

The Engineer's Guild

A guild was where masters trained apprentices, judged one another's work and kept the standards of a craft. This volume is about that last stage: depth in an area you chose, taste built from having been wrong, and the ability to make other engineers better. It covers reading papers and production code, open source and design review, honest trade-offs and estimates, leading projects and mentoring people, the business you work for, and the habits that make a career compound. Every chapter turns its advice into a small program you can run, so the claims are checkable.

CHAPTERS	CODE EXAMPLES	EXERCISES	STAGE BUDGET
16	16	48	Continuous

Contents

How to use this volume	3
Part I · Depth	5
Chapter 01 · What mastery is	6
Chapter 02 · Reading a paper	13
Chapter 03 · The canon	20
Chapter 04 · Reading production code	27
Part II · Working in the open	34
Chapter 05 · Open source, from patch to reviewer	35
Chapter 06 · Design documents	43
Chapter 07 · Trade-offs, argued both ways	51
Chapter 08 · Writing in public	58
Part III · Leading	65
Chapter 09 · Estimating honestly	66
Chapter 10 · Leading a project	73
Chapter 11 · Status, stakeholders and influence	80
Chapter 12 · Mentoring	87
Part IV · The long game	94
Chapter 13 · Learning the business	95
Chapter 14 · Measuring delivery	101
Chapter 15 · Finishing things	108
Chapter 16 · A career that compounds	115
The deliverable	122
The sixteen rules	124
Reading list	126

This stage has no end date. Every other stage in the syllabus can be finished; this one is practised for the rest of a career. The syllabus says it plainly: expertise isn't one more subject but depth in a chosen area, taste built from having been wrong, and the ability to move other engineers, and the technical half is the easy half. So this volume isn't a course to complete in a month. Read it through once, then return to the part that matches what you're doing: Part I when choosing and deepening an area, Part II when working in the open, Part III when leading, Part IV when planning the long run. The table shows where each of the stage's practices is taught.

STAGE 15 PRACTICE	CHAPTERS	READ ALONGSIDE
Pick one track and go three levels deeper than your team needs	Ch. 1	Will Larson's Staff Engineer guides
Read primary sources: GFS, MapReduce, Bigtable, Dynamo, Spanner, Raft	Ch. 2–3	The papers themselves; Papers We Love
Read production source code of tools you use daily, and fix something in one	Ch. 4	The Architecture of Open Source Applications
Contribute to open source until you're trusted with review	Ch. 5	Producing Open Source Software
Write design documents that survive sceptical senior engineers	Ch. 6	Design Docs at Google
Develop taste in trade-offs: argue both sides honestly	Ch. 7	How to Measure Anything
Write publicly	Ch. 8	Google's technical writing courses
Lead a project end to end: scoping, sequencing, risk, stakeholders, landing it	Ch. 9–11	The Staff Engineer's Path; The Mythical Man-Month
Mentor deliberately	Ch. 12	The Manager's Path, chapter 1
Learn the business: what it sells, what it costs, where engineering moves the number	Ch. 13	Your own company's numbers
Cultivate the boring virtues: reliability, clear communication, finishing things	Ch. 14–16	Accelerate; DORA's research

Every chapter has a program. Advice about careers and leadership is usually unfalsifiable, which is why so much of it is useless. Here, each chapter's central claim is turned into a small Python program that runs, using only the standard library: a decision tested for fragility, a paper's idea built small, a codebase mapped and traced, a patch reviewed, a schedule simulated, a status report computed from data, a team's flow measured. The programs use invented numbers, said so in the text, so read their outputs for the shape of the result rather than as measurements of your situation. Then change the numbers to yours. That's the point of them.

Do the exercises in your own work. Most of them ask you to apply the chapter to something real: your area, your codebase, your next design document, your current project. They're more useful done slowly, over months, than all at once. Several, such as the decision journal in chapter 1 and the work log in chapter 16, are habits to start now and keep.

The deliverable is a body of work, not a project: a maintained open-source contribution, a written archive others cite, and at least one system in production that other engineers depend on and that you didn't build alone. That takes years, and the appendix describes what each part looks like when it's done. Begin all three at once, small, and let them grow together.

THE ONE HABIT

Almost everything in this volume depends on one habit: **write it down.** The decision and what you expected, in a journal. The paper's argument, in a summary. The design, in a document. The trade-off, as ranges. The project's state, as a forecast. What you did this week, in a log.

Writing is how judgement becomes something you can check, share and improve. Memory flatters its owner and fades; a written record does neither. The engineers people trust most are usually the ones whose thinking they can read.

PART I

Depth

Four chapters on going deep in one area. First, what expertise is and how to choose where to build it, honestly, with an eye on how fragile the choice is. Then the two best ways into any area: its founding papers, read by building their core ideas small enough to run, and its production code, read with a map and a trace rather than from the first line down. By the end of this part you have an area, a reading list, and a method for each.

What mastery is

3 exercises

Expertise isn't one more subject to study. It's three things that grow together: depth in an area you chose, taste built from having been wrong, and the ability to make other engineers better. The first is the one you can plan. Pick the area deliberately, go several levels deeper than your team needs, and expect it to take years rather than months. This chapter is about choosing well, and about knowing how fragile your choice is.

