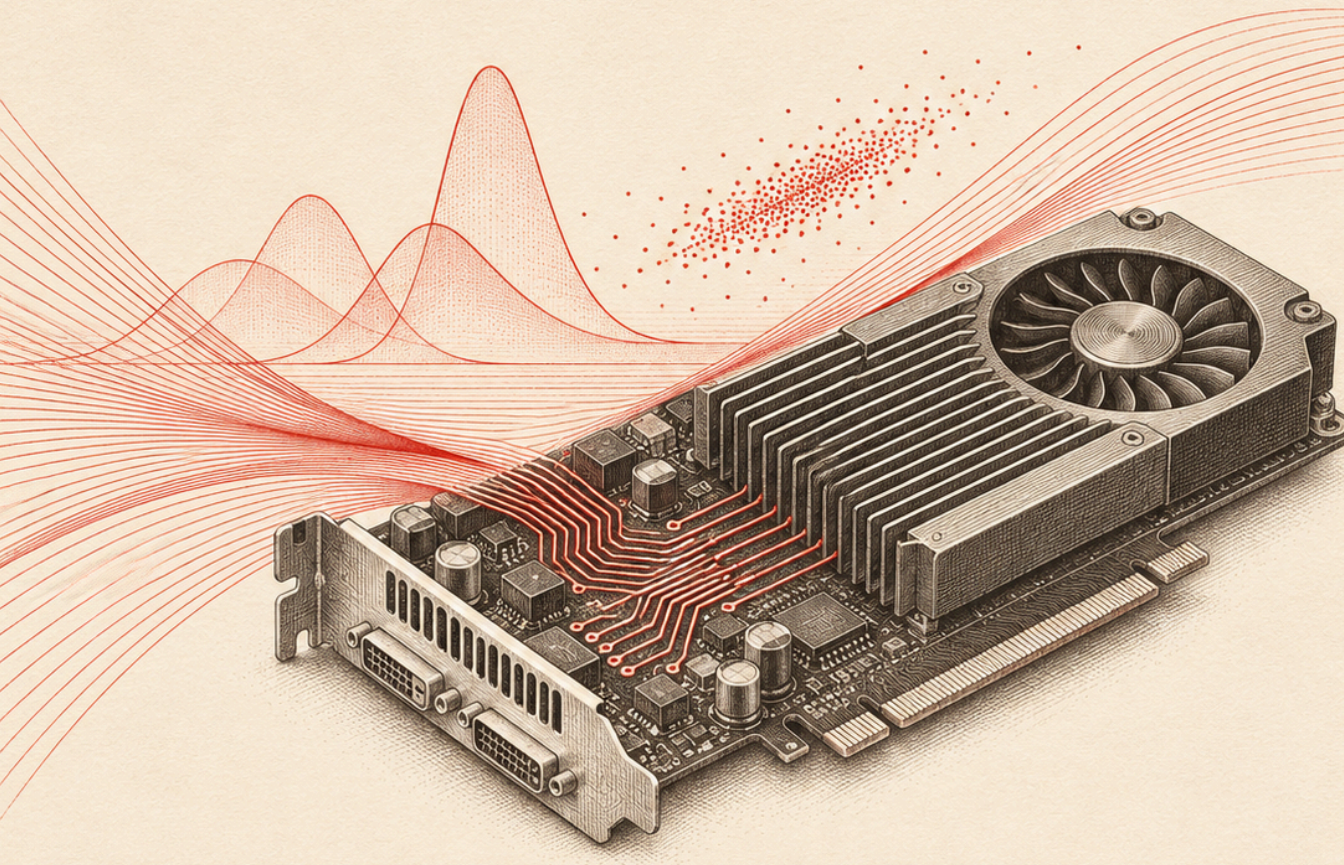
VECTORFALL

THE FREE GUIDE

PERSONAL SUPERCOMPUTING WITH VECTORFALL

A GPU-hosted algorithms lab for you and your AI agents



AMIR HUSAIN

VectorFall 0.1.0

CONTENTS

CHAPTER 1	Install and start the lab	3
CHAPTER 2	Run your first experiment	6
CHAPTER 3	Find the right workspace	10
CHAPTER 4	Map your question to a model	15
CHAPTER 5	Find unusual days in a table	21
CHAPTER 6	Build teams with rules	26
CHAPTER 7	Estimate outcomes under uncertainty	29
CHAPTER 8	Teach the lab a sound to find	33
CHAPTER 9	Trace text to its sources	37
CHAPTER 10	Develop and compare a photograph	41
CHAPTER 11	Write an analysis in Python	44
CHAPTER 12	Connect an agent to the lab	48
CHAPTER 13	Compare your card with historical systems	51

CHAPTER 1

INSTALL AND START THE LAB

A VectorFall installation consists of one container, one NVIDIA GPU, and a data directory on the host. Start with the card you already have. The installer checks the host and the NVIDIA container runtime, then asks which card to use if there is a choice. Open the browser on the same computer for the first session.

CHECK THE HOST

Use a Linux x86-64 host, an NVIDIA driver appropriate for the card, and Docker with the NVIDIA Container Toolkit or Podman with NVIDIA CDI. The host needs Python 3.6 or newer for `vfctl`, about 3 GB for the container image, and space for your files. Run `nvidia-smi -L` before installation. It should show each card you might want to use.

CARD FAMILY	WHAT TO CHECK
Kepler, such as Quadro K420	Use NVIDIA's 470 driver branch.
Maxwell and Pascal, such as Quadro K1200 and GTX 1080	Choose a driver branch supported by the card. The release uses CUDA 10.2 in the container.
Volta, Turing and newer	Check the card and driver against the repository's support table. Run a smoke test before relying on the combination.
Windows	Follow the Docker Desktop and WSL2 route, and check NVIDIA GPU access inside WSL2.

The installation notes cover container runtime setup and driver compatibility. A computer that shows its GPU in `nvidia-smi` can still have a container-runtime configuration problem. The installer tests both layers.

INSTALL THE RELEASE

```
curl -fsSLO
https://raw.githubusercontent.com/ahtx/vectorfall/v0.1.0/install.sh
bash install.sh
```

The installer prints each check as it goes. If a prerequisite needs administrative access, it prints the command for you to review and run. For a noninteractive installation you can give `--yes --gpu N` with an index from `nvidia-smi -L`. Add `--engine podman` to select Podman. Run `bash install.sh --help` for the other options.

Open http://localhost:8092 when the health check passes. The default data and configuration folder is `~/vectorfall`. The installer puts `vfctl` in `~/.local/bin`. If your shell cannot find it, run `~/.local/bin/vfctl status` or add that directory to your shell's path.

CHECK THE SERVER AND KEEP ITS DATA

COMMAND	WHAT TO USE IT FOR
`vfctl status`	Confirm the running version, chosen GPU, and address.
`vfctl doctor --quick`	Check the host, runtime and server configuration.
`vfctl gpu`	See which GPUs the host can offer.
`vfctl gpu select`	Change the card selected for this server.
`vfctl logs -f`	Read server logs.
`vfctl backup`	Archive the data directory. The default backup stops the server for a consistent copy.
`vfctl update`	Fetch the release image and restart the server with the existing data.

Shell examples use the `vf` client. Fetch it from the running server and make it executable:

```
mkdir -p "$HOME/.local/bin"
curl -fsS http://localhost:8092/v1/cli -o "$HOME/.local/bin/vf"
chmod +x "$HOME/.local/bin/vf"
export PATH="$HOME/.local/bin:$PATH"
```

Examples with `jq` use it to select fields from JSON. The `$` at the start of a line in a recorded example is the shell prompt. Type the command after it.

NETWORK ACCESS AND THE MICROPHONE

The server binds to loopback by default. Keep it on a trusted computer or private network. The Console executes Python with the server process's permissions. Anyone

allowed to use it should have that level of trust. The browser can use a microphone from `localhost` or HTTPS. For an authorized lab computer on a private network, see the HTTPS setup and `VF_TLS_HOSTS` in the installation notes.

CHAPTER 2

RUN YOUR FIRST EXPERIMENT

Start with a calculation that you can compare across the CPU and GPU. Then test an estimate whose exact answer is known. These are two habits worth keeping when you bring your own question to the lab. Measure the computation, and find a separate way to check the result.

THE FIRST SESSION

Open `http://localhost:8092` in a browser. Click *Console* in the rail on the left. Type this program and press *Run* (or `Ctrl`+`Enter`):

CONSOLE

```
def matmul(xp, n):
    a = xp.ones((n, n), dtype=xp.float32)  ❶
    return a @ a

vf.race(matmul, 2048)  ❷
```

OUTPUT · QUADRO K1200

6.9× faster

CPU	130.8 ms
Card	18.84 ms

done in 0.93 s

❶ xp is np (CPU) or cp (the card)

❷ the same function on the CPU and on the card

`vf.race` runs your function two times: one time with NumPy on the CPU, and one time with CuPy on the card. A 2048 by 2048 matrix product is about 17 billion floating-point operations. The card did it in less than 20 milliseconds, which is close to one trillion operations each second. The sample output came from a Quadro K1200 and a host with a 12-core CPU and a tuned BLAS library. Try the same program with

128 in place of 2048 and see how the gap changes. Kernel launches and data transfers matter more when there is little arithmetic to do.

