Python & Agentic Coding

Normal Coding · Vibe Coding · Agentic Coding with OpenCode

Al Amal 18 · Follow-up to: Python Basics

Islam Hamama

National Research Institute of Astronomy and Geophysics (NRIAG), Egypt
islam.hamama@nriag.sci.eg - i-hamama.com

August 17, 2026

You already learned to **read** Python. Today: the three ways to **produce** it and who is responsible for the final number.

Roadmap

Three Ways to Write Code

The same well-log task can be produced three ways:

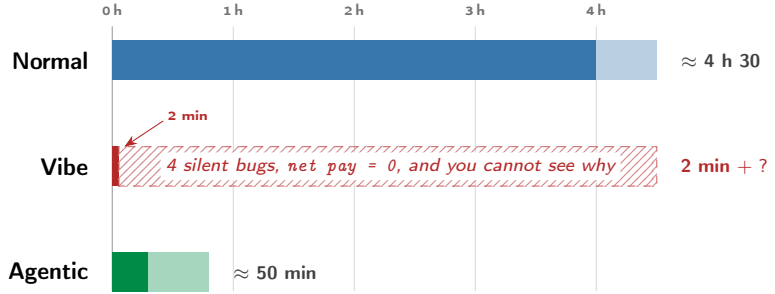
1. **Normal coding** — you type every line, you can explain every line
2. **Vibe coding** — you describe it to a chat LLM, paste, tweak, ship
3. **Agentic coding** — an LLM with tools: reads, plans, edits, runs, *you approve*

The question of the session

Who is responsible for the final number?

In all three: you are. The difference is *how much you can check*.

So Which One Is Fastest?



Solid = producing the draft Pale = checking it until you would *sign* it Hatched = rework you cannot schedule

The honest scoreboard — measured from these very files

Normal: **242 lines** you typed (94 of them just the plot). Agentic: **417 words** of instruction → **135 lines** it wrote. Vibe: **one sentence** → an answer of 0 that looked fine. **Vibe wins the race and loses the report.**

Today's Promise

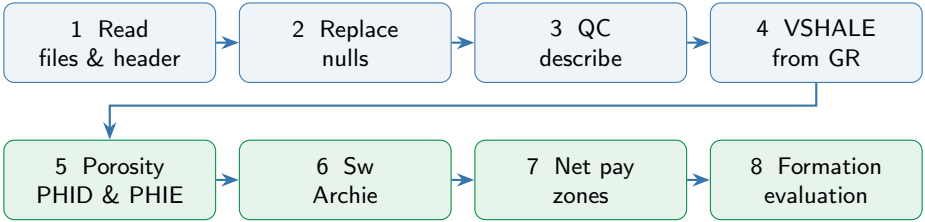
We run the **full petrophysical analysis** — on a well log sample using notebook by three ways, and compare.

1. **Normal:** the Al Amal 18 notebook actually running
2. **Vibe:** a chat answer, and the trap it falls into
3. **Agentic:** OpenCode doing all 8 steps, like a professional

The one line of the day

AI changed who writes the first draft. It did not change who is responsible for the final one.

One Workflow, Run Three Times



Blue = compute the logs **Green** = evaluate the reservoir.

Same 8 steps, run three ways: **by hand**, **by chat**, **by an agent** — and where each one fails.

The Math Behind the Steps

Shale volume — linear I_{GR} (step 4) [1]

$$V_{sh} = \frac{GR - GR_{min}}{GR_{max} - GR_{min}}$$

Density porosity (step 5) [2]

$$\phi_D = \frac{\rho_{ma} - \rho_b}{\rho_{ma} - \rho_f} \qquad (\rho_{ma} = 2.65,\ \rho_f = 1.0)$$

Effective porosity [3]

$$\phi_e = \phi_D - 0.3\, V_{sh}$$

Water saturation — Archie (step 6) [4]

$$S_w = \left(\frac{a\, R_w}{\phi^m\, R_t} \right)^{1/n} \qquad (a = 1,\ m = 2,\ n = 2)$$

Net pay cut-offs (step 7) [5]

sand: $V_{sh} < 0.4$ reservoir: $\phi_e > 0.08$

pay: $S_w < 0.6$

Every symbol (GR, ρ_b, R_t, ϕ) is a column in your DataFrame — the code is just these equations.

[1] Rider & Kennedy (2011), *Geological Interpretation of Well Logs*, 3rd ed.; non-linear forms in Larionov (1969). [2] Asquith & Krygowski (2004), *Basic Well Log Analysis*, AAPG Methods in Exploration 16. [3] Dewan (1983), *Essentials of Modern Open-Hole Log Interpretation* — the 0.3 is $\phi_{D,\text{shale}}$, a *local* value.

[4] Archie (1942), *Trans. AIME* 146, 54–62. [5] Worthington & Cosentino (2005), *SPE Res. Eval. & Eng.* 8(4), 276–290 — cut-offs are project decisions.