Depth, taste and leverage

Every stage before this one taught breadth: enough of each layer to work across all of them. That's the foundation, and it's what makes an engineer reliable. What makes one exceptional is different. When the database is losing writes at three in the morning, or a design review needs someone who has seen this exact failure before, teams don't call the person who knows a little of everything. They call the person who knows one thing all the way down.

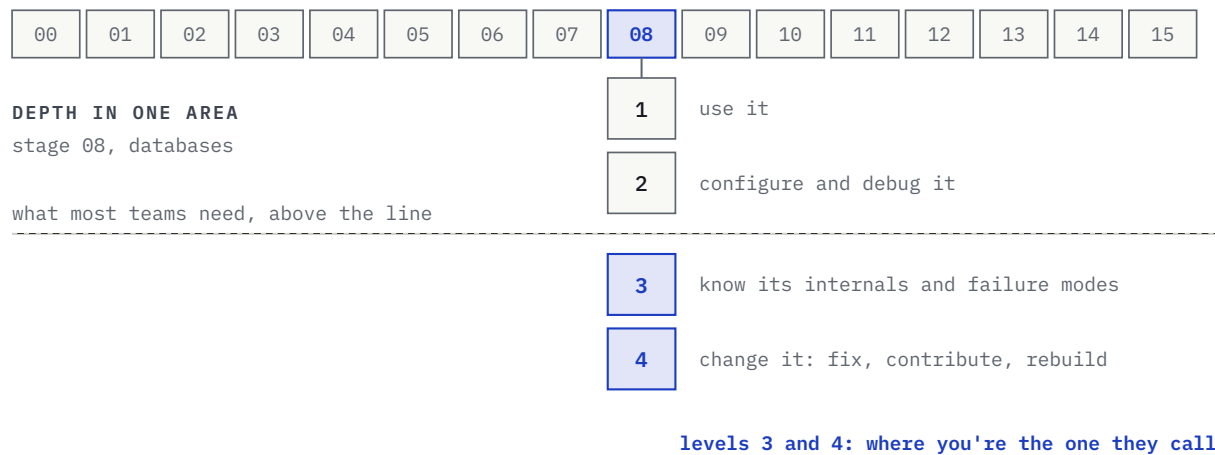
That depth is the first part of expertise. The second is **taste**: a sense of which designs will age well and which will hurt, which risks are real and which are noise. Taste isn't mysterious. It's a large store of cases, most of them mistakes, yours and other people's, compressed into fast judgement. The third is **leverage**: the ability to make that judgement useful to others, through review, design documents, mentoring and writing. An expert whose knowledge stays in their head helps one project at a time. One who can teach it changes how a whole organisation builds.

This volume covers all three. Part I is about depth: choosing an area and reading your way into it, through papers and production code. Part II is about working in the open, where taste is built and tested: open source, design documents, trade-offs and public writing. Part III is about leading: estimation, sequencing, communication and mentoring. Part IV is about the long game: the business, measurement, finishing things and a career that compounds.

Three levels deeper

The syllabus asks you to go "three levels deeper than your team needs", which needs unpacking. Think of knowing a technology at four levels. At level 1 you can **use** it: write the queries, call the API, follow the guide. At level 2 you can **configure and debug** it: tune it, read its logs, explain why it's slow. At level 3 you know its **internals and failure modes**: how it stores data, what it does under partition or overload, which of its guarantees are weaker than they sound. At level 4 you can **change** it: fix bugs in its source, contribute features, or build a smaller version yourself and explain every trade-off you made.

BREADTH: THE SIXTEEN STAGES



Breadth across every stage, depth in one. Most teams need levels 1 and 2 of everything they use; the expert is the person who has gone past the dashed line in one area.

Most teams need level 1 of most of what they use and level 2 of a few things. "Three levels deeper" means being at level 3 or 4 in something your team relies on, so that you're the person who can answer the questions nobody else can. It doesn't mean ignoring everything else. The T shape, broad across the top and deep in one place, is the usual picture, and the breadth matters: depth in databases is worth far more to someone who also understands the network, the operating system and the product than to someone who doesn't.

Choosing where to go deep

The hub's tracks, backend and distributed systems, platform, AI engineering, low-level systems, product and frontend, and security, are broad families. Inside each are areas narrow enough to master: storage engines, consensus and replication, query planning, cryptographic engineering, rendering performance. Choose an area, not a track. Four questions help.

- **Leverage:** how often do your team's hardest problems land in this area? Depth that your work uses every week compounds; depth that your work never touches decays.
- **Curiosity:** would you read about it on a free weekend, for years? Mastery takes a decade of attention, and only genuine interest survives that long.
- **Adjacency:** how much of what you already know carries over? An area next to your strengths gets you to level 3 sooner, and the early progress keeps you going.
- **Demand:** how many teams, anywhere, need a deep expert in it? This matters less than people think, because real depth in almost anything is rare, but it matters.

None of these is decisive on its own, and the honest way to combine them is to write them down, weight them, and then ask how much your answer depends on numbers you guessed. The example does exactly that, for a backend engineer on a payments team weighing five areas.