Now ask a question that has a known answer. What fraction of random points in a square falls inside the circle that touches its sides? The answer is π/4. Throw two hundred million points on the card and see how close the estimate comes:

CONSOLE

```
n, inside = 0, 0
for _ in range(20):  (1)
    x = cp.random.random(10_000_000, dtype=cp.float32)
    y = cp.random.random(10_000_000, dtype=cp.float32)
    inside += int(((x * x + y * y) ⩽ 1.0).sum())
    n += x.size
est = 4 * inside / n
print("estimate of pi:", est)
print("error:         ", abs(est - math.pi))
print("expected error from chance alone: about", round(4 * math.sqrt((math.pi
/ 4) * (1 - math.pi / 4) / n), 6))
```

OUTPUT · QUADRO K1200

```
estimate of pi: 3.14148692
error:          0.00010573358979293346
expected error from chance alone: about 0.000116
done in 0.62 s
```

(1) 20 batches of 10 million points fit on a 1 GB card

A Monte Carlo estimate varies from run to run. The final line gives its standard error, about 0.00012 here. Differences of that order are expected. If much larger differences recur, check that the random points are uniform, the circle test is correct and the count includes every batch.

For the third experiment, leave the browser. Open a terminal and ask the server how many people must be in a room before two of them probably share a birthday:

SHELL

```
$ vf run forecast.birthday
```

OUTPUT · QUADRO K1200

```
23 people
With only 23 people in a room, it is more probable than not that two have the
same birthday.
  tone: good
  paths: 23700000
```

```
    ms: 214.1
    (also: chart, facts — use --json or --out to see them)
  [forecast.birthday: 214.4 ms wall · 214.2 ms on the GPU in 1 runs]
```

The `vf` client sent one request. The server simulated rooms on the card and answered with a headline, a sentence and the numbers behind them. The browser, the API, the command line and an AI agent all reach the same models.

WHAT HAPPENED

Three things happened in this session, and they are the core of VectorFall.

The card did the arithmetic. CuPy is NumPy for the GPU. Each operation on a `cp` array starts a *kernel* that runs on every element in parallel, on hundreds or thousands of cores. Python itself still runs on the CPU. It only tells the card what to do.

The answer came with a check. The π estimate came with the error that chance alone gives. The birthday answer is a simulation that you can compare with the exact formula. The planning and forecasting examples use the same idea with another check, such as a greedy plan or last week's sales.

Every operation is a model. The birthday simulation is the model `forecast.birthday`. VectorFall 0.1 has more than two hundred models. Each one has a name, a title, typed parameters with limits, and a note that says if it runs on the GPU, on the CPU or on both. You can call each model in five ways:

WAY IN	HOW	WHO USES IT
The web interface	`http://localhost:8092`	you, with a mouse
The `vf` client	`vf run sim.budget --budget 200000`	you, in a terminal or a script
REST and OpenAPI 3.1	`POST /v1/models/sim.budget`	programs in any language
JSON-RPC 2.0	`POST /v1/rpc` with `"method": "sim.budget"`	programs that send batches
MCP	`http://localhost:8092/mcp`, tool `sim_budget`	AI agents

One function checks the parameters for all five. A bad value gets the same message everywhere.

HOW THE SERVER USES THE CARD

VectorFall is one server process in one container. It owns one card. A single *GPU worker* thread runs all the work for the card, one job at a time, from a queue. The queue keeps two registered jobs from competing for memory. The worker writes each job in the *Log*, and the top bar of the web interface shows the card's activity, memory, temperature and the speed of the last job.

Long work runs as a *job*. A job reports its progress, and you can cancel it. The interface stays responsive while the card works.

The Console is different. Your code runs in its own Python process, with its own access to the card. If your code loops forever, the *Stop* button ends that process and the rest of VectorFall continues.

> **NOTE** A card that also drives a display has a watchdog. A single long kernel can time out. VectorFall splits long work into short kernels. When you write a kernel, keep individual launches short, and test small workloads first.

CHAPTER 3

FIND THE RIGHT WORKSPACE

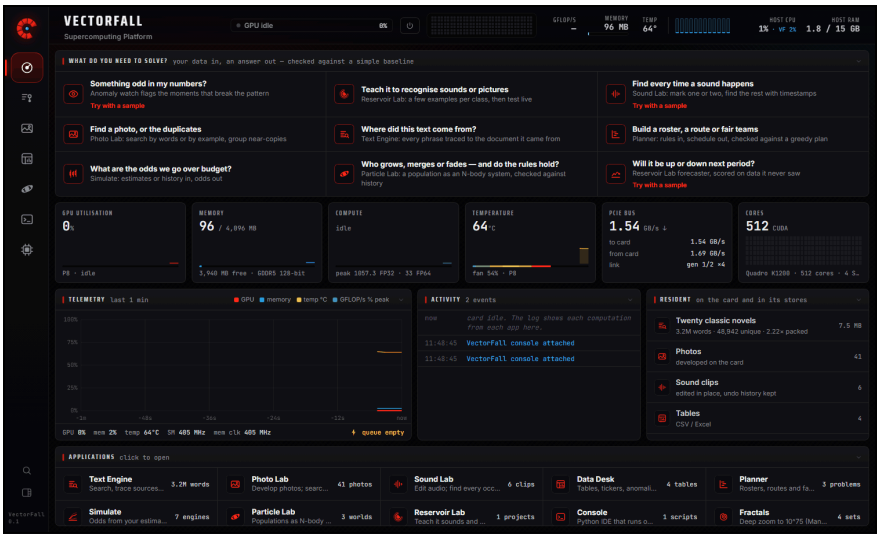
Home shows questions you can try with sample data. Start there when you are exploring. Once the shape of a question is clear, the rail takes you directly to a workspace. When you need the same operation in a program, Models gives you its typed parameters and copies the appropriate call.

THE FRAME: TOP BAR, RAIL AND COMMAND BAR

Every screen has the same frame. The rail on the left opens the applications. The top bar shows the card at all times: a strip of lights for the cores, the GFLOP/s of the last job, the memory in use, the temperature, and the use of the host CPU and memory. The *job pill* in the middle shows the job that runs now, with its progress and a button to cancel it. The power button next to it is *Reset card*: it stops everything on the GPU and starts the service again.

Press `Ctrl` + `K` anywhere to open the command bar. Type a few letters of an application, a table, a clip or a command, and press `Enter`. The rail can also sit at the bottom as a dock, and a *Desktop* mode puts the Log, a command tool, the Machine view and recent items in a track beside the application.

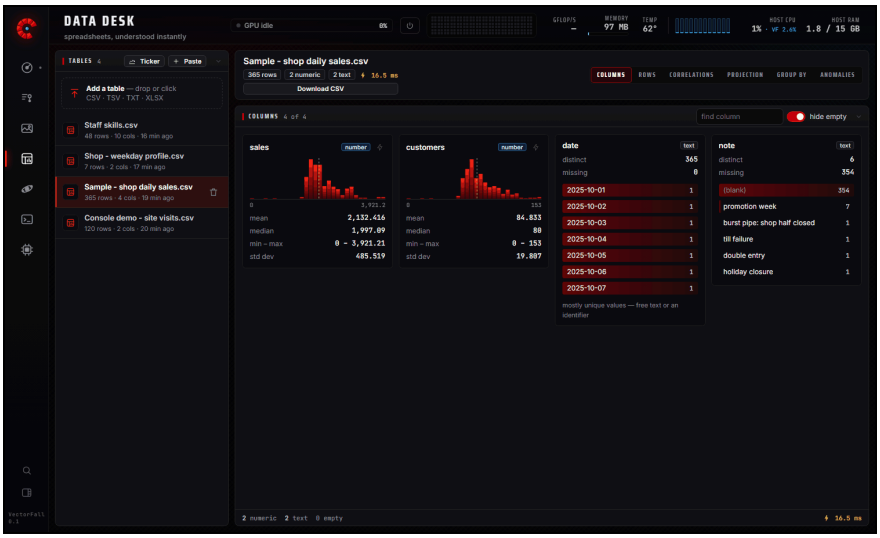
HOME



Home: everyday problems at the top, then the gauges of the card, its telemetry and activity, and the applications.

The top of Home lists eight everyday problems, such as *Something odd in my numbers?* and *What are the odds we go over budget?*. Each one opens the tool that answers it, and the ones that need data have *Try with a sample*. Below are gauges for the card with short histories, a telemetry trace of the last minutes, an activity feed, the list of what is resident on the card and in its stores, and the applications with their live status.

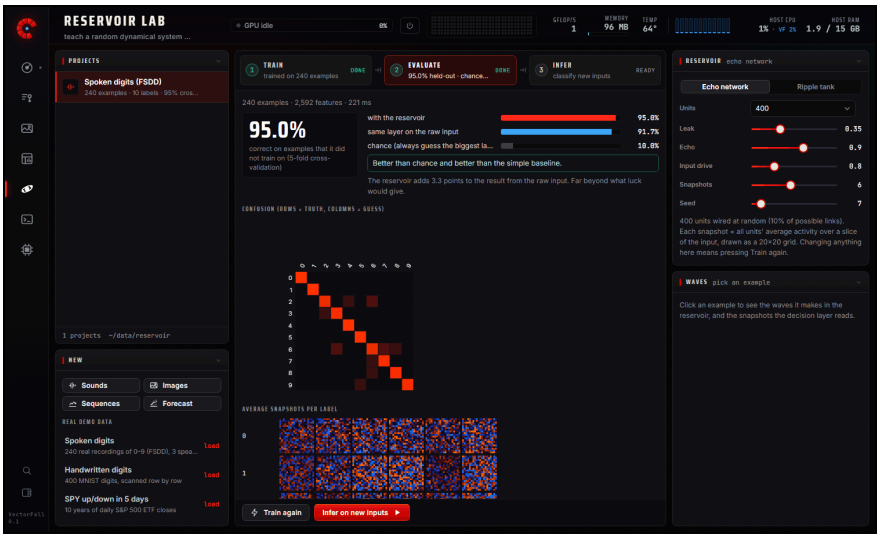
DATA DESK



The Data Desk with the sample shop table. Each column has a summary card.