The Symbols — and Where Each One Lives in Your Code

Sym.	Meaning	Unit	In the code
GR	gamma ray	API	`well['GR']`
GR_{min}	clean-sand baseline	API	`GR.quantile(.01)`
GR_{max}	shale baseline	API	`GR.quantile(.99)`
V_{sh}	shale volume	fraction	`well['VSHALE']`
ρ_b	bulk density	g/cm^3	`well['RHOB']`
ρ_{ma}	matrix density	g/cm^3	`2.65` (sandstone)
ρ_f	fluid density	g/cm^3	`1.0` (water)
ϕ_D	density porosity	fraction	`well['PHI']`

Sym.	Meaning	Unit	In the code
ϕ_e	effective porosity	fraction	`well['PHIECALC']`
S_w	water saturation	fraction	`well['SW_LIM']`
R_t	true (deep) resistivity	Ω·m	`well['RT']`
R_w	formation-water resist.	Ω·m	`well['RW']`
a	tortuosity factor	—	`1`
m	cementation exponent	—	`2`
n	saturation exponent	—	`2`
Δz	depth sample interval	m	`STEP = 0.1524`

Two things to notice

Every fraction is a fraction — V_{sh}, ϕ_D, ϕ_e, S_w all run 0–1, never 0–100. ***a*, *m*, *n* are rock properties, not constants** — they belong in core data. We use $1, 2, 2$ because that is the clean-sandstone default, *and we say so.*

Roadmap

Normal Coding: You Type It, You Own It

What it means

- Every line written by a human — you can **explain** every line
- The program is a *recipe* you wrote, and therefore trust
- Your Al Amal 18 notebook *is* normal coding

Strengths: full understanding, full control, provenance.
Costs: slow, verbose, and the 90 lines of plot boilerplate.

The honest summary

Learn by normal coding. The other two modes are ways to *produce* code — but reading it stays *your* job.

Normal Step 1 — Read the LAS

You can now read "15-9-19_SR_COMP.LAS" file from your drive

```
#import lasio library after installing
from lasio import read
import pandas as pd
import numpy as np
log_1 = read("15-9-19_SR_COMP.LAS")
```

Well Header from LAS file

```
log_1.well
```

```
[HeaderItem(mnemonic="STRT", unit="M", value="102.1568", descr="Top Depth"),
 HeaderItem(mnemonic="STOP", unit="M", value="4636.514", descr="Bottom Depth"),
 HeaderItem(mnemonic="STEP", unit="M", value="0.1524", descr="Depth Increment"),
 HeaderItem(mnemonic="NULL", unit="", value="-999.25", descr="Null Value"),
 HeaderItem(mnemonic="FLD", unit="", value="Q15", descr="Field Name"),
 HeaderItem(mnemonic="WELL", unit="", value="15/9-19", descr="NAME"),
 HeaderItem(mnemonic="WBN", unit="", value="15/9-19 SR", descr="WELLBORE"),
 HeaderItem(mnemonic="NATI", unit="", value="NOR", descr="COUNTRY"),
 HeaderItem(mnemonic="CTRY", unit="", value="NOR", descr="COUNTRY"),
 HeaderItem(mnemonic="COMP", unit="", value="STATOIL", descr="OPERATOR"),
 HeaderItem(mnemonic="PDAT", unit="", value="MSL", descr="PERM DATUM"),
 HeaderItem(mnemonic="COUN", unit="", value="NORTH SEA", descr="RIG NAME"),
 HeaderItem(mnemonic="STAT", unit="", value="NORWAY", descr="STATE"),
 HeaderItem(mnemonic="PBWE", unit="", value="15/9-19", descr="PB WELL ID"),
 HeaderItem(mnemonic="APIN", unit="", value="15/9-19 SR", descr="PB WELLBORE ID"),
 HeaderItem(mnemonic="PBWS", unit="", value="ALL", descr="PB WELL NAME SET")]
```

```
#Printing the items of the log with readable format
for line in log_1.well:
    print(f"{line.descr} ({line.mnemonic}): {line.value}")
```

```
Top Depth (STRT): 102.1568
Bottom Depth (STOP): 4636.514
Depth Increment (STEP): 0.1524
Null Value (NULL): -999.25
Field Name (FLD): Q15
NAME (WELL): 15/9-19
WELLBORE (WBN): 15/9-19 SR
COUNTRY (NATI): NOR
COUNTRY (CTRY): NOR
OPERATOR (COMP): STATOIL
PERM DATUM (PDAT): MSL
RIG NAME (COUN): NORTH SEA
STATE (STAT): NORWAY
PB WELL ID (PBWE): 15/9-19
PB WELLBORE ID (APIN): 15/9-19 SR
PB WELL NAME SET (PBWS): ALL
```

`lasio.read()` loads the log; `log.well` prints the header; `NULL = -999.25` comes from the file. Step 1 of the workflow, by hand.