EXAMPLE

track.py · five candidate areas scored against four criteria, then tested for how fragile the winner is

```
"""Choosing a track to go deep in: score the candidates, then ask how fragile the answer is."""
```

```
# A backend engineer on a payments team, scoring five candidate tracks from 1 to 5.
```

```
CRITERIA = {
    "leverage": 3,      # how often their team's hard problems land here
    "curiosity": 3,     # would they read about it on a free weekend, for years?
    "adjacency": 2,    # how much of what they already know carries over
    "demand": 1,       # how many teams, anywhere, need a deep expert in it
}

TRACKS = {
    "databases":      {"leverage": 5, "curiosity": 3, "adjacency": 4, "demand": 4},
    "distributed":    {"leverage": 4, "curiosity": 4, "adjacency": 3, "demand": 4},
    "security":       {"leverage": 3, "curiosity": 5, "adjacency": 2, "demand": 5},
    "performance":   {"leverage": 3, "curiosity": 3, "adjacency": 3, "demand": 3},
    "ml-infrastructure": {"leverage": 1, "curiosity": 4, "adjacency": 1, "demand": 5},
}
```

```
def total(track, weights):
    return sum(weights[c] * TRACKS[track][c] for c in weights)
```

```
def ranking(weights):
    return sorted(TRACKS, key=lambda t: (-total(t, weights), t))
```

```
weights = dict(CRITERIA)
order = ranking(weights)
print("track          " + ".join(f"{c:>11}" for c in CRITERIA) + "      total")
for t in order:
    print(f"{t:20}" + ".join(f"{TRACKS[t][c]:>11}" for c in CRITERIA) + f"{total(t,
        weights):>11}")
winner, runner_up = order[0], order[1]
lead = total(winner, weights) - total(runner_up, weights)
print(f"\nwinner: {winner}, {lead} points ahead of {runner_up}")
```

```
# How far can each weight move, with the others held, before the winner changes?
```

```
print("\nweight range that keeps the same winner (tried from 0 to 6 in steps of 0.5):")
for c in CRITERIA:
    keep = [w / 2 for w in range(0, 13) if ranking({**weights, c: w / 2})[0] == winner]
    flips = sorted({ranking({**weights, c: w / 2})[0] for w in range(0, 13)} - {winner})
    note = "never flips" if not flips else "otherwise " + ", ".join(flips) + " wins"
    print(f" {c:10} now {weights[c]}: {min(keep):g} to {max(keep):g} ({note})")
```

```
# What would have to be true for the runner-up to win? One score at a time.
```

```
def winner_if(track, criterion, score):
    saved, TRACKS[track][criterion] = TRACKS[track][criterion], score
    try:
        return ranking(weights)[0]
    finally:
        TRACKS[track][criterion] = saved

print(f"\nwhat would have to be true for {runner_up} to win, one score at a time:")
print(f" {'criterion':10} {winner + ' falls to':>18} {runner_up + ' rises to':>20}")
for c in CRITERIA:
    now_w, now_r = TRACKS[winner][c], TRACKS[runner_up][c]
    fall = next((s for s in range(now_w - 1, 0, -1) if winner_if(winner, c, s) == runner_up),
        None)
    rise = next((s for s in range(now_r + 1, 6) if winner_if(runner_up, c, s) == runner_up),
```

```

None)
show = lambda now, s: f"{now} -> {s}" if s else "no score does it"
print(f" {c:10} {show(now_w, fall):>18} {show(now_r, rise):>20}")

```

OUTPUT

track	leverage	curiosity	adjacency	demand	total
databases	5	3	4	4	36
distributed	4	4	3	4	34
security	3	5	2	5	33
performance	3	3	3	3	27
ml-infrastructure	1	4	1	5	22

winner: databases, 2 points ahead of distributed

weight range that keeps the same winner (tried from 0 to 6 in steps of 0.5):

```

leverage now 3: 1.5 to 6 (otherwise security wins)
curiosity now 3: 0 to 4.5 (otherwise security wins)
adjacency now 2: 0.5 to 6 (otherwise security wins)
demand now 1: 0 to 4 (otherwise security wins)

```

what would have to be true for distributed to win, one score at a time:

```

criterion databases falls to distributed rises to
leverage      5 -> 4                4 -> 5
curiosity      3 -> 2                4 -> 5
adjacency      4 -> 2                3 -> 5
demand         4 -> 1      no score does it

```

What the example shows

The weights matter less than you'd expect. Databases wins, two points ahead of distributed systems, and it stays the winner over wide ranges of every weight. Leverage could fall from 3 to 1.5, or curiosity rise from 3 to 4.5, before security overtook it. Arguing about whether curiosity deserves a weight of 3 or 4 is mostly wasted breath.

The scores matter more. A single point, databases' leverage at 4 instead of 5, or its curiosity at 2 instead of 3, hands the win to distributed systems. That's the useful output. It tells you the decision is a near tie between two areas, decided by two judgements you could check. Is the team's hardest work really in the database? Spend a week reading database internals and see whether your curiosity score holds up. A decision matrix is worth less for its answer than for showing you which question to go and answer.

