROTHERS

ELEMENT	FABRIC / M / PYTHON
WEAPON	CRAZY IDEAS
QUEST	MAKE DATA FUN

SHORT DESCRIPTION

I work with Microsoft Fabric every day.

I help companies move from "we have data" to "we DO things with our data" — Lakehouses, Eventstreams, KQL, DAX, the usual suspects.

And in my spare time, I do this:

I make Fabric do things ***it was never supposed to do, or its APIs ;)***

Today's mission: show you what running DOOM and Pokémon in a Fabric notebook taught me about real-time data pipelines.

SELECT YOUR LEVEL

STAGE 01

1UP: DOOM

running an FPS where it shouldn't be

STAGE 02

CATCH 'EM ALL: POKÉMON YELLOW

live game state → Eventstream → dashboard

STAGE 03

BOSS FIGHT: A WILD PYSPARK APPEARED

why Python, not Spark, is your default

STAGE 04

POWER-UPS

lessons retro games teach modern Fabric

STAGE 05

CONTINUE?

what to build next

STAGE 01

1UP: DOOM

"running an FPS where it shouldn't be"

→ PRESS A TO START ←

DOOM RUNS ON EVERYTHING.

Why? Because the source code is small (~36k lines of C), portable (no fancy graphics APIs) and with best community!

 a literal potato

 a digital pregnancy test

 a smart fridge

 a piano keyboard

 an arcade marquee

 a baguette (yes, really)

Fabric notebook is just a Python kernel. So... why NOT?

ATTEMPT #1: THE NAIVE WAY

```
// code cell


import vizdoom as vzd
import matplotlib.pyplot as plt

game = vzd.DoomGame()
game.load_config("basic.cfg")
game.init()

for tic in range(1000):
    game.make_action([0,0,0,1])
    frame = game.get_state().screen_buffer
    plt.imshow(frame)
```

RESULT

0.3 fps

feels like 1995

but worse.

PROBLEM

matplotlib re-renders the entire figure every frame. Each plot is a fresh PNG. The kernel cries.
The fans spin. Your CU budget weeps.

ATTEMPT #2: GO HEADLESS

If the notebook can't host a window, give it a stream of pictures instead.

01 `SDL_VIDEODRIVER=dummy`

Tell the OS: "there's no display. so, don't care about it."

02 `frame → PNG → base64 → data: URI`

Encode each frame in-memory. No file I/O. Microseconds, not milliseconds.

03 `display_handle.update(HTML(<img>))`

IPython updates the SAME <img> element. No new cells. No flicker (almost). Can go up to 30-60 FPS.

Result: ~45 FPS inside a notebook cell (under certain conditions). No native window. No GPU. Just bytes.

LIVE DEMO #1

DOOM

INSIDE A FABRIC NOTEBOOK.

→ switch to notebook now ←

(if it crashes, blame the WiFi)

FIRST LESSON.

"A Fabric notebook isn't BI. It's a Python kernel with a Lakehouse next door."

— me, after staying up too late playing DOOM at work

STAGE 02

CATCH 'EM ALL

”Every game is a database. You just have to know where to read."

DOOM was fun.

But where are the DATA?

DOOM was a CPU benchmark. Cool. But we are at a data conference.
So... let's add a real-time pipeline to a **Game Boy**.



ENTER: POKÉMON YELLOW

EVERY GAME IS A DATABASE.

*Pokémon Yellow, on a Game Boy Color, in **2 megabytes** of ROM:*

151
Pokémon species

165
moves

44
bytes per Pokémon

19
bytes for the entire Pokédex

8 KB
battery save (SRAM)

6
active party slots — your CAPACITY

*A modern "basic" Python service eats more memory just printing **"hello world."***

WRAM IS A LIVE FEED.

Every Pokémon Yellow runtime variable lives at a known address in memory. PyBoy lets me read it. Live. Every frame.

```
// WRAM offsets — pokemon_yellow.py



 WRAM_PLAYER_NAME    = 0xD157   # 11 bytes
 WRAM_PARTY_COUNT    = 0xD162   # 1 byte
 WRAM_PARTY_DATA     = 0xD16A   # 6 × 44 bytes
 WRAM_POKEDEX_OWNED = 0xD2F6   # 19 bytes (151 bits)

 def read_party() -> list[dict]:
     count = pyboy.memory[WRAM_PARTY_COUNT]
     party = []
     for i in range(count):
         off = WRAM_PARTY_DATA + i * 44
         pdata = pyboy.memory[off:off+44]
         party.append(decode(pdata))
     return party
```

THE INSIGHT

Pokémon Yellow doesn't have an API.