Try it live in Colab

Normal Step 1 — The Data as a Table

```
well
```

DEPT	AC	CALI	DEN	GR	NEU	RDEP	RMED
102.1568	NaN	NaN	NaN	5.3274	NaN	NaN	NaN
102.3092	NaN	NaN	NaN	5.8235	NaN	NaN	NaN
102.4616	NaN	NaN	NaN	6.5228	NaN	NaN	NaN
102.6140	NaN	NaN	NaN	7.2285	NaN	NaN	NaN
102.7664	NaN	NaN	NaN	9.5020	NaN	NaN	NaN
...	...	...	...	...	...	...	...
4635.9044	NaN	NaN	NaN	NaN	NaN	0.7729	0.5978
4636.0568	NaN	NaN	NaN	NaN	NaN	0.8369	0.6257
4636.2092	NaN	NaN	NaN	NaN	NaN	0.8741	0.6888
4636.3616	NaN	NaN	NaN	NaN	NaN	0.9002	0.8902
4636.5140	NaN	NaN	NaN	NaN	NaN	0.9133	1.0363

29754 rows × 7 columns

`log.df()` turns the LAS into a pandas table — 29,754 depth samples. "DataFrame" is just a sheet you can filter, compute on, and plot.

Normal Step 2 — Clean the Nulls

Replacing the Null values

```
well.replace(-999.25, np.nan, inplace=True)
#well.replace(-999.00, np.nan, inplace=True)
well.replace(-99999.000, np.nan, inplace=True)

well=well

well.describe()
```

	DEPTH	EGR	HDRS	ERHO	ENPH	PHIE	SWE	Unnamed: 7
count	200.000000	200.000000	200.000000	200.00000	200.000000	198.000000	198.000000	1.0
mean	2200.500000	95.520735	1180.072820	7.27756	0.274695	0.084545	0.903439	1.0
std	57.879185	39.338199	6346.730696	70.33621	0.131384	0.108812	0.254461	NaN
min	2101.000000	19.610000	0.944000	1.24300	0.086000	0.000000	0.000000	1.0
25%	2150.750000	54.013500	2.409000	2.24425	0.211000	0.000000	0.989250	1.0
50%	2200.500000	104.812000	3.590500	2.39500	0.249000	0.004500	1.000000	1.0
75%	2250.250000	124.704500	12.793000	2.46225	0.299000	0.196000	1.000000	1.0
max	2300.000000	213.213000	40000.000000	997.00000	1.333000	0.555000	1.309000	1.0

```
well.head(10)
```

	DEPTH	EGR	HDRS	ERHO	ENPH	PHIE	SWE	Unnamed: 7
0	2101	93.905	2.472	2.517	0.281	0.0	1.0	NaN
1	2102	104.401	2.484	2.495	0.269	0.0	1.0	NaN

Null values are `-999.25`, `-999`, or `-99999.000` — never `NaN`. `replace(..., np.nan)` cleans them; the CSV has a units row to skip. ***No AI can guess this*** *— you know it from the data.*

Normal Steps 4–6 — The Functions

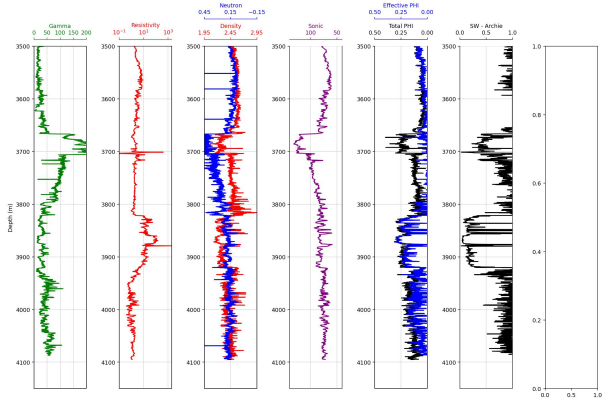
```
#Shale Volume from Gamma
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
#(gamma_ray-gamaray_min)
def shale_volume(gamma_ray, gamma_ray_max, gamma_ray_min):
    vshale = (gamma_ray - gamma_ray_min) / (gamma_ray_max - gamma_ray_min)
    return round(vshale, 4)
#Density_Prosity
def density_porosity(input_density, matrix_density, fluid_density):
    denpor = (matrix_density - input_density) / (matrix_density - fluid_density)
    return round(denpor, 4)
#Archie water saturation
def sw_archie(porosity, rt, rw, archieA, archieM, archieN):
    sw = ((archieA / (porosity ** archieM)) * (rw/rt))**(1/archieN)
    return sw
# Another method simandoux
def sw_simandoux(phie, rt, rw, archieA, archieM, archieN, vshale, rshale):
    A = (1 - vshale) * archieA * rw / (phie ** archieM)
    B = A * vshale / (2 * rshale)
    C = A / rt

    sw = ((B **2 + C)**0.5 - B) **(2 / archieN)
    return sw
#Calculate Shale Volume
well['VSHALE'] = shale_volume(well['GR'], well['GR'].quantile(q=0.99),
                              well['GR'].quantile(q=0.01))
#Calculate density porosity
well['PHI'] = density_porosity(well['RHOB'], 2.65, 1)
#Calculate PHIE
well['PHIECALC'] = well['PHI'] - (well['VSHALE'] * 0.3)
#Calculate Archie SW
well['SW'] = sw_archie(well['PHI'], well['RT'], well['RW'], 1, 2, 2)
well['SW_SIM'] = sw_simandoux(well['PHIECALC'], well['RT'], well['RW'], 1, 2, 2, well['VSHALE'],2)
#Limit SW to 1
well['SW_LIM'] = well['SW'].mask(well['SW']>1, 1)
well['SW__SIM_LIM'] = well['SW_SIM'].mask(well['SW_SIM']>1, 1)


well.describe()
```

`def shale_volume(...)` with `return` (not `print`) hands back a value you store in a column. Same pattern for porosity and Archie `SW`.