And notice what didn't come close: machine-learning infrastructure, with the highest demand of all, came last, because nothing in this engineer's work or experience touches it. Demand without leverage or adjacency makes for a slow, lonely start.

How long it takes

Peter Norvig's essay *Teach Yourself Programming in Ten Years* borrows a finding from research on expertise in chess, music and sport: world-class performance takes about ten years of deliberate practice, and there's no known shortcut. Engineering is no different. That

sounds discouraging until you notice what it implies: if you start now and keep going, you'll have it, and most people never start. A useful level 3 in one area takes perhaps two years of steady effort alongside a job, and level 4 several more.

The effort needs a shape, or it dissolves into reading whatever crosses your feed. Chapter 16 of *The Engineer's Whetstone* covers deliberate practice; here, the concrete version is a standing list of the next three things you'll read or build in your area, renewed whenever one is done. The rest of Part I is about two of the best sources: the papers that founded your area, and the production code that implements it.

Taste comes from being wrong

Taste grows from cases, and the most instructive cases are your own mistakes, but only if you notice them. Memory is generous to its owner: a year later, you'll remember having been fairly sure the migration would be risky, whatever you actually said. The cure is a **decision journal**. Before a decision that matters, write a few lines: what you decided, what you expected to happen, how confident you were, and what would prove you wrong. Months later, read it back. The surprises are the lessons, and they're the raw material of taste.

Other people's mistakes count too, and they're cheaper. Postmortems, published incident reports, design documents that describe a rejected alternative and why: read them in your area as you'd read case law. Over time you'll find you can predict the failure before reading how it ended. That's what taste is.

COMMON TRAPS

Specialising before you have breadth, so the depth has nothing to connect to.

Choosing by demand alone, an area that's popular but that your work never touches.

Staying at level 2 everywhere, because each new tool feels like progress.

Treating a decision matrix's answer as the decision, rather than as a pointer to what to check.

Remembering your judgement as better than it was, because nothing was written down at the time.

EXERCISES

EXERCISE 1.1 Score your own areas

Change the example's areas, scores and weights to your own. Which area wins, and what's the smallest single change that would make the runner-up win? What would you do in the next month to find out whether that change is true?

SOLUTION

Your numbers are yours, but the shape of a good answer is the same. Suppose storage engines beat stream processing by three points, and the smallest flip is your curiosity about storage engines dropping by one. Then the next month's task is a test of that score: read the first four chapters of *Database Internals* and one paper, LSM trees or a B-tree variant, and notice whether you look forward to it. If the flip were leverage instead, the test would be different: list the last ten hard problems your team hit and count how many were really storage problems. Either way, the exercise turns a vague career question into a small experiment with an answer. If no single change flips the result, the decision isn't close, and you can stop deliberating and start.

EXERCISE 1.2 Describe level 3

Pick a technology your team uses every day. Write down three things someone at level 3 would know about it that someone at level 2 wouldn't.

SOLUTION

For PostgreSQL, for example. At level 2 you can read `EXPLAIN ANALYZE` and add an index. At level 3 you know, first, that an `UPDATE` writes a whole new row version and leaves the old one for vacuum, so a table updated constantly bloats, and a long-running transaction anywhere stops vacuum from reclaiming it. Second, that the planner's row estimates come from sampled statistics with an independence assumption, so correlated columns produce wild misestimates that no index fixes. Third, that `synchronous_commit`, `fsync` and replication settings decide what a "committed" transaction survives, and that the defaults trade some of that for speed. Each of these is a failure mode, not a feature, which is the mark of level 3: you know how it breaks. If your three items read like documentation of features, push further.

EXERCISE 1.3 A decision journal entry

Write a decision journal entry for a technical decision you're facing this month, in no more than eight lines.

SOLUTION

2026-10-23 Decision: add a read replica for reporting, rather than a separate warehouse.
Why: reporting queries are 40% of primary CPU; a replica takes a week, a warehouse a quarter.
Expect: primary CPU down to about 55% at peak within two weeks of the switch.
Confidence: 70%.
I'm wrong if: replica lag makes the finance team's end-of-day report unusable, or CPU stays above 70% because the real cost was something else.
Alternatives rejected: query tuning only (tried, saved 10%); a warehouse (too slow to deliver).
Review on: 2026-12-01.

The entry is short enough that you'll actually write it, and specific enough to be checked: a number, a confidence, a date, and the evidence that would prove it wrong. The "I'm wrong if" line is the most valuable, because it's the one memory rewrites. When you review it in December, record what happened in one line beneath it. After a year of entries, read them all in one sitting: your recurring errors, perhaps always underestimating migration time, or overrating new tools, become obvious, and that's the start of taste.

CHECKLIST

- ☐ Can you name the area you're going deep in, and say why it beat the alternatives?
- ☐ Can you describe what level 3 and level 4 mean for it, concretely?
- ☐ Do you know which of your judgements your choice depends on, and how you'd test them?
- ☐ Do you have a standing list of the next three things to read or build?
- ☐ Do you keep a decision journal, and review it?

