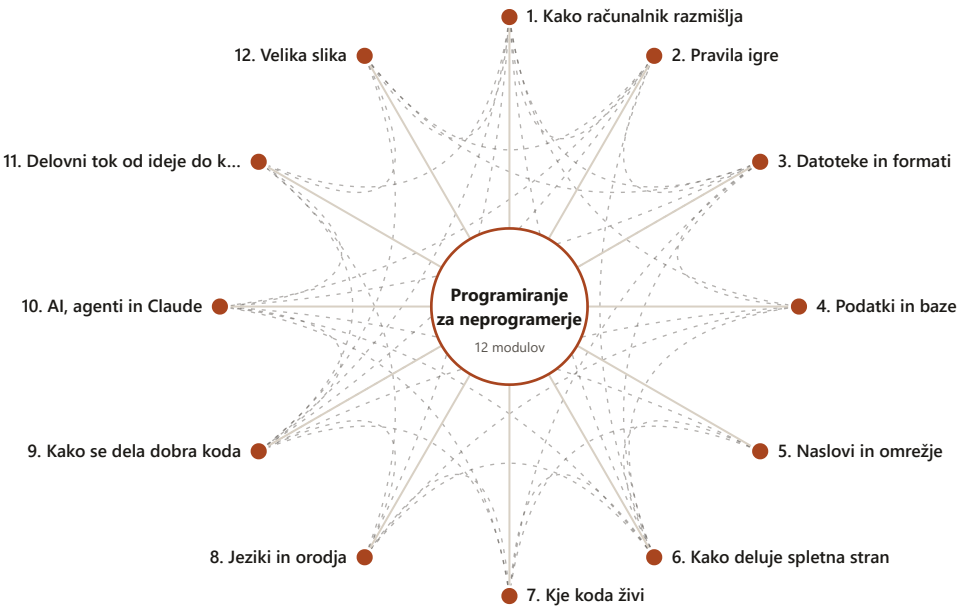
Programiranje za neprogramerje

Kaj moraš razumeti, da lahko gradiš z AI agenti

Ni namen, da bi postal programer. Namen je, da razumeš besede, ki jih uporablja tvoj AI agent, in da znaš problem razstaviti tako, da ga zna rešiti.



Kazalo

MODUL 1

Kako računalnik razmišlja

Devet gradnikov, iz katerih je sestavljen vsak program na svetu — in črno okno, v katerem se to zgodi.

Kaj je program `program / code`

Program je zaporedje navodil, ki jih računalnik izvede eno za drugim, od zgoraj navzdol.

Navodila so napisana v jeziku, ki ga bere tudi človek, računalnik pa si ga sproti prevede v svoje ukaze.

Od recepta se loči samo po bralcu: recept bere kuhar, program bere stroj — zato mora biti neprimerno bolj natančen.

prestej.py

```
# Prestej vrstice v datoteki.
pot = "gostje.txt"

datoteka = open(pot, encoding="utf-8")
vrstice = datoteka.readlines()
datoteka.close()

print("Stevilo vrstic:", len(vrstice))
```

IZPIS

```
Stevilo vrstic: 248
```

Sest vrstic, ki se izvedejo od prve do zadnje. To je cel program.

> Kuharju lahko napišeš *sol po okusu* in bo kosilo dobro. Računalniku moraš povedati, koliko gramov, sicer se ustavi ali pa doda nič. Ta razlika — da računalnik nikoli ne ugiba — je vzrok za skoraj vse, kar se začetniku zdi nelogično. Ko boš kasneje pisal navodila AI agentu, boš v resnici pisal recept za nekoga vmes: dovolj pameten, da ugane manjkajoče, a ravno zato nevaren, kadar ugane narobe.
>
> VEČ O TEM
>
> - Automate the Boring Stuff — Python Basics (brezplačna knjiga) — https://automatetheboringstuff.com/2e/chapter1/
> - Python: uradni uvodni vodič — https://docs.python.org/3/tutorial/introduction.html

Spremenljivka `variable`

Spremenljivka je poimenovano mesto, kjer je shranjena ena vrednost.

Levo od enačaja je ime `name`, desno vrednost `value`.

Vrednost zamenjaš na enem mestu in vse, kar jo uporablja, računa po novem.

ddv.py

```
1  ddv_stopnja = 22        # spremeni samo tukaj
2  cena_brez = 100
3
4  ddv = cena_brez * ddv_stopnja / 100
5  cena_z_ddv = cena_brez + ddv
6
7  print(ddv, cena_z_ddv)
```

IZPIS

```
22.0 122.0
```

Ce se stopnja spremeni na 9.5, popravis eno vrstico - ne sedemnajstih.

> Zakaj to šteje pri delu z AI: če je stopnja DDV zapisana neposredno na sedemnajstih mestih v kodi, je sprememba tvegana in agent jo bo skoraj zagotovo nekje spregledal. Če je zapisana v eni spremenljivki, je sprememba ena vrstica. Ko agentu rečeš *spremeni davčno stopnjo*, v ozadju prosiš ravno za to. Vrednost, zapisana kar sredi kode, ima svoje ime: `hard-coded value`.
>
> **VEČ O TEM**
>
> - Real Python — Variables in Python — https://realpython.com/python-variables/

Podatkovni tipi `data types`

Računalnik strogo loči med številom `int`, besedilom `str`, da/ne vrednostjo `bool` in datumom `date`.

Število `22` in besedilo `"22"` nista isto: prvo lahko sešteješ, drugo samo zlepiš.

Narekovaji so tisti, ki odločajo — kar je med njimi, je za računalnik besedilo, tudi če so same številke.

Presenetljivo velik del napak so podatki, ki izgledajo kot število, v resnici pa so besedilo.

tipi.py

```
1  kolicina = 22          # int    - stevilo
2  kolicina_txt = "22"    # str    - besedilo
3  je_placano = True      # bool   - da / ne
4
5  print(kolicina + 1)
6  print(kolicina_txt + "1")
7  print(type(kolicina), type(kolicina_txt))
```

IZPIS

```
23
221
<class 'int'> <class 'str'>
```

Vrstica 5 sesteje, vrstica 6 zlepi. Ista tipka, drugacen pomen.

> Klasičen primer iz pisarne: v Excelu je stolpec z zneski, v katerem je nekje vrednost `1.200,00 €`. Za človeka je to tisoč dvesto evrov, za računalnik pa niz znakov z evrom in vejico. Seštevek se zato ne izide ali pa program obstane. Kadar ti agent javi napako `TypeError: expected number, got string`, je to natanko ta primer — in rešitev je pretvorba besedila v število, ne popravljanje matematike.
>
> **VEČ O TEM**
>
> - Real Python — Basic Data Types in Python — https://realpython.com/python-data-types/
> - Python: števila, nizi in seznami — https://docs.python.org/3/tutorial/introduction.html

Seznam `list / array`

Seznam je oštevilčeno zaporedje vrednosti pod enim samim imenom.

Do posameznega elementa `item` prideš prek njegove številke `index`, pri čemer Python šteje od nič.

Seznam uporabiš vedno, kadar imaš *več istega*: vrstice v tabeli, datoteke v mapi, prijavljene udeležence.

seznam.py

```
1  gostje = ["Ana", "Bor", "Cvet", "Dan"]
2
3  print(gostje[0])        # prvi
4  print(gostje[3])        # zadnji
5  print(len(gostje))      # koliko jih je
6
7  gostje.append("Eva")    # dodaj na konec
8  print(gostje)
```

IZPIS

```
Ana
Dan
4
['Ana', 'Bor', 'Cvet', 'Dan', 'Eva']
```

gostje[4] bi vrgel napako IndexError - tak predal ne obstaja.

> Štetje od nič ni muha programerjev, ampak posledica tega, kako je seznam shranjen v pomnilniku: številka pove, koliko mest za začetkom je element. V praksi si zapomni samo to, da je prvi element pod številko 0 in zadnji pod *dolžina − 1*. Napaka `IndexError: list index out of range` pomeni, da si segel v predal, ki ne obstaja — skoraj vedno zato, ker si štel od ena.
>
> **VEČ O TEM**
>
> - Real Python — Lists and Tuples — https://realpython.com/python-lists-tuples/
> - Automate the Boring Stuff — Lists — https://automatetheboringstuff.com/2e/chapter4/

Objekt `object / dict / key-value`

Objekt je izpolnjen obrazec: vsako polje ima svoje ime `key` in svojo vrednost `value`.

Do podatka ne prideš prek številke, ampak prek imena polja.

Tako izgleda velika večina podatkov v datotekah `.json` in v odgovorih spletnih storitev.

objekt.py

```python
stranka = {
    "ime": "Novak",
    "email": "novak@primer.si",
    "znesek": 120,
    "placano": False,
}

print(stranka["email"])
print(stranka["znesek"] * 1.22)
```

IZPIS

```
novak@primer.si
146.39999999999998
```

Seznam objektov = tabela: vrstice so elementi, stolpci so imena polj.

> Seznam in objekt sta dva gradnika, iz katerih je sestavljeno skoraj vse. Seznam objektov (*več izpolnjenih obrazcev*) je natanko to, kar je tabela v Excelu: vrstice so elementi seznama, stolpci pa polja objekta. Mimogrede — izpis `146.39999999999998` ni napaka programa, ampak posledica tega, kako računalnik hrani decimalna števila `floating point`; pri denarju se temu izogneš z zaokroževanjem ali s štetjem celih centov.
>
> **VEČ O TEM**
>
> - Real Python — Dictionaries in Python — https://realpython.com/python-dicts/
> - Automate the Boring Stuff — Dictionaries and Structuring Data — https://automatetheboringstuff.com/2e/chapter5/

Pogoj `if / else`

Pogoj je kretnica: če nekaj drži, gre program po eni poti, sicer po drugi.

Zapiše se skoraj tako, kot ga izgovoriš — *če je znesek nad sto, daj popust, sicer polno ceno*.

Vsako poslovno pravilo, ki ga znaš povedati z besedo *če*, je mogoče zapisati kot kodo.

popust.py

```python
znesek = 120

if znesek > 100:
    popust = 0.10
elif znesek > 50:
    popust = 0.05
else:
    popust = 0.0

print("Popust:", popust * 100, "%")
```

IZPIS

```
Popust: 10.0 %
```

Zamik (presledki na zacetku vrstice) v Pythonu doloca, kaj spada pod kateri pogoj.

> Največ nesporazumov nastane pri pravilih, ki jih v pogovoru izrečemo nedokončana. *Popust dobijo stalne stranke* ne pove, kaj je stalna stranka in kaj se zgodi z ostalimi. Računalnik potrebuje vse veje `branch`. Zato te bo dober agent — in skill `grilling` iz Modula 11 — silil, da poveš tudi drugo polovico stavka.
>
> **VEČ O TEM**
>
> - Real Python — Conditional Statements — https://realpython.com/python-conditional-statements/
> - Automate the Boring Stuff — Flow Control — https://automatetheboringstuff.com/2e/chapter2/

Zanka for `for loop`

Zanka ponovi isti postopek za vsak element seznama.

`for` uporabiš takrat, ko že vnaprej veš, čez kaj greš: čez tristo vrstic, čez dvanajst mesecev, čez vse datoteke v mapi.

Namesto tristo ročnih ponovitev napišeš postopek enkrat in poveš, čez kaj naj teče.

zanka.py

```
1  racuni = [120, 45, 310]
2  skupaj = 0
3
4  for znesek in racuni:
5      skupaj = skupaj + znesek
6      print("Dodal", znesek, "-> skupaj", skupaj)
7
8  print("Konec:", skupaj)
```

IZPIS

```
Dodal 120 -> skupaj 120
Dodal 45 -> skupaj 165
Dodal 310 -> skupaj 475
Konec: 475
```

Zamaknjeni vrstici 5 in 6 se ponovita trikrat, zadnja vrstica samo enkrat.

> Zanka je razlog, zakaj se programiranje sploh splača. Delo, ki traja tri ure ročno, traja v zanki tri sekunde — in kar je pomembneje, drugič traja spet tri sekunde. Kadar se zalotiš, da nekaj v Excelu ponavljaš za vsako vrstico posebej, imaš pred sabo zanko. Ena ponovitev zanke se imenuje `iteration`.
>
> **VEČ O TEM**
>
> - Real Python — For Loops — https://realpython.com/python-for-loop/
> - Python: nadzor poteka programa — https://docs.python.org/3/tutorial/controlflow.html

Zanka while `while loop`

`while` ponavlja, dokler je nek pogoj izpolnjen — koliko ponovitev bo, vnaprej ne veš. Če se pogoj nikoli ne neha izpolnjevati, se program vrti v neskončnost `infinite loop` in ga moraš ustaviti ročno.

while.py

```
1  zaloga = 3
2
3  while zaloga > 0:
4      print("Prodano. Ostalo:", zaloga)
5      zaloga = zaloga - 1
6
7  print("Zaloge ni vec.")
```

IZPIS

```
Prodano. Ostalo: 3
Prodano. Ostalo: 2
Prodano. Ostalo: 1
Zaloge ni vec.
```

Ce izbrises vrstico 5, se pogoj nikoli ne spremeni - neskoncna zanka, Ctrl + C.

> Razlika je preprosta: `for` je *štirikrat*, `while` je *dokler*. Neskončna zanka je eden redkih primerov, ko se računalnik obnaša, kot da se je sesul, čeprav pridno dela točno to, kar si mu naročil.
>
> **VEČ O TEM**
>
> - Real Python — While Loops — https://realpython.com/python-while-loop/

Funkcija `function`

Funkcija je poimenovan kos kode: noter daš podatke `arguments`, ven dobiš rezultat `return value`.

Napišeš jo enkrat, nato jo po imenu pokličeš, kolikorkrat hočeš.

Dobra funkcija počne eno samo stvar in ima ime, iz katerega je razvidno, katero.

funkcija.py

```python
def z_ddv(cena, stopnja=22):
    return cena * (1 + stopnja / 100)

print(z_ddv(100))
print(z_ddv(250, 9.5))
print(z_ddv(80))
```

IZPIS

```
122.0
273.75
97.6
```

def = definiraj. return = kaj funkcija vrne. stopnja=22 je privzeta vrednost.

> Funkcija je najmanjša enota, ki se jo da samostojno preveriti s testom (Modul 9). Prav zato je pri delu z AI agentom pomembna: agent, ki ti napiše eno funkcijo z jasnim vhodom in izhodom, je preverljiv. Agent, ki ti napiše tristo vrstic v enem kosu, ni. Kadar rečeš *razbij to na manjše dele*, v resnici prosiš za funkcije.
>
> **VEČ O TEM**
>
> - Real Python — Defining Your Own Python Function — https://realpython.com/defining-your-own-python-function/
> - Automate the Boring Stuff — Functions — https://automatetheboringstuff.com/2e/chapter3/

Komentar `comment`

Komentar je vrstica, ki jo računalnik v celoti prezre, človek pa jo prebere.

Služi razlagi, *zakaj* je nekaj narejeno tako — kaj koda počne, se vidi iz kode same.

Pri delu z AI so komentarji dvojno koristni: agent jih prebere kot navodilo, ne le kot opombo.

komentar.py

```python
import math


# Racunovodstvo zahteva zaokrozevanje NAVZGOR
# (dogovor 2024). Zato ni round(), ampak ceil().
znesek = math.ceil(12.31)

print(znesek)
```

IZPIS

```
13
```

Slab komentar bi bil: "zaokrozi znesek". To se vidi ze iz kode.

> Slab komentar ponovi kodo z drugimi besedami. Dober komentar pove tisto, česar iz kode ni mogoče razbrati: da je ta izjema posledica zahteve računovodstva, da tega vrstnega reda ne smeš spremeniti, da je bilo prej drugače in zakaj se ni obneslo.
>
> **VEČ O TEM**
>
> - Real Python — Writing Comments in Python — https://realpython.com/python-comments-guide/

Terminal `terminal / command line / cmd`

Terminal je okno, v katerega pišeš ukaze, namesto da klikaš po gumbih.

En ukaz je ena vrstica: ime programa in za njim nastavitve `arguments`.

Poziv `prompt` na levi pove, v kateri mapi trenutno si — tam bodo ukazi tudi delovali.

Skoraj vse, kar počneš z miško, se da narediti tudi tu — le da je hitreje in ponovljivo.

Ukazni poziv (cmd)

```
C:\Users\Klemen> cd Documents\projekt



C:\Users\Klemen\Documents\projekt> dir

  prestej.py

  gostje.txt



C:\...\projekt> python prestej.py

Stevilo vrstic: 248



C:\...\projekt> _
```

Poziv se spremeni, ko z ukazom cd vstopis v drugo mapo.

`cd ime_mape`	vstopi v mapo
`cd ..`	nazaj v nadrejeno mapo
`cd /d D:\projekt`	skok na drug disk in v mapo
`dir`	izpiši, kaj je v mapi (na Macu in Linuxu `ls`)
`type gostje.txt`	izpiši vsebino datoteke (drugod `cat`)
`python prestej.py`	poženi program
`cls`	počisti zaslon
`Tab`	dopolni začeto ime datoteke
`↑ ↓`	prikliči prejšnje ukaze
`Ctrl + C`	ustavi, kar trenutno teče

> Črno okno, ki se odpre ob dvokliku na `zazeni-lokalno.bat`, je terminal. Vse, kar v njem piše, je program, ki govori s tabo. Dve stvari sta vredni zaupanja: `Ctrl + C` ustavi, kar teče, in zadnja vrstica napake je skoraj vedno najbolj uporabna. Na Windowsu obstajata dve različici — starejši `cmd` in novejši `PowerShell`; ukazi so večinoma isti, razlike pa se pokažejo šele pri zahtevnejših rečeh. Agenti, kot je Claude Code, živijo prav tu — zato je koristno, da te okno ne straši.
>
> **VEČ O TEM**
>
> - Microsoft — seznam ukazov za Windows — https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/windows-commands
> - Microsoft — ukaz cd — https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/cd

MODUL 2

Pravila igre

Zakaj se stvari pokvarijo, kako se napaka bere in v čem se koda bistveno razlikuje od umetne inteligence.

Smeti noter, smeti ven `garbage in, garbage out`

Računalnik ne preverja, ali so podatki smiselni — preveri samo, ali jih zna obdelati.

Če je med zneski eden zapisan kot besedilo, se seštevek ne zmoti, ampak se ustavi.

Večina *napak programa* v resnici ni napaka programa, ampak napaka podatkov, ki so vanj prišli.

gigo.py

```
1  zneski = [120, 45, "310"]   # zadnji je besedilo!
2  skupaj = 0
3
4  for z in zneski:
5      skupaj = skupaj + z
6
7  print(skupaj)
```

IZPIS

```
TypeError: unsupported operand type(s)
for +: 'int' and 'str'
```

Program ni pokvarjen. Vhodni podatek je.

> Enako velja za AI agente, le da je posledica drugačna: program se ustavi, agent pa nadaljuje. Če mu daš tabelo z nejasnimi stolpci in polovico manjkajočih vrednosti, ti bo vrnil odgovor — samozavesten in napačen. Zato se največ dela pri delu z AI ne zgodi pri pisanju navodil, ampak pri pripravi vhodnih podatkov.

VEČ O TEM

- Wikipedia — Garbage in, garbage out — https://en.wikipedia.org/wiki/Garbage_in,_garbage_out

Kaj je bug bug

Bug je razlika med tem, kar si mislil, da si naročil, in tem, kar si dejansko naročil.

Skoraj nikoli ne gre za to, da bi računalnik naredil napako — naredil je točno to, kar piše.

Najpogostejši bug je primer, na katerega nihče ni pomislil: prazen seznam, negativno število, manjkajoče polje.

bug.py

```
1  def povprecje(stevila):
2      return sum(stevila) / len(stevila)
3
4  print(povprecje([2, 4, 6]))
5  print(povprecje([]))        # kaj pa, ce je seznam prazen?
```

IZPIS

```
4.0
ZeroDivisionError: division by zero
```

Funkcija je pravilna za vse primere, razen za tistega, na katerega nismo pomislili.