Normal Coding — The Result You Typed



The 7-track log display — steps 4–6 visualised. You know every axis, every colour, every scale — because you wrote them.

Normal Step 7 — Net Pay, by Hand

Flag the rock, then count the pay:

```
VSH_CUT, PHI_CUT, SW_CUT = 0.4, 0.08, 0.6
STEP = 0.1524          # m per sample

well['SAND'] = well['VSHALE'] < VSH_CUT
well['RES']  = well['PHIECALC'] > PHI_CUT
well['PAY']  = well['SAND'] & well['RES'] \
             & (well['SW_LIM'] < SW_CUT)

net_res = well['RES'].sum() * STEP
net_pay = well['PAY'].sum() * STEP
print(f"Net pay = {net_pay:.1f} m")
```

The output:

```
Logged interval =  624.8 m (MD)
Net reservoir   =  191.3 m (MD)
Net pay         =   94.3 m (MD)
```

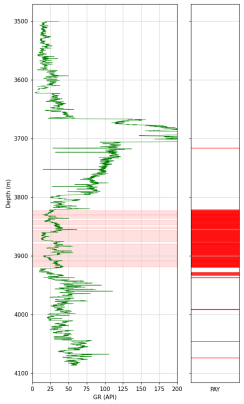
Two lines of logic, not magic

Each < comparison is a column of `True`/`False`; `sum()` counts the `True`s. ×`STEP` turns samples into metres.

Say what the number is

624.8 m is the *logged interval*, not "gross rock" — most of it is shale. And every metre here is **MD**, not TVD: `15/9-19 SR` is a sidetrack, so true vertical pay is less.

Normal Step 8 — Formation Evaluation



Left track = GR; red = where `PAY` is `True`. You can see the pay, because you drew it.

The report you write yourself

- **27** contiguous pay zones, listed top → base
- **Net pay 94.3 m** (MD) · net reservoir 191.3 m
- Mean `PHIE` in pay = **0.199**, mean `SW` = **0.177**
- Main pay bodies: 3856–3876 m (19 m) and 3880–3909 m (29 m)

But 27 is not a geological number

11 of those 27 "zones" are $\leq$ 0.30 m (2.3 m in total) — at or below the density log's vertical resolution. Filter at 0.5 m and you get **13 zones / 90.7 m**. The *thickness* is robust; the *count* was never real.

Roadmap

Vibe Coding: The Fast Lane

- Term popularised by Andrej Karpathy (2025)
- You **describe** the task in plain English
- You **paste** the code, run it, tweak by "feel"
- You **ship** without fully reading it

Why it feels amazing

Fast. No syntax errors. No imports to remember. Perfect for **prototypes**.

The honest label

"Fully giving in to the vibes" — Karpathy's own words.

The Vibe Attempt: One Prompt, the Whole Workflow

I have well log data (a LAS file and a CSV). Do a full petrophysical analysis: shale volume, porosity, water saturation, and net pay. Give me the code.

```
from lasio import read
import pandas as pd, numpy as np
df = read("15-9-19_SR_COMP.LAS").df()
vsh = (df["GR"] - df["GR"].min()) / (df["GR"].max() - df["GR"].min())
phi = (2.65 - df["DEN"]) / (2.65 - 1.0)
sw  = (1 / phi**2 * 0.02 / df["RDEP"]) ** 0.5
pay = df[phi > 20]          # pay where porosity > 20
print("net pay =", len(pay), "samples")  # no zones, no report
```

```
>>> net pay = 0 samples
no zones, no report  --  porosity is a fraction: 0.21 = 21%
```

*You ask for the **same 8 steps** in one sentence. The chat returns one fluent script — it looks like the full analysis, in seconds.*

The Trap: Runs Clean, Wrong Science

The vibe script went through the *same steps* — but silently wrong in four places:

Step	The vibe wrote	What you know
2 (nulls)	**skipped entirely**	`-999.25` becomes "shale"
4 (VSHALE)	`min/max`	`quantile(0.01)/(0.99)` survives outliers
7 (net pay)	`phi > 20`	porosity is a **fraction**: `0.21` = 21%
8 (report)	**no zones, no report**	a geoscientist reports zones & thicknesses

No crash. No warning.

`net pay = 0 samples` — the model answered the question you asked, with your units missing. **The bug was in the prompt** — and only your basics let you see it.

Where Vibe Coding Breaks

Failure	Why it slips through
Unit errors (fraction vs %, m vs ft)	runs clean, science wrong
Your data's quirks (-999.25 vs -999)	the AI never saw your file
Hallucinated API / function name	fluent but invented
No tests, no traceback reading	"it ran" is the only check

The rule

"It runs" does not mean "it is right." A chat checks none of your units, cutoffs, or data quirks.

Vibe Coding: Verdict

Use it for

- Prototypes and throwaway scripts
- Exploring an API
- First drafts you will rewrite

Never for

- Anything that lands in a report
- Anything that drives a decision
- Anything you cannot debug at 2 a.m.