GO DEEPER

GUIDES Staff Engineer: guides Will Larson

ESSAY Teach Yourself Programming in Ten Years Peter Norvig

TALK You and Your Research Richard Hamming, 1986

Reading a paper

3 exercises

The papers behind your area are its primary sources: the ideas as their inventors stated them, with the reasoning, the alternatives they rejected and the measurements they had. They're short, and most engineers never read them. Read each one in passes, with a question in mind. Then build its central idea small enough to run, because until you've built it you've only read about it.

Why go to the source

Everything you've learned about distributed systems passed through several hands before it reached you: the paper, the talk about the paper, the blog post about the talk, the vendor's documentation, the colleague's summary. Each retelling keeps the conclusion and drops the reasoning. The paper still has it. It says what problem the authors faced, which designs they considered, why they chose this one, what they measured, and, often in a sentence nobody quotes, what it doesn't handle well.

That last part is why primary sources build taste. Dynamo's authors say plainly that their design suits applications that can tolerate conflicting versions. MapReduce's authors say its model fits computations that can be expressed as map and reduce, and leave the rest unsaid. Read the paper and you know the system's boundaries as its designers saw them, which no summary preserves. And most of the classics are twelve to fifteen pages long. A paper a month is a modest habit, and in five years it's a foundation few engineers have.

Read in three passes

Srinivasan Keshav's short note *How to Read a Paper* describes the method most researchers use, and it works just as well for engineers. Never read a paper once from start to end. Read it three times, each pass deeper, and stop after any pass if the paper isn't worth more.

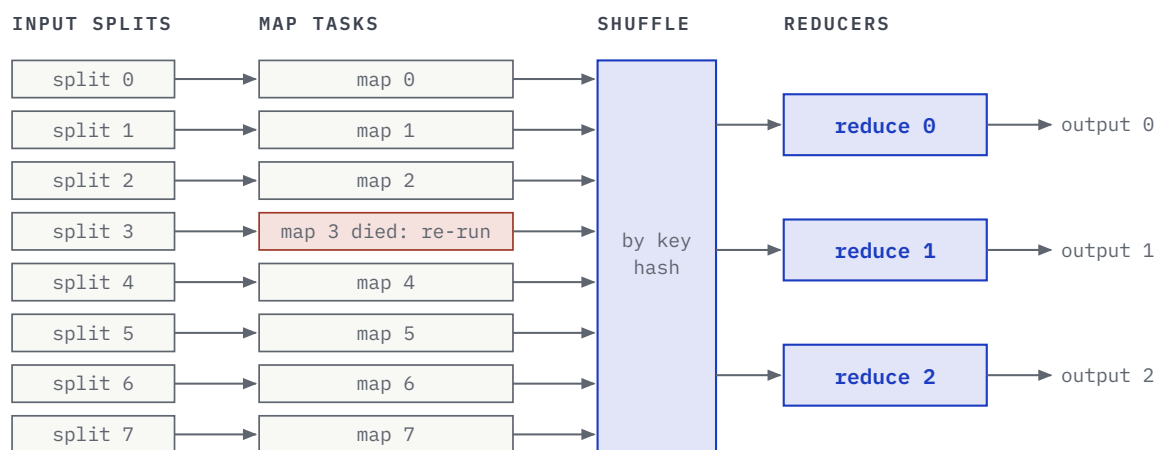
- **First pass, ten minutes.** Title, abstract and introduction; the section headings; the conclusion; a glance at the references. Afterwards, answer Keshav's five Cs: the **category** of paper (a system, a measurement, an analysis), its **context** (what it builds on), whether its assumptions look **correct**, its main **contributions**, and its **clarity**. Most papers stop here.
- **Second pass, an hour.** Read the whole paper but skip proofs and fine detail. Study the figures and the evaluation: what workload, on what hardware, compared with what? Note what you don't understand, and the references you'd need.
- **Third pass, several hours.** Reconstruct the paper: make the same assumptions and rebuild the design yourself, then compare. This is where you find the hidden assumptions and the weak spots, and it's the pass that makes the ideas yours.

Go in with a question, as you would with a debugger. "How does GFS handle a chunkserver dying mid-write?" focuses a second pass far better than "learn about GFS". And read the evaluation sceptically. Every paper measures the workloads its design is good at. The interesting question is always which workloads it didn't measure, and why.

Building the core small

For an engineer, the best third pass is to build the paper's central idea, small enough to run in a page of code. Not the system, which took a team years, but the mechanism the paper is about. Building forces every vague sentence into a decision. "The master re-executes failed tasks" becomes the question of what state a failed task leaves behind, and whether running it twice could double-count anything. The places where you get stuck are the places where you didn't understand the paper, even though you thought you did.

MapReduce, the 2004 paper by Jeffrey Dean and Sanjay Ghemawat, is the classic case. Its idea fits in a sentence: express a computation as a **map** function, applied to each piece of input independently, and a **reduce** function, applied to all the values for each key, and let the system handle distributing the work, moving the data and surviving failures. The figure shows the flow, and the example builds it: a word count over eight input splits, with three of the paper's refinements.



each map writes one partition per reducer; a combiner adds up repeats first

Map tasks work on splits independently and write one partition per reducer; the shuffle brings each key's values together; reducers produce the output. Because map is a pure function of its split, a failed task can simply run again.