The Data Desk holds your tables. Add a CSV, TSV or Excel file, paste numbers, import a ticker, or extract the tables of a web page. The tabs show the columns, the rows (filter and sort), correlations, a projection with clusters, group-by totals and *Anomaly watch*. Every other tool can read these tables.

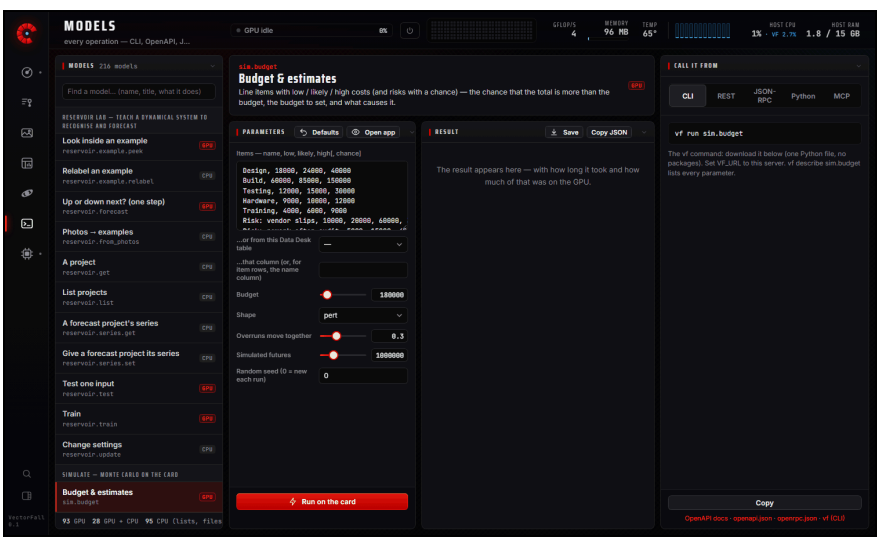
RESERVOIR LAB



The Reservoir Lab with the spoken digits: 95 percent cross-validated, against chance and the raw input, with the confusion matrix and the examples.

The Reservoir Lab teaches a dynamical system to recognise sounds, images and sequences from a few examples, and to forecast up or down. Every score is cross-validated and shown against chance and the raw input.

MODELS



Models: every operation as a form, and as CLI, REST, JSON-RPC, Python and MCP calls.

The Models screen lists every model with a form for its parameters. For the model that you choose, it shows the same call as a `vf` command, a REST request, a JSON-RPC request, Python code and an MCP tool call, ready to copy. It is the best place to learn the API.

PICK THE STARTING POINT FOR YOUR DATA

WHAT YOU HAVE	WHERE TO START	WHAT TO CHECK
People, constraints or appointments	Planner	The proposed plan and a simpler baseline
Inputs that could turn out several ways	Simulate or Forecast	The range and the chance of crossing a threshold
A recording and examples of an event	Sound Lab	Which events were missed and which flags were false
A passage, corpus or reference question	Text Engine or Language	The exact match, the source passage or the reason it was retrieved
Photos to develop and choose	Photo Lab	Clipping, the edit at full size and which files will export
A population, state or rule to explore	Particle Lab, Patterns or Fractals	Which starting conditions and numeric precision you used
Your own analysis or a result to save	Console and Data Desk	Repeat the computation with a small input whose result you can check

Keep named input and result tables distinct. That makes the path from one lab to the next easier to repeat. The Planner can open a Data Desk table and save a plan back there. Console can load both for a second check, then write the calculation as a new table or chart.

CHAPTER 4

MAP YOUR QUESTION TO A MODEL

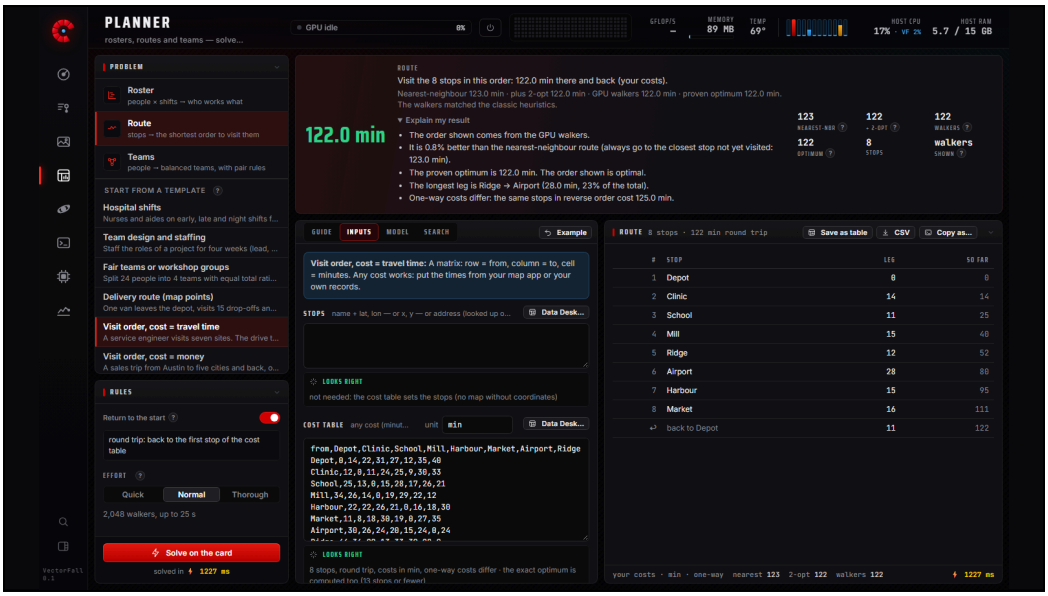
A service engineer must visit four sites. The order can change the day's drive time, and the time from A to B can differ from the time from B to A. Choose the objective before entering numbers. Minutes on the road, tolls and the chance of being late belong in different models unless you decide how to compare them.

START WITH THE COST TEMPLATE

A clinic manager has three questions for next week:

1. Who works the early, late and night shifts, with the right skills and eleven hours of rest between shifts?
2. A service engineer visits seven sites. The drive times differ by direction. What order takes the least time?
3. Twelve new staff cost money to train. What is the chance that the training budget is not enough?

Open the **Planner**. Under *Start from a template* on the left, the list names everyday problems: *Hospital shifts*, *Team design and staffing*, *Fair teams*, *Delivery route*, and three *Visit order* problems in which the cost is time, money or risk. Click **Visit order, cost = travel time**. The cost table on the *Inputs* tab fills with an example. Each row is a departure, each column is a destination, and each entry is the travel time in minutes. Under each box, a check reads the text while you type and says what it understood, or which line is wrong. Press **Solve on the card**.



The Planner with the Visit order, cost = travel time template, solved. Left: the templates. Middle: the cost table, checked by the server as you type. Top: the answer, the baseline and Explain my result.

The answer comes with *Explain my result*: which method produced the plan that you see, how it compares with the simple method, and which limits bind (for a route, changes in leg cost and whether the return trip is included). The *Copy as…* button gives the same problem as Console code, as an MCP call for an AI agent, as `curl`, or as a `vf` command.

The same templates are a model, so the command line sees them too:

SHELL

```
$ vf run problem.templates --json | jq -r '.result.templates[] | "\(.id)\t\
(.model)\t\(.title)"'
```

OUTPUT · QUADRO K1200

```
hospital-shifts    plan.roster Hospital shifts
team-staffing   plan.roster Team design and staffing
fair-teams  plan.teams  Fair teams or workshop groups
delivery-route  plan.route  Delivery route (map points)
visit-time  plan.route  Visit order, cost = travel time
visit-money plan.route  Visit order, cost = money
visit-risk  plan.route  Visit order, cost = risk
job-changeover  plan.route  Job order, cost = changeover time
project-budget  sim.budget  Project budget risk
project-deadline    forecast.project    Project deadline
```

```
call-centre forecast.queue  Queue or call-centre staffing
sales-history   sim.history Odds from your own history
```

Each template names the model that solves it. `problem.template` shows one in full: the problem in words, how a problem of your own maps onto the text, and the filled-in inputs:

SHELL

```
$ vf run problem.template --id team-staffing --json | jq -r '.result |
.problem, "", .maps'
```

OUTPUT · QUADRO K1200

```
Staff the roles of a project for four weeks (lead, backend, frontend, QA, data,
design) from people with different skills.

A role in a week is a 'shift': shift = the role, day = the week, needed = how
many people the role needs, skill = what the role requires. max_shifts = how many
weeks a person can give; unavailable = weeks away (holiday, other project);
prefers = roles they want.
```

THE SHAPES OF QUESTIONS

Most practical questions have one of a small number of shapes. When you see the shape, you know the family of algorithms, and VectorFall has a tool, and usually a template, for each family.