It doesn't need one.

The cartridge IS the API. You just have to know which byte at which address means what.

This is CDC.

From 1998.

THE POKÉDEX MYSTERY.

How does Pokémon Yellow store "which of 151 Pokémon have I caught"?

MODERN ENTERPRISE

```
[
  { id: 1, name: "Bulbasaur", caught: true },
  { id: 2, name: "Ivysaur",   caught: false },
  ...
  { id: 151, name: "Mew",    caught: false }
]
```

~4,800 bytes

of JSON, gzip'd, S3'd, queried via Spark.

POKÉMON YELLOW, 1998

```
11010110 00100101 ...
^^^^^^^^
bits 0-7 = Pokémon #1-8
bit set = caught
```

19 bytes

of pure, bit-packed elegance.

→ 252× smaller. Same information. Modern Delta Lake calls this "bloom filters." Game Boy called it Tuesday.

THE SNAPSHOT ENGINE.

Reading WRAM every frame would flood the pipeline. Only emit when state actually changes.




```
if current_hash == last_hash:
    return None  # 0 bytes sent — kindest thing you can do to your Event Hub
last_hash = current_hash
queue.put_nowait(snapshot)   # ← bounded, non-blocking, sleeps the producer if Spark is dead
```

THE WHOLE PIPELINE.

Six nodes. ~400 lines of Python. Zero Spark. Real-time.

PYBOY EMULATOR
Pokémon Yellow

+

SNAPSHOT ENGINE
+ MD5 delta

+

AZURE EVENT HUB
JSON events

+

EVENT-STREAM
filter by type

+

EVENT-HOUSE (KQL)
3× destinations

+

REAL-TIME DASHBOARD
auto-refresh 5s

`event_type` IS THE ROUTER

party_snapshot

level, HP, moves, status

pokedex_snapshot

owned + seen bitmaps

session_log

player name, party count

LIVE DEMO #2

POKÉMON

YELLOW + REAL-TIME DASHBOARD

MISSION

catch a Pokémon → watch the party count tick in the dashboard.

→ PRESS A TO START ←

STAGE 03 · BOSS FIGHT

A WILD PYSPARK APPEARED!

▸ FIGHT ITEM

PKMN RUN

WHY I TRIED SPARK FIRST.

"because

FABRIC = SPARK,

right?"

This is the marketing reflex.

Capacity = Spark cluster. Notebook = Spark notebook. Engineering = Spark engineering.

Fabric DOES have a Python kernel. It's just one click away in the dropdown. And not many people click it.

WHAT KILLED ME.

Three crit hits. From a PySpark cluster, on a single-threaded emulator. Predictable in hindsight.

-30 HP

CU COST

capacity units burning idle

A Pokémon emulator is single-threaded. Spark spun up 8 cores. 7 were doing nothing — but billed.

-60 HP

STARTUP OVERHEAD

60-second cold start vs 60 fps target

PyBoy needs to tick every 16 ms. Spark needs a minute to wake up. Math says no.

SUPER EFFECTIVE

-90 HP

KERNEL HANGS

no clean restart, debugging hell

Trial-and-error on a frozen Spark kernel means re-attaching, re-mounting, re-everything. Iteration died.

Result: my Pokémon died before it left the lab.

SO I CLICKED THE DROPDOWN.

	PYSPARK	PLAIN PYTHON
STARTUP TIME	~60s	~1s
CU COST / hour	8× cores billed	1× core billed
ITERATION LOOP	minutes	seconds
POKÉMON FPS	<5	~60
DRAMA WHEN BROKEN	you know	kernel restart, you're back

PYTHON USED "FAST." IT'S SUPER EFFECTIVE.

THE TAKEAWAY YOU SHOULD STEAL.

DEFAULT PYTHON.
PROMOTE TO SPARK WHEN PROVEN NEEDED.

When SHOULD you use Spark?