EXAMPLE

mapreduce.py · the paper's model in sixty lines: a word count with a combiner, a failed task re-run, and backup tasks for stragglers

```
"""Reading the MapReduce paper by building its core: map, shuffle, reduce, a combiner,
re-execution after a failure, and backup tasks for stragglers."""
import random
import re
from collections import defaultdict
```

```

DOCS = [ # eight "input splits", one per map task
    "the log is the database the rest is cache",
    "a cache is a log you are allowed to forget",
    "the database keeps the log and the log keeps the truth",
    "replicas apply the log in order and agree",
    "the truth is in the log not in the cache",
    "forget the cache and replay the log",
    "order is the hard part of the log",
    "agree on the order and the rest follows",
]

def map_fn(doc):
    # user code: emit (word, 1) for every word
    for word in re.findall(r"[a-z]+", doc):
        yield word, 1

def reduce_fn(word, counts):
    # user code: add up one word's counts
    return sum(counts)

def run_job(docs, reducers=3, combiner=False, fail_task=None):
    """One MapReduce job. Each map task writes one partition per reducer, as the paper's do."""
    partitions = [defaultdict(list) for _ in range(reducers)]
    pairs_sent = 0
    for task, doc in enumerate(docs):
        attempts = 2 if task == fail_task else 1 # the master re-runs a task whose worker died
        for attempt in range(attempts):
            local = [defaultdict(list) for _ in range(reducers)]
            for word, n in map_fn(doc):
                local[sum(map(ord, word)) % reducers][word].append(n) # partition by hash of the key
            if task == fail_task and attempt == 0:
                continue # the worker died: its local output is lost, nothing was sent
            for r, part in enumerate(local):
                for word, ns in part.items():
                    if combiner:
                        ns = [reduce_fn(word, ns)] # pre-aggregate on the map side
                    partitions[r][word].extend(ns)
                pairs_sent += len(ns)
    result = {}
    for part in partitions: # each reducer sees its keys in sorted order
        for word in sorted(part):
            result[word] = reduce_fn(word, part[word])
    return result, pairs_sent

plain, sent_plain = run_job(DOCS)
combined, sent_combined = run_job(DOCS, combiner=True)
rerun, _ = run_job(DOCS, fail_task=3)
top = sorted(plain.items(), key=lambda kv: (-kv[1], kv[0]))[:5]
print("top words:", ", ".join(f"{w} {n}" for w, n in top))
print(f"pairs shuffled to reducers: {sent_plain} without a combiner, {sent_combined} with one "
      f"f"({1 - sent_combined / sent_plain:.0%} less)")
print("same answer with the combiner:", combined == plain)
print("same answer after map task 3's worker died and the task was re-run:", rerun == plain)

# Stragglers: the job finishes when its slowest task does. The paper's fix is a backup copy of
# each task still running near the end; whichever copy finishes first wins.
rng = random.Random(7)
def job_time(tasks=200, machines=50, backups=False):
    times = []
    for _ in range(tasks):

```

```

t = rng.uniform(1.0, 1.4)
if rng.random() < 0.03:                # a few machines are slow: a bad disk, a noisy
    neighbour                           neighbour
    t *= rng.uniform(5, 10)
if backups and t > 2.0:                # still running at the end: a backup copy
    starts there,
    t = min(t, 2.0 + rng.uniform(1.0, 1.4)) # and it usually lands on a healthy
    machine
times.append(t)
waves = [times[i:i + machines] for i in range(0, tasks, machines)]
return sum(max(w) for w in waves)      # each wave of tasks waits for its slowest
member
runs = 1000
without = sorted(job_time() for _ in range(runs))
with_b = sorted(job_time(backups=True) for _ in range(runs))
med = lambda xs: xs[len(xs) // 2]
print(f"\n200 tasks on 50 machines, 3% of tasks on a slow machine, {runs} simulated jobs:")
print(f"  median job time without backup tasks: {med(without):5.1f}")
print(f"  median job time with backup tasks:    {med(with_b):5.1f}  "
      f"({1 - med(with_b) / med(without):.0%} faster)")

```

OUTPUT

```

top words: the 17, log 8, is 5, and 4, cache 4
pairs shuffled to reducers: 71 without a combiner, 56 with one (21% less)
same answer with the combiner: True
same answer after map task 3's worker died and the task was re-run: True

200 tasks on 50 machines, 3% of tasks on a slow machine, 1000 simulated jobs:
  median job time without backup tasks:  32.1
  median job time with backup tasks:     11.2  (65% faster)

```

What the example shows

A combiner cuts the shuffle, by how much depends on the data. Running the reduce function on each map task's own output before sending it reduced the pairs shuffled from 71 to 56, 21% fewer. The saving is modest here because each split is one short sentence, so few words repeat within a split. On real inputs, where a 64 MB split contains "the" thousands of times, the paper's combiner cuts traffic by orders of magnitude. The example also shows the condition the paper relies on: the combiner gave the same answer only because addition doesn't care how you group it. Try it with an average instead of a sum, and exercise 2.2 shows what breaks.