IF YOUR QUESTION SOUNDS LIKE THIS	THE SHAPE	USE	TEMPLATE
"Who works which shift?"	scheduling with rules	Planner (`plan.roster`)	`hospital-shifts`
"Who fills which role each week?"	assignment of people to slots	Planner (`plan.roster`)	`team-staffing`
"Split these people fairly"	assignment with constraints	Planner (`plan.teams`)	`fair-teams`
"What order of visits is shortest?"	travelling salesman	Planner (`plan.route`)	`delivery-route`
"What order costs the least time, money or risk?"	travelling salesman on a cost table	Planner (`plan.route --costs`)	`visit-time`, `visit-money`, `visit-risk`
"In what order do we run the jobs?"	sequencing with changeover costs	Planner (`plan.route --costs`, one way)	`job-changeover`
"What is the chance that it costs more than X?"	Monte Carlo over estimates	Simulate (`sim.budget`)	`project-budget`
"Will we finish by the date?"	Monte Carlo of a schedule	Simulate (`forecast.project`)	`project-deadline`
"How many people at the counter?"	a queue	Simulate (`forecast.queue`)	`call-centre`
"Given my history, what happens next?"	bootstrap of a series	Simulate (`sim.history`)	`sales-history`
"Will my series go up or down?"	a forecast scored on unseen data	Reservoir Lab (`reservoir.updown`)	`series-updown`
"Which kind of recording is this?"	classification from labelled examples	Reservoir Lab (`reservoir.classify`)	`sensor-classify`
"How will my group of companies, countries or teams change?"	an N-body model of a population	Particle Lab (`particles.group`)	`group-change`
"Is something odd in these numbers?"	anomaly detection on a series	Data Desk, *Anomaly watch* (`data.anomalies`)	
"Which of these things are alike?"	clustering, projection	Data Desk, *Projection* (`data.pca`)	

IF YOUR QUESTION SOUNDS LIKE THIS	THE SHAPE	USE	TEMPLATE
"Which values make all of these rules true?"	constraint satisfaction	Simulate, *Walk solver* (`sim.solve`)	
"Find every time this sound happens"	detection in a signal	Sound Lab, *Find sounds* (`sound.find`)	
"Where did this text come from?"	search, n-gram tracing	Text Engine (`text.trace`, `text.search`)	
"None of these"	your own algorithm	Console	

Some questions have two shapes. "Will we finish the project with this team?" is an assignment (which people) and a Monte Carlo (how long the tasks take). Do the two steps in order, and keep the output of the first step as a table so that the second step can read it. The Planner and Simulate can share the table through the Data Desk.

BRING YOUR OWN DATA

Every tool reads your data from the same places:

- **Tables** go into the Data Desk: CSV, TSV or Excel files, pasted numbers, a stock ticker, or tables from a web page. The Planner, Simulate and Anomaly watch read these tables, and their results can be saved back as new tables. Every solver parameter that holds a table has a twin that ends in `_table` (`--people_table "Staff skills"`), and `vf.solve` takes a table name in place of text.
- **Sound** goes into the Sound Lab: MP3, WAV, M4A, FLAC or OGG, up to five minutes for each clip.
- **Text** goes into the Text Engine: files, a ZIP of files, pasted text, a web page, or books from Project Gutenberg.
- **Photos** go into the Photo Lab, one at a time or many at a time.
- **Examples** for the Reservoir Lab come from your microphone, your camera, your photos or a table.

Each upload is also a model, so an agent or a script can do the same. For example, `vf run data.upload --file @staff.csv` adds a table, as the team exercise shows.

KEEP A BASELINE

Add up the travel times for the proposed route directly from the table, including the return if you asked for one. Compare the total with the nearest-next-stop rule. If they tie, the simple rule may be enough for these inputs. If they differ, look at the particular legs that changed. That can show you where a better estimate of travel time would matter.

A solution can be optimal for your table and still be impractical. If a visit has a fixed appointment or a street closes after a certain hour, check that condition separately unless the model explicitly represented it. Keep a copy of the input, the goal, and the check with the route you send out.

CHAPTER 5

FIND UNUSUAL DAYS IN A TABLE

You keep a table of daily sales for a small shop. A year of numbers goes up and down with the days of the week, the seasons and a promotion. Somewhere in the year there are days that do not fit: a till that failed, a day entered two times, a burst pipe. You want VectorFall to find those days, and to tell you how far you can trust the search.

THE SESSION

VectorFall has a sample table of this kind, so that every reader can follow along. On the Home screen, the problem *Something odd in my numbers?* has the link *Try with a sample*. The link calls the server, and you can do the same from a terminal:

SHELL

```
$ curl -s -X POST http://localhost:8092/api/samples/shop
$ vf run data.list
```

OUTPUT · QUADRO K1200

```
$ curl -s -X POST http://localhost:8092/api/samples/shop
{"kind":"table","id":"7952e7e969","name":"Sample - shop daily
sales.csv","new":false}

$ vf run data.list
2 items
  {"id": "7952e7e969", "name": "Sample - shop daily sales.csv", "rows": 365,
"numeric": ["sales", "customers"], "categorical": ["date", "note"], "added":
17954550
  {"id": "6f4c4b5400", "name": "Console demo - site visits.csv", "rows": 120,
"numeric": ["day", "visits"], "categorical": [], "added": 1795455031.8234148,
"colum
[data.list: 0.2 ms wall · 0.0 ms on the GPU in 0 runs]
```

The table has 365 rows: a date, the sales, the number of customers and a note. Ask Anomaly watch to look at the sales, and give it the note column, which marks the days that you already know about:

SHELL

```
$ vf run data.anomalies --table "Sample - shop daily sales" --cols sales --
label_col note
```

OUTPUT · QUADRO K1200

```
3 unusual moments
3 stretches of sales break the pattern (3 points, 0.82% of the data). The
strongest is at 2026-07-04: sales was 0, but the expected value was about 2688
(12.9x the typical error). 3 of your 5 known incidents were caught.
On planted glitches the pattern model (reservoir) caught 3 of 8 at a 1-in-100
alarm rate (the moving median found 0).
  echoes: 0
  detector: pattern
  best: pattern
  threshold: 5.0
  incident_count: 3
  flagged: 3
  n: 365
  names: ['sales']
  period: 7
  lags: 6
  notes: []
  table: Sample - shop daily sales.csv
  date_col: date
  ms: 0.0
  compute_s: 0.34
  (also: tests, incidents, known, chart — use --json or --out to see them)
[data.anomalies: 0.6 ms wall · 0.0 ms on the GPU in 0 runs]
```

Read the three lines at the top. The first is the *headline*. The second, the *sentence*, says what was found: the strongest break is 4 July, a day with zero sales when the pattern expected about 2,700. The third, the *check*, is the honest part. Before it looked at your real data, the tool put eight glitches of its own into a copy of the table: three spikes, three dips and two level shifts. It then counted how many each detector found, at a false-alarm rate of one in a hundred days. The pattern model found three of the eight. The moving median found none.

WHAT HAPPENED

Anomaly watch has three detectors:

- **Rolling median.** The expected value is the median of the days around it. This is simple and robust, but it does not know about weekly patterns.

- **Linear model.** The expected value comes from a regression on the recent days and on the other columns.
- **Pattern model.** A *reservoir*, a large random network of nodes that the card runs, learns the shape of the series. The expected value is its prediction.

For each day, the tool computes how far the real value is from the expected value, in units of the typical error (a robust z-score). Days beyond the threshold (default 5) are flagged. With `--detector auto`, the tool picks the detector that found the most planted glitches. That is why the answer names the pattern model as `best`.

The planted test answers the question "how good is this search on data like mine?". Three of eight is not excellent. It means that small glitches hide well in this series, because the weekly swing is large. A large glitch, such as a closed day, is easy. A day that is 15 percent low is hard. The tool found 3 of the 5 marked days. Look at the list to see which ones it missed:

SHELL

```
$ vf run data.anomalies --table "Sample - shop daily sales" --cols sales --
label_col note --json \
    | python3 -c "import json,sys; r=json.load(sys.stdin)['result']; [print(i)
for i in r['incidents']]"
```

OUTPUT · QUADRO K1200

```
{'start': 276, 'end': 276, 'peak': 276, 'col': 0, 'column': 'sales', 'score':
12.92, 'value': 0.0, 'expected': 2687.7236995301696, 'label': '2026-07-04',
'filled': False}
{'start': 164, 'end': 164, 'peak': 164, 'col': 0, 'column': 'sales', 'score':
8.72, 'value': 455.239990234375, 'expected': 2616.6360850517626, 'label': '2026-
03-14', 'filled': False}
{'start': 217, 'end': 217, 'peak': 217, 'col': 0, 'column': 'sales', 'score':
7.97, 'value': 3703.6298828125, 'expected': 1826.7020891246016, 'label': '2026-
05-06', 'filled': False}
```

The burst pipe of 20 January is not in the list. Its score was below the threshold of 5. Lower the threshold and look again:

SHELL

```
$ vf run data.anomalies --table "Sample - shop daily sales" --cols sales --
label_col note --threshold 3.5 --json     | python3 -c "import json,sys;
r=json.load(sys.stdin)['result']; print(r['headline']);
[print(i['label'], i['score'], round(i['value']), round(i['expected'])) for i
in r['incidents']]"
```

OUTPUT · QUADRO K1200

```
8 unusual moments
2026-07-04 12.92 0 2688
2026-03-14 8.72 455 2617
2026-05-06 7.97 3704 1827
2026-01-20 4.8 711 1815
2026-08-15 4.57 3921 3065
2025-12-18 4.18 2448 3411
2026-01-01 4.1 2014 2958
2026-08-10 3.68 2273 1633
```

Now the burst pipe is in fourth place, with a score of 4.8. Two of the new flags fall in the promotion week, and two are ordinary days in December and January. A lower threshold finds more of the real incidents, and it also flags more normal days. There is no free lunch: the threshold trades missed incidents against false alarms. The planted test tells you where your series sits on that trade.

> **NOTE** The note column marks five days, but only four of them are incidents. The fifth is the promotion week, a planned rise. The tool cannot know this. When you give it a label column, every row with a note counts as a known incident. Keep planned events in a different column.