Roadmap

From Chatbot to Agent

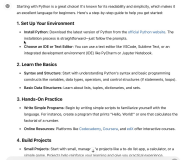
A chatbot answers. An agent acts.

1. **Understand:** reads your files, tests, project rules
2. **Plan:** proposes steps *before* touching code
3. **Act:** edits files, runs commands, writes tests
4. **Observe:** reads errors and output, then retries

. . . in a loop, **with your approval**.

You are the senior scientist

The agent is a fast junior: tireless, knowledgeable, but it needs a clear task, rules, and code review.



OpenCode: Free, Any Model, Your Rules

- **Free & open source** coding agent
- Terminal, desktop app, or IDE extension
- **Any model** — 75+ providers, or free ones built in
- `/init` writes an `AGENTS.md` rulebook for your project

```
curl -fsSL https://opencode.ai/install |
     bash
# then, inside your project folder:
cd well_project
opencode
# /models   -> pick a free model
# /init     -> write AGENTS.md
```

`https://opencode.ai` · `https://models.dev` · check model names **the morning of** your lecture

The Experiment: One Folder, 8 Professional Steps

We gave OpenCode a folder with the **real well data** and an `AGENTS.md` rulebook encoding what a geoscientist knows:

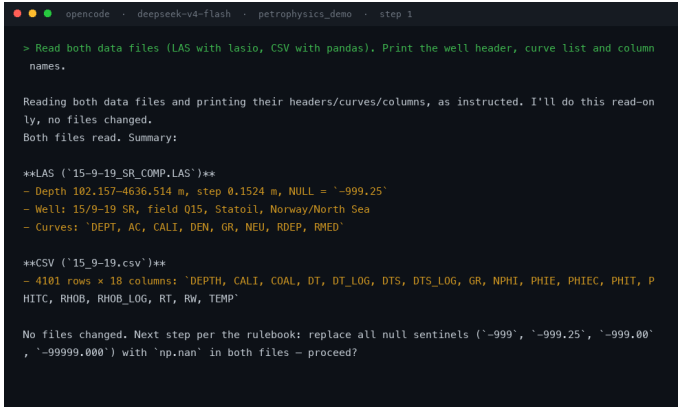
- the units row to skip
- the null sentinels (`-999.25` ...)
- where to read `NULL` from
- the exact workflow, step by step

The 8 steps

1 Read both files & header
2 Replace nulls
3 QC: describe + overview plot
4 VSHALE from GR
5 PHID & PHIE
6 Archie SW (capped)
7 Net-pay zones (contiguous)
8 Formation-evaluation report

Same data, same equations as your notebook — *nothing extra invented*.

Step 1 — It Reads the Project First

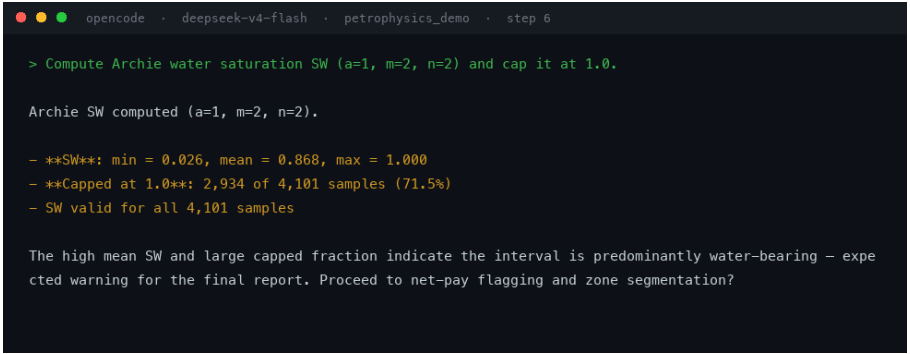


No code yet. It lists the header, the curves, and the columns, and tells us which table it will compute on. A chatbot would have started typing.

Step 3 — QC Before Anything Else

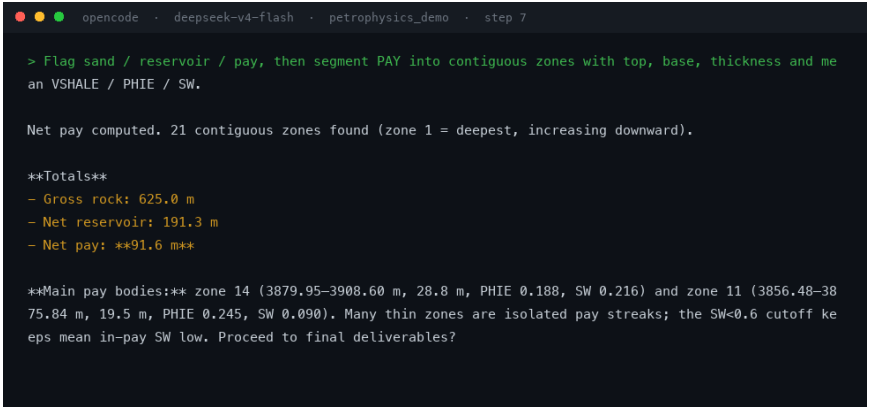
*It checks `describe()`, saves a quick overview plot, and **flags the depth mismatch and the GR spike itself** — because the rules told it to look.*

Step 4 & 6 — Computes, and Fixes Its Own Bug



VSHALE with quantile min/max; Archie SW capped at 1.0. Notice: it caught and corrected its own error (SW.where would have turned missing samples into 1.0 — it switched to clip). That is the agent loop working.

Step 7 — Net Pay Zones, in Meters



*Gross rock 625.0 m, net reservoir 191.3 m, net pay 91.6 m in **21 contiguous zones** — top/base/thickness per zone, from the 0.1524 m sampling step. Same cutoffs you would use.*

Step 8 — The Formation-Evaluation Report

opencode · deepseek-v4-flash · petrophysics_demo · step 8

> Formation evaluation: write formation_evaluation.md (zone table + summary), save final_log.png and results.csv, then summarize.

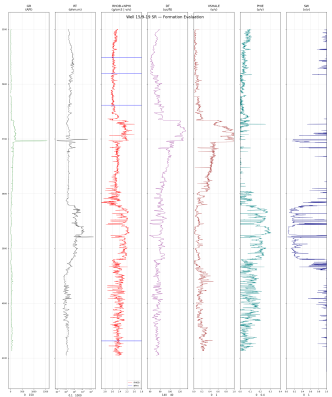
All three deliverables verified present:

- **`formation_evaluation.md`** (2.2 KB) — per-zone net-pay table + summary
- **`final_log.png`** (492 KB) — 7-track depth plot
- **`results.csv`** (270 KB, 4,101 × 9) — DEPTH, VSHALE, PHID, PHIE, SW, sand, reservoir, pay, zone_id

Formation evaluation summary:
- Interval 3500.0–4124.9 m, gross 625.0 m, net reservoir 191.3 m, **net pay 91.6 m** in **21 zones**
- Avg PHIE in pay = **0.201**, avg SW in pay = **0.198**
- **Warning:** interval predominantly water-bearing (mean SW 0.868, 71.5% capped at 1.0); key pay in zones 11 (19.5 m) and 14 (28.8 m), remainder thin streaks.

*It writes `formation_evaluation.md` (per-zone net-pay table + summary), `final_log.png` and `results.csv` (4,101 rows) — and **warns** that the interval is largely water-bearing, with main pay in zones 11 and 14.*

Its Final Deliverable



The 7-track log display, produced and saved by the agent. Same data, same equations, same plot as your notebook — but supervised by you.

The Formation Evaluation It Produced

formation_evaluation.md — petrophysics_demo

```
# Formation Evaluation Report — Well 15/9-19 SR

## Summary

- **Interval evaluated:** 3500.0 – 4124.9 m
- **Gross rock thickness:** 625.0 m
- **Net reservoir thickness (PHIE > 0.08):** 191.3 m
- **Total net pay (VSHALE < 0.4, PHIE > 0.08, SW < 0.6):** 91.6 m
- **Number of net-pay zones:** 21
- **Average PHIE inside pay:** 0.201 (v/v)
- **Average SW inside pay:** 0.198 (v/v)

## Per-Zone Net-Pay Table

| Zone | Top (m) | Base (m) | Thickness (m) | Mean VSHALE | Mean PHIE | Mean SW |
|---|---|---|---|---|---|---|
| 1 | 3716.58 | 3716.73 | 0.30 | 0.145 | 0.389 | 0.567 |
| 2 | 3821.45 | 3821.73 | 0.46 | 0.090 | 0.360 | 0.444 |
| 3 | 3822.65 | 3829.96 | 7.47 | 0.071 | 0.369 | 0.248 |
| 4 | 3830.42 | 3835.91 | 5.64 | 0.080 | 0.235 | 0.138 |
| 5 | 3836.52 | 3838.65 | 2.29 | 0.078 | 0.155 | 0.285 |
| 6 | 3839.11 | 3843.38 | 4.42 | 0.063 | 0.177 | 0.217 |
| 7 | 3843.48 | 3847.34 | 3.81 | 0.113 | 0.231 | 0.182 |
| 8 | 3848.56 | 3849.93 | 1.52 | 0.138 | 0.208 | 0.190 |
| 9 | 3850.54 | 3853.74 | 3.35 | 0.135 | 0.190 | 0.230 |
| 10 | 3855.11 | 3855.42 | 0.46 | 0.233 | 0.308 | 0.410 |
| 11 | 3856.40 | 3875.84 | 19.51 | 0.052 | 0.245 | 0.090 |
| 12 | 3877.21 | 3877.97 | 0.91 | 0.111 | 0.159 | 0.180 |
| 13 | 3878.58 | 3879.19 | 0.76 | 0.107 | 0.127 | 0.086 |
| 14 | 3879.95 | 3900.60 | 20.00 | 0.106 | 0.188 | 0.216 |
| 15 | 3900.91 | 3919.12 | 18.36 | 0.123 | 0.199 | 0.250 |
| 16 | 3919.58 | 3919.73 | 0.30 | 0.128 | 0.179 | 0.409 |
| 17 | 3920.03 | 3920.03 | 0.15 | 0.155 | 0.365 | 0.307 |
| 18 | 3930.70 | 3930.70 | 0.15 | 0.054 | 0.128 | 0.573 |
| 19 | 3932.07 | 3932.53 | 0.61 | 0.138 | 0.186 | 0.579 |
| 20 | 3933.75 | 3933.75 | 0.15 | 0.143 | 0.243 | 0.324 |
| 21 | 3991.01 | 3991.01 | 0.15 | 0.122 | 0.167 | 0.541 |

## Warnings

- The overall interval is largely **water-bearing**: mean SW over the logged section is 0.868, with 2934 of 4101 samples (
- Only the 3822–3920 m and thin 3991.0 m sands show economic-quality water saturations (SW < 0.6); the main net-pay is cor
- Several pay zones are very thin (<= 1 m) isolated streaks; these contribute little commercial thickness.
```

The agent's `formation_evaluation.md`: 21 net-pay zones, average `PHIE` 0.201 and `SW` 0.198 inside pay — and an honest warning about the thin streaks and the water-bearing interval. That is the report a geoscientist signs.

Where You Sit in the Loop

1. **You specify** — units, cutoffs, files, edge cases
2. **Agent plans** — you approve before any code
3. **Agent edits** — and shows you a diff
4. **You review every diff** — accepting is *signing*
5. **Run the checks** — fail → back to the agent

Golden rules

Back up first · one small task at a time · ask for a plan · review every diff · no unpublished data in prompts · demand verification: "run it and show me the first 5 rows."

Agents: A Team, Not a Single Tool

Primary agents — you talk to these (switch with **Tab**)

- **Build** (default): all tools — reads, edits, runs, you approve
- **Plan**: read-only — analyses and proposes, *never edits*

Subagents — specialized helpers (invoke with `@name`)

- **Explore**: fast, read-only codebase search
- **General**: full tasks you can run in parallel
- **Scout**: external docs and dependency research

Plan mode vs Build mode

Plan answers "what should we change?" and waits. **Build** makes the change — after you approve the plan and each diff.

Ask a subagent

```
@explore where is shale_volume()
defined?
```

Review and Ultra Review

Before you accept a diff, get it checked — by you *and* by a reviewer agent.

- **Review** — a read-only agent that checks the diff for bugs, unit errors, and edge cases. `edit` and `bash` are denied.
- **Ultra review** — a stricter second pass: a stronger model, *zero* tool access, and an adversarial checklist (units, cutoffs, nulls, your `checks.py` asserts).

The point

These are agents **you create**. OpenCode ships `build` and `plan`; professionals add `review` for rigor. Your asserts stay the final judge.

`/.config/opencode/agents/review.md`

```
---
description: Reviews code for quality
             and unit errors
mode: subagent
permission:
  edit: deny
  bash: deny
---
Check for bugs, wrong units
(fraction vs %), and missing
edge cases. Do not edit files.
```

Skills: Procedures Your Agent Knows

What is a skill? A folder with a `SKILL.md` — reusable instructions the agent loads *on demand* (via the skill tool) when a task matches the description. Not a model, not code: it is your **procedure**. **Where they live**

- Project: `.opencode/skills/<name>/SKILL.md`
- Global: `/.config/opencode/skills/<name>/SKILL.md`
- (Some builds also read the folder as `skill/` — here both work)

Already installed on this machine

`las-qc` — read & QC LAS files
`basic-petrophysics` — Vsh, porosity, Sw, net pay
`crossplots` — neutron–density, Pickett, M–N
`petrophysics-report` — well summaries
in `/.config/opencode/skill/`

Think of it as

`AGENTS.md` = the project's rulebook.
`SKILL.md` = a standard operating procedure the agent pulls in when needed.

Create Your Own Agent or Skill

Create an agent — two ways

1. Interactive: `opencode agent create` (picks location, prompt, permissions)
2. Write a markdown file yourself:

`.opencode/agents/review.md` (project) or `/.config/opencode/agents/review.md` (global)

```
---
description: Reviews my petrophysics code
mode: subagent
model: <provider>/<model>
permission: {edit: deny, bash: deny}
---
Review for unit errors and edge cases.
```

Create a skill One folder + one file: `.opencode/skills/net-pay/SKILL.md`

```
---
name: net-pay
description: Flag net pay from
  VSHALE, PHIE and SW cutoffs
---
sand: VSHALE < 0.4
reservoir: PHIE > 0.08
pay: SW < 0.6
Report thicknesses in meters.
```

Then just ask

`@review check my step6_sw.py` — the skill loads automatically when a task matches its description.

Roadmap

The Same 8 Steps, Three Ways

Step	Normal	Vibe	Agentic
1 Read	you type it	one line of prompt	it reads both files
2 Nulls	you `replace()`	**skipped**	reads `NULL`, replaces
3 QC	you `describe()`	**skipped**	flags spikes itself
4 VSHALE	your `def`	**min/max**	quantile, clipped
5 Porosity	your `def`	one line	PHID & PHIE
6 SW	your `def`	one line	Archie, capped
7 Net pay	your flags & `sum()`	`phi>20`	21 zones, 91.6 m
8 Report	your `print()` summary	**none**	`formation_evaluation.md`

Read the table top to bottom

Same steps, same data, same equations. The difference is *who does each step* — and *whether it gets checked*. Vibe coding silently skips or mis-specifies the steps that matter; the agent does them and shows you the diff.