Re-execution is safe because map is pure. Map task 3's worker died, its local output vanished, the task ran again elsewhere, and the result was identical. This is the paper's deepest design decision, easy to miss on a first read: by asking users for functions with no side effects, the system can retry anything without coordination. A map function that wrote to a database, or read the clock, would make retries visible and the whole fault-tolerance story would collapse.

Backup tasks attack the tail. A job finishes when its slowest task does, and a few slow machines, a bad disk, a noisy neighbour, can hold up hundreds of healthy ones. The paper's fix is to start a second copy of each task still running near the end and take whichever finishes first. In the simulation, the median job finished 65% sooner. Its constants are invented, so read that for its direction; the paper itself reports that one sort job took 44%

longer with backup tasks switched off. The same idea reappears a decade later in Dean and Barroso's *The Tail at Scale* as hedged requests, which is one reason to read papers in sequence.

Reading the critics too

A paper is an argument, and arguments have opponents. In 2008, David DeWitt and Michael Stonebraker published "MapReduce: A major step backwards", arguing that it ignored forty years of database research: no schemas, no indexes, brute-force scans. Some of the criticism aged badly and some aged well; within a few years, SQL layers such as Hive appeared on top of Hadoop, and later systems such as Spark kept the fault-tolerance idea while dropping the rigid two-stage model. Reading the paper, its critics and what came after tells you more about the design space than any of them alone. When you finish a paper, look for who argued with it.

Making it a habit

Habits survive where intentions don't. Three that work: a **reading list** of five papers in your area, ordered so each builds on the last; a **one-page summary** for each, written after the second pass, with the problem, the key idea, the evidence and your doubts; and **company**. A reading group of three colleagues meeting every fortnight to discuss one paper will carry you through papers you'd abandon alone, and the discussion surfaces what you missed. Papers We Love, a community with chapters in many cities, keeps a curated repository of papers by topic and is a good place to start. Chapter 3 suggests a first reading list for distributed data systems.

COMMON TRAPS

Reading a paper once, front to back, and getting lost in section 3.

Reading the summary instead, and losing the reasoning and the stated limits.

Trusting the evaluation, without asking which workloads it left out.

Stopping at understanding, and never building the core to test it.

Reading alone, without writing anything down, so a month later little remains.

EXERCISES

EXERCISE 2.1 A first pass

Do a ten-minute first pass of *The Google File System* (2003) and answer the five Cs in one line each.

SOLUTION

Category: a systems paper: the design of a production storage system, with measurements.

Context: distributed file systems such as AFS and NFS, rethought for Google's workloads.

Correctness: assumes failures are normal, files are huge, writes are mostly appends and reads mostly streaming; plausible for batch data processing, not for databases.

Contributions: a single master holding metadata, 64 MB chunks replicated three times, relaxed consistency with atomic record append, and lessons from running it.

Clarity: clear, with a design rationale for each decision and honest about weak points.

The "correctness" line is the important one, because it states the assumptions the whole design rests on. Every unusual choice in GFS, the single master, the large chunks, the weak consistency, follows from them. A first pass that captures the assumptions tells you when the paper applies to your problem and when it doesn't, before you've read the details.

EXERCISE 2.2 A combiner that lies

Change the example to compute the average length of the words starting with each letter, and use the reduce function as the combiner. Why is the answer wrong, and how would you fix it?

SOLUTION

An average of averages isn't the overall average. If one map task saw two words starting with "t" of lengths 3 and 5, its combiner sends 4; if another saw one word of length 10, it sends 10; the reducer averages 4 and 10 to 7, but the true average of 3, 5 and 10 is 6. Addition is associative and commutative, so partial sums combine correctly; averaging isn't. The fix is to combine something that does compose: have map emit (length, 1) pairs, have the combiner and reducer add both parts, giving (total length, count), and divide only at the very end. The paper's condition, that the combining function must be commutative and associative, is exactly this.

EXERCISE 2.3 A map that isn't pure

A map function samples 10% of its input at random to make the job faster. What goes wrong when the system re-runs a failed task or a backup task, and what would you do instead?

SOLUTION

Each run of the task picks a different random 10%. If a map task fails after some of its output was read, or a backup copy and the original both finish, the reducers can see output from two different samples, so the totals are neither one sample nor the other, and the job's result changes from run to run. The paper's guarantee, that the output equals what a single sequential run would produce, holds only for deterministic functions. The fix is to make the sampling deterministic: decide whether to keep each record by hashing its key or position with a fixed seed, so every run of the task keeps exactly the same records. The general lesson applies far beyond MapReduce: anything that a system might retry must give the same result each time, which is the same property chapter 10 of *The Engineer's Fleet* asks of message consumers.

CHECKLIST

- ☐ Can you do a first pass in ten minutes and answer the five Cs?
- ☐ Do you read the evaluation asking which workloads were left out?
- ☐ Have you built the central idea of at least one paper small enough to run?
- ☐ Can you explain why MapReduce's fault tolerance depends on pure functions?
- ☐ Do you have a reading list in your area, and a summary for each paper read?