YOUR OWN ANALYSIS IN THE CONSOLE

The tools cover the common questions. For the others, load the table in the Console. This program tests a forecast rule against a baseline. The rule is "sales are the same as on the same weekday last week". The baseline is "sales are the same as yesterday":

CONSOLE

```
t = vf.data.load("Sample - shop daily sales")
s = cp.asarray(t["sales"])
ok = cp.asarray([n == "" for n in t["note"]])  (1)
def mae(lag):
    err = cp.abs(s[lag:] - s[:-lag])
    keep = ok[lag:] & ok[:-lag]
    return float(err[keep].mean())
print("error of 'same as yesterday':          %6.1f" % mae(1))
print("error of 'same weekday last week':     %6.1f" % mae(7))
print("typical daily sales:                   %6.1f" % float(cp.median(s)))
```

OUTPUT · QUADRO K1200

```
error of 'same as yesterday':          360.2
error of 'same weekday last week':     168.7
typical daily sales:                  1997.1
```

```
done in 0.60 s
```

1. leave out the days with a note

The weekly rule has less than half the error of the daily rule. This tells you that the week is the main pattern in this shop, and that any forecast for it must know about weekdays. You found this with eight lines of code, and every operation ran on the card.

For your own table, compare the rules on the same dates and check that no future values were used to make a prediction. A baseline that follows weekly seasonality is a better check for this shop than a simple daily lag. Make that comparison before you fit a more complex model.

CHAPTER 6

BUILD TEAMS WITH RULES

You run a company of 48 people in six departments. Next month the company holds a two-week project sprint, and you must form six teams of eight. Each team must have a fair share of experience and a mix of departments. Two people must not be on the same team, and two others asked to work together. After the sprint, a four-week project needs three Python developers and two people for each other skill every week. A spreadsheet and an afternoon give you a plan. VectorFall gives you a plan, a comparison with the usual method, and a measure of how good the plan is against chance.

THE SESSION

Download the fictional staff table. It has a name, a department, the years of experience, a score from 1 to 5 for six skills, and the last performance rating. (The people are invented.) In the web interface, drop the file on the **Data Desk**. From the command line, upload it:

SHELL

```
$ head -4 staff.csv
$ vf run data.upload --file @staff.csv --filename "Staff skills.csv"
```

OUTPUT · QUADRO K1200

```
$ head -4 staff.csv
name,dept,years,python,stats,sales,design,writing,ops,rating
Avery,Engineering,8,3,3,1,2,3,2,7.4
Blake,Data,16,4,5,1,2,3,3,9.6
Casey,Sales,4,1,2,4,1,3,1,5.0
$ vf run data.upload --file @staff.csv --filename "Staff skills.csv"
  id: 3b305d5126
  name: Staff skills.csv
  rows: 48
  numeric: ['years', 'python', 'stats', 'sales', 'design', 'writing', 'ops',
'rating']
  categorical: ['name', 'dept']
  added: 1795476392.932527
```

```
  notes: []
  (also: columns — use --json or --out to see them)
[data.upload: 7.4 ms wall · 0.0 ms on the GPU in 0 runs]
```

Now open the **Planner** and choose **Teams**. On the *Inputs* tab, press **Data Desk…** above the *People* box and pick *Staff skills*. The box fills with the table, and the check under it says what the Planner will use: the first number column, `years`, is the score to balance, and the first text column with a few values, `dept`, is the category to mix. If you want another column, pick it in the role selector under the box. Type the two rules in the *Pair rules* box, one on each line:

```
Gray - Blake
Indy + Casey
```

Set *Teams* to 6 and press **Solve on the card**. The same problem from the command line:

SHELL

```
$ vf run plan.teams --people_table "Staff skills" --k 6 --pairs $'Gray -
Blake\nIndy + Casey' --json \
    | jq -r '.result | .sentence, .compare_text, (.explain[] | "  - " + .)'
```

OUTPUT · QUADRO K1200

```
6 teams of 8. Team totals of years differ by 7 at most. dept is spread as evenly
as the numbers allow. 2 of 2 pair rules hold.
Snake draft: total spread 3, mix error 28, 1 broken · GPU walkers: spread 7, mix
error 10, 0 broken.
  - The teams shown come from the GPU walkers.
  - Balance of years: the team totals differ by 7 (snake draft 3). Totals: 73,
77, 80, 74, 77, 77.
  - Mix of dept: error 10 (snake draft 28; 0 means each team has its exact share
of each value).
  - Pair rules: 2 of 2 hold (snake draft 1 of 2).
```

Read *Explain my result*. The Planner made two plans. The first is a *snake draft*, the method of school sports: sort the people by score, then give them out 1, 2, 3, 4, 5, 6, 6, 5, 4, 3, 2, 1. The second comes from thousands of GPU walkers that search for a plan with equal totals, a good mix and no broken rules. The snake draft balances the totals well, but it ignores the departments, so its *mix error* is large, and it can break a pair rule. The walkers trade a little balance for a much better mix, and they keep both rules.

WHAT HAPPENED

The Planner compiles your problem into a model for the *walk solver*. Each walker is one complete plan. On each step, every walker tries a small change, such as a swap of two people, and keeps it with a probability that depends on how much the change improves the score. This is *simulated annealing*, with thousands of walkers in parallel on the card. The score has three parts:

- **Balance:** the spread between the largest and the smallest team total (weight `--balance`).
- **Mix:** how far the count of each category in each team is from an even share (weight `--mix`).
- **Rules:** a large penalty for each broken rule, so a valid plan always wins.

You can change the weights. With `--balance 3 --mix 0.5` the result is closer to the snake draft. The `--effort` option sets how long the walkers search. The *Model* tab shows the generated model, and you can edit it.

SAVE THE PLAN

In the web interface, the Planner has a *Save as table* button, and `plan.save` does the same for a script or an agent. From the Data Desk, Simulate can read the teams, and you can export a CSV for the company calendar.

CHAPTER 7

ESTIMATE OUTCOMES UNDER UNCERTAINTY

A caf? refit can run over budget in several ways. The electrician may charge more than expected, a repair may take longer, or two costs may rise together. You can write an estimate for each item and use the card to see what those estimates imply for the whole project. A queue raises a related question about timing.

THE SESSION

You plan to refit the café. You have a low, a likely and a high cost for each item, and two risks that may or may not happen. Open **Simulate**. Under *Start from a template*, click **Project budget risk**. The items box fills with an example, and the note above it says how a project maps onto it: one line for each item, `name, low, likely, high`, and a fifth value for a risk, its chance. Replace the lines with your own items (or press **Data Desk…** to read them from a table with low, likely and high columns), set the budget, and press **Run Monte Carlo**. Download the caf? refit example for the command-line version.

SHELL

```
$ cat refit.txt
$ vf run sim.budget --budget 50000 --sims 1000000 --seed 1 --items @refit.txt
```

OUTPUT · QUADRO K1200

```
$ cat refit.txt
Counter and shelves, 9000, 12000, 18000
Coffee machine, 8000, 9000, 11000
Electrical work, 4000, 6000, 12000
Plumbing, 3000, 5000, 9000
Furniture, 5000, 7000, 10000
Painting and signs, 2000, 3000, 5000
Risk: permit delay, 3000, 5000, 9000, 0.3
Risk: old wiring must go, 4000, 6000, 10000, 0.2

$ vf run sim.budget --budget 50000 --sims 1000000 --seed 1 --items @re ...
27%
There is a 27% chance the total goes over 50,000. To be 80% sure, budget 51,508
```

```
(+15% on the sum of the likely values, 44,700); for 90%, 54,483. The plain sum of
likely values is enough only 38% of the time. Most of the uncertainty comes from
Risk: old wiring must go, Risk: permit delay, Counter and shelves.
  tone: warn
  paths: 1000000
  ms: 529.9
  (also: chart, chart2, facts — use --json or --out to see them)
[sim.budget: 530.0 ms wall · 182.2 ms on the GPU in 1 runs]
```

The answer is the chance that the total goes over 50,000, the budget for 80 and 90 percent confidence, and how often the plain sum of the likely values would have been enough. That last number is the baseline: the plan that most people make. The model samples each item from a PERT distribution between the low and the high value. The `--correlation` option (default 0.3) makes overruns move together, as they do in real projects. Ask for the facts table with `--json` to see the P10, P50 and P90 values.



Simulate with the Queue or call-centre staffing template: the note says how a queue maps onto the fields.

HOW MANY PEOPLE AT THE COUNTER?

A queue is the same problem in a shop, a call centre, a clinic or a help desk: people arrive at random, and each service takes a random time. The template **Queue or call-centre staffing** has 120 calls an hour, 4 minutes for each call and 9 agents. `problem.solve` runs the template, and for a queue it also runs one agent less and one more, because that comparison answers "how many?":

SHELL

```
$ vf run problem.solve --id call-centre --json | jq -r '.result | .sentence,
(.explain[] | "  - " + .)'
```

OUTPUT · QUADRO K1200

```
On average people wait 2.3 minutes; 1 in 10 waits more than 7.6. The counters are
busy 89% of the time.
  - The work is 8 servers' worth (arrivals x service / 60), so at least 9 servers
are needed to keep up; with fewer the line grows all day.
  - With 8 servers: On average people wait 10.3 minutes; 1 in 10 waits more than
25.9. The counters are busy 100% of the time — that is more than the capacity, so
the line continues to get longer.
  - With 10 servers: On average people wait 0.8 minutes; 1 in 10 waits more than
2.7. The counters are busy 80% of the time.
```

The work is eight agents' worth: 120 calls of 4 minutes is 480 minutes of work in each hour of 60 minutes. With 8 agents, average demand matches service capacity. A stable steady-state average wait requires spare capacity. With 9 the average wait is about two minutes, and with 10 it is less than one. Arithmetic gives the capacity threshold. The simulation estimates what one more agent buys under the modeled arrivals and service times.