> Beseda izvira iz leta 1947, ko so v računalnik Mark II dejansko zletele vešče in jih je bilo treba pobrati iz rele. Pomembnejše od zgodbe je to, kar iz nje sledi: iskanje buga ni iskanje krivca, ampak iskanje primera, ki ga navodila niso pokrila. Prav zato so testi (Modul 9) tako močno orodje — so seznam primerov, na katere si se spomnil, zapisan tako, da se preveri sam.
>
> **VEČ O TEM**
>
> - Wikipedia — Software bug — https://en.wikipedia.org/wiki/Software_bug

Napaka ali sesutje `error / exception`

Ko program naleti na nekaj, česar ne zna, sproži *izjemo* `exception` — in se ustavi.

Isto situacijo lahko vnaprej predvidiš in jo obvladaš, namesto da se program sesuje.

Razlika med *program se je sesul* in *program je javil, da manjka podatek* je samo v tem, ali je nekdo na to pomislil.

izjema.py

```
1  vnos = "dvanajst"
2
3  try:
4      stevilo = int(vnos)
5  except ValueError:
6      stevilo = 0
7      print("To ni stevilo, uporabljam 0.")
8
9  print("Rezultat:", stevilo)
```

IZPIS

```
To ni stevilo, uporabljam 0.
Rezultat: 0
```

Brez try/except bi se program na vrstici 4 ustavil z ValueError.

> Past je v tem, da se da z `try/except` napako tudi *pomesti pod preprogo*: če na tihem nadomestiš vsako napako z ničlo, bo program deloval naprej in tiho računal narobe. To je slabše od sesutja, ker sesutje vsaj opaziš. Dobro pravilo: obvladaj samo tiste napake, za katere veš, kaj z njimi.
>
> **VEČ O TEM**
>
> - Python — napake in izjeme — https://docs.python.org/3/tutorial/errors.html

Kako brati sporočilo o napaki `traceback / stack trace`

Sporočilo o napaki ni kazen, ampak najbolj natančen opis problema, kar ga boš dobil.

Bere se *od spodaj navzgor: zadnja vrstica pove,* kaj *je narobe, vrstice nad njo pa,* kje*.

Če boš agentu prilepil celotno sporočilo namesto opisa *ne dela*, bo rešitev prišla veliko hitreje.

Izpis napake

```
Traceback (most recent call last):

  File "racun.py", line 5, in <module>

    skupaj = skupaj + z

             ~~~~~~~^~~

TypeError: unsupported operand type(s)

for +: 'int' and 'str'
```

Isti primer kot pri 2.1, tokrat v celoti.

`TypeError: ...`	zadnja vrstica: kaj je narobe — preberi jo prvo
`File "racun.py", line 5`	katera datoteka in katera vrstica
`skupaj = skupaj + z`	vrstica kode, ki je sprožila napako
`~~~~~~~^~~`	puščica pokaže na točno mesto v vrstici
`Traceback ...`	pot, po kateri je program prišel do te vrstice

> Pri daljših programih je *traceback* dolg deset vrstic in vse izgledajo enako pomembne. Niso: zanima te zadnja vrstica z imenom napake in zadnja omemba *tvoje* datoteke. Vse vmes so notranje datoteke knjižnic, ki si jih ti nisi pisal. Ko delaš z agentom, prilepi celotno sporočilo — agent zna iz njega prebrati več kot ti, a samo, če ga dobi v celoti.
>
> **VEČ O TEM**
>
> - Real Python — Understanding Tracebacks — https://realpython.com/python-traceback/

Razhroščevanje `debugging`

Razhroščevanje je preverjanje domnev: kaj mislim, da je v tej spremenljivki, in kaj je v resnici.

Najstarejša metoda je še vedno med najboljšimi — izpiši vrednost in poglej.

Kadar nekaj *ne dela*, skoraj vedno drži ena tvoja domneva manj, kot si mislil.

preveri.py

```
1  zneski = [120, 45, "310"]
2
3  for z in zneski:
4      print(repr(z), type(z))   # zacasno: kaj je res notri?
```

IZPIS

```
120 <class 'int'>
45 <class 'int'>
'310' <class 'str'>
```

repr() pokaze narekovaje - takoj se vidi, kateri element je besedilo.

> Profesionalna orodja (*debugger*) znajo program ustaviti sredi izvajanja in pokazati vse vrednosti hkrati, a za začetek je izpis povsem dovolj. Pomembnejša od orodja je navada: namesto ugibanja, kje je napaka, razpolovi program — preveri na sredini, ali so vrednosti še prave. Tako v nekaj korakih zožiš iskanje s tristo vrstic na tri.
>
> **VEČ O TEM**
>
> - Real Python — Python Debugging With pdb — https://realpython.com/python-debugging-pdb/

Koda je predvidljiva, AI ni `deterministic / non-deterministic`

Ista koda z istim vhodom da *vsakič* isti rezultat — to se imenuje determinizem `deterministic`.

Jezikovni model z istim vprašanjem ne da nujno istega odgovora; deluje z verjetnostmi, ne s pravili.

To ni napaka modela, ampak njegova narava — in je razlog, zakaj kodo, ki jo napiše AI, vedno preveriš.

Zato se pravilo glasi: AI naj piše kodo, koda naj računa. Nikoli obratno.

determinizem.py

```python
def z_ddv(cena):
    return cena * 1.22

print(z_ddv(100))
print(z_ddv(100))
print(z_ddv(100))
```

IZPIS

```
122.0
122.0
122.0
```

Trikrat isto vprasanje, trikrat isti odgovor. Pri AI to ne drzi.

> Iz tega sledi praktično pravilo, ki ti bo prihranilo največ težav: AI agenta ne prosi, naj *izračuna* vsoto tvojih tristo računov — prosi ga, naj napiše program, ki jo izračuna. V prvem primeru dobiš številko, ki ji ne moreš zaupati in je ne moreš ponoviti. V drugem dobiš orodje, ki ga preveriš enkrat in uporabljaš stokrat. Ista razlika loči *vprašal sem AI* od *zgradil sem si aplikacijo*.
>
> **VEČ O TEM**
>
> - Anthropic — kako Claude deluje (dokumentacija) — https://docs.claude.com/en/docs/about-claude/models/overview

MODUL 3

Datoteke in formati

Katere datoteke zna AI brati in katerih ne — ter zakaj je ta ena razlika pomembnejša od vseh ostalih.

Datoteka, končnica, pot `file, extension, path`

Datoteka je kos podatkov z imenom; končnica `extension` za piko je samo obljuba, kaj je notri.

Pot `path` je naslov datoteke na disku — mape, ločene s poševnicami, in na koncu ime.

Ko agentu poveš *tale datoteka*, mu v resnici moraš povedati pot; drugače je ne najde.

poti.py

```python
from pathlib import Path

pot = Path("D:/projekt/racuni/januar.pdf")

print(pot.name)      # ime z koncnico
print(pot.suffix)    # samo koncnica
print(pot.parent)    # mapa, v kateri je
print(pot.exists())  # ali sploh obstaja
```

IZPIS

```
januar.pdf
.pdf
D:\projekt\racuni
False
```

Windows uporablja \, splet in Python pa /. Python razume oboje.

> Preimenovanje `podatki.txt` v `podatki.xlsx` ne naredi iz datoteke Excelove preglednice — spremeni samo obljubo. Windows privzeto končnice skriva, kar je vir presenetljivo velikega števila zmed; v Raziskovalcu jih vklopiš pod *Pogled → Pokaži → Končnice imen datotek*. Pri delu z agentom vedno navedi celotno pot ali pa datoteko postavi v mapo projekta.
>
> VEČ O TEM
>
> - Python — pathlib (delo s potmi) — https://docs.python.org/3/library/pathlib.html

Tekstovno ali binarno `text vs. binary`

Vse datoteke so v osnovi zaporedje števil; razlika je samo v tem, ali ta števila predstavljajo črke.

Tekstovno datoteko lahko odpreš v Beležnici in jo razumeš — binarno tudi odpreš, a vidiš smeti.

To je najpomembnejša delitev v celotnem gradivu: kar je tekst, AI bere neposredno; vse ostalo je treba najprej pretvoriti.

tekst_ali_binarno.py

```python
# tekstovna datoteka - berljiva
print(open("gostje.txt", encoding="utf-8").read(22))

# slika - isti postopek, drugacen rezultat
print(open("logo.png", "rb").read(12))
```

IZPIS

```
Ana Novak
Bor Kovac

b'\x89PNG\r\n\x1a\n\x00\x00\x00\r'
```

Oboje so datoteke. Prvo agent prebere, drugo mora najprej pretvoriti.

> Zato je smiselno vprašanje pred vsakim opravilom z AI: *ali je moj vhod tekst?* Če je, gre skoraj vse gladko. Če ni — slika, PDF iz skenerja, zaslonska slika tabele — je prvi korak pretvorba, ne pa boljši prompt. Velik del razočaranj nad AI izvira iz tega, da je bil vhod binaren, pričakovanje pa tekstovno.
>
> **VEČ O TEM**
>
> - Wikipedia — Optical character recognition — https://en.wikipedia.org/wiki/Optical_character_recognition

Navaden tekst in Markdown `.txt / .md`

`.txt` je gola vsebina brez oblikovanja — najbolj nedolžen format, kar jih je.

`.md` (Markdown) je isti navaden tekst, le z nekaj dogovorjenimi znaki za naslove, sezname in krepko pisavo.

Markdown je jezik, v katerem se pogovarjaš z agenti in v katerem pišeš datoteko `arhitektura.md` iz Modula 9.

markdown.py

```python
vsebina = """# Zapisnik sestanka

## Sklepi
- Rok: 30. september
- Odgovoren: **Klemen**

Podrobnosti so v [datoteki](porocilo.pdf).
"""

open("zapisnik.md", "w", encoding="utf-8").write(vsebina)
```

IZPIS

```
(datoteka zapisnik.md je zapisana)
```

Znaki #, - in ** so ves Markdown, ki ga potrebujes v 90 % primerov.

> Prednost Markdowna je, da je berljiv v obeh smereh: človek ga razume tudi brez prikazovalnika, računalnik pa ga zna spremeniti v lepo oblikovan dokument, spletno stran ali PDF. Prav zato je privzeti jezik dokumentacije pri razvijalcih in privzeti izhod jezikovnih modelov.

VEČ O TEM

- Markdown Guide — osnovna sintaksa — https://www.markdownguide.org/basic-syntax/

CSV — tabela kot tekst `.csv`

CSV je tabela, zapisana kot navaden tekst: ena vrstica je ena vrstica tabele, vejica loči stolpce.

Nima formul, barv, več listov in ne ve, kaj je datum — zna samo vrstice in stolpce.

Ravno zato je najzanesljivejši način, da podatke iz Excela spraviš v program ali v AI agenta.

beri_csv.py

```python
# gostje.csv je navaden tekst:
#   ime,email,znesek
#   Novak,novak@primer.si,120
#   Kovac,kovac@primer.si,45

import csv

with open("gostje.csv", encoding="utf-8") as f:
    for vrstica in csv.DictReader(f):
        print(vrstica["ime"], "->", vrstica["znesek"])
```

IZPIS

```
Novak -> 120
Kovac -> 45
```

Prva vrstica so imena stolpcev. Vsaka naslednja je en objekt iz pojma 1.5.

> Dve pasti sta slovenski: Excel pri nas privzeto loči stolpce s *podpičjem, ne z vejico, in decimalke piše z vejico. Če program prebere CSV narobe, je to skoraj vedno vzrok. Druga past je kodiranje: če se šumniki spremenijo v `Ĺˇ`, je datoteka shranjena v drugem naboru znakov — shrani jo kot* CSV UTF-8*.
>
> **VEČ O TEM**
>
> - Wikipedia — Comma-separated values — https://en.wikipedia.org/wiki/Comma-separated_values
> - Python — modul csv — https://docs.python.org/3/library/csv.html

JSON — struktura kot tekst `.json`

JSON je zapis objektov in seznamov iz Modula 1 v obliki navadnega teksta. Uporablja zavite oklepaje za objekte, oglate za sezname in narekovaje za imena polj. Skoraj vse, kar si dve napravi na internetu povesta, je zapisano v JSON.

json_primer.py

```python
import json

besedilo = '{"ime": "Novak", "znesek": 120, "placano": false}'

stranka = json.loads(besedilo)     # iz teksta v objekt
print(stranka["ime"], stranka["znesek"])

print(json.dumps(stranka, indent=2))   # nazaj v tekst
```

IZPIS

```
Novak 120
{
  "ime": "Novak",
  "znesek": 120,
  "placano": false
}
```

loads = beri, dumps = zapisi. Isti podatki, dve obliki.

> JSON je za razliko od CSV sposoben zapisati gnezdenje — stranka ima seznam naročil, vsako naročilo ima seznam postavk. Prav zato je privzeti jezik spletnih storitev (Modul 6) in tudi način, kako agent opiše orodje, ki ga zna uporabiti (MCP, Modul 10).
>
> **VEČ O TEM**
>
> - json.org — uradni opis formata — https://www.json.org/json-en.html
> - Python — modul json — https://docs.python.org/3/library/json.html

PDF `.pdf`

PDF je narejen za to, da je povsod videti enako — ne za to, da bi iz njega jemali podatke.

Notri sta lahko dve povsem različni stvari: pravo besedilo ali zgolj slika strani.

Od tega, katera od obeh je, je odvisno, ali bo agent datoteko prebral v sekundi ali sploh ne.

beri_pdf.py

```python
# pip install pypdf
from pypdf import PdfReader

stran = PdfReader("racun.pdf").pages[0]
besedilo = stran.extract_text()

print(len(besedilo), "znakov")
print(besedilo[:40])
```

IZPIS

```
0 znakov
```

Nic znakov pomeni, da je PDF nastal iz skena - potreben je OCR (3.11).

> Preprost test brez programiranja: odpri PDF in poskusi z miško označiti besedilo. Če se označi, je notri pravi tekst in ga bo agent prebral. Če se ne da označiti ničesar, gledaš sliko. Tretja, najbolj zoprna možnost so tabele: besedilo se izlušči, a se postavitev stolpcev izgubi, zato tabele iz PDF-jev vedno preveri.
>
> **VEČ O TEM**
>
> - pypdf — dokumentacija — https://pypdf.readthedocs.io/en/stable/

Wordove in Excelove datoteke `.docx / .xlsx`

`.docx` in `.xlsx` sta v resnici stisnjeni mapi (`.zip`) z datotekami XML notri. Vsebina je torej tekst, le zapakiran — zato jo programi in agenti berejo zanesljivo. Preimenuj `porocilo.docx` v `porocilo.zip`, odpri in prepričaj se sam.

docx_je_zip.py

```python
import zipfile


with zipfile.ZipFile("porocilo.docx") as z:
    for ime in z.namelist()[:4]:
        print(ime)
```

IZPIS

```
[Content_Types].xml
_rels/.rels
word/document.xml
word/styles.xml
```

word/document.xml je celotno besedilo dokumenta.

> To pojasni, zakaj agent brez težav prebere Wordov dokument, s sliko iste strani pa ima velike težave: prvo je zapakiran tekst, drugo so pike. Velja tudi obratno — ko od agenta zahtevaš Wordov dokument, ga v resnici sestavi iz teh XML delov.

VEČ O TEM

- Wikipedia — Office Open XML — https://en.wikipedia.org/wiki/Office_Open_XML

Slika ali risba `raster vs. vector (.png / .svg)`

`.jpeg` in `.png` sta mreži pik — povečaj ju in postanejo mehki.

`.svg` je navodilo za risanje, zapisano kot tekst: *črta od tu do tam, krog s tem polmerom*. SVG lahko agent prebere in spremeni; sliko iz pik lahko samo pogleda.

svg_je_tekst.py

```
1  svg = open("assets/img/logo.svg", encoding="utf-8").read()
2
3  print(svg[:90])
```

IZPIS

```
<svg role="img" aria-label="Zlata ovca" version="1.1"
 xmlns="http://www.w3.org/2000/svg" viewBox=
```

To je logotip te predstavitve. Je navaden tekst - zato se barva po temi strani.

> Praktična posledica: logotipe, ikone, diagrame in grafe hrani v SVG, fotografije pa v JPEG. SVG je poljubno velik brez izgube kakovosti, zavzame malo prostora in ga lahko kadarkoli prebarvaš — tudi z agentom, ki mu preprosto rečeš, naj spremeni barvo. Pri fotografiji to ni mogoče, ker v njej ni oblik, samo pike.
>
> **VEČ O TEM**
>
> - MDN — SVG — https://developer.mozilla.org/en-US/docs/Web/SVG

HTML `.html`

HTML je tekst z oznakami, ki povedo, kaj je naslov, kaj odstavek in kaj povezava.

Brskalnik ga ne prikaže kot tekst, ampak ga *izvede* in nariše stran.

Datoteka `.html` je samostojna — dvoklik jo odpre tudi brez interneta.

naredi_stran.py

```
1 html = """<!doctype html>
2 <html lang="sl">
3   <body>
4     <h1>Pozdravljeni</h1>
5     <p>To je <b>spletna stran</b>.</p>
6   </body>
7 </html>"""
8
9 open("stran.html", "w", encoding="utf-8").write(html)
```

IZPIS

```
(dvoklik na stran.html odpre brskalnik)
```

Gradivo, ki ga berete, je ena taka datoteka. Vec v Modulu 6.

> Prav ta lastnost — da je stran samo datoteka — je razlog za Modul 7: spletno tehnologijo lahko uporabiš za povsem lokalno aplikacijo, ki ne potrebuje niti interneta niti namestitve. Na službenem računalniku, kjer ne smeš nameščati programov, je to pogosto edina pot do lastnega orodja.

VEČ O TEM

- MDN — HTML — https://developer.mozilla.org/en-US/docs/Web/HTML

Kateri formati so AI-prijazni `AI-friendly formats`

Pravilo je preprosto: bližje kot je format navadnemu tekstu, manj težav boš imel.

Če imaš na voljo izvor podatkov, ga uporabi — izvozi CSV namesto da pošlješ zaslonsko sliko tabele.

Pretvorba je vedno prvi korak, ne boljši prompt.

`.txt .md`	odlično — čisti tekst, nobene pretvorbe
`.csv`	odlično — tabela kot tekst
`.json`	odlično — struktura kot tekst
`.html .svg`	odlično — tekst z oznakami
`.docx .xlsx`	dobro — zapakiran tekst, bere se zanesljivo
`.pdf iz besedila`	srednje — besedilo se izlušči, postavitev tabel se izgubi
`.pdf iz skena`	slabo — najprej OCR
`.jpeg .png`	slabo — model vidi sliko, ne podatkov
`zaslonska slika tabele`	najslabše — skoraj vedno obstaja izvorna datoteka

> Vrstni red v tabeli je hkrati vrstni red, po katerem naj iščeš vir podatkov. Preden fotografiraš zaslon, se vprašaj, ali obstaja izvoz. Preden pošlješ PDF, se vprašaj, ali obstaja Excel, iz katerega je nastal. Vsak korak nazaj proti izvoru ti prihrani eno pretvorbo in eno mesto, kjer se lahko izgubijo podatki.
>
> **VEČ O TEM**
>
> - Anthropic — delo z datotekami in dokumenti — https://docs.claude.com/en/docs/build-with-claude/files

OCR — iz slike v tekst OCR

OCR `optical character recognition` je postopek, ki iz slike prepozna črke in vrne navaden tekst.

Deluje v korakih: poravna sliko, jo očisti, poišče vrstice in oblike znakov ter za vsako ugane črko.