GO DEEPER

NOTE How to Read a Paper S. Keshav

PAPER MapReduce: Simplified Data Processing on Large Clusters Dean & Ghemawat, 2004

COMMUNITY Papers We Love a curated repository of papers, and local chapters

The canon

3 exercises

A handful of papers, published over about ten years, founded the distributed data systems most of the industry now runs on. Read them in order and they become a conversation: each one answers a problem the last one left open. This chapter gives a reading order with the one lesson each paper teaches, then reads the most surprising of them, Spanner, closely enough to build its central trick: making timestamps agree with real time by waiting out the clocks' uncertainty.

A conversation in seven papers

The syllabus names six papers: *The Google File System*, MapReduce, Bigtable, Dynamo, Spanner and Raft. Read in publication order, with Chubby added, they tell a story about one question: how do you build a dependable system out of thousands of machines that fail all the time?

YEAR	PAPER	THE LESSON THAT TRANSFERS
2003	The Google File System	Treat failure as the normal case, and design for your real workload, not a general one.
2004	MapReduce	A restricted programming model buys automatic distribution and fault tolerance.
2006	Bigtable	A sorted, versioned map, built on a file system and a lock service, can serve both batch and online work.
2006	Chubby	Put consensus in one small, well-tested service, and let everything else lean on it.
2007	Dynamo	When availability matters more than agreement, accept conflicting versions and reconcile them.
2012	Spanner	With bounded clock uncertainty, a global database can have real-time ordering, at the price of waiting.
2014	Raft	Consensus can be designed to be understood, and understandability is a correctness property.

Two lines of descent run through the table. Google's papers keep strong coordination and spend engineering on making it cheap: a single master in GFS, a lock service under Bigtable, Paxos inside Spanner. Amazon's Dynamo goes the other way: no master, writes

accepted anywhere, conflicts resolved afterwards, because a shopping cart that's sometimes inconsistent beats one that's sometimes unavailable. *The Engineer's Fleet* covers the theory behind that choice. The papers show it being made, by people with real constraints, and that's the part that builds judgement.

Read the first pass of all seven in a weekend, then one a month in depth. For each, find the sentence where the authors state the assumption everything else depends on. GFS assumes large files written by appending. Dynamo assumes applications can merge conflicting versions. Spanner assumes clocks whose error is known and small. When a system built on one of these designs misbehaves, it's very often because that assumption stopped being true.

Spanner's problem

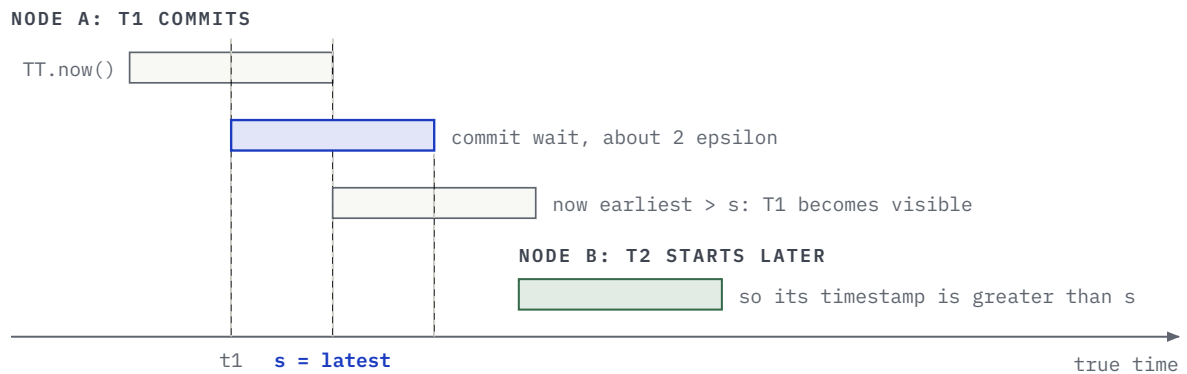
Spanner, from James Corbett and colleagues at Google in 2012, is a database spread across data centres on several continents that still gives transactions a property called **external consistency**: if one transaction commits before another starts, in real time, the first gets the smaller timestamp. That sounds obvious, and it's very hard, because the two transactions may run on machines thousands of kilometres apart whose clocks disagree.

Why it matters: suppose an administrator revokes Bob's access to a folder, and a moment later, a colleague uploads a salary file to it. The revoke committed first, in real time. If the machines' clocks disagree, the upload can still get the earlier timestamp. Then a read of the database "as of" some moment between the two timestamps, the kind of consistent snapshot every backup, report and replica uses, sees the salary file in the folder and Bob still allowed to read it. No single machine did anything wrong; the timestamps simply lied about the order.

TrueTime and commit wait

Spanner's answer has two parts. The first is **TrueTime**, a clock API that doesn't return a time but an interval: `TT.now()` returns `[earliest, latest]`, guaranteed to contain the true time. Google keeps the interval narrow with GPS receivers and atomic clocks in every data centre; the paper reports an uncertainty, called epsilon, that typically varied between about 1 and 7 milliseconds.

The