THE SIMULATE ENGINES

ENGINE	MODEL	USE IT FOR
Budget and estimates	`sim.budget`	line items with low, likely, high and risks
From history	`sim.history`	the odds for the next periods, from your own past, with a backtest
Project a series	`sim.series`	a cone of futures for one series
Stock or financial	`sim.finance`	price paths, value at risk, drawdowns, the chance to hit a target
Count scenarios	`sim.counts`	a count with explicit inflow and outflow assumptions
Declarative scenario	`sim.formula`	your own equations, as JSON
State machine	`sim.states`	people or machines that move between states with given chances
Custom projection	`sim.custom`	a Markov model of a series, or a chain of states
Walk solver	`sim.solve`	rules and an objective, with no randomness in the problem

The planning presets are `forecast.retirement`, `forecast.project` (when will the project finish?) and `forecast.queue` (how long is the wait?). The templates (`vf run problem.templates`) fill them with an example.

FIND THE ASSUMPTION WORTH IMPROVING

Keep the seed, your cost ranges and your reason for choosing each range. Then run two versions in which one assumption changes. When the chance of an overrun moves because two costs are related, better information about that relationship will help more than another million trials. More runs reduce sampling noise. They cannot repair a cost that was left out of the model.

CHAPTER 8

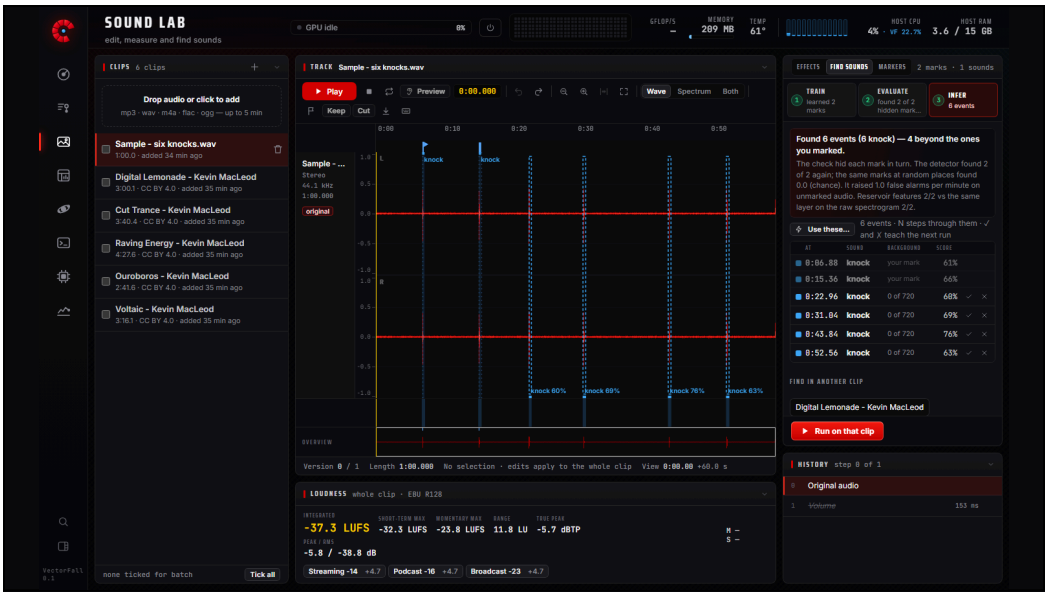
TEACH THE LAB A SOUND TO FIND

Suppose you have a recording of a room and want to mark every knock. In the waveform, a knock looks like a brief jump in level. In a spectrogram it spans a band of frequencies. A door or a dropped object can look similar, so use the detector to find candidates and listen to them before you count them.

THE SESSION: FIND EVERY KNOCK

VectorFall has a test recording for this problem: one minute of quiet room noise with six knocks in it. On the Home page, choose *Try with a sample* and then *Knocks*. Then open the Sound Lab and choose *Sample - six knocks* in the *Clips* list. You can see the knocks as thin lines in the waveform, but say that you have listened only to the start, and you heard the first two at 7.0 s and 15.5 s.

1. Choose the *Find sounds* tab on the right.
2. Drag across the first knock on the track, type `knock` in the name box, and press *Mark*. Do the same for the second knock.
3. Press *Learn and find*.



Find sounds after Learn and find: the two marks (top of the track), the four knocks that it found (dashed boxes), the detector lane under the track, and each event with its background rate.

The card listens to the whole minute in a few seconds. The *Evaluate* step shows the check, and the *Infer* step lists every event. Each event has a *background* column: how many windows of unmarked audio score as high as this event does. Here every knock reads "0 of 720", which is the strongest evidence that the test can give. Press `N` to step from event to event, and listen. The ✓ and ✗ buttons teach the next run: ✓ makes the event a mark, ✗ makes it a "not this" example. *Use these…* saves the events as a Data Desk table, makes them markers, mutes them, or sends them to the Reservoir Lab as training examples.

The same in the shell

The `vf` client calls the same models. The clip name can be any part of the name:

SHELL

```
$ vf run sound.find --sound "six knocks" --json | jq -r '.result | .lomo |
"found \(.found) of \(.tested) hidden marks; chance \(.chance)"'
```

OUTPUT · QUADRO K1200

```
found 2 of 2 hidden marks; chance 0.0
```

WHAT HAPPENED

The sound finder does three things on the card:

1. It turns the audio into a *log-mel spectrogram*: short overlapping windows, a Fourier transform of each, and the energy in bands that follow the human ear. A sound becomes a picture of frequency against time.
2. It runs the spectrogram through a *reservoir*: a large, fixed, random network of nodes with feedback. The state of the reservoir at each moment remembers the recent past, so a knock becomes a pattern in time and not only a column of numbers.
3. It fits a *ridge regression* from the reservoir state to your labels, and scores every window of the clip.

The check is the part that you can trust. With only two marks, a model can learn them by heart. Thus the finder hides each mark in turn, trains on the others, and tries to find the hidden one. Then it does the same with your marks moved to random places in the recording. A detector that learned from random places learned nothing, so its score is the level of chance. The knocks: 2 of 2 hidden marks found again, against 0 for the random marks. The finder also compares the reservoir with the raw spectrogram, so you can see if the reservoir helped, and it counts the false alarms per minute on audio that you did not mark.

The background rate of each event uses the same idea. The finder scores each unmarked window with a detector that did not train on that window. An event that scores higher than all 720 windows is unusual in this recording. Repeat the check on other clips before relying on the detector.

Random folds would give a better score, because neighbouring windows overlap and leak the answer. The finder does not use them.

> TIP If the finder calls a look-alike a knock (a cough, a door), select it and press *Mark as "not this"*. The label `nothing` makes it an example of what is not the sound, and the next *Learn and find* uses it.

CHECK A SECOND RECORDING

Use Infer to run the detector on another clip. Listen to a few of the strongest events and several near the cutoff. Save the times and scores to Data Desk if someone else

needs to review them. After you change the recording conditions, compare false alarms per minute and missed knocks against events you marked by ear.

CHAPTER 9

TRACE TEXT TO ITS SOURCES

A colleague forwards you a sentence whose source they cannot remember. If it appears word for word in a known collection, the Text Engine can give you the document and the surrounding passage. Ranked retrieval offers another starting point when someone remembers the idea but has changed the wording.

THE SESSION

VectorFall comes with two corpora: twenty classic novels from Project Gutenberg, and a set of essays. Put the novels on the card:

SHELL

```
$ vf run text.corpus.list
$ vf run text.corpus.activate --corpus "Twenty classic novels"
```

OUTPUT · QUADRO K1200

```
$ vf run text.corpus.list
2 items
  {"id": "988f7c7514", "name": "Twenty classic novels", "bytes": 17575289,
"docs": 20, "version": 0, "created": 1795477284.2541506, "active": true, "index":
{"ver
  {"id": "913a12504d", "name": "Essays by Amir Husain (Forbes)", "bytes": 260990,
"docs": 35, "version": 0, "created": 1795477284.1266599, "active": false, "index
[text.corpus.list: 0.4 ms wall · 0.0 ms on the GPU in 0 runs]

$ vf run text.corpus.activate --corpus "Twenty classic novels"
  id: 988f7c7514
[text.corpus.activate: 0.0 ms wall · 0.0 ms on the GPU in 0 runs]
```

Now trace a suspect paragraph. Its first sentence comes word for word from *Pride and Prejudice*. The second was written for this example.

SHELL

```
$ vf run text.trace --text "It is a truth universally acknowledged, that a
single man in possession of a good \
fortune, must be in want of a wife. The essay argues that money shapes every
marriage in the story."
```

OUTPUT · QUADRO K1200

```
68% of your text appears word for word in the corpus, in 2 stretches of 6+ words
(the same words shuffled: 0%). The longest runs 16 words — from Pride and
Prejudice.
  coverage: 0.6765
  coverage_shuffled: 0.0
  words: 34
  spans: 2
  min_words: 6
  longest: 16
  longest_doc: Pride and Prejudice
  scans: 28
  unknown: 0
  ms: 123.1
  (also: segments, top_docs — use --json or --out to see them)
[text.trace: 125.5 ms wall · 36.1 ms on the GPU in 28 runs]
```

The tracer found the stretches that match the corpus word for word and named the document. It also ran a control: the same words in a shuffled order. Shuffled words almost never match long stretches, so a long match in the real text is not chance. Short matches of three or four words happen everywhere ("in the middle of the") and mean nothing. That is why the tracer counts a match only from six words, and why it reports the control.