Ker gre za ugibanje, dela napake — najpogosteje zamenja `0` in `O` ter `1` in `l`.

Rezultat OCR zato vedno preveri, še posebej pri zneskih in davčnih številkah.

ocr.py

```python
# pip install pytesseract pillow  (+ program Tesseract)
import pytesseract
from PIL import Image

slika = Image.open("racun_sken.png")
besedilo = pytesseract.image_to_string(slika, lang="slv")

print(besedilo[:60])
```

IZPIS

```
RACUN st. 2026-0148
Datum: 12.09.2026
Znesek: 1.2O0,00 EUR
```

Poglej zadnjo vrstico: 1.2O0 - OCR je namesto nicle prepoznal crko O.

> Sodobni jezikovni modeli znajo sliko prebrati tudi sami, brez posebnega OCR programa, in so pri razgibanih postavitvah pogosto boljši. Slabost je ista kot pri klasičnem OCR — ugibanje ostane ugibanje. Dobro pravilo za pisarno: OCR uporabi za iskanje in pregled, nikoli pa ne prepiši zneska iz OCR v računovodstvo brez pogleda na izvirnik.
>
> VEČ O TEM
>
> - Tesseract OCR (odprtokodni program) — https://github.com/tesseract-ocr/tesseract
> - Wikipedia — Optical character recognition — https://en.wikipedia.org/wiki/Optical_character_recognition

MODUL 4

Podatki in baze

Od Excelove tabele do prave baze podatkov, v štirih korakih in brez preskoka.

Excel kot izhodišče `spreadsheet`

Excelova tabela je že skoraj baza podatkov: ima stolpce z imeni in vrstice z vrednostmi.

Formula v Excelu je isto kot vrstica kode — le da jo moraš vsakič znova povleči po stolpcu.

Program naredi isti izračun, a ga lahko jutri ponoviš na drugi datoteki, ne da bi se česa dotaknil.

sestej.py

```
# V Excelu bi napisal:  =SUM(C2:C4)
# V Pythonu:

import csv

with open("gostje.csv", encoding="utf-8") as f:
    zneski = [int(v["znesek"]) for v in csv.DictReader(f)]

print(sum(zneski))
```

IZPIS

```
475
```

Isti rezultat. Razlika je v tem, da to kodo jutri pozenes na 300 datotekah.

> Meja Excela se pokaže na treh mestih: ko podatkov ni več mogoče obdržati v eni tabeli, ko mora do njih hkrati dostopati več ljudi, in ko je treba isti postopek ponoviti vsak teden. Prvi dve težavi rešuje baza, tretjo program. Dokler nobena od teh treh ne boli, je Excel povsem spodobna izbira in tega ti ne bo povedal noben razvijalec.

VEČ O TEM

- Python — modul csv — https://docs.python.org/3/library/csv.html

Isti podatki, štiri oblike `same data, four shapes`

Ena sama vrstica podatkov se zapiše na štiri načine, ki jih boš srečeval povsod.

Vsebina je v vseh štirih enaka — razlikuje se le zapis in to, kdo ga bere.

Ko to enkrat vidiš, prenehajo biti CSV, JSON in SQL tri različne skrivnosti.

stiri_oblike.py

```python
# 1) Excel - vrstice in stolpci
#      ime    | znesek
#      Novak  |    120

# 2) CSV - tabela kot tekst
csv_zapis = "ime,znesek\nNovak,120"

# 3) JSON - objekt kot tekst
json_zapis = '[{"ime": "Novak", "znesek": 120}]'

# 4) SQL - ukaz bazi
sql_zapis = "INSERT INTO stranke (ime, znesek) VALUES ('Novak', 120);"
```

Stirje zapisi, ena sama informacija: Novak, 120.

> Izbira med njimi ni vprašanje okusa, ampak namena. CSV je za izmenjavo. JSON je za pogovor med programi. SQL je za shranjevanje in iskanje. Excel je za človeka, ki hoče videti in popraviti. Skoraj vsako delo s podatki je v resnici prehajanje med temi štirimi oblikami.

VEČ O TEM

- Wikipedia — Comma-separated values — https://en.wikipedia.org/wiki/Comma-separated_values

Kaj je baza podatkov `database`

Baza je program, ki hrani podatke in zna hitro odgovarjati na vprašanja o njih.

Od datoteke se loči po treh stvareh: vodi red, zna iskati med milijoni vrstic in prenese več hkratnih uporabnikov.

Najpreprostejša baza je ena sama datoteka — SQLite je vgrajen že v Python, brez namestitve.

baza.py

```python
import sqlite3

baza = sqlite3.connect("podjetje.db")    # nastane datoteka

baza.execute("CREATE TABLE IF NOT EXISTS stranke ("
             "id INTEGER PRIMARY KEY, ime TEXT, znesek INTEGER)")
baza.execute("INSERT INTO stranke (ime, znesek) VALUES ('Novak', 120)")
baza.commit()

print(baza.execute("SELECT * FROM stranke").fetchall())
```

IZPIS

```
[(1, 'Novak', 120)]
```

Cela baza je ena datoteka podjetje.db. Prav to uporablja tvoj telefon za sporocila.

> Velike baze (PostgreSQL, MySQL, SQL Server) so isti koncept, le da tečejo kot samostojen strežnik, do katerega se poveže več programov hkrati. Za osebno orodje ali prototip je SQLite skoraj vedno prava izbira: nič za namestiti, nič za nastaviti, in celotno bazo prekopiraš tako, da prekopiraš datoteko.

VEČ O TEM

- Python — modul sqlite3 — https://docs.python.org/3/library/sqlite3.html
- SQLite — kdaj ga uporabiti — https://www.sqlite.org/whentouse.html

Relacijska baza in ključi `relational database, primary/foreign key`

Relacijska baza hrani podatke v več tabelah, ki so med sabo povezane.

Vsaka vrstica ima svojo enolično številko — primarni ključ `primary key`.

Druga tabela se nanjo sklicuje s to isto številko — tujim ključem `foreign key`.

Tako se podatek o stranki zapiše enkrat, čeprav ima stranka petdeset naročil.

kljuci.py

```
baza.execute("CREATE TABLE narocila ("
             "id INTEGER PRIMARY KEY,"
             "stranka_id INTEGER,"      # tuji kljuc -> stranke.id
             "znesek INTEGER)")

baza.execute("INSERT INTO narocila (stranka_id, znesek) VALUES (1, 120)")
baza.execute("INSERT INTO narocila (stranka_id, znesek) VALUES (1, 45)")
baza.commit()

vrstice = baza.execute(
    "SELECT stranke.ime, narocila.znesek "
    "FROM narocila JOIN stranke ON stranke.id = narocila.stranka_id"
).fetchall()

print(vrstice)
```

IZPIS

```
[('Novak', 120), ('Novak', 45)]
```

Ime 'Novak' je v bazi zapisano enkrat samkrat, pa se pojavi v obeh vrsticah.

> Prav zaradi te lastnosti je relacijska baza tako močna: ko stranka spremeni e-naslov, ga popraviš na enem mestu in pravilen je povsod. V Excelu bi ga moral popraviti v vseh petdesetih vrsticah — in ena bi zagotovo ostala stara. To je isti princip kot spremenljivka iz pojma 1.2, le na ravni podatkov.

VEČ O TEM

- Wikipedia — Relational database — https://en.wikipedia.org/wiki/Relational_database

SQL `SQL`

SQL je jezik za pogovor z bazo in je presenetljivo podoben angleškemu stavku.

Povedati ti ni treba, *kako* naj baza podatke poišče — samo, *kaj* hočeš.

Štiri besede pokrijejo večino vsega: `SELECT`, `FROM`, `WHERE`, `ORDER BY`.

poizvedba.sql

```sql
SELECT stranke.ime,
       SUM(narocila.znesek) AS skupaj
FROM narocila
JOIN stranke ON stranke.id = narocila.stranka_id
WHERE narocila.znesek > 50
GROUP BY stranke.ime
ORDER BY skupaj DESC
LIMIT 10;
```

IZPIS

```
Novak | 120
```

Prevod: vzemi naročila nad 50, sestej jih po strankah, uredi padajoce, vrni prvih deset.

SQL je vreden poznavanja tudi, če ne boš nikoli programiral: ko AI agentu rečeš, naj ti pripravi poročilo iz baze, bo napisal SQL — in ti boš moral znati prebrati, ali je zajel prave vrstice. Najpogostejša napaka ni sintaksa, ampak pozabljen pogoj `WHERE`, zaradi katerega poročilo tiho vključi tudi storniranje in testne vnose.

VEČ O TEM

- SQLite — jezik SQL — https://www.sqlite.org/lang.html
- Wikipedia — SQL — https://en.wikipedia.org/wiki/SQL

Dokumentna baza `document database / NoSQL`

Dokumentna baza ne hrani vrstic, ampak cele objekte — take, kot si jih videl pri JSON.

Vsak zapis ima lahko drugačna polja; vnaprej dogovorjene oblike ni.

Prednost je svoboda, cena pa je, da red nad podatki ni več naloga baze, ampak tvoja.

dokument.py

```python
# En dokument - v relacijski bazi bi bili to dve tabeli
stranka = {
    "ime": "Novak",
    "email": "novak@primer.si",
    "narocila": [
        {"datum": "2026-09-01", "znesek": 120},
        {"datum": "2026-09-14", "znesek": 45},
    ],
}

print(stranka["narocila"][0]["znesek"])
```

IZPIS

```
120
```

MongoDB, Firestore in podobne baze hranijo natanko taksne dokumente.

> Nevarnost dokumentnih baz je tiha: ker vnaprejšnje oblike ni, se čez leto dni v isti zbirki znajdejo zapisi s poljem `email`, `e-mail` in `mail`. Baza ne bo javila ničesar, poročilo pa bo izpustilo tretjino strank. V relacijski bazi bi tak vnos preprosto zavrnila.

VEČ O TEMI

- Wikipedia — Document-oriented database — https://en.wikipedia.org/wiki/Document-oriented_database

Kdaj katera `choosing a store`

Izbira je skoraj vedno odvisna od enega vprašanja: ali so podatki povsod enake oblike?

Če so, vzemi relacijsko bazo — pomagala ti bo obdržati red.

Če niso in se oblika še išče, je dokumentna baza hitrejša pot do delujočega prototipa.

`Excel / CSV`	do nekaj tisoč vrstic, en uporabnik, ročno delo
`SQLite`	osebno orodje, prototip, ena aplikacija — brez namestitve
`PostgreSQL / MySQL`	več uporabnikov hkrati, pomembni podatki, poročila
`MongoDB / Firestore`	oblika zapisov se še spreminja, veliko gnezdenja
`JSON datoteka`	nastavitve in majhni seznami — ne pa evidenca

> Praktičen nasvet za delo z agentom: povej mu, koliko vrstic pričakuješ, koliko ljudi bo uporabljalo orodje in ali se oblika podatkov še spreminja. To so tri vprašanja, iz katerih izhaja odločitev — brez njih bo agent izbral tisto, kar je videl najpogosteje, kar je pogosto pretirano za tvoj primer.
>
> **VEČ O TEM**
>
> - SQLite — kdaj ga uporabiti — https://www.sqlite.org/whentouse.html

MODUL 5

Naslovi in omrežje

Kaj je pravzaprav naslov spletne strani in kako računalnik najde drug računalnik.

Anatomija naslova URL

Naslov spletne strani ni ena beseda, ampak šest ločenih delov, od katerih ima vsak svojo nalogo.

Ko enkrat vidiš, kje se en del konča in drug začne, naslovi nehajo biti skrivnost.

Del za vprašajem je poizvedba `query` — prav ta v tem gradivu nosi izbiro jezika.

naslov.py

```python
from urllib.parse import urlparse

u = urlparse("https://www.zlataovca.si:443/gradivo/index.html?lang=sl#modul3")

print(u.scheme)     # protokol
print(u.hostname)   # gostitelj (domena)
print(u.port)       # vrata
print(u.path)       # pot do datoteke
print(u.query)      # poizvedba
print(u.fragment)   # odsek na strani
```

IZPIS

```
https
www.zlataovca.si
443
/gradivo/index.html
lang=sl
modul3
```

Poglej naslov te strani v brskalniku - ?lang=sl je isti del kot v tem primeru.

> Dve praktični posledici. Prvič: vse za vprašajem je viden vsakomur na poti in se zapisuje v dnevnike strežnikov, zato tja nikoli ne sodijo gesla ali osebni podatki. Drugič: `#odsek` se sploh ne pošlje strežniku — to je navodilo brskalniku, kam naj se pomakne na že naloženi strani.
>
> **VEČ O TEM**
>
> - MDN — kaj je URL — https://developer.mozilla.org/en-US/docs/Web/API/URL
> - Python — urllib.parse — https://docs.python.org/3/library/urllib.parse.html

Domena in DNS `domain, DNS`

Računalniki se med sabo ne kličejo po imenih, ampak po številkah.

DNS je imenik, ki ime `example.com` prevede v številko naslova.

Domeno si najameš pri registrarju za nekaj deset evrov na leto — to je najem imena, ne strežnika.

dns.py

```python
import socket


print(socket.gethostbyname("example.com"))
print(socket.gethostbyname("www.python.org"))
```

IZPIS

```
172.66.147.243
151.101.64.223
```

Ista domena lahko cez teden vrne drugo stevilko - imenik se spreminja.

> Zato ob selitvi spletne strani traja nekaj ur, preden jo vsi vidijo na novem mestu: imenik se ne posodobi povsod hkrati, saj si vmesni strežniki odgovore nekaj časa zapomnijo. Ista lastnost pojasni, zakaj stran včasih deluje tebi, kolegu pa ne — pri njem je še vedno shranjen star odgovor.
>
> **VEČ O TEM**
>
> - Wikipedia — Domain Name System — https://en.wikipedia.org/wiki/Domain_Name_System

IP naslov `IP address`

IP naslov je hišna številka računalnika v omrežju, zapisana kot štiri števila med 0 in 255.

Nekateri razponi so rezervirani za domača in pisarniška omrežja in na internetu ne obstajajo.

Prav zato tvojega računalnika od zunaj ni mogoče doseči brez posebne pomoči.

`127.0.0.1`	ta računalnik, vedno in povsod (`localhost`)
`192.168.x.x`	domače ali pisarniško omrežje
`10.x.x.x`	večja podjetniška omrežja
`172.16–31.x.x`	prav tako zasebno
`vse ostalo`	javni naslov, dosegljiv z interneta

> Ker so zasebni razponi povsod isti, ima tvoj računalnik doma in računalnik v pisarni lahko popolnoma enak naslov `192.168.1.10` — in to ni težava, ker sta v ločenih omrežjih. Posledica pa je, da nekdo z interneta ne more preprosto vpisati tvojega naslova in odpreti tvoje strani. To rešuje tunel iz pojma 7.3.
>
> **VEČ O TEM**
>
> - Wikipedia — Private network — https://en.wikipedia.org/wiki/Private_network

localhost `localhost / 127.0.0.1`

`localhost` pomeni *ta računalnik* — naslov, ki nikoli ne zapusti tvoje naprave.

Ko poženeš `zazeni-lokalno.bat`, teče strežnik prav tu in nihče drug ga ne vidi.

To je najvarnejši prostor za preizkušanje: internet o njem ne ve ničesar.

server.py

```
1  import http.server, socketserver
2
3  with socketserver.TCPServer(("127.0.0.1", 8080),
4                              http.server.SimpleHTTPRequestHandler) as srv:
5      print("Odpri http://localhost:8080")
6      srv.serve_forever()
```

IZPIS

```
Odpri http://localhost:8080
```

Tri vrstice so cel spletni streznik. To gradivo tece na skoraj isti kodi.

> Opazi `127.0.0.1` v kodi: strežnik posluša samo na tem naslovu, zato je dosegljiv izključno s tega računalnika. Če bi hotel, da ga vidijo tudi drugi v pisarni, bi moral poslušati na `0.0.0.0`, kar pomeni *na vseh omrežnih karticah*. Prav v tej eni številki je razlika med pojmoma 5.4 in 5.5.
>
> **VEČ O TEM**
>
> - Python — http.server — https://docs.python.org/3/library/http.server.html

Lokalno omrežje `LAN, 192.168.x.x`

Ko strežnik posluša na vseh omrežnih karticah, ga vidijo vsi, ki so na isti wifi mreži.

Takrat ne uporabiš `localhost`, ampak naslov tvojega računalnika v tej mreži.

To je najhitrejši način, da nekaj pokažeš kolegu ali odpreš na svojem telefonu.

lan.py

```python
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.connect(("8.8.8.8", 80))      # ne poslje nicesar, samo izbere kartico
print("http://" + s.getsockname()[0] + ":8080")
s.close()
```

IZPIS

```
http://192.168.1.24:8080
```

Prav to vrstico izpise zazeni-lokalno.bat - odpri jo na telefonu.

> Omejitev je, da mora biti druga naprava v isti mreži. Na telefonu to pomeni, da mora biti na wifi in ne na mobilnih podatkih. V podjetjih je pogosto tudi gostujoče wifi omrežje ločeno od službenega, zato naslov deluje z enega, z drugega pa ne — in to ni napaka programa, ampak nastavitev omrežja.
>
> **VEČ O TEM**
>
> - Wikipedia — Private network — https://en.wikipedia.org/wiki/Private_network

Vrata `port`

Na enem računalniku teče več programov hkrati; vrata `port` povedo, kateri od njih naj sprejme povezavo.

Naslov je stavba, vrata so številka stanovanja.

Če dobiš napako *port already in use*, v tem stanovanju že nekdo stanuje — izberi drugo številko.

`80`	navadne spletne strani (http)
`443`	varne spletne strani (https) — privzeto, zato ga ne vidiš
`8080 8000 5000`	razvojni strežniki, ki jih poganjaš sam
`3306 5432`	baze podatkov (MySQL, PostgreSQL)
`22`	oddaljena prijava na strežnik (SSH)

> Zato v naslovu spletne strani vrat običajno ne vidiš: brskalnik pri `https://` sam doda 443. Pri lastnem strežniku jih moraš navesti, ker 8080 ni privzet. Če te zanima, kaj na tvojem računalniku trenutno zaseda vrata, ti to v ukaznem pozivu pove `netstat -ano | findstr 8080`.
>
> **VEČ O TEM**
>
> - Wikipedia — seznam vrat TCP in UDP — https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

Zahteva in odgovor `HTTP request / response`

Splet deluje po preprostem vzorcu: brskalnik pošlje zahtevo, strežnik vrne odgovor, povezava se zapre.

Zahteva pove metodo (`GET` ali `POST`), pot in nekaj dodatnih podatkov v glavah `headers`.

Odgovor vrne statusno kodo, glave in vsebino — datoteko, sliko ali JSON.

zahteva.py

```python
import urllib.request

with urllib.request.urlopen("http://localhost:8080/index.html") as odgovor:
    print(odgovor.status)
    print(odgovor.headers["Content-Type"])
    print(len(odgovor.read()), "bajtov")
```

IZPIS

```
200
text/html; charset=utf-8
2184 bajtov
```

Tocno to naredi brskalnik, ko vpises naslov - le da vsebino se narise.

> Pomembna lastnost je, da si strežnik med dvema zahtevama ničesar ne zapomni — vsaka zahteva je nova in prazna. Zato obstajajo piškotki in prijavni žetoni: da lahko brskalnik ob vsaki zahtevi znova pove, kdo si. Ta ista lastnost je razlog, da AI agenti potrebujejo kontekst, ki se pošlje ob vsakem sporočilu — o tem v Modulu 10.
>
> **VEČ O TEM**
>
> - MDN — pregled protokola HTTP — https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

Statusne kode `status codes`

Vsak odgovor se začne s trimestno številko, ki pove, kako se je zahteva iztekla.

Prva števka je vse, kar si moraš zapomniti: 2 je v redu, 3 je preusmeritev, 4 je tvoja napaka, 5 je njihova.

Znamenita 404 torej ne pomeni *pokvarjeno*, ampak *tega naslova ni*.

`200 OK`	vse je v redu, vsebina je priložena
`301 / 302`	vsebina se je preselila, pojdi drugam
`400 Bad Request`	zahteva je napačno sestavljena
`401 / 403`	nisi prijavljen / nimaš dovoljenja
`404 Not Found`	na tem naslovu ni ničesar
`429 Too Many Requests`	preveč zahtev prehitro — počakaj
`500 Internal Server Error`	strežnik se je sesul — napaka je pri njih
`503 Service Unavailable`	strežnik je preobremenjen ali na vzdrževanju

> Razlika med 4xx in 5xx je pri delu z agenti zelo praktična: če dobiš 401 ali 403, je treba popraviti ključ ali dovoljenja na tvoji strani. Če dobiš 500, popravljanje prompta ne bo pomagalo — težava je pri storitvi in edino smiselno je počakati ali javiti. Koda 429 pomeni, da si dosegel omejitev števila zahtev; takrat se splača vgraditi premor med klici.
>
> **VEČ O TEM**
>
> - MDN — statusne kode HTTP — https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

MODUL 6

Kako deluje spletna stran

Kaj se v resnici zgodi med klikom na gumb in prikazom rezultata.

Brskalnik je izvajalec `browser / rendering engine`

Brskalnik ni okno v internet, ampak program, ki datoteke prevzame in jih *izvede* na tvojem računalniku.

Ko vpišeš naslov, prenese nekaj datotek, jih prebere in iz njih nariše stran.

Vse, kar potem vidiš in klikaš, se dogaja lokalno — ne na strežniku.

Kaj brskalnik prenese

```
GET /index.html          -> 2 KB  (vsebina in zgradba)

GET /assets/css/style.css -> 12 KB  (videz)

GET /assets/js/app.js     -> 18 KB  (obnasanje)

GET /assets/img/logo.svg  -> 25 KB  (slika)



Skupaj: 4 zahteve, 57 KB. Stran je narisana.
```

To so dejanske datoteke tega gradiva. Odpri F12 -> Network in poglej sam.

> Prav zato je mogoče stran odpreti tudi brez strežnika: če datoteke že imaš na disku, brskalnik nima kaj prenašati in jih preprosto prebere. Dvoklik na `index.html` naredi natanko to. Ta lastnost je temelj Modula 7 in razlog, zakaj lahko na službenem računalniku, kjer ne smeš ničesar nameščati, vseeno poganjaš lastno aplikacijo.

VEČ O TEM

- MDN — tvoja prva spletna stran — https://developer.mozilla.org/en-US/docs/Learn_web_development/Getting_started/Your_first_website

HTML — zgradba HTML

HTML pove, *kaj* je na strani: naslov, odstavek, seznam, gumb, slika.

Oznake so v koničastih oklepajih in se skoraj vedno pojavijo v paru — odprta in zaprta.

O videzu HTML ne pove ničesar; to je naloga CSS.

index.html

```
<!doctype html>
<html lang="sl">
  <body>
    <h1>Gostje</h1>
    <ul id="seznam">
      <li>Ana Novak</li>
    </ul>
    <button id="dodaj">Dodaj gosta</button>
  </body>
</html>
```

Vsak element ima lahko id - to je ime, po katerem ga pokliceta CSS in JavaScript.

> Ko brskalnik HTML prebere, ga spremeni v drevo elementov, ki mu pravimo DOM. To drevo je tisto, kar JavaScript spreminja — in prav zato se stran lahko spremeni, ne da bi se znova naložila. Ko agentu rečeš *dodaj gumb*, bo v resnici dodal eno vrstico v to strukturo.
>
> **VEČ O TEM**
>
> - MDN — HTML — https://developer.mozilla.org/en-US/docs/Web/HTML
> - MDN — kaj je DOM — https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

CSS — videz `CSS`

CSS pove, *kako* naj stvari izgledajo: barve, velikosti, razmiki, postavitev.

Pravilo izbere elemente in jim določi lastnosti — nič več kot to.

Ista HTML stran z drugim CSS je videti kot povsem drug izdelek.

style.css

```css
:root {
  --accent: #d97757;      /* ena barva na enem mestu */
}

button {
  background: var(--accent);
  color: white;
  border: 0;
  border-radius: 8px;
  padding: 0.5rem 1rem;
}
```

To je res koda tega gradiva. Spremeni --accent in vse skupaj zamenja barvo.

> Zapis `--accent` je spremenljivka iz pojma 1.2, le v CSS. Prav na tem temelji preklop med temno in svetlo temo na tej strani: HTML se ne spremeni niti za znak, zamenja se samo peščica barvnih spremenljivk. Če boš kdaj agentu rekel *spremeni barvno shemo*, je to mesto, kjer bo delal.
>
> **VEČ O TEM**
>
> - MDN — CSS — https://developer.mozilla.org/en-US/docs/Web/CSS

JavaScript — obnašanje `JavaScript / TypeScript`

JavaScript je edini programski jezik, ki ga brskalnik razume neposredno.

Poskrbi za vse, kar se zgodi *potem*: klik, vnos, preverjanje, spreminjanje strani.

TypeScript je isti jezik z dodanimi podatkovnimi tipi iz pojma 1.3 — napake se pokažejo že med pisanjem.

app.js

```
1  const gumb = document.getElementById("dodaj");
2  const seznam = document.getElementById("seznam");
3
4  gumb.addEventListener("click", function () {
5      const vrstica = document.createElement("li");
6      vrstica.textContent = "Nov gost";
7      seznam.appendChild(vrstica);
8  });
```

Klik na gumb doda vrstico. Stran se pri tem ne nalozi znova.

> Opazi, da ta primer nikoli ne pokliče strežnika — vrstica se doda samo v brskalniku in ob osvežitvi strani izgine. To je ključna razlika med *videti se je spremenilo* in *shranjeno je*. Za drugo potrebuješ API klic in bazo, kar sta naslednja dva pojma.

VEČ O TEM

- MDN — JavaScript — https://developer.mozilla.org/en-US/docs/Web/JavaScript
- TypeScript — dokumentacija — https://www.typescriptlang.org/docs/

Frontend in backend `frontend / backend`

Frontend je vse, kar teče v brskalniku pri uporabniku — HTML, CSS, JavaScript.

Backend je program, ki teče na strežniku in hrani podatke ter skrbi za pravila.

Meja med njima je pomembna zato, ker je frontend viden in spremenljiv vsakomur.

kje_tece.py

```
# BACKEND - tece na strezniku, uporabnik ga ne vidi
def sme_videti_racun(uporabnik, racun):
    return racun.lastnik_id == uporabnik.id       # <- pravo preverjanje

# FRONTEND - tece v brskalniku, vsak ga lahko prebere in spremeni
# if (uporabnik.jeLastnik) { prikaziGumb(); }     <- samo videz
```

Skrivanje gumba ni varnost. Varnost je preverjanje na strezniku.

> To je najpogostejša varnostna napaka začetnikov in tudi AI agentov: pravilo se preveri samo v brskalniku. Kdorkoli lahko odpre razvijalska orodja, spremeni kodo in gumb prikaže nazaj. Pravilo se zato glasi: frontend skrbi za udobje, backend za resnico. Kar je pomembno, mora biti preverjeno tam, kjer uporabnik nima dostopa.

VEČ O TEM

- MDN — kaj je API — https://developer.mozilla.org/en-US/docs/Glossary/API

API klic `API call`

API je dogovorjen seznam vprašanj, ki jih program lahko postavi drugemu programu. Brskalnik pošlje zahtevo na naslov, strežnik vrne JSON — brez slik in brez oblikovanja. Ista vrata uporabljajo mobilne aplikacije, drugi programi in AI agenti.

klic.js

```
async function shraniGosta(ime) {
    const odgovor = await fetch("/api/gostje", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ ime: ime })
    });

    if (!odgovor.ok) throw new Error("Napaka " + odgovor.status);
    return await odgovor.json();
}
```

IZPIS

```
{ "id": 42, "ime": "Nov gost", "shranjeno": true }
```

POST pomeni 'shrani', GET pomeni 'daj mi'. Odgovor je JSON iz pojma 3.5.

> Zdaj se sestavi celotna slika: JSON iz Modula 3 je oblika sporočila, statusna koda iz Modula 5 pove, kako se je izteklo, baza iz Modula 4 pa je tisto, kamor se podatek dejansko zapiše. Prav tako deluje tudi MCP iz Modula 10 — agent pošlje zahtevo orodju in dobi nazaj JSON.

VEČ O TEM

- MDN — uporaba fetch — https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

Pot enega klika `the journey of one click`

Klik na gumb sproži verigo šestih korakov, ki se zgodi v nekaj stotinkah sekunde.

Vsak korak lahko odpove — in statusna koda ti pove, kateri.

Ko to pot enkrat vidiš, znaš vprašanje *zakaj ne dela* postaviti veliko bolj natančno.

`1. Brskalnik`	uporabnik klikne gumb, sproži se JavaScript
`2. Zahteva`	`POST /api/gostje` z JSON vsebino
`3. Strežnik`	preveri dovoljenja in pravila (`6.5`)
`4. Baza`	`INSERT INTO gostje ...` — podatek se zapiše
`5. Odgovor`	`200 OK` in JSON z novim zapisom
`6. Brskalnik`	prejme odgovor in posodobi stran

> Diagnostika po korakih: če se v razvijalskih orodjih zahteva sploh ne pojavi, je težava v koraku 1. Če dobiš 401 ali 403, je v koraku 3. Če dobiš 500, je v koraku 3 ali 4 na strežniku. Če dobiš 200, a se na strani nič ne spremeni, je v koraku 6. Prav ta razčlenitev je tisto, kar boš povedal agentu, in razlika med *ne dela* in *zahteva vrne 500* je razlika med uro in minuto iskanja.
>
> **VEČ O TEM**
>
> - MDN — statusne kode HTTP — https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Statično in dinamično `static vs. dynamic`

Statična stran je datoteka, ki se vsem prikaže enako — strežnik jo samo pošlje naprej. Dinamična stran se sestavi za vsakega uporabnika posebej, ker mora pogledati v bazo. To gradivo je statično, zato deluje tudi brez interneta in brez strežnika.

razlika.py

```
# STATICNO - streznik samo poslje datoteko
#   GET /index.html  ->  index.html

# DINAMICNO - streznik stran sestavi ob vsaki zahtevi
def stran_za(uporabnik):
    narocila = baza.execute(
        "SELECT * FROM narocila WHERE stranka_id = ?", (uporabnik.id,)
    ).fetchall()
    return sestavi_html(narocila)
```

Staticno je hitrejse in varnejse. Dinamicno je potrebno, ko je vsebina osebna.

> Pogosta in zelo praktična kombinacija je vmesna pot: stran je statična, podatke pa si pridobi z API klici iz pojma 6.6. Tako dobiš hitrost statične strani in svežino dinamične. Prav to počne večina sodobnih aplikacij — in prav to počne tudi gradivo, ki ga bereš, le da namesto strežnika bere svoje podatkovne datoteke.

VEČ O TEM

- Wikipedia — Static web page — https://en.wikipedia.org/wiki/Static_web_page

MODUL 7

Kje koda živi

Lestvica gostovanja od tvojega računalnika do oblaka — s ceno, dovoljenji in pastmi vsake stopnje.

Prva stopnja: moj računalnik `localhost`

Najnižja stopnja gostovanja je tvoj računalnik: program teče, ko ga poženeš, in umre, ko okno zapreš.

Nič ne stane, nikogar ni treba prositi za dovoljenje in nihče drug ne vidi ničesar.

Za učenje, preizkušanje in osebna orodja je to povsem dovolj — in pogosto najboljša izbira.

zazeni-lokalno.bat

```
D:\projekt> python tools\server.py



  Na tem racunalniku:   http://localhost:8080

  V lokalnem omrezju:   http://192.168.1.24:8080



  Ustavi z:  Ctrl + C
```

Dokler to okno tece, stran zivi. Ko ga zapres, je ni vec.

> Prav ta lastnost — da nič ni trajno — je za začetek prednost, ne pomanjkljivost. Ničesar ne moreš pokvariti, nikomur ne moreš razkriti podatkov in vse lahko kadarkoli izbrišeš. Preden karkoli postaviš višje po tej lestvici, naj na tej stopnji deluje brezhibno.

VEČ O TEM

- Python — http.server — https://docs.python.org/3/library/http.server.html

Druga stopnja: pisarna `LAN hosting`

Če strežnik posluša na vseh omrežnih karticah, ga vidijo vsi na isti wifi mreži.

Še vedno ne stane nič in še vedno ni treba ničesar nameščati na tuje naprave.

Za predstavitev sodelavcem ali za orodje, ki ga uporablja ena ekipa, je to pogosto dovolj.

`Cena`	nič
`Dovoljenja`	nobenih — razen če IT blokira porte
`Kdo vidi`	vsi na isti mreži
`Ko ugasneš računalnik`	nedosegljivo
`Tipična raba`	orodje za ekipo, demo na sestanku

> Omejitev, ki v podjetjih največkrat zagode, ni tehnična, ampak organizacijska: gostujoče wifi omrežje je pogosto ločeno od službenega, požarni zid pa lahko blokira vse razen brskanja. Če naslov deluje tebi in kolegu ne, najprej preveri, ali sta res na isti mreži.
>
> **VEČ O TEM**
>
> - Wikipedia — Private network — https://en.wikipedia.org/wiki/Private_network

Tretja stopnja: tunel `tunnel (trycloudflare, ngrok)`

Tunel je program, ki iz tvojega računalnika vzpostavi povezavo navzven in ti podeli javni naslov.

Ker povezavo vzpostaviš ti, požarnemu zidu ni treba ničesar odpirati — in prav zato deluje tudi v službi.

Naslov je začasen: velja, dokler okno teče, in ob naslednjem zagonu je drugačen.

zazeni-javno.bat

```
D:\projekt> python tools\tunnel.py


  Odpiram javni tunel ...


  +-------------------------------------------+
  |  https://nekaj-nakljucnih-besed.trycloudflare.com  |
  +-------------------------------------------+


  Naslov deluje, dokler tece to okno.
```

Tako je nastal naslov, prek katerega morda prav zdaj berete to gradivo.

> Varnostna opomba, ki jo je treba izreči na glas: ta naslov je javen. Kdorkoli ga dobi, vidi vse, kar strežnik streže — brez gesla in brez prijave. Zato v mapo, ki jo streže tunel, nikoli ne postavi osebnih podatkov, ključev ali internih dokumentov. Za resno rabo obstajajo tuneli z računom, ki znajo zahtevati prijavo.
>
> **VEČ O TEM**
>
> - Cloudflare — TryCloudflare (začasni tuneli) — https://developers.cloudflare.com/cloudflare-one/connections/connect-networks/do-more-with-tunnels/trycloudflare/

Četrta stopnja: najet strežnik VPS

VPS je računalnik v tujem podatkovnem centru, ki ga najameš za nekaj evrov na mesec.

Teče neprekinjeno, ima stalen javni naslov in nanj lahko namestiš karkoli.

Cena te svobode je, da si zanj odgovoren sam: posodobitve, varnost, varnostne kopije.

`Cena`	približno 5–20 € na mesec
`Dovoljenja`	potrebuješ kartico in nekoga, ki to odobri
`Kdo vidi`	ves internet
`Ko ugasneš računalnik`	deluje naprej
`Tipična raba`	prava aplikacija z bazo in uporabniki

> Najpogostejša napaka na tej stopnji je, da se strežnik postavi in nato pozabi. Nevzdrževan strežnik z javnim naslovom postane v nekaj mesecih tarča; zato velja pravilo, da VPS najameš šele takrat, ko veš, kdo ga bo posodabljal. Za mnoge projekte je naslednja stopnja boljša izbira.

VEČ O TEM

- Wikipedia — Virtual private server — https://en.wikipedia.org/wiki/Virtual_private_server

Peta stopnja: statično gostovanje `static hosting`

Če je stran statična, je ni treba nikjer poganjati — dovolj je, da nekdo streže datoteke.

Ponudniki kot GitHub Pages ali Netlify to počnejo brezplačno, s stalnim naslovom in potrdilom HTTPS.

Za gradivo, predstavitev, dokumentacijo ali osebno stran je to skoraj vedno prava izbira.

`Cena`	brezplačno za manjše strani
`Dovoljenja`	samo račun pri ponudniku
`Kdo vidi`	ves internet, s stalnim naslovom
`Ko ugasneš računalnik`	deluje naprej
`Omejitev`	ni baze in ni strežniške kode

> To gradivo bi lahko prav tako gostoval na tak način: ker je statično, bi ga bilo treba samo naložiti in bi imelo stalen naslov brez tvojega računalnika. Odločili smo se za tunel, ker je za prototip enostavnejši in ne zahteva računa — a če boš gradivo delil pogosteje, je statično gostovanje naslednji smiselni korak.
>
> **VEČ O TEM**
>
> - GitHub Pages — https://pages.github.com/
> - Netlify — dokumentacija — https://docs.netlify.com/

Šesta stopnja: oblak `cloud (AWS, Azure)`

Oblak ni en računalnik, ampak stotine storitev, ki jih sestaviš po potrebi in plačaš po porabi.

Prednost je, da se sam prilagodi obremenitvi; slabost, da je zapleten in da računa ni lahko napovedati.

Za orodje, ki ga uporablja ekipa, je skoraj vedno pretiran.

`Cena`	po porabi — od centov do presenečenj
`Dovoljenja`	poslovni račun, pogosto tudi nabava
`Kdo vidi`	kar nastaviš
`Zahtevnost`	visoka — svoj poklic
`Tipična raba`	veliko uporabnikov, veliko podatkov, zahteve po razpoložljivosti

> Pri delu z AI agentom je vredno vedeti, da bo ta pogosto predlagal oblačno rešitev, ker jo je videl največkrat. Če mu v `arhitektura.md` (Modul 9) zapišeš, da gre za orodje za deset ljudi brez zunanjega dostopa, bo predlagal nekaj precej preprostejšega — in to boš dejansko znal vzdrževati.
>
> **VEČ O TEM**
>
> - Wikipedia — Cloud computing — https://en.wikipedia.org/wiki/Cloud_computing

Spletna tehnologija brez spleta `local web app, offline`

HTML, CSS in JavaScript ne potrebujejo interneta — potrebujejo samo brskalnik.

Mapa z datotekami je lahko popolnoma delujoča aplikacija, ki jo odpreš z dvoklikom.

Na službenem računalniku, kjer ne smeš nameščati programov, je to pogosto edina pot do lastnega orodja.

Gradivo, ki ga berete, je natanko tak primer: deluje tudi brez strežnika in brez omrežja.

Dve poti do iste strani

```
1) S streznikom:

   D:\projekt> python tools\server.py

   http://localhost:8080



2) Brez streznika:

   dvoklik na  D:\projekt\index.html

   file:///D:/projekt/index.html



Enaka stran. Brez namestitve, brez interneta.
```

Zato v tem gradivu ni niti ene povezave na zunanjo knjiznico.

> Meje te poti so tri: brez strežnika ni skupne baze (podatki ostanejo v brskalniku enega uporabnika), brskalnik iz varnostnih razlogov ne sme brati poljubnih datotek z diska, in nekatere zmožnosti so dovoljene samo prek `https`. Za osebni kalkulator, pregledovalnik, kontrolni seznam ali prav takšno gradivo pa je to povsem dovolj.
>
> **VEČ O TEM**
>
> - MDN — progresivne spletne aplikacije — https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps

Kaj smeš dati na javni naslov `what to expose`

Ko nekaj dobi javni naslov, predpostavi, da bo to nekdo našel — tudi če naslova nisi nikomur dal.

Iskalniki, samodejni pregledovalniki in radovedneži najdejo naslove hitreje, kot si misliš.

Pravilo je preprosto: na javni naslov sodi samo tisto, kar bi mirno pustil na oglasni deski.

`Gradivo, predstavitve`	brez težav
`Osebni podatki strank`	nikoli brez prijave
`Interni dokumenti`	nikoli
`Datoteke `.env`, ključi, gesla`	nikoli — tudi ne v mapi projekta
`Baza podatkov`	nikoli neposredno; samo prek strežnika s preverjanjem
`Testni podatki z realnimi imeni`	raje ne — uporabi izmišljene

> Pri delu z AI agenti se doda še ena past: agent, ki gradi aplikacijo, rad na hitro naredi stran za pregled podatkov brez prijave, ker o njej nihče ni rekel drugače. Zato v `arhitektura.md` (Modul 9) izrecno zapiši, kdo sme videti kaj — sicer bo privzeta nastavitev *vsi*. In preden karkoli pošlješ v tunel, poglej, kaj je v mapi.
>
> **VEČ O TEM**
>
> - MDN — varnost na spletu — https://developer.mozilla.org/en-US/docs/Web/Security
> - Wikipedia — požarni zid — https://en.wikipedia.org/wiki/Firewall_(computing)

MODUL 8

Jeziki in orodja

Zakaj obstaja toliko programskih jezikov, za kaj se vsak uporablja in kaj so knjižnice.

Zakaj obstaja toliko jezikov `programming languages`

Vsi jeziki znajo isto — razlikujejo se po tem, kaj olajšajo in kaj otežijo.

Eni so bližje stroju in zato hitri, drugi bližje človeku in zato hitro napisani.

Izbira jezika je skoraj vedno odločitev o okolju, v katerem bo program tekel, ne o okusu.

isti_program.txt

```
# Python
print("Pozdravljeni")

// JavaScript
console.log("Pozdravljeni");

// C
printf("Pozdravljeni\n");

-- SQL
SELECT 'Pozdravljeni';
```

Stirje jeziki, ena naloga. Razlike so v podrobnostih, ne v zamisli.

> Za netehničnega uporabnika je najbolj uporabna posledica ta: ko ti agent predlaga jezik, vprašaj *zakaj tega*. Dober odgovor se glasi *ker mora to teči v brskalniku* ali *ker je knjižnica za branje PDF najboljša v Pythonu*. Slab odgovor je *ker je priljubljen*.

VEČ O TEM

- Python — pogosta vprašanja — https://docs.python.org/3/faq/general.html

Prevajani in interpretirani `compiled vs. interpreted`

Prevajani jezik se pred zagonom v celoti prevede v strojno kodo — nastane samostojen program `.exe`.

Interpretirani jezik se bere sproti, vrstico za vrstico, zato za zagon potrebuješ nameščeno okolje.

Prevajani so hitrejši pri izvajanju, interpretirani pa pri pisanju in popravljanju.

Dva načina zagona

```
PREVAJANO (C):

  gcc program.c -o program.exe    <- prevedi (enkrat)

  program.exe                     <- pozeni (velikokrat)



INTERPRETIRANO (Python):

  python program.py               <- prevedi IN pozeni, vsakic sproti
```

Zato za Python potrebujes nameščen Python, za .exe pa nic.

> Ta razlika pojasni tudi, zakaj so zagonske datoteke tega gradiva `.bat` in ne `.exe`: `.bat` je navaden tekst, ki ga Windows prebere sproti, zato ga lahko odpreš in preveriš, kaj počne. Pri nepodpisani `.exe` datoteki tega ne moreš — in prav zato jih službeni računalniki pogosto blokirajo.

VEČ O TEM

- Wikipedia — Interpreter — https://en.wikipedia.org/wiki/Interpreter_(computing)

C in C++ `C / C++`

Najbližja strojni ravni; uporabljata se tam, kjer šteje vsaka tisočinka sekunde.

Operacijski sistemi, gonilniki, igre, krmilniki naprav — in temelji skoraj vseh drugih jezikov.

primer.c

```c
#include <stdio.h>

int main(void) {
    int znesek = 100;
    printf("Z DDV: %.2f\n", znesek * 1.22);
    return 0;
}
```

Tip spremenljivke (int) je treba napisati. Prevajalnik nic ne ugiba.

> Za neprogramerja je pomembno le to: če ti agent predlaga C ali C++, se vprašaj, ali res rešuješ problem hitrosti. V pisarniškem svetu je odgovor skoraj vedno ne — in Python bo dal isti rezultat v desetkrat manj vrsticah.

VEČ O TEM

- isocpp.org — o C++ — https://isocpp.org/

Python `Python`

Berljiv, zelo razširjen in odličen za obdelavo podatkov, avtomatizacijo in umetno inteligenco.

Ima ogromno knjižnic za branje datotek, tabel, PDF-jev in spletnih storitev.

Za netehničnega uporabnika je skoraj vedno prava prva izbira.

primer.py

```python
import csv

with open("racuni.csv", encoding="utf-8") as f:
    skupaj = sum(int(v["znesek"]) for v in csv.DictReader(f))

print(f"Skupaj z DDV: {skupaj * 1.22:.2f}")
```

IZPIS

```
Skupaj z DDV: 579.50
```

Pet vrstic naredi to, kar bi v C zahtevalo petdeset.

> Python je tudi jezik, v katerem AI agenti pišejo najbolj zanesljivo, preprosto zato, ker ga je v učnih podatkih največ. Če nimaš posebnega razloga za kaj drugega, je Python tudi najboljša izbira za sodelovanje z agentom — in na tvojem računalniku je že nameščen.

VEČ O TEM

- Python — uradni vodič — https://docs.python.org/3/tutorial/introduction.html

JavaScript in TypeScript `JavaScript / TypeScript`

JavaScript je edini jezik, ki teče neposredno v brskalniku — zato je neizogiben za spletne strani.

TypeScript je JavaScript z dodanimi tipi; napake se pokažejo že med pisanjem, ne šele pri uporabniku.

Z okoljem Node.js lahko isti jezik teče tudi zunaj brskalnika, na strežniku.

primer.ts

```ts
function zDdv(cena: number, stopnja: number = 22): number {
    return cena * (1 + stopnja / 100);
}

console.log(zDdv(100));
console.log(zDdv("sto"));   // <- TypeScript javi napako ze tukaj
```

Zapis ': number' je vse, kar loci TypeScript od JavaScripta.

> Praktično pravilo: če izdelek živi v brskalniku, bo v njem JavaScript ali TypeScript, ne glede na to, kaj si želiš. Če pa gre za obdelavo podatkov, je Python običajno krajša pot. Mnogi projekti uporabljajo oboje — in to ni znak zmede, ampak normalno stanje.

VEČ O TEM

- MDN — JavaScript — https://developer.mozilla.org/en-US/docs/Web/JavaScript
- TypeScript — dokumentacija — https://www.typescriptlang.org/docs/

Java in C# `Java / C#`

Jezika velikih poslovnih sistemov: bančništvo, zavarovalništvo, ERP, javna uprava.

Strožja sta in bolj zgovorna, kar se obrestuje, ko na isti kodi dela petdeset ljudi deset let.

Primer.java

```java
public class Primer {
    public static void main(String[] args) {
        double znesek = 100;
        System.out.println("Z DDV: " + znesek * 1.22);
    }
}
```

Sest vrstic za isti izpis. Struktura je cena za velike ekipe.

> Če v tvojem podjetju obstaja interni sistem, je verjetnost velika, da je napisan v enem od teh dveh jezikov. Za lastno orodje ju skoraj zagotovo ne potrebuješ — sta pa razlog, zakaj bo integracija z internim sistemom vedno zahtevala razvijalca.
>
> **VEČ O TEM**
>
> - dev.java — uradna stran — https://dev.java/
> - Microsoft — C# — https://learn.microsoft.com/en-us/dotnet/csharp/

SQL SQL

SQL ni jezik za pisanje programov, ampak za spraševanje baz — in prav zato ga srečaš povsod.

Naučiti se ga prebrati je najhitreje povrnjena investicija za netehničnega uporabnika.

poizvedba.sql

```sql
SELECT mesec, SUM(znesek) AS skupaj
FROM racuni
WHERE leto = 2026
GROUP BY mesec
ORDER BY mesec;
```

Podrobneje je SQL obdelan v Modulu 4.

> Posebnost SQL je, da opisuješ *rezultat*, ne postopka. Zato je tako kratek — in zato je tudi tako nevaren, kadar pozabiš pogoj: `DELETE FROM racuni` brez `WHERE` izbriše vse.
>
> **VEČ O TEM**
>
> - SQLite — jezik SQL — https://www.sqlite.org/lang.html

Bash in PowerShell `Bash / PowerShell`

To sta jezika ukazne vrstice: namenjena sta vodenju drugih programov, ne pisanju aplikacij.

Z njima avtomatiziraš opravila, ki bi jih sicer klikal — kopiranje, preimenovanje, zaganjanje.

PowerShell

```powershell
# preimenuj vse PDF-je v mapi po datumu spremembe

Get-ChildItem *.pdf | ForEach-Object {

    Rename-Item $_ ("racun_" + $_.LastWriteTime.ToString("yyyy-MM-dd") + ".pdf")

}
```

Windows uporablja PowerShell, Mac in Linux pa Bash. Zamisel je ista.

> Datoteke `.bat`, ki poganjajo to gradivo, so najpreprostejša oblika istega. Za neprogramerja je to pogosto najhitrejša zmaga z AI: prosiš agenta za skripto, ki uredi mapo z dvesto datotekami, in jo dobiš v desetih sekundah. Le prej si naredi kopijo mape.
>
> **VEČ O TEM**
>
> - Microsoft — PowerShell — https://learn.microsoft.com/en-us/powershell/scripting/overview
> - GNU — priročnik za Bash — https://www.gnu.org/software/bash/manual/bash.html

Izvajalno okolje in Node.js `runtime / Node.js`

Jezik sam po sebi ne teče — potrebuje program, ki ga izvaja; temu pravimo izvajalno okolje `runtime`.

Za Python je to `python.exe`, za JavaScript v brskalniku je to brskalnik sam.

Node.js je isto okolje za JavaScript, le da teče zunaj brskalnika — zato lahko bere datoteke in streže strani.

Ko kdo reče *rabiš Node*, misli prav to: brez okolja jezika ni mogoče pognati.

Kaj je nameščeno

```
D:\projekt> python --version

Python 3.13.5



D:\projekt> node --version

'node' is not recognized as an internal or external command



-> Python je nameščen, Node.js ni.
```

Prav zato to gradivo ne potrebuje Node.js - vse tece v brskalniku in v Pythonu.

> To je tudi najpogostejši razlog, zakaj navodila z interneta *ne delujejo*: zahtevajo okolje, ki ga na tvojem računalniku ni. Na službenem računalniku namestitev pogosto ni dovoljena, zato je vredno agentu že na začetku povedati, kaj imaš na voljo — sicer bo predlagal rešitev, ki je ne boš mogel pognati.
>
> **VEČ O TEM**
>
> - Node.js — o projektu — https://nodejs.org/en/about

Knjižnice `libraries (pip, npm)`

Knjižnica je koda, ki jo je nekdo drug že napisal in jo lahko uporabiš namesto svoje.

Za Python jih namestiš z ukazom `pip install`, za JavaScript z `npm install`.

Branje PDF-jev, delo z Excelom, risanje grafov, pošiljanje pošte — za vse to knjižnice že obstajajo.

Namestitev knjižnice

```
D:\projekt> pip install pypdf

Collecting pypdf

  Downloading pypdf-6.0.0-py3-none-any.whl

Successfully installed pypdf-6.0.0



D:\projekt> python

>>> from pypdf import PdfReader        <- zdaj je na voljo
```

Namesto tisoc vrstic za branje PDF napises eno vrstico uvoza.

> Hkrati je to tudi vstopna točka za težave: vsaka knjižnica je tuja koda, ki teče na tvojem računalniku. Preden namestiš nekaj, česar ne poznaš, preveri, ali je ime pravilno zapisano — obstajajo knjižnice z namerno podobnimi imeni. Pri službenem računalniku je `pip install` pogosto tudi tisto, kar IT blokira.
>
> **VEČ O TEM**
>
> - PyPI — zbirka knjižnic za Python — https://pypi.org/
> - npm — o registru — https://docs.npmjs.com/about-npm

Knjižnica ali lastna koda `build or borrow`

Uporabi knjižnico, kadar je problem splošen in dobro rešen: datumi, PDF, šifriranje, omrežje.

Napiši sam, kadar je problem tvoj, majhen in specifičen — pravila tvojega podjetja niso v nobeni knjižnici.

Nikoli ne piši sam šifriranja, obdelave datumov in preverjanja gesel; tam je *sam* skoraj vedno napačno.

`Branje PDF, Excel, slik`	knjižnica — zanesljivo
`Datumi, časovni pasovi`	knjižnica — vedno
`Šifriranje, gesla`	knjižnica — nikoli sam
`Tvoja poslovna pravila`	lastna koda — nihče je ne pozna
`Nekaj deset vrstic logike`	lastna koda — manj odvisnosti
`Knjižnica za trivialno stvar`	raje ne — dodana odvisnost ni zastonj

> Vsaka knjižnica je dolg: nekdo jo mora vzdrževati, posodabljati in nekega dne bo nehala delovati z novo različico jezika. Zato velja: velika knjižnica za velik problem je prihranek, majhna knjižnica za majhen problem pa je pogosto breme. Pri agentu to povej izrecno, sicer bo dodal odvisnost za vsako malenkost.
>
> **VEČ O TEM**
>
> - Python — nameščanje paketov — https://packaging.python.org/en/latest/tutorials/installing-packages/

Odvisnosti in različice `dependencies, versions`

Program, ki uporablja pet knjižnic, je odvisen od petih tujih projektov in njihovih sprememb.

Zato se različice zapišejo v datoteko — da bo program čez leto dni tekel enako kot danes.

Oznaka `6.0.0` pomeni večja.manjša.popravek; prva številka se spremeni, ko nekaj ni več združljivo.

requirements.txt

```
pypdf==6.0.0
pytesseract==0.3.13
pillow==11.0.0
```

Ukaz 'pip install -r requirements.txt' namesti natanko te razlicice.

> Ločena okolja (`python -m venv .venv`) poskrbijo, da imata dva projekta na istem računalniku lahko različni različici iste knjižnice. Za netehničnega uporabnika je pomembno predvsem to, da ob težavi *včeraj je delalo* najprej pomisli na spremembo različice — in da datoteke `requirements.txt` ne briše.
>
> **VEČ O TEM**
>
> - Semantično označevanje različic — https://semver.org/
> - Python — virtualna okolja (venv) — https://docs.python.org/3/library/venv.html

MODUL 9

Kako se dela dobra koda

Razgradnja problema, moduli, testi in datoteka, iz katere gradi AI agent.

Razgradnja problema `decomposition`

Velikega problema ni mogoče rešiti; rešiti je mogoče le zaporedje majhnih.

Razgradnja pomeni, da nalogo razbiješ na korake, ki jih znaš opisati vsakega posebej.

Če koraka ne znaš opisati v enem stavku, še ni dovolj majhen.

To je veščina, ki jo pri delu z AI potrebuješ bolj kot znanje kateregakoli jezika.

razgradnja.txt

```
NALOGA: "uredi mi racune"        <- preveliko, agent bo ugibal

RAZGRAJENO:
  1. preberi vse .pdf iz mape "prejeto"
  2. iz vsakega izlusci datum, stevilko in znesek
  3. ce izluscanje ne uspe, datoteko daj v mapo "rocno"
  4. zapisi vrstice v racuni.csv
  5. preimenuj datoteko v LLLL-MM-DD_stevilka.pdf
  6. izpisi, koliko jih je uspelo in koliko ne
```

Sest stavkov. Vsak se da preveriti posebej - in vsak je lahko svoja funkcija.

> Opazi tretjo točko: opisuje, kaj naj se zgodi, ko gre nekaj narobe. Prav ta točka loči navodilo, po katerem nastane uporabno orodje, od navodila, po katerem nastane program, ki se sesuje ob prvi nenavadni datoteki. Pri vsakem koraku se vprašaj: *kaj pa, če ne gre?*

VEČ O TEM

- Wikipedia — ločevanje odgovornosti — https://en.wikipedia.org/wiki/Separation_of_concerns

Main in moduli `main + modules`

Koda se razdeli na module: vsaka datoteka opravlja eno vrsto dela.

Glavna datoteka `main` samo vodi vrstni red — sama ne počne ničesar zahtevnega.

Tako lahko en del popraviš ali zamenjaš, ne da bi se dotaknil ostalih.

main.py

```python
from branje import preberi_pdf
from izlusci import izlusci_podatke
from zapisi import zapisi_csv

def main():
    for datoteka in preberi_pdf("prejeto"):
        podatki = izlusci_podatke(datoteka)
        zapisi_csv(podatki, "racuni.csv")

if __name__ == "__main__":
    main()
```

Devet vrstic pove, kaj program pocne. Podrobnosti so v treh drugih datotekah.

> Ta razdelitev je pri delu z AI agentom praktično nujna iz povsem drugega razloga: agent lahko popravi eno datoteko, ne da bi prebral in prepisal celoten program. Manj kode, ki jo mora obdelati naenkrat, pomeni manj tokenov, manj napak in manjšo verjetnost, da bo med popravljanjem pokvaril nekaj drugega.

VEČ O TEM

- Wikipedia — ločevanje odgovornosti — https://en.wikipedia.org/wiki/Separation_of_concerns

Poimenovanje `naming`

Ime naj pove, kaj stvar je ali počne — ne, kako je narejena.

Dobro ime prihrani komentar; slabo ime zahteva razlago vsakič znova.

imena.py

```python
# slabo
def obd(d, x):
    return [i for i in d if i[1] > x]

# dobro
def racuni_nad_zneskom(racuni, prag):
    return [r for r in racuni if r.znesek > prag]
```

Druga razlicica ne potrebuje komentarja. Ime je komentar.

> Pri delu z agentom je poimenovanje presenetljivo močno orodje: agent iz imen sklepa o namenu. Če funkcijo poimenuješ `racuni_nad_zneskom`, bo napisal kodo, ki filtrira račune. Če jo poimenuješ `obd`, bo ugibal — in ugibal bo na podlagi tega, kar je videl v drugih projektih, ne v tvojem.
>
> **VEČ O TEM**
>
> - PEP 8 — slogovna navodila za Python — https://peps.python.org/pep-0008/

Kaj je test `test`

Test je majhen program, ki požene tvojo kodo in preveri, ali je rezultat pričakovan.

Napišeš ga enkrat, poganja pa se samodejno ob vsaki spremembi.

Njegova vrednost ni v tem, da dokaže pravilnost danes, ampak da opozori, ko se jutri nekaj pokvari.

test_ddv.py

```python
from ddv import z_ddv

def test_osnovni_primer():
    assert z_ddv(100) == 122.0

def test_nizja_stopnja():
    assert z_ddv(100, 9.5) == 109.5

def test_nic():
    assert z_ddv(0) == 0
```

IZPIS

```
3 passed in 0.01s
```

assert pomeni 'trdim, da'. Ce trditev ne drzi, test pade.

> Najkoristnejši testi niso tisti za običajne primere, ampak za robove: prazen seznam, negativno število, manjkajoče polje, zelo velika vrednost. Prav tam se skriva bug iz pojma 2.2 — in prav tam AI agent najpogosteje spregleda primer.
>
> **VEČ O TEM**
>
> - pytest — dokumentacija — https://docs.pytest.org/en/stable/
> - Python — unittest — https://docs.python.org/3/library/unittest.html

Vrste testov `unit / integration / end-to-end`

Testi enot preverjajo posamezno funkcijo; hitri so in jih je lahko veliko.

Integracijski testi preverjajo, ali deli delujejo skupaj — na primer program in baza.

Testi od konca do konca preverijo celotno pot uporabnika; so najpočasnejši in najbolj krhki.

`Test enote`	ena funkcija · milisekunde · veliko jih je
`Integracijski test`	več delov skupaj · sekunde · nekaj deset
`Test od konca do konca`	cela aplikacija · minute · le za ključne poti
`Ročni test`	človek pred zaslonom · vedno potreben za videz

> Klasično priporočilo je piramida: veliko hitrih testov enot spodaj, malo počasnih testov na vrhu. Razlog je preprost — če pade test enote, veš točno, katera funkcija je kriva; če pade test od konca do konca, veš le, da nekje na poti nekaj ne deluje.
>
> **VEČ O TEM**
>
> - Martin Fowler — testna piramida — https://martinfowler.com/bliki/TestPyramid.html
> - Wikipedia — avtomatizirano testiranje — https://en.wikipedia.org/wiki/Test_automation

Zakaj so testi pri AI še pomembnejši `tests with AI`

Agent piše kodo hitreje, kot jo ti utegneš prebrati — testi so edini način, da mu sledi preverjanje.

Ker je agent nedeterminističen (pojem 2.6), ne moreš predpostaviti, da je rešitev pravilna, ker je videti prepričljivo.

Test je zahteva, zapisana tako, da se preveri sama — in prav zato je boljše navodilo kot prošnja v besedah.

Najboljši vrstni red je: najprej testi, nato koda, nato preverjanje.

Vrstni red dela z agentom

```
1. Ti:      opisi zahtevo
2. Agent A: napise TESTE (in nic drugega)
3. Ti:      preberes teste - so to res tvoje zahteve?
4. Agent B: napise KODO, testov ne vidi
5. Racunalnik: pozene teste


   3 passed, 1 failed  ->  agent B popravlja, dokler ni zeleno
```

Agent B ne vidi testov, zato jim ne more pisati kode na kozo.

> Zakaj ločena agenta: če isti agent napiše teste in kodo, bo teste nezavedno prilagodil kodi, ki jo je pravkar napisal — in oboje bo ustrezalo isti napačni predpostavki. Ločitev je poceni in učinkovita: agent, ki testov ne vidi, mora napisati kodo, ki dejansko dela, ne pa kodo, ki izgleda kot rešitev. Podrobno v pojmu 11.7.
>
> **VEČ O TEM**
>
> - Claude Code — pregled — https://docs.claude.com/en/docs/claude-code/overview

Git in commit `Git, commit`

Git si zapomni vsako stanje kode, ki ga potrdiš — in omogoči vrnitev na katerokoli od njih.

Commit je posnetek s sporočilom: *kaj sem spremenil in zakaj*.

Pri delu z agentom je to varnostna mreža: če ti pokvari kodo, se v eni vrstici vrneš na zadnje delujoče stanje.

Osnovni git

```
D:\projekt> git init

D:\projekt> git add .

D:\projekt> git commit -m "Delujoce branje PDF racunov"



D:\projekt> git log --oneline

a3f9c21 Delujoce branje PDF racunov



D:\projekt> git restore .        <- razveljavi vse od zadnjega commita
```

Commit naredi PREDEN pustis agenta popravljati. Vedno.

`git init`	začni slediti spremembam v tej mapi
`git add .`	pripravi vse spremembe za commit
`git commit -m "..."`	shrani posnetek s sporočilom
`git log --oneline`	pokaži zgodovino
`git diff`	kaj se je spremenilo od zadnjega commita
`git restore .`	zavrzi vse neshranjene spremembe

> Za netehničnega uporabnika je najpomembnejše eno samo pravilo: **commit pred vsako sejo z agentom**. Ko agent naredi nekaj čudnega — in bo — je razlika med `git restore .` in popoldnevom reševanja. Git ni potreben za majhne projekte zaradi sodelovanja, ampak zaradi te ene same možnosti vrnitve.
>
> **VEČ O TEM**
>
> - Git — dokumentacija — https://git-scm.com/doc

Datoteka arhitektura.md `architecture.md`

To je dokument, v katerem opišeš, kaj gradiš, preden agent napiše prvo vrstico kode.

Vsebuje cilj, vhode, izhode, omejitve, strukturo datotek in odločitve, ki so že sprejete.

Agent ga prebere na začetku vsake seje, zato se ne vrača k vprašanjem, na katera si že odgovoril.

Je najcenejši način, da preprečiš, da bi agent gradil nekaj, česar nisi naročil.

arhitektura.md

```
# Orodje za obdelavo prejetih racunov

## Cilj
Iz PDF racunov v mapi "prejeto" zgraditi CSV s podatki in
preimenovati datoteke po datumu in stevilki.

## Vhod
- PDF datoteke, ~200 na mesec, nekateri skenirani

## Izhod
- racuni.csv (datum, stevilka, znesek, dobavitelj, datoteka)
- neuspeli primeri v mapo "rocno"

## Omejitve
- Windows, Python 3.13, brez Node.js
- brez namescanja programov z administratorskimi pravicami
- podatki ne smejo zapustiti racunalnika

## Odlocitve
- knjiznica pypdf, OCR samo ce extract_text() vrne prazno
- brez baze, CSV zadosca
```

Ta datoteka je vredna vec kot ura pogovora z agentom.

> Razdelek *Omejitve* je tisti, ki ga ljudje najpogosteje izpustijo in ki največ prinese. Brez njega bo agent predlagal rešitev, ki zahteva Node.js, oblak ali administratorske pravice — vse troje na službenem računalniku pogosto ni mogoče. Ko povzetek pogovora preraščа v odločitve, jih zapiši sem; to je tudi tisto, kar ohrani smisel, ko ti zmanjka context window (Modul 10).

VEČ O TEM

- Claude Code — datoteka s spominom projekta — https://docs.claude.com/en/docs/claude-code/memory

MODUL 10

AI, agenti in Claude

Tokeni, context window, agenti, skills, MCP — in kaj od tega dejansko vpliva na tvoje delo.

Kaj je jezikovni model LLM

Jezikovni model napoveduje, kateri del besedila najverjetneje sledi temu, kar je že napisano.

Ne išče po bazi in ne razume v človeškem pomenu — izračuna najverjetnejše nadaljevanje.

Ker je napoved verjetnostna, je odgovor lahko odličen in prepričljivo napačen z enako lahkoto.

Kako nastane odgovor

```
Vhod:   "Glavno mesto Slovenije je"


Model izracuna verjetnosti naslednjega dela:

   " Ljubljana"   98.2 %

   " mesto"        0.7 %

   " znano"        0.4 %

   ...


Izbere najverjetnejsega, ga doda na konec in ponovi.
```

Odgovor nastaja del za delom, ne kot celota. Zato ga vidis, kako se izpisuje.

> Iz tega izhaja vse ostalo v tem modulu. Model nima spomina med pogovori, ne ve, kaj je res, in nima dostopa do tvojih datotek — razen če mu to izrecno daš. Vse, kar zna, je nadaljevati besedilo, ki ga vidi. Presenetljivo veliko koristnega dela se da narediti prav s to eno sposobnostjo, a le če razumeš njene meje.
>
> VEČ O TEM
>
> - Anthropic — pregled modelov — https://docs.claude.com/en/docs/about-claude/models/overview

Token `token`

Model ne bere črk in ne besed, ampak tokene — koščke besedila, dolge povprečno tri do štiri znake.

Pogoste angleške besede so pogosto en sam token, redkejše in slovenske besede pa se razbijejo na več.

Zato ista vsebina v slovenščini porabi občutno več tokenov kot v angleščini — in s tem več denarja.

Isti stavek, dva jezika

```
EN: "The invoice was paid yesterday."

    [The][ invoice][ was][ paid][ yesterday][.]        ~6 tokenov



SL: "Racun je bil vceraj placan."

    [Rac][un][ je][ bil][ vc][eraj][ pla][can][.]      ~9 tokenov



Razmerje se pri daljsih besedilih ustali okoli 1.5x do 2x.
```

Priblizek: 1 token ~ 4 znake. Natancno stevilo zna povedati samo model sam.

> Praktična posledica je dvojna. Prvič, če delaš z zelo velikimi besedili in te skrbi cena, je angleščina cenejša — tudi če je vsebina slovenska, so navodila lahko angleška. Drugič, ocene *koliko strani gre v pogovor* so pri slovenščini bolj pesimistične, kot kažejo angleški primeri iz dokumentacije.
>
> **VEČ O TEM**
>
> - Anthropic — štetje tokenov — https://docs.claude.com/en/docs/build-with-claude/token-counting

Kaj stanejo tokeni `pricing`

Plačaš dvoje: tokene, ki jih pošlješ (vhod), in tokene, ki jih model napiše (izhod).

Izhod je približno petkrat dražji od vhoda, zato dolgi odgovori stanejo bistveno več kot dolga vprašanja.

Pri naročnini tega ne plačuješ po tokenih, a iste številke določajo, kdaj dosežeš omejitev porabe.

`Claude Opus 5`	$5 vhod / $25 izhod na milijon tokenov · 1M kontekst
`Claude Sonnet 5`	$2 vhod / $10 izhod · 1M kontekst
`Claude Haiku 4.5`	$1 vhod / $5 izhod · 200K kontekst
`Milijon tokenov`	približno 700.000 angleških besed
`Tipično vprašanje`	nekaj sto tokenov — torej stotinke centa
`Cela knjiga v pogovoru`	nekaj sto tisoč tokenov — evri, ne centi

> Cene se spreminjajo, razmerja pa ostajajo: močnejši model je dražji, izhod je dražji od vhoda, in ponavljajoče se pošiljanje istega dolgega besedila je največji tihi strošek. Za to obstaja predpomnjenje `prompt caching`, ki ponovno poslani del zaračuna bistveno ceneje. Številke v tabeli veljajo ob nastanku tega gradiva (september 2026) — preveri jih na uradni strani.
>
> **VEČ O TEM**
>
> - Anthropic — cenik — https://docs.claude.com/en/docs/about-claude/pricing
> - Anthropic — predpomnjenje vsebine — https://docs.claude.com/en/docs/build-with-claude/prompt-caching

Prompt in kontekst `prompt / context`

Prompt je to, kar napišeš; kontekst je vse, kar model ob tem vidi.

Kontekst vključuje celoten dosedanji pogovor, priložene datoteke in navodila, ki jih je dobil na začetku.

Model ne ve ničesar, česar ni v kontekstu — zato je *saj sem ti prej povedal* smiselno le, če je tisto prej še vedno zraven.

Kaj model dejansko prejme

```
[sistemska navodila]     kdo si, kako odgovarjaj

[arhitektura.md]        1.200 tokenov

[prejsnjih 14 sporocil] 8.400 tokenov

[priloga racun.pdf]     3.100 tokenov

[tvoje vprasanje]          40 tokenov

                        -----------------

                        12.740 tokenov gre v model ob VSAKEM sporocilu
```

Zato drugo vprasanje v dolgem pogovoru stane vec kot prvo.

> To pojasni tudi vedenje, ki se zdi nelogično: če v pogovoru na sredini popraviš navodilo, model še vedno vidi tudi prvotno. Kar je zapisano, ostane zapisano. Zato je pri daljšem delu boljše kot dolg popravljen pogovor: kratek povzetek odločitev, shranjen v `arhitektura.md`, in nov pogovor.

VEČ O TEM

- Anthropic — kontekstno okno — https://docs.claude.com/en/docs/build-with-claude/context-windows

Context window `context window`

Kontekstno okno je zgornja meja, koliko tokenov lahko model naenkrat vidi.

Pri Claude Opus 5 in Sonnet 5 je to milijon tokenov, pri Haiku 4.5 dvesto tisoč.

Ko se okno polni, se pogovor upočasni in podraži; ko je polno, je treba nekaj izpustiti.

Milijon tokenov zveni ogromno — a velik projekt s kodo in dokumenti ga porabi hitreje, kot pričakuješ.

Polnjenje okna

```
[##########.............................]  25 %   udobno
[####################...................]  50 %   se vedno v redu
[################################.......]  80 %   cas za povzetek
[########################################] 100 %   najstarejsi del pade ven
```

V Claude Code vidis odstotek porabljenega okna sproti.

> Zakaj se odgovori v zelo dolgih pogovorih poslabšajo: model mora upoštevati vedno več besedila, med katerim je veliko nepomembnega — starih poskusov, opuščenih idej, napak, ki so že popravljene. To ni okvara, ampak logična posledica. Rešitev ni boljši prompt, ampak čiščenje konteksta.

VEČ O TEM

- Anthropic — kontekstno okno — https://docs.claude.com/en/docs/build-with-claude/context-windows

Povzemi in začni znova `summarise and continue`

Ko se okno napolni okoli osemdeset odstotkov, agenta prosi za povzetek stanja.

Povzetek naj vsebuje: kaj je cilj, kaj je že narejeno, katere odločitve so sprejete in kaj je naslednji korak.

Nato odpri nov pogovor, prilepi povzetek in nadaljuj — z desetino porabe in boljšimi odgovori.

To je najkoristnejša navada pri delu z AI in tista, ki jo ljudje najdlje odlašajo.

povzetek-prompt.txt

```
Pripravi povzetek te seje za nadaljevanje v novem pogovoru.
Vkljuci:
  1. cilj projekta v dveh stavkih
  2. kaj je ze narejeno in deluje
  3. sprejete odlocitve in zakaj (npr. zakaj CSV in ne baza)
  4. kaj je naslednji korak
  5. znane pasti in kaj NE poskusati znova

Napisi ga tako, da bo nekdo brez konteksta lahko nadaljeval.
```

Povzetek shrani v arhitektura.md - tako preziviv tudi zaprtje pogovora.

> Peta točka je tista, ki jo ljudje izpuščajo in ki največ prinese: seznam stvari, ki so bile že poskušene in niso delovale. Brez nje bo novi pogovor samozavestno predlagal isto slepo ulico. V Claude Code obstaja za to tudi skill `handoff`, ki povzetek pripravi po preverjenem vzorcu (Modul 11).
>
> **VEČ O TEM**
>
> - Claude Code — datoteka s spominom projekta — https://docs.claude.com/en/docs/claude-code/memory

Kako porabiti manj tokenov `saving tokens`

Poraba ni odvisna od tega, koliko napišeš, ampak koliko model vsakič prebere.

Največji prihranki so v krajših pogovorih, manjših datotekah in natančnejših vprašanjih.

`Nov pogovor za novo temo`	največji prihranek od vseh
`Povzetek pri 80 % okna`	desetkratno zmanjšanje konteksta
`Manjše datoteke (moduli)`	agent prebere 200 vrstic namesto 3.000
`Natančno vprašanje`	krajši odgovor, manj popravljanja
`Pošlji CSV, ne slike tabele`	manj tokenov in boljši rezultat
`Ne prilagaj cele mape`	priloži le datoteke, ki so res potrebne
`Šibkejši model za preprosta opravila`	Haiku za urejanje, Opus za razmislek
`Ne ponavljaj konteksta`	model vidi celoten pogovor, ne rabi ponovitve

> Prva vrstica v tabeli je vredna vseh ostalih skupaj. Ljudje pustijo en pogovor teči cel teden in se čudijo porabi — pri tem pa vsako novo vprašanje potegne s seboj vse od ponedeljka. Pravilo: en pogovor, ena naloga. Ko je naloga končana, pogovor zapri.
>
> **VEČ O TEM**
>
> - Anthropic — predpomnjenje vsebine — https://docs.claude.com/en/docs/build-with-claude/prompt-caching

Halucinacije `hallucination`

Halucinacija je odgovor, ki zveni pravilno, a ni resničen — izmišljena številka, neobstoječa funkcija, napačna povezava.

Ni laž in ni napaka v običajnem pomenu: model je izračunal najverjetnejše nadaljevanje, ki se je izkazalo za neresnično.

Najpogostejša je tam, kjer model nima podatkov: pri natančnih številkah, imenih, datumih, povezavah in navedbah.

Zato pravilo: vse, kar je preverljivo, preveri — in raje zahtevaj vir kot trditev.

Kje najpogosteje zataji

VISOKO TVEGANJE	NIZKO TVEGANJE
natancne stevilke	razlaga pojma
imena in datumi	preoblikovanje besedila
povezave in citati	predlogi struktur
imena funkcij knjiznic	koda, ki jo lahko pozenes
pravni in davcni podatki	povzetek besedila, ki si ga dal

Desni stolpec: model dela s tem, kar mu das. Levi: model ugiba.

> Najboljša obramba ni boljši prompt, ampak preverljiv izhod. Koda se da pognati in testirati (Modul 9). Povzetek se da primerjati z izvirnikom. Številka iz zraka se ne da preveriti z ničimer. Zato velja pravilo iz pojma 2.6: naj AI napiše program, ki izračuna, ne pa da izračuna sam. Vse povezave v tem gradivu so bile na primer preverjene s programom, ne prepisane iz spomina.

VEČ O TEM

- Anthropic — pregled modelov — https://docs.claude.com/en/docs/about-claude/models/overview

Kaj ne sme v AI `privacy`

Vse, kar vpišeš v pogovor, zapusti tvoj računalnik in potuje do storitve.

V poslovnem okolju je pomembno, kateri paket uporabljaš in kaj piše v pogodbi tvojega podjetja.

Nekatere stvari ne sodijo v pogovor v nobenem primeru — ne zaradi nezaupanja, ampak ker to ni potrebno.

`Gesla, API ključi, žetoni`	nikoli — tudi ne 'samo za test'
`Osebni podatki strank`	le če podjetje to izrecno dovoljuje
`Zdravstveni in finančni podatki`	praviloma ne
`Pogodbe in interni dokumenti`	po pravilih podjetja
`Koda brez skrivnosti`	običajno v redu
`Izmišljeni testni podatki`	vedno v redu — in pogosto dovolj

> Praktičen prijem, ki reši večino primerov: pred pošiljanjem podatke anonimiziraj. Agent ne potrebuje pravih imen strank, da napiše program za obdelavo računov — potrebuje le pravo *obliko* podatkov. Zamenjaj imena, skrajšaj datoteko na deset vrstic in pošlji to. Rezultat bo enak, tveganje pa nič.
>
> **VEČ O TEM**
>
> - Anthropic — pomoč in pogosta vprašanja — https://support.claude.com/

Kaj je agent `agent`

Agent je jezikovni model, ki ima na voljo orodja in teče v zanki: razmisli, uporabi orodje, poglej rezultat, ponovi.

Razlika med klepetom in agentom ni v modelu, ampak v tem, ali sme kaj narediti.

Orodja so lahko branje datoteke, iskanje po spletu, zagon programa ali klic tuje storitve.

Zanka se konča, ko agent oceni, da je naloga opravljena — ali ko ga ustaviš ti.

Zanka agenta

```
Ti:     "Preimenuj vse racune po datumu."



Agent:  razmisli -> "moram videti, kaj je v mapi"

        orodje   -> dir prejeto\

        rezultat <- 214 datotek

        razmisli -> "preberem prvi PDF in poiscem datum"

        orodje   -> python preberi.py

        rezultat <- napaka: manjka knjiznica

        razmisli -> "namestim jo"

        orodje   -> pip install pypdf

        ...

        konec    -> "Preimenoval sem 209 datotek, 5 jih je v mapi rocno."
```

Vsak korak zanke je nov klic modela - in vsak stane tokene.

> Iz izpisa je razvidna tudi cena: agent, ki naredi trideset korakov, pošlje model tridesetkrat, vsakič z vsem, kar je do tedaj videl. Zato so pri agentih majhne datoteke, jasna navodila in omejen obseg naloge neposredno vidni v računu — in zato je `arhitektura.md` iz pojma 9.8 tako koristna.

VEČ O TEM

- Claude Code — pregled — https://docs.claude.com/en/docs/claude-code/overview

Claude: klepet `Claude chat`

Klepet je pogovorno okno brez dostopa do tvojih datotek — razen tistih, ki jih sam priložiš.

Namenjen je razmisleku, razlagi, pisanju in načrtovanju, ne izvajanju.

Prav zato je najboljši prostor za prvi korak vsakega projekta: da idejo najprej razjasniš.

`Za kaj je najboljši`	razmislek, razlaga, pisanje, načrt
`Kaj vidi`	samo pogovor in kar priložiš
`Kaj lahko spremeni`	nič na tvojem računalniku
`Tipična uporaba`	*razloži mi*, *razmisli z mano*, *napiši osnutek*

> Napaka, ki jo naredi skoraj vsak na začetku, je, da gre takoj v orodje, ki piše kodo. Pol ure pogovora v navadnem klepetu, kjer razjasniš, kaj sploh hočeš, prihrani nekajkrat toliko dela kasneje. Recept za to je v pojmu 11.5.
>
> **VEČ O TEM**
>
> - Claude — pomoč in navodila — https://support.claude.com/en/collections/4078531-claude-apps

Claude: Cowork `Claude Cowork`

Cowork je agentsko okolje za delo, ki ni programiranje: dokumenti, preglednice, predstavitve, raziskave.

Claude tam ne le svetuje, ampak sam opravi zaporedje korakov in vrne izdelek.

Namenjen je ljudem, ki jih ne zanima koda, zanima pa jih rezultat.

`Za kaj je najboljši`	daljše opravilo z izdelkom na koncu
`Kaj vidi`	datoteke, ki mu jih daš, in povezane storitve
`Kaj lahko spremeni`	ustvari in ureja datoteke v svojem prostoru
`Tipična uporaba`	*preglej teh 40 poročil in naredi povzetek*

> Meja med Cowork in Code ni v težavnosti, ampak v vrsti izdelka: če je rezultat dokument, tabela ali analiza, je Cowork prava izbira; če je rezultat koda, ki se bo poganjala in vzdrževala, je Code. Ker se izdelki hitro razvijajo, preveri trenutne zmožnosti na uradni strani.
>
> **VEČ O TEM**
>
> - Claude — pomoč in navodila — https://support.claude.com/en/collections/4078531-claude-apps

Claude: Code `Claude Code`

Claude Code je agent, ki teče na tvojem računalniku in ima dostop do mape projekta.

Bere in piše datoteke, poganja ukaze, preizkuša kodo in popravlja, kar ne deluje.

Prav v njem je nastalo to gradivo — vključno z zagonskimi datotekami in tem besedilom.

`Za kaj je najboljši`	graditi in vzdrževati kodo in orodja
`Kaj vidi`	mapo projekta, ki mu jo določiš
`Kaj lahko spremeni`	datoteke v tej mapi in ukaze, ki jih odobriš
`Varovalka`	vsak nevaren ukaz mora potrditi človek
`Tipična uporaba`	*zgradi orodje po arhitektura.md*

> Ker ima dostop do datotek, je tu Git iz pojma 9.7 nepogrešljiv. Pravilo, ki ga ni vredno kršiti: commit pred vsako sejo. In drugo: agentu določi mapo projekta, ne celotnega diska — tako je obseg morebitne škode vnaprej omejen.
>
> **VEČ O TEM**
>
> - Claude Code — pregled — https://docs.claude.com/en/docs/claude-code/overview
> - Claude Code — predstavitev izdelka — https://claude.com/product/claude-code

Artifacts `artifacts`

Artifact je samostojen izdelek, ki nastane med pogovorom in se prikaže ob njem: dokument, stran, aplikacija ali diagram.

Ni le besedilo v pogovoru — je stvar, ki jo lahko odpreš, uporabljaš in deliš.

Uporaben je takrat, ko rezultat ni odgovor, ampak nekaj, kar bo nekdo dejansko uporabljal.

> Za netehničnega uporabnika je to najhitrejša pot od ideje do uporabne stvari: opišeš, kaj potrebuješ, in dobiš delujoč pripomoček, ne navodil, kako ga narediti. Ista tehnologija kot v Modulu 6 — HTML, CSS in JavaScript — le da je vmesni korak odpadel.
>
> **VEČ O TEM**
>
> - Claude — pomoč in navodila — https://support.claude.com/en/collections/4078531-claude-apps

Connectors `connectors`

Connector je povezava med Claudom in storitvijo, ki jo že uporabljaš — koledarjem, pošto, diskom, sistemom za naloge.

Ko je vzpostavljena, Claude te podatke lahko bere in po potrebi tudi spreminja.

Vsaka taka povezava razširi, kaj agent zmore — in hkrati, kaj lahko pokvari.

> Pri povezavah velja isto pravilo kot pri javnih naslovih iz pojma 7.8: vklopi le tiste, ki jih res potrebuješ, in preveri, kaj vsaka dovoljuje. Razlika med *sme brati koledar* in *sme pošiljati vabila v tvojem imenu* je velika, čeprav sta obe povezavi videti enako. Tehnično connectorji stojijo na protokolu MCP iz pojma 10.17.
>
> **VEČ O TEM**
>
> - Anthropic — oddaljeni strežniki MCP — https://docs.claude.com/en/docs/agents-and-tools/remote-mcp-servers

Skills `skills`

Skill je zapisan postopek, ki ga agent prebere takrat, ko ga potrebuje — priročnik na polici, ne znanje na pamet.

Je navadna mapa z datoteko `SKILL.md`, v kateri piše, kdaj se uporabi in kako se naloga opravi.

Ker se naloži le ob pravem trenutku, ne obremenjuje konteksta, dokler ni potreben.

Pomembnejše od tehnike je posledica: dober postopek zapišeš enkrat in ga imaš vsakič.

SKILL.md

```
---
name: mesecno-porocilo
description: Uporabi, kadar uporabnik zahteva mesecno porocilo
             o prejetih racunih.
---

# Mesecno porocilo

1. Preberi racuni.csv
2. Filtriraj na zahtevani mesec
3. Sestej po dobaviteljih, uredi padajoce
4. Izpisi tabelo in skupni znesek
5. Opozori na racune brez dobavitelja
```

Pet vrstic navodila nadomesti pet minut razlaganja - vsakic znova.

> Opis v glavi je najpomembnejši del: iz njega agent ugotovi, *kdaj* naj skill sploh uporabi. Slab opis pomeni, da skill obstaja, a se nikoli ne sproži. Skills so tudi najlažji način, da svoje delovne postopke deliš s sodelavci — mapo preprosto pošlješ naprej.

VEČ O TEM

- Anthropic — Agent Skills — https://docs.claude.com/en/docs/agents-and-tools/agent-skills
- Claude Code — skills — https://docs.claude.com/en/docs/claude-code/skills

MCP — kako agent vidi svet MCP

MCP je dogovorjen način, kako se agentu pove, katera orodja ima na voljo in kako jih pokliče.

Vsako orodje se predstavi z imenom, opisom in obliko vhoda — agent ga nato kliče kot API iz Modula 6.

Ker je dogovor enoten, isti strežnik MCP deluje z različnimi agenti in različnimi modeli.

Preprosto povedano: MCP je vtičnica, v katero priklopiš svoje sisteme.

orodje.json

```
{
  "name": "poisci_racun",
  "description": "Poisce racun po stevilki in vrne znesek in datum.",
  "inputSchema": {
    "type": "object",
    "properties": {
      "stevilka": { "type": "string" }
    },
    "required": ["stevilka"]
  }
}
```

Agent prebere OPIS in iz njega sklepa, kdaj naj orodje uporabi.

> Zdaj se sestavi cela veriga iz prejšnjih modulov: opis orodja je JSON (3.5), klic je zahteva z odgovorom (5.7, 6.6), rezultat se vrne kot JSON in se prilepi v kontekst (10.4). Zato je kakovost opisa tako pomembna — natanko tako kot dobro ime funkcije iz pojma 9.3. Orodje s slabim opisom agent bodisi spregleda bodisi uporabi narobe.

VEČ O TEM

- Model Context Protocol — uradna stran — https://modelcontextprotocol.io/
- Anthropic — MCP — https://docs.claude.com/en/docs/mcp

Pod-agenti `sub-agents`

Pod-agent je ločen agent s svojim kontekstom, ki ga glavni agent pokliče za zaokroženo opravilo.

Prednost je dvojna: glavni kontekst ostane čist, pod-agent pa vidi samo to, kar res potrebuje.

Prav ta omejenost je uporabna — pod-agent, ki ne vidi testov, jim ne more pisati kode na kožo.

Delitev dela

```
GLAVNI AGENT  (ve vse o projektu)
  |
  +-- pod-agent A: "napisi teste za funkcijo z_ddv"
  |      vidi: zahtevo + arhitektura.md
  |      NE vidi: obstojeco kodo
  |
  +-- pod-agent B: "implementiraj funkcijo z_ddv"
  |      vidi: zahtevo + arhitektura.md
  |      NE vidi: teste agenta A
  |
  +-- pozene teste -> 3 passed, 1 failed -> B popravlja
```

Loceni konteksti niso omejitev, ampak namen.

> Cena so dodatni tokeni: vsak pod-agent je svoj pogovor. Zato se prijem splača pri opravilih, kjer je natančnost pomembnejša od cene — pri pisanju testov, pri pregledu kode in pri raziskovanju, kjer bi glavni kontekst hitro zapolnili nepomembni podatki.

VEČ O TEM

- Claude Code — pod-agenti — https://docs.claude.com/en/docs/claude-code/sub-agents

Dobre prakse pri gradnji agentov `building agents well`

Agent je toliko dober, kolikor natančno je opisana naloga in kolikor omejena so njegova orodja.

Najpogostejši vzrok slabega rezultata ni model, ampak preohlapno navodilo in prevelik obseg.

`Zapiši` `` `arhitektura.md` ``	brez nje agent ugiba namen
`Omeji mapo projekta`	manj dostopa, manj škode
`Ena naloga, en pogovor`	kontekst ostane čist
`Zahtevaj načrt pred izvedbo`	napako v načrtu je ceneje popraviti
`Testi pred implementacijo`	in ločen pod-agent zanje
`Commit pred vsako sejo`	varnostna mreža
`Preveri, kaj je naredil`	diff prebere človek, ne agent
`Zapiši, kaj ni delovalo`	sicer bo poskusil znova

> Vse te prakse imajo skupen imenovalec: zmanjšajo negotovost. Model je nedeterminističen (2.6), zato ga ne moreš narediti zanesljivega — lahko pa zmanjšaš prostor, v katerem se lahko zmoti. Jasen cilj, majhen obseg, preverljiv izhod in možnost vrnitve nazaj so štirje vzvodi, ki jih imaš v rokah.
>
> **VEČ O TEM**
>
> - Claude Code — pregled — https://docs.claude.com/en/docs/claude-code/overview

MODUL 11

Delovni tok od ideje do kode

Celoten recept: od stavka v klepetu do preverjene kode, korak za korakom.

Skill: grilling `grilling`

Grilling je skill, ki obrne vloge: namesto da ti agent odgovarja, te začne spraševati.

Vprašanja postavlja v krogih — vsak krog odpre le tista vprašanja, na katera je mogoče odgovoriti zdaj.

Pri vsakem vprašanju ponudi svoje priporočilo, zato je odgovarjanje hitro tudi, kadar nimaš mnenja.

Rezultat je, da ideja postane odločitev, preden nastane ena sama vrstica kode.

Kako izgleda

```
Ti:    "Rad bi orodje za obdelavo racunov."


Agent: Q1 - Vhod: so racuni PDF, skenirani ali oboje?
          -> Predlog: oboje, s samodejnim zaznavanjem.

       Q2 - Kaj naj se zgodi, ce izluscanje ne uspe?
          -> Predlog: datoteka gre v mapo "rocno" in se zabelezi.

       Q3 - Kje bodo podatki koncali: CSV ali baza?
          -> Predlog: CSV, ker je 200 racunov na mesec.
```

To gradivo je nastalo natanko tako - z 28 vprasanji v stirih krogih.

> Vrednost ni v vprašanjih samih, ampak v tem, da te prisilijo odgovoriti na stvari, ki bi jih sicer agent tiho ugibal. Vsako neodgovorjeno vprašanje je mesto, kjer bo nastalo nekaj, česar nisi hotel. Uporabi ga vedno, kadar je naloga večja od ene datoteke.

VEČ O TEM

- grilling — izvorna koda skilla — https://github.com/mattpocock/skills/tree/main/skills/productivity/grilling

Skill: handoff `handoff`

Handoff pripravi predajo dela: povzetek stanja, ki ga lahko prevzame nov pogovor ali drug človek.

Reši natanko težavo iz pojma 10.6 — kaj narediti, ko se kontekstno okno polni.

Dober handoff vsebuje tudi, kaj je bilo že poskušeno in ni delovalo.

predaja.md

```
# Stanje projekta - 21. 9. 2026

## Cilj
Orodje za obdelavo prejetih racunov (glej arhitektura.md).

## Narejeno
- branje PDF, izluscanje datuma in zneska (deluje na 209/214)
- zapis v racuni.csv

## Odprto
- 5 skeniranih racunov potrebuje OCR

## NE poskusaj znova
- pdfminer: pocasnejsi in slabsi rezultat kot pypdf
- regex za znesek brez konteksta vrstice: prevec lazno pozitivnih
```

Zadnji razdelek je tisti, ki loci dobro predajo od seznama zelja.

> Isto datoteko uporabiš tudi, ko delo predaš sodelavcu ali ko se k projektu vrneš čez mesec dni. Pravzaprav je to isti dokument kot `arhitektura.md`, le da opisuje trenutno stanje namesto namena — mnogi ju zato držijo skupaj v eni datoteki z dvema razdelkoma.
>
> **VEČ O TEM**
>
> - handoff — izvorna koda skilla — https://github.com/mattpocock/skills/tree/main/skills/productivity/handoff

Skill: improve `improve`

Improve pregleda obstoječo kodo kot izkušen svetovalec in pripravi načrt izboljšav.

Sam kode ne spreminja — njegov izdelek je načrt, dovolj natančen, da ga izvede drug agent.

Prav ta ločitev je bistvo: razmislek opravi močan model, izvedbo pa lahko cenejši.

Kaj naredi

```
1. Prebere projekt: jezik, zgradbo, teste, navade

2. Poisce priloznosti: napake, varnost, hitrost, manjkajoci testi

3. Vsako ugotovitev PREVERI v kodi (ne zaupa svojemu prvemu vtisu)

4. Predstavi seznam, urejen po razmerju ucinek/trud

5. Ti izberes, kaj naj postane nacrt

6. Za vsako izbrano napise samostojen nacrt v mapo plans/
```

Nacrt je samostojen: izvajalec ga razume brez tega pogovora.

> Tretji korak je tisti, ki ga ljudje spregledajo in ki največ prinese: skill svoje ugotovitve preveri, preden jih predstavi, ker so prvi vtisi pogosto napačni. Isti prijem je vreden posnemanja tudi ročno — preden agentu verjameš, da je nekaj narobe, ga prosi, naj pokaže vrstico.
>
> **VEČ O TEM**
>
> - improve — izvorna koda skilla — https://github.com/shadcn/improve

Kje dobiš skills `skills.sh`

Skills so navadne mape z datoteko `SKILL.md` — lahko jih napišeš sam ali prevzameš od drugih.

Zbirka preverjenih skillov je dosegljiva na *skills.sh*.

Preden skill uporabiš, ga preberi: to so navodila, ki jih bo agent izvedel na tvojem računalniku.

`skills.sh`	zbirka skillov za brskanje in namestitev
`` `SKILL.md` ``	srce vsakega skilla — preberi ga pred uporabo
`Polje` `` `description` ``	odloča, kdaj se skill sproži
`Lasten skill`	najhitrejša pot: zapiši postopek, ki ga ponavljaš
`Deljenje`	mapo pošlješ sodelavcu, deluje takoj

> Enako pravilo kot pri knjižnicah iz pojma 8.10: tuja navodila so tuja koda. Skill, ki ga ne razumeš, lahko agentu naroči karkoli — tudi brisanje ali pošiljanje podatkov. Pri znanih virih je tveganje majhno, a branje pred uporabo je navada, ki se splača.
>
> **VEČ O TEM**
>
> - skills.sh — https://skills.sh/
> - Anthropic — Agent Skills — https://docs.claude.com/en/docs/agents-and-tools/agent-skills

Recept 1: od ideje do arhitekture `recipe 1`

Prvi del recepta se odvije v navadnem klepetu — brez kode, brez orodij, brez hitenja.

Cilj je ena sama datoteka `arhitektura.md`, iz katere bo agent kasneje gradil.

Šele ko ta datoteka obstaja, se splača odpreti orodje, ki piše kodo.

`1. Opiši idejo`	v navadnem klepetu, s svojimi besedami
`2. Povej vhode`	kaj imaš: datoteke, tabele, dostopi
`3. Povej izhode`	kaj pričakuješ: datoteka, poročilo, orodje
`4. Zahtevaj grilling`	*grill me* — odgovarjaj po krogih
`5. Navedi omejitve`	Windows, brez namestitev, podatki ostanejo doma
`6. Zahtevaj` `arhitektura.md`	in jo preberi, preden greš naprej
`7. Shrani jo v mapo projekta`	tam jo bo agent našel

> Šesta točka ni formalnost. Preberi datoteko kot naročnik, ne kot bralec: je v njej kaj, česar nisi rekel? Manjka kaj, kar si mislil, da je očitno? Vsak stavek, ki ga popraviš tu, je popravek, ki ga kasneje ne bo treba delati v kodi — in to je razlika med desetimi minutami in dvema dnevoma.
>
> **VEČ O TEM**
>
> - Claude Code — datoteka s spominom projekta — https://docs.claude.com/en/docs/claude-code/memory

Recept 2: od arhitekture do kode `recipe 2`

Drugi del se odvije v orodju, ki ima dostop do mape — na primer v Claude Code.

Vrstni red je vedno isti: načrt, potrditev, testi, izvedba, preverba.

Nikoli ne preskoči potrditve načrta; napaka v načrtu je stokrat cenejša od napake v kodi.

`1. \`git init\` in prvi commit`	varnostna mreža, preden se karkoli začne
`2. Daj mu mapo projekta`	in `arhitektura.md` vanjo
`3. Če kaj ni jasno: grilling`	raje vprašanja zdaj kot ugibanje kasneje
`4. Če je jasno: improve`	naj pripravi načrt implementacije
`5. Preberi načrt`	in ga potrdi ali popravi
`6. Testi s pod-agentom`	ločen agent, ki ne bo pisal kode
`7. Implementacija`	drug agent, ki testov ne vidi
`8. Poženi teste`	in pusti agenta popravljati, dokler ni zeleno
`9. Preglej diff`	to je edini korak, ki ga opravi človek
`10. Commit`	in šele nato naslednja naloga

> Deveta točka je tista, ki je ni mogoče prenesti na stroj. Ni treba, da razumeš vsako vrstico — dovolj je, da preveriš, ali je agent spremenil tisto, kar si pričakoval, in ali ni mimogrede spremenil še česa drugega. Prav zato so majhni koraki in pogosti commiti tako pomembni: diff, ki obsega tri datoteke, se da prebrati, diff s tridesetimi pa ne.
>
> **VEČ O TEM**
>
> - Claude Code — pregled — https://docs.claude.com/en/docs/claude-code/overview
> - Git — dokumentacija — https://git-scm.com/doc

Implementator ne vidi testov `blind implementer`

Teste naj napiše en agent, kodo pa drug, ki testov ne vidi.

Če isti agent napiše oboje, bo teste nezavedno prilagodil kodi, ki jo je pravkar napisal.

Tako oboje ustreza isti napačni predpostavki in testi ne dokazujejo ničesar.

Ločitev je poceni, izvedljiva v enem stavku navodila in opazno zmanjša število napak.

Navodilo, ki to doseže

```
Ti (glavnemu agentu):



  "Za funkcijo z_ddv najprej s pod-agentom napisi teste

   po arhitektura.md. Nato z LOCENIM pod-agentom, ki testov

   NE vidi, implementiraj funkcijo. Nato pozeni teste in

   pusti drugega pod-agenta popravljati, dokler vsi ne gredo skozi.

   Testov med popravljanjem ne spreminjaj."
```

Zadnji stavek je kljucen: sicer bo agent popravil test namesto kode.

> Zadnja vrstica ni pretirana previdnost. Ko test pade, je popravljanje testa najkrajša pot do zelene barve — in agent bo, če mu tega ne prepoveš, to pot ubral. Test, ki je bil prilagojen kodi, ni več zahteva, ampak opis tega, kar koda pač počne. Če se med delom izkaže, da je test res napačen, ga popravi ti, zavestno in z razlogom.

VEČ O TEM

- Claude Code — pod-agenti — https://docs.claude.com/en/docs/claude-code/sub-agents

MODUL 12

Velika slika

Kako se vse našteto poveže in kam naprej.

Kako se vse poveže `the big picture`

Vse gradivo stoji na štirih idejah, ki se ponavljajo v vsakem modulu.

Prva: kar je tekst, je za računalnik in za AI uporabno; kar ni, je treba najprej pretvoriti.

Druga: ena resnica na enem mestu — spremenljivka, ključ v bazi, `arhitektura.md`.

Tretja: koda je predvidljiva, AI ni, zato mora biti izhod preverljiv.

Četrta: velik problem se reši samo tako, da postane zaporedje majhnih.

Ista ideja skozi module

```
ENA RESNICA NA ENEM MESTU

  1.2  spremenljivka        ena skatla, ne sedemnajst

  4.4  kljuc v bazi         ime stranke zapisano enkrat

  6.3  barva v CSS          --accent na enem mestu

  9.8  arhitektura.md       odlocitve na enem mestu

 10.6  povzetek seje        stanje na enem mestu


TEKST JE UPORABEN, BINARNO NI

  3.2  tekst vs. binarno    temeljna delitev

  3.10 AI-prijazni formati  posledica

  3.11 OCR                  resitev, ko je prepozno
```

Ko opazis vzorec, si nehas zapomnjevati podrobnosti.

> Če boš čez pol leta pozabil vse podrobnosti — kaj počne `git restore`, katera vrata uporablja PostgreSQL, kako se piše `SELECT` — se ne bo zgodilo nič hudega. Vse to je mogoče poiskati ali vprašati agenta. Štiri ideje zgoraj pa so tiste, zaradi katerih boš znal postaviti pravo vprašanje, in prav to je razlika med uporabnikom in graditeljem.

VEČ O TEM

- Anthropic — Agent Skills — https://docs.claude.com/en/docs/agents-and-tools/agent-skills

Kam naprej `where to go next`

Ne začni z velikim projektom, ampak z opravilom, ki te vsak teden jezi.

Naj bo dovolj majhno, da se konča v enem popoldnevu, in dovolj tvoje, da veš, kdaj je prav.

Prvi uspeh naj bo nekaj, kar boš dejansko uporabljal — takrat se vse iz tega gradiva usede samo od sebe.

`Prvi projekt`	opravilo, ki ga ponavljaš vsak teden
`Prvo orodje`	navaden klepet — razjasni idejo
`Prva datoteka`	`arhitektura.md`, napisana z grillingom
`Prva navada`	commit pred vsako sejo z agentom
`Druga navada`	en pogovor, ena naloga
`Ko se zatakne`	preberi zadnjo vrstico napake in jo prilepi agentu
`Ko okno poči`	povzetek in nov pogovor

> Najpogostejša napaka po takšni delavnici ni tehnična, ampak izbira prvega projekta: ljudje se lotijo nečesa velikega, ker je navdušenje sveže, in obtičijo. Majhno opravilo, ki ga dokončaš, nauči več kot velik projekt, ki ga opustiš — in ti da edino stvar, ki resnično prepriča sodelavce: delujoč primer.
>
> **VEČ O TEM**
>
> - skills.sh — zbirka skillov — https://skills.sh/
> - Claude Code — pregled — https://docs.claude.com/en/docs/claude-code/overview

SLOVARČEK

Slovarček

Vsi pojmi po abecedi, slovensko ↔ angleško.

Anatomija naslova `URL`

Naslov spletne strani ni ena beseda, ampak šest ločenih delov, od katerih ima vsak svojo nalogo.

API klic `API call`

API je dogovorjen seznam vprašanj, ki jih program lahko postavi drugemu programu.

Artifacts `artifacts`

Artifact je samostojen izdelek, ki nastane med pogovorom in se prikaže ob njem: dokument, stran, aplikacija ali diagram.

Bash in PowerShell `Bash / PowerShell`

To sta jezika ukazne vrstice: namenjena sta vodenju drugih programov, ne pisanju aplikacij.

Brskalnik je izvajalec `browser / rendering engine`

Brskalnik ni okno v internet, ampak program, ki datoteke prevzame in jih izvede na tvojem računalniku.

C in C++ `C / C++`

Najbližja strojni ravni; uporabljata se tam, kjer šteje vsaka tisočinka sekunde.

Claude: Code `Claude Code`

Claude Code je agent, ki teče na tvojem računalniku in ima dostop do mape projekta.

Claude: Cowork `Claude Cowork`

Cowork je agentsko okolje za delo, ki ni programiranje: dokumenti, preglednice, predstavitve, raziskave.

Claude: klepet `Claude chat`

Klepet je pogovorno okno brez dostopa do tvojih datotek — razen tistih, ki jih sam priložiš.

Connectors `connectors`

Connector je povezava med Claudom in storitvijo, ki jo že uporabljaš — koledarjem, pošto, diskom, sistemom za naloge.

Context window `context window`

Kontekstno okno je zgornja meja, koliko tokenov lahko model naenkrat vidi.

CSS — videz `CSS`

CSS pove, kako naj stvari izgledajo: barve, velikosti, razmiki, postavitev.

CSV — tabela kot tekst `.csv`

CSV je tabela, zapisana kot navaden tekst: ena vrstica je ena vrstica tabele, vejica loči stolpce.

Četrta stopnja: najet strežnik `VPS`

VPS je računalnik v tujem podatkovnem centru, ki ga najameš za nekaj evrov na mesec.

Datoteka arhitektura.md `architecture.md`

To je dokument, v katerem opišeš, kaj gradiš, preden agent napiše prvo vrstico kode.

Datoteka, končnica, pot `file, extension, path`

Datoteka je kos podatkov z imenom; končnica extension za piko je samo obljuba, kaj je notri.

Dobre prakse pri gradnji agentov `building agents well`

Agent je toliko dober, kolikor natančno je opisana naloga in kolikor omejena so njegova orodja.

Dokumentna baza `document database / NoSQL`

Dokumentna baza ne hrani vrstic, ampak cele objekte — take, kot si jih videl pri JSON.

Domena in DNS `domain, DNS`

Računalniki se med sabo ne kličejo po imenih, ampak po številkah.

Druga stopnja: pisarna `LAN hosting`

Če strežnik posluša na vseh omrežnih karticah, ga vidijo vsi na isti wifi mreži.

Excel kot izhodišče `spreadsheet`

Excelova tabela je že skoraj baza podatkov: ima stolpce z imeni in vrstice z vrednostmi.

Frontend in backend `frontend / backend`

Frontend je vse, kar teče v brskalniku pri uporabniku — HTML, CSS, JavaScript.

Funkcija `function`

Funkcija je poimenovan kos kode: noter daš podatke arguments, ven dobiš rezultat return value.

Git in commit `Git, commit`

Git si zapomni vsako stanje kode, ki ga potrdiš — in omogoči vrnitev na katerokoli od njih.

Halucinacije `hallucination`

Halucinacija je odgovor, ki zveni pravilno, a ni resničen — izmišljena številka, neobstoječa funkcija, napačna povezava.

HTML `.html`

HTML je tekst z oznakami, ki povedo, kaj je naslov, kaj odstavek in kaj povezava.

HTML — zgradba `HTML`

HTML pove, kaj je na strani: naslov, odstavek, seznam, gumb, slika.

Implementator ne vidi testov `blind implementer`

Teste naj napiše en agent, kodo pa drug, ki testov ne vidi.

IP naslov `IP address`

IP naslov je hišna številka računalnika v omrežju, zapisana kot štiri števila med 0 in 255.

Isti podatki, štiri oblike `same data, four shapes`

Ena sama vrstica podatkov se zapiše na štiri načine, ki jih boš srečeval povsod.

Izvajalno okolje in Node.js `runtime / Node.js`

Jezik sam po sebi ne teče — potrebuje program, ki ga izvaja; temu pravimo izvajalno okolje runtime.

Java in C# `Java / C#`

Jezika velikih poslovnih sistemov: bančništvo, zavarovalništvo, ERP, javna uprava.

JavaScript — obnašanje `JavaScript / TypeScript`

JavaScript je edini programski jezik, ki ga brskalnik razume neposredno.

JavaScript in TypeScript `JavaScript / TypeScript`

JavaScript je edini jezik, ki teče neposredno v brskalniku — zato je neizogiben za spletne strani.

JSON — struktura kot tekst `.json`

JSON je zapis objektov in seznamov iz Modula 1 v obliki navadnega teksta.

Kaj je agent `agent`

Agent je jezikovni model, ki ima na voljo orodja in teče v zanki: razmisli, uporabi orodje, poglej rezultat, ponovi.

Kaj je baza podatkov `database`

Baza je program, ki hrani podatke in zna hitro odgovarjati na vprašanja o njih.

Kaj je bug `bug`

Bug je razlika med tem, kar si mislil, da si naročil, in tem, kar si dejansko naročil.

Kaj je jezikovni model `LLM`

Jezikovni model napoveduje, kateri del besedila najverjetneje sledi temu, kar je že napisano.

Kaj je program `program / code`

Program je zaporedje navodil, ki jih računalnik izvede eno za drugim, od zgoraj navzdol.

Kaj je test `test`

Test je majhen program, ki požene tvojo kodo in preveri, ali je rezultat pričakovan.

Kaj ne sme v AI `privacy`

Vse, kar vpišeš v pogovor, zapusti tvoj računalnik in potuje do storitve.

Kaj smeš dati na javni naslov `what to expose`

Ko nekaj dobi javni naslov, predpostavi, da bo to nekdo našel — tudi če naslova nisi nikomur dal.

Kaj stanejo tokeni `pricing`

Plačaš dvoje: tokene, ki jih pošlješ (vhod), in tokene, ki jih model napiše (izhod).

Kako brati sporočilo o napaki `traceback / stack trace`

Sporočilo o napaki ni kazen, ampak najbolj natančen opis problema, kar ga boš dobil.

Kako porabiti manj tokenov `saving tokens`

Poraba ni odvisna od tega, koliko napišeš, ampak koliko model vsakič prebere.

Kako se vse poveže `the big picture`

Vse gradivo stoji na štirih idejah, ki se ponavljajo v vsakem modulu.

Kam naprej `where to go next`

Ne začni z velikim projektom, ampak z opravilom, ki te vsak teden jezi.

Kateri formati so AI-prijazni `AI-friendly formats`

Pravilo je preprosto: bližje kot je format navadnemu tekstu, manj težav boš imel.

Kdaj katera `choosing a store`

Izbira je skoraj vedno odvisna od enega vprašanja: ali so podatki povsod enake oblike?

Kje dobiš skills `skills.sh`

Skills so navadne mape z datoteko SKILL.md — lahko jih napišeš sam ali prevzameš od drugih.

Knjižnica ali lastna koda `build or borrow`

Uporabi knjižnico, kadar je problem splošen in dobro rešen: datumi, PDF, šifriranje, omrežje.

Knjižnice `libraries (pip, npm)`

Knjižnica je koda, ki jo je nekdo drug že napisal in jo lahko uporabiš namesto svoje.

Koda je predvidljiva, AI ni `deterministic / non-deterministic`

Ista koda z istim vhodom da vsakič isti rezultat — to se imenuje determinizem deterministic.

Komentar `comment`

Komentar je vrstica, ki jo računalnik v celoti prezre, človek pa jo prebere.

localhost `localhost / 127.0.0.1`

localhost pomeni ta računalnik — naslov, ki nikoli ne zapusti tvoje naprave.

Lokalno omrežje `LAN, 192.168.x.x`

Ko strežnik posluša na vseh omrežnih karticah, ga vidijo vsi, ki so na isti wifi mreži.

Main in moduli `main + modules`

Koda se razdeli na module: vsaka datoteka opravlja eno vrsto dela.

MCP — kako agent vidi svet `MCP`

MCP je dogovorjen način, kako se agentu pove, katera orodja ima na voljo in kako jih pokliče.

Napaka ali sesutje `error / exception`

Ko program naleti na nekaj, česar ne zna, sproži izjemo exception — in se ustavi.

Navaden tekst in Markdown `.txt / .md`

.txt je gola vsebina brez oblikovanja — najbolj nedolžen format, kar jih je.

Objekt `object / dict / key-value`

Objekt je izpolnjen obrazec: vsako polje ima svoje ime key in svojo vrednost value.

OCR — iz slike v tekst `OCR`

OCR optical character recognition je postopek, ki iz slike prepozna črke in vrne navaden tekst.

Odvisnosti in različice `dependencies, versions`

Program, ki uporablja pet knjižnic, je odvisen od petih tujih projektov in njihovih sprememb.

PDF `.pdf`

PDF je narejen za to, da je povsod videti enako — ne za to, da bi iz njega jemali podatke.

Peta stopnja: statično gostovanje `static hosting`

Če je stran statična, je ni treba nikjer poganjati — dovolj je, da nekdo streže datoteke.

Pod-agenti `sub-agents`

Pod-agent je ločen agent s svojim kontekstom, ki ga glavni agent pokliče za zaokroženo opravilo.

Podatkovni tipi `data types`

Računalnik strogo loči med številom int, besedilom str, da/ne vrednostjo bool in datumom date.

Pogoj `if / else`

Pogoj je kretnica: če nekaj drži, gre program po eni poti, sicer po drugi.

Poimenovanje `naming`

Ime naj pove, kaj stvar je ali počne — ne, kako je narejena.

Pot enega klika `the journey of one click`

Klik na gumb sproži verigo šestih korakov, ki se zgodi v nekaj stotinkah sekunde.

Povzemi in začni znova `summarise and continue`

Ko se okno napolni okoli osemdeset odstotkov, agenta prosi za povzetek stanja.

Prevajani in interpretirani `compiled vs. interpreted`

Prevajani jezik se pred zagonom v celoti prevede v strojno kodo — nastane samostojen program .exe.

Prompt in kontekst `prompt / context`

Prompt je to, kar napišeš; kontekst je vse, kar model ob tem vidi.

Prva stopnja: moj računalnik `localhost`

Najnižja stopnja gostovanja je tvoj računalnik: program teče, ko ga poženeš, in umre, ko okno zapreš.

Python `Python`

Berljiv, zelo razširjen in odličen za obdelavo podatkov, avtomatizacijo in umetno inteligenco.

Razgradnja problema `decomposition`

Velikega problema ni mogoče rešiti; rešiti je mogoče le zaporedje majhnih.

Razhroščevanje `debugging`

Razhroščevanje je preverjanje domnev: kaj mislim, da je v tej spremenljivki, in kaj je v resnici.

Recept 1: od ideje do arhitekture `recipe 1`

Prvi del recepta se odvije v navadnem klepetu — brez kode, brez orodij, brez hitenja.

Recept 2: od arhitekture do kode `recipe 2`

Drugi del se odvije v orodju, ki ima dostop do mape — na primer v Claude Code.

Relacijska baza in ključi `relational database, primary/foreign key`

Relacijska baza hrani podatke v več tabelah, ki so med sabo povezane.

Seznam `list / array`

Seznam je oštevilčeno zaporedje vrednosti pod enim samim imenom.

Skill: grilling `grilling`

Grilling je skill, ki obrne vloge: namesto da ti agent odgovarja, te začne spraševati.

Skill: handoff `handoff`

Handoff pripravi predajo dela: povzetek stanja, ki ga lahko prevzame nov pogovor ali drug človek.

Skill: improve `improve`

Improve pregleda obstoječo kodo kot izkušen svetovalec in pripravi načrt izboljšav.

Skills `skills`

Skill je zapisan postopek, ki ga agent prebere takrat, ko ga potrebuje — priročnik na polici, ne znanje na pamet.

Slika ali risba `raster vs. vector (.png / .svg)`

.jpeg in .png sta mreži pik — povečaj ju in postanejo mehki.

Smeti noter, smeti ven `garbage in, garbage out`

Računalnik ne preverja, ali so podatki smiselni — preveri samo, ali jih zna obdelati.

Spletna tehnologija brez spleta `local web app, offline`

HTML, CSS in JavaScript ne potrebujejo interneta — potrebujejo samo brskalnik.

Spremenljivka `variable`

Spremenljivka je poimenovano mesto, kjer je shranjena ena vrednost.

SQL `SQL`

SQL je jezik za pogovor z bazo in je presenetljivo podoben angleškemu stavku.

SQL `SQL`

SQL ni jezik za pisanje programov, ampak za spraševanje baz — in prav zato ga srečaš povsod.

Statično in dinamično `static vs. dynamic`

Statična stran je datoteka, ki se vsem prikaže enako — strežnik jo samo pošlje naprej.

Statusne kode `status codes`

Vsak odgovor se začne s trimestno številko, ki pove, kako se je zahteva iztekla.

Šesta stopnja: oblak `cloud (AWS, Azure)`

Oblak ni en računalnik, ampak stotine storitev, ki jih sestaviš po potrebi in plačaš po porabi.

Tekstovno ali binarno `text vs. binary`

Vse datoteke so v osnovi zaporedje števil; razlika je samo v tem, ali ta števila predstavljajo črke.

Terminal `terminal / command line / cmd`

Terminal je okno, v katerega pišeš ukaze, namesto da klikaš po gumbih.

Token `token`

Model ne bere črk in ne besed, ampak tokene — koščke besedila, dolge povprečno tri do štiri znake.

Tretja stopnja: tunel `tunnel (trycloudflare, ngrok)`

Tunel je program, ki iz tvojega računalnika vzpostavi povezavo navzven in ti podeli javni naslov.

Vrata `port`

Na enem računalniku teče več programov hkrati; vrata port povedo, kateri od njih naj sprejme povezavo.

Vrste testov `unit / integration / end-to-end`

Testi enot preverjajo posamezno funkcijo; hitri so in jih je lahko veliko.

Wordove in Excelove datoteke `.docx / .xlsx`

.docx in .xlsx sta v resnici stisnjeni mapi (.zip) z datotekami XML notri.

Zahteva in odgovor `HTTP request / response`

Splet deluje po preprostem vzorcu: brskalnik pošlje zahtevo, strežnik vrne odgovor, povezava se zapre.

Zakaj obstaja toliko jezikov `programming languages`

Vsi jeziki znajo isto — razlikujejo se po tem, kaj olajšajo in kaj otežijo.

Zakaj so testi pri AI še pomembnejši `tests with AI`

Agent piše kodo hitreje, kot jo ti utegneš prebrati — testi so edini način, da mu sledi preverjanje.

Zanka for `for loop`

Zanka ponovi isti postopek za vsak element seznama.

Zanka while `while loop`

while ponavlja, dokler je nek pogoj izpolnjen — koliko ponovitev bo, vnaprej ne veš.