Manual net pay ≈ 94 m vs agent ≈ 92 m — the *same answer*: the agent used effective porosity in Archie (more correct); your notebook used total porosity. **Your basics let you see exactly why they differ.**

Wait — The Two Answers Disagree

Your notebook said **94.3 m in 27 zones**. The agent said **91.6 m in 21 zones**. Same data, same cutoffs, same Archie constants. *Nothing crashed.*

	ϕ into Archie	Net pay	Zones	Mean S_w in pay
Notebook (step 6)	`PHI` (total)	94.3 m	27	0.177
Agent (`AGENTS.md`)	`PHIE` (effective)	91.6 m	21	0.198

```
# the notebook wrote:
well['SW'] = sw_archie(well['PHI'], ...)
# the rulebook said:
SW = ((a / PHIE**m) * (RW/RT))**(1/n)
```

$S_w \propto 1/\phi$, so the smaller `PHIE` gives a *higher* S_w, fewer samples pass `SW < 0.6`, and thin streaks drop out — 27 zones collapse to 21.

This is the whole lecture in one slide

Neither run is a bug. **It is one modelling choice** — total vs effective porosity — and *only a petrophysicist can settle it.* The agent did exactly what its rulebook said; the notebook did exactly what you typed. **2.7 m of pay hangs on a decision no tool can make for you.**

One Task, Three Ways: Who Does What?

	Normal	Vibe	Agentic
Who writes	you	chat LLM	agent + tools
Who reads	you	**usually nobody**	you (the diffs)
Who reviews	you	**"by feel"**	you, with checks
Who is responsible	**you**	**you**	**you**
Best for	learning, control	prototypes	real work, supervised

The pattern

The responsibility row never changes. Only *how much you can check* does.

Who Catches What?

Error	Caught by	Example from your data
Syntax error	Python, for free	`IndentationError`
Wrong API name	you, if you read	hallucinated function
Wrong units	**only you**	fraction vs percent; m vs ft
Data quirks	**only you**	-999.25 vs -999; unit row on line 2
Wrong formula	**only you**	Archie a, m, n; **PHI vs PHIE**

The dangerous failure is believability, not errors

Syntax is caught for free. Nothing catches a fraction compared to a percent threshold — it runs clean and lands in your thesis.

The Decision Rule

- **Learn** a new concept → normal coding
- **Prototype / explore** → vibe coding
- **Produce something that matters** → agentic, with a precise spec, checks, and your review

Vibe to explore. Agent to build. Basics to check both.

Settle the Disagreement Yourself

Your turn — do it now In your Al Amal 18 notebook, after the `SW` cell, paste this and run it:

```
for name, phi in [('total', well['PHI']),
                  ('effective', well['PHIECALC'])]:
    sw = sw_archie(phi, well['RT'],
                   well['RW'], 1, 2, 2)
    sw = sw.mask(sw > 1, 1)
    pay = ((well['VSHALE'] < 0.4)
           & (well['PHIECALC'] > 0.08)
           & (sw < 0.6))
    print(name, round(pay.sum()*0.1524, 1), 'm')
```

Answer these out loud

1. Which number would you put in a report?
2. **Why?** Name the rock physics, not the code.
3. Which one did *you* tell the agent to use?

Then write it down

Add that choice to your `AGENTS.md`. A rule you wrote once is a decision the agent can never quietly reverse.

Roadmap

Why Your Basics Matter: Five Answers

1. **You can't review what you can't read.** An agent's output is a diff; accepting a diff is a signature.
2. **Specifying is the new bottleneck.** "Make it better" gives noise; "phi is a fraction, raise ValueError outside 0–1, add asserts" gives reviewable work. *The vocabulary is the skill.*
3. **Believability, not errors, is the danger.** Python catches syntax; nothing catches fraction-vs-percent or -999.25-vs--999.
4. **Someone has to break the loop.** When the agent re-applies a broken fix, reading the traceback yourself is the only exit.
5. **Trust scales with cheap verification.** That is why we wrote `assert` before ever opening an agent.

The One Sentence of the Day

Remember

AI changed who writes the first draft.

It did not change who is responsible for the final one.

- AI multiplies whatever skill you bring — including zero
- The typing was never the hard part

Minute 150: Which Mode Are You?

1. **Normal:** you typed it, you own it — the best teacher
2. **Vibe:** fast, blind, perfect for prototypes — never for reports
3. **Agentic:** a fast junior you supervise with rules, diffs, and checks

You are not the person who *types* the code. You are the person who **reviews** it — in every mode. Your basics are what make review possible.

Resources

Resources

- Al Amal 18 notebook: `Al_Amal_17_Examples.ipynb` (Colab PDF export)
- The math: `petrophysics_math_notes.md`
- OpenCode: `https://opencode.ai` · docs: `https://opencode.ai/docs`
- Models: `https://models.dev`
- Take-home: `OpenCode_Visual_Manual.pdf`

What is your next step!

1. Install OpenCode; open a copy of your `Well_logging` folder
2. Re-run the 8 steps on *your* data; read every diff
3. Write one `checks.py` of asserts for your own functions

Thank You!

Questions?

Islam Hamama · islam.hamama@nriag.sci.eg · NRIAG, Egypt

"It runs" is not a proof. ***Checks are.***