WHAT HAPPENED

When a corpus goes on the card, the Text Engine replaces each word with a 2-byte code. Common words get the small codes, and rare words get an escape code and a side table. English packs about 2.2 times smaller this way, so a card with 1 GB holds well over a gigabyte of text. A search is then a scan of the code stream by thousands of threads:

SHELL

```
$ vf run text.stats
```

OUTPUT · QUADRO K1200

```
  bytes: 17575289
  tokens: 3950131
  words: 3183355
  vocab: 56721
  unique_words: 48942
  sentences: 202648
  paragraphs: 69841
  hapax: 17255
  type_token: 0.0154
  words_per_sentence: 15.7
  letters_per_word: 4.28
  syllables_per_word: 1.4
(7 more lines)
```

`text.search` finds every occurrence of a word or a phrase, `text.near` counts the words that come after (or before) it, and `text.retrieve` ranks passages by how well they answer a question (BM25). The full answer of each model is JSON; the examples in this chapter pipe it through `jq` to print only the parts that matter:

SHELL

```
$ vf run text.search --q "white whale" --json \
    | jq -r '.result | "\(.count) hits in \(.gpu_ms) ms", (.hits[:3][] | "...\
(.left[-36:])[\(.match)]\(.right[:30])..." | gsub("\n"; " "))'
$ vf run text.near --q "dear" --top 6 --json | jq -r '.result.items[] | "\
(.count)  \(.word)"'
```

OUTPUT · QUADRO K1200

```
$ vf run text.search --q "white whale" --json \
94 hits in 1.38 ms
...f something queer about that, eh? A [white whale]--did ye mark that, man? Look
...
...re now heard me give orders about a [white whale]. Look ye! d'ye see this
Spani...
...whosoever of ye raises me that same [white whale], he shall have this gold
ounc...
$ vf run text.near --q "dear" --top 6 --json | jq -r '.result.items[] ...
104  boy
91  fellow
63  sir
52  friend
46  count
40  Miss
```

INSPECT A RETRIEVED ANSWER

The *Language* screen has VectorLM. VectorLM is a prototype that answers a simple question from the corpus: it retrieves passages with BM25, picks the best sentence, and takes a short answer from it, with the source passage. It can also continue the answer with the small language model that it trains on the card.

> **WARNING** VectorLM is a prototype for studying retrieval. Check the source passage for every answer you use.

SHELL

```
$ vf run text.vectorlm.ask --prompt "Who is the captain of the Pequod?" --top
5
```

OUTPUT · QUADRO K1200

```
It belongs to me and Captain Bildad to see the Pequod fitted out for the voyage,
and supplied with all her needs, including crew.
  question: Who is the captain of the Pequod?
  kind: person
  found: True
  answer: Bildad
  span: Bildad
  confidence: 1.0
  doc: Moby Dick; Or, The Whale
  byte: 903784
  end: 904561
  sentence_byte: 904052
  model: VectorLM (retrieval + extraction, prototype)
  ms: 6.0
  (also: passage, passages, terms, warning — use --json or --out to see them)
(1 more lines)
```

Look at the source passage under the answer. When the answer is right, the passage shows why. When it is wrong, the passage shows what misled it: retrieval puts the right text in front of a model, and the model can still pick the wrong words from it. The Language screen also has a small graph store (`graph.*` models) for facts as nodes and edges, which an agent can use as a memory.

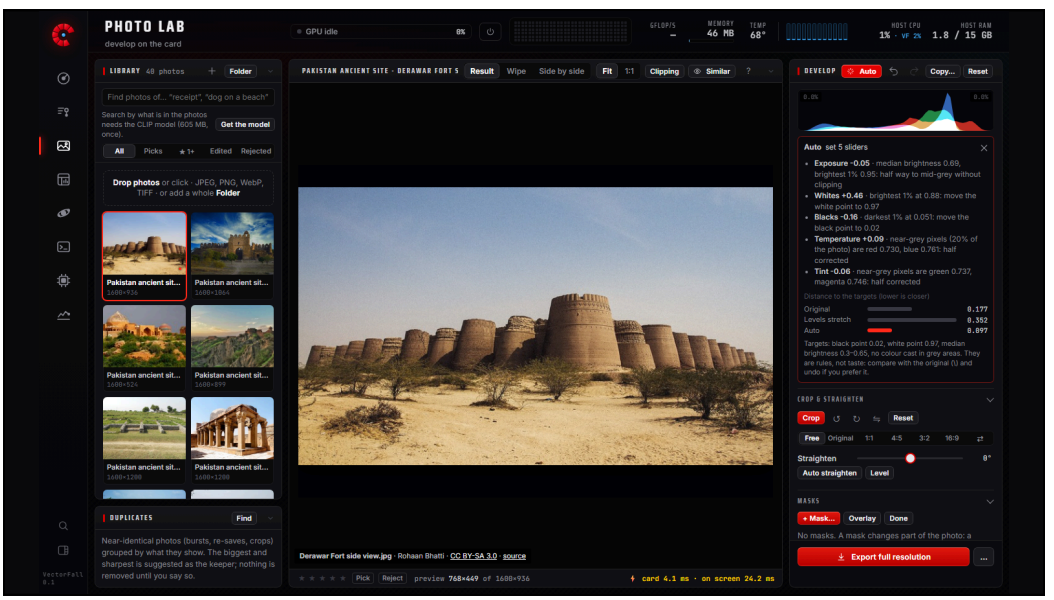
CHAPTER 10

DEVELOP AND COMPARE A PHOTOGRAPH

You have a folder of trip photos. Some are too dark, some are copies, and you want to develop and share a selection. The Auto panel shows which measurements prompted an adjustment. Inspect the particular frame at full size to see whether the change helps the picture.

THE SESSION: DEVELOP, CHOOSE, EXPORT

Open the Photo Lab and click a photo in the library. Press **Auto** (or Shift+A). The card measures the photo and moves ordinary sliders, and the panel tells you which sliders it moved and why:



The Photo Lab after Auto. The panel lists each slider that Auto set with its reason, and scores the result against the original and against a plain levels stretch.

Auto is not a black box. Each line is a measurement ("darkest 1% at 0.051: move the black point to 0.02"), and the bars compare three things on the same fixed targets: the

original, a plain *levels stretch* (the simple baseline that pulls each colour channel to full range), and Auto. Lower is closer. The targets are rules, not taste, so look:

- **Hold the mouse on the photo**, or press `\`, to see the original. Release to see the result.
- Press `W` for a **wipe**: drag the line across the photo. Press `Y` for side by side.
- Press `J` to paint the clipped pixels: red where a channel is pure white, blue where the pixel is pure black. The two numbers on the histogram give the same counts in percent.

Now make it yours. Click a look, for example *Film*, and move **Amount** to 60%. Drag a slider: while it moves, the card sends a half-size draft, and the full preview follows when you release it. Double-click a slider, or the name of a group such as *Light*, to reset it. Click a value to type it.

Every change is a named step in **History**, and nothing changes the original file. Press Ctrl+Z to undo and Ctrl+Shift+Z to redo, or click any step to go back to it. The edit is kept on the server, not in the browser, so it is the same in every browser, in the CLI and for an agent.

Choose the photos for the ZIP file with the keys that most photo editors use: `P` picks, `X` rejects, `U` removes the flag, and `1` to `5` give stars. After `P` or `X` the next photo opens. Filter the library with *Picks*, *Edited* or *Rejected*. A reject is only a mark: no photo is deleted. To give the same edit to several photos, use **Copy…**: paste everything, or only *Light*, *Colour* or *the look*, to this photo, to the picks or to the search results. Each photo gets a *Pasted* step, so Undo on that photo brings its own edit back. Last, the **…** button next to *Export* writes the picks, each with its own edit, at full resolution into a ZIP file.

HOW THE DEVELOP PIPELINE RUNS ON THE CARD

A preview is not an approximation of the export: both run the same six CUDA kernels. The first gathers the frame: quarter turns, mirror, straighten and crop in one bilinear pass. The next does all the light and white balance steps of a pixel, and the masks' local changes, in one pass; the others do clarity, colour, the separable blurs of noise reduction and sharpening, and the effects. A mask is not a separate image: each kernel computes the weight of every mask at the pixel from its few numbers (or samples the brush map) and adds the mask's sliders, weighted. Each run of per-pixel steps is one pass through the card's memory, which matters on a card with 29 GB/s. A

full-resolution export runs in strips sized to the free memory, each with enough overlap for the blurs, so a 24-megapixel photo fits on a 1 GB card. A 1:1 view is a padded tile of the frame. The tests check that a 1:1 tile and the same pixels of the full export are the same, also with a rotation, a crop and masks. Grain is a hash of the pixel position, so it does not change between renders and the strips join without a seam.

The status line under the photo shows two times: *card* is the GPU work, and *on screen* is the time from the request to the decoded picture. On a Quadro K1200, a draft takes 15 to 50 ms on screen, and a full preview of a 6000×3510 photo takes 16 to 65 ms. On the same card, a 2560 px preview of a 21-megapixel photo with a rotation, a crop, three masks and clarity costs 52 ms of card time, against 25 ms with no edit.

CHAPTER 11

WRITE AN ANALYSIS IN PYTHON

You have an idea that no tool in VectorFall covers yet. Perhaps you want to filter a signal, to test a formula on a million random inputs, or to draw a picture of a function. The Console is the place for this. It is a Python editor in the browser, and your code runs on the server with the card ready to use.

WHERE YOUR CODE RUNS

The Console starts each program in a new Python process, in the folder `~/scripts` on the server. These names are ready, and you do not import them:

NAME	WHAT IT IS
`np`	NumPy. It runs on the CPU.
`cp`	CuPy. The same functions as NumPy, but the arrays live on the card and the operations run there.
`vf`	The VectorFall helpers: `show`, `plot`, `table`, `race`, `timeit`, `data`, `text`, `machine`.
`math`, `time`	The standard Python modules.

Python itself runs on the CPU. Only operations on `cp` arrays, and kernels that you write, run on the card. A Python loop over a million elements is slow on any computer. A CuPy operation on a million elements is one kernel.