YES

→ data exceeds single-node memory (>100 GB)

→ a single operation requires a true cluster (joins on TB-scale)

→ you genuinely benefit from horizontal scale-out

NO

→ "because the docs said so"

→ "because the architect on the project demands it"

→ "because it sounds more enterprise"

STAGE 04

POWER-UPS

"lessons retro games taught me about modern data engineering"



POWER-UP 01

BIT-PACKING IS A SUPERPOWER.

The Pokédex bitmap (19 bytes / 151 species) maps directly onto a stack of techniques you should already be reaching for in 2026:

GAME BOY (1998)	MODERN EQUIVALENT	WHAT IT DOES
Pokédex caught bitmap	**Roaring Bitmap / Bloom filters**	*presence checks at ~zero cost*
Status flags (1 byte)	**Delta column stats**	*min/max in metadata, skip files entirely*
Save file fixed offsets	**Columnar (Parquet) layout**	*read only the bytes you need*
Move PP packed 6+2 bits	**Bit-packed integer columns**	*Parquet's RLE + dictionary encoding*

Stop sending JSON when it is not needed.

POWER-UP 02

HEADLESS + CUSTOM UI — INSIDE A SESSION.

This is the most underused part of Fabric notebooks. The notebook isn't a presentation surface. It's a tiny web app you can BUILD.

RECIPE

- SDL_VIDEODRIVER=dummy
- frame → PNG → base64
- display_handle.update()
- ipywidgets for input
- thread for the game loop

WHAT YOU CAN BUILD

- live dashboards in the notebook
- interactive data explorers
- annotation UIs for ML labeling
- real-time alerts with buttons
- ANY headless service

REAL TALK

- you're 50 lines from a tool
- no React. no deployment.
- Lakehouse next door for state
- ship it to your team Monday
- this scales further than you think

→ if it runs headless in Linux, you can probably stream it from a Fabric notebook.

POWER-UP 03

SAVE POINTS ARE EVERYWHERE.

SAVE STATE

(the emulator one)

- full snapshot of memory + registers
- ~32 KB per save
- instant rewind
- *modern equivalent:*
  - **Delta Lake snapshot, time travel**
  - **git tag, container image**

BATTERY SAVE

(the cartridge one)

- incremental, persistent to disk
- ~8 KB per game
- survives shutdown
- *modern equivalent:*
  - **Delta change feed, CDC**
  - **OneLake — your battery save**

If your data platform doesn't let you BOTH rewind AND incrementally persist — it's missing the cartridge slot.

BONUS POWER-UPS · RAPID FIRE

6-Pokémon PARTY	+	max concurrent workloads / parallelism
Move PP	+	rate limits / quotas / call budget
Burned / Frozen	+	throttling / backpressure states
Paralyzed	+	intermittent retries (50% chance)
Random encounters	+	sampling strategy / stochastic tests
Type effectiveness	+	compatibility matrix (engine × workload)
EXP / Level cap	+	warm-up / training data plateau
HM Cut, HM Surf	+	permission-gated capabilities

STAGE 05 · GAME OVER

CONTINUE?

▶ YES

no

→ wrap-up + next adventure

WHAT AM I THINKING OF BUILD NEXT?

Three teasers. If any of these get built before the next conference, you'll see them.

PROTO 01

TETRIS

→ Delta Lake compaction VIZ ←

Visualize OPTIMIZE/Z-ORDER as Tetris stacking. Watch your data files COMPACT live.

PROTO 02

ZELDA

→ lineage as a dungeon map ←

Walk your data lineage like a 2D dungeon. Each table is a room. Each join is a door.

PROTO 03

STREET FIGHTER

→ engine showdown ←

Spark vs DuckDB vs Polars vs Python — same query, same data, picks-the-winner combo meter.

(If you want any of these, tell me afterward.)

THE TAKEAWAY

▶ PRESS START TO CONTINUE.

the moment you stop accepting "it can't be done in Fabric" — that's when you find what Fabric really is.

ENGINEERING MAY BE FUN.

GO PLAY.

THANK YOU.

Questions? Find me at the grill.

ŠTĚPÁN REŠL (STEVE)

Lead Technical Consultant · DataBrothers

EMAIL stepan.resl@databrothers.cz

LINKEDIN /in/stepan-resl

BLOG datameerkat.ccom

☆ GAME OVER ☆ · thanks for staying through the boss fight

FEEDBACK



THANK YOU