Ask the card about itself:

CONSOLE

```
p = cp.cuda.runtime.getDeviceProperties(0)
free, total = cp.cuda.runtime.memGetInfo()
print(p["name"].decode(), "compute capability", "%d.%d" % (p["major"],
p["minor"]))
print("multiprocessors:", p["multiProcessorCount"], " clock:", p["clockRate"]
// 1000, "MHz")
print("memory: %.0f MB free of %.0f MB" % (free / 2**20, total / 2**20))
print("kernel time limit (display watchdog):",
bool(p["kernelExecTimeoutEnabled"]))
```

OUTPUT · QUADRO K1200

```
Quadro K1200 compute capability 5.0
multiprocessors: 4  clock: 1032 MHz
memory: 3871 MB free of 4035 MB
kernel time limit (display watchdog): False
```

done in 0.17 s

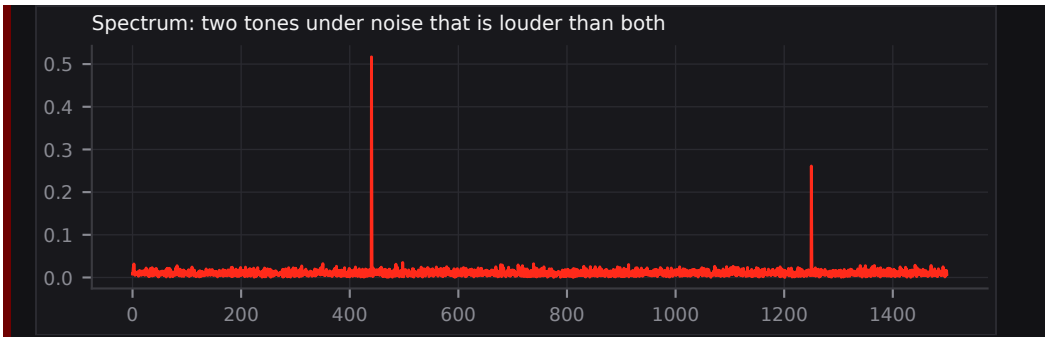
SHOWING RESULTS

`vf.plot` draws a line, bar or scatter chart. `vf.table` shows rows. Here is a noisy signal, its spectrum, and a table of the strongest frequencies:

CONSOLE

```
sr = 8000
t = cp.arange(sr * 2, dtype=cp.float32) / sr  (1)
sig = cp.sin(2 * math.pi * 440 * t) + 0.5 * cp.sin(2 * math.pi * 1250 * t) +
cp.random.normal(0, 1.5, t.size, dtype=cp.float32)
spec = cp.abs(cp.fft.rfft(sig)) / t.size
freq = cp.fft.rfftfreq(t.size, 1 / sr)
vf.plot(freq[:3000], spec[:3000], title="Spectrum: two tones under noise that
is louder than both")
top = cp.argsort(spec)[::-1][:4].get()
vf.table([{"Hz": float(freq[i]), "strength": round(float(spec[i]), 4)} for i
in top])
```

OUTPUT · QUADRO K1200



HZ	STRENGTH
440	0.5169
1250	0.2611
1980	0.0362
2114	0.0356

done in 0.75 s

1 two seconds

The noise is louder than the two tones, so you cannot see them in the waveform. The Fourier transform finds both at once. Try repeating the analysis on a shorter window. That localizes short events in time, but makes adjacent frequencies harder to tell apart. Sound Lab uses this relationship to visualize changes over time.

TABLES FROM THE DATA DESK, BOTH WAYS

`vf.data` connects your code to the tables of the Data Desk. Save a table from the Console, and every tool can use it:

CONSOLE

```
rs = np.random.RandomState(1)
days = np.arange(120)
visits = 500 + 3 * days + 60 * np.sin(days / 7 * 2 * np.pi) + rs.normal(0, 25,
days.size)
tid = vf.data.save("Console demo - site visits", {"day": days, "visits":
visits.round()})
print("saved table", tid)
t = vf.data.load("Console demo - site visits")
print(len(t["day"]), "rows; mean visits %.1f" % np.nanmean(t["visits"]))
```

OUTPUT · QUADRO K1200

```
saved table 6f4c4b5400
120 rows; mean visits 679.4
```

```
done in 0.21 s
```

Text columns come back as lists, and number columns come back as NumPy arrays. A blank cell becomes `nan`.

CHAPTER 12

CONNECT AN AGENT TO THE LAB

A program can ask the same question as the browser. VectorFall uses one model registry for the UI, CLI, REST, JSON-RPC and Model Context Protocol (MCP). Choose the interface that suits your client. Give it the input, units, and the result you want to check.

CALL A MODEL FROM YOUR PROGRAM

The Models screen shows each operation's parameters and examples. Find `forecast.birthday` and run its sample in the UI. Then request the same model from the shell:

```
curl -fsS -H 'Content-Type: application/json' -d '{}'
http://localhost:8092/v1/models/forecast.birthday
```

The service publishes an OpenAPI document at `/v1/openapi.json` and a JSON-RPC endpoint at `/v1/rpc`. When a model returns a job, keep its ID. If the client times out, look for that job before starting another one. The card may still be working. The top bar in the browser lists current jobs and lets you stop one.

CONNECT A PLANNING AGENT

For Claude Code, use a tool set with the Planner and Problem Templates groups:

```
claude mcp add --transport http vectorfall   'http://localhost:8092/mcp?
tools=plan,problem'
```

For OpenCode, add a remote MCP entry to its configuration:

```
{
  "$schema": "https://opencode.ai/config.json",
  "mcp": {
    "vectorfall": {
      "type": "remote",
```

```
      "url": "http://localhost:8092/mcp?tools=plan,problem",
      "enabled": true
    }
  }
}
```

Give the agent the directed travel-time table from the route example. Ask it to validate the table, solve the order with a return to the depot, and return the total in minutes and the baseline. Confirm the order against the input table. If a leg is absent or the units are unclear, fix the input and rerun.

LOAD ONLY THE TOOLS YOU NEED

ADDRESS SUFFIX	WHEN TO USE IT
`/mcp?tools=plan,problem`	The task is planning, and the agent may use a problem template.
`/mcp?tools=find`	A short keyword search should return matches with their schemas.
`/mcp?tools=discover`	The agent should first browse families, then select the operations it needs.
`/mcp`	The client needs the complete catalog and has enough context for it.

In discovery mode, `vf_groups` lists the tool families. Use `vf_tools` to browse one family or `vf_find` to search by words. Then use `vf_describe` for selected schemas and `vf_run` for execution. Reading a few definitions at a time preserves context. The extra calls add round trips. A fixed group gives a smaller client a shorter path when you already know the task.

Use `vf_start` and `vf_job` for background work. Tool sets scope a client's catalog. Trust and access control still depend on the client and the server's network configuration.

ASK FOR AN ANSWER YOU CAN CHECK

A local 27B model was asked to solve two routes, with five runs per condition per problem. It found the proven optimum in 0 of 10 runs on its own and in 10 of 10 with VectorFall over MCP. On the six-stop problem the median end-to-end time rose from 13 seconds to 332 seconds. VectorFall used 1.3 to 5.1 seconds of GPU time for that case. The path includes the local model processing the prompt, calling tools and reading the results. These figures describe the recorded setup with a fixed tool set.

The recorded runs show the route and the independent check. For your first use, give the agent a tiny case you can solve by hand. Ask it to name any input it could not interpret and retain the baseline in its report.

CHAPTER 13

COMPARE YOUR CARD WITH HISTORICAL SYSTEMS

Lineage compares a measurement from your GPU with historical supercomputer measurements. The useful number is the rate for a defined calculation, at a specified precision and problem size. Since 1993 the TOP500 has ranked machines using a double-precision LINPACK benchmark for solving dense systems of linear equations.

KEEP PRECISION AND RATE TOGETHER

A single-precision matrix product, an advertised theoretical peak and a double-precision solver describe different quantities. Start with the FP64 result in Lineage and check that the residual is small. If you share a measurement, include the matrix size, the card and the precision. Memory and communication capacity matter too, especially for problems too large for one GPU.

THE LINEAGE SCREEN

The *Lineage* screen of VectorFall runs a LINPACK test of its own on your card, in FP64 and FP32, with the residual check that the real benchmark uses. It then places the result on every TOP500 list since 1993: the last list where your card would have been number 1, its estimated rank on each later list, and the last list that it would still have entered. Models of the famous machines stand next to the result.

The same test and the same comparison are models, so the shell and your agents can use them too. A quick run takes a few seconds and is not saved; `--size auto` fills the free memory and saves the result for the card:

SHELL

```
$ vf run lineage.linpack --precision fp64 --size quick
$ vf run lineage.compare --json | jq -r '.result.headline'
```

OUTPUT · QUADRO K420

```
$ vf run lineage.linpack --precision fp64 --size quick
```

```
6.0 GFLOP/s FP64
FP64 n=3,488: 6.0 GFLOP/s in 4.7 s, residual 0.0018 (passed)
  (also: card, fp64, fp32, source, crays, runs — use --json or --out to see them)
[lineage.linpack: 5002.2 ms wall · 5000.7 ms on the GPU in 1 runs]

$ vf run lineage.compare --json | jq -r '.result.headline'
the fastest computer in 1989
```

The ranking shown in the app is an estimate based on the rate. For the historical records, see the TOP500 lists.

REFERENCE AND CREDITS

- VectorFall release and documentation
- Sample data and attribution
- Software licence
- Commercial licensing

VectorFall uses the PolyForm Noncommercial License 1.0.0. Commercial use requires a commercial licence. The application and guide use Inter, JetBrains Mono and Saira Condensed under the SIL Open Font License 1.1. The shop, staff, caf? and knock samples are invented. The novels are from Project Gutenberg. Photo Lab's sample images carry individual credits in the repository.

Copyright ? 2026 Amir Husain. All rights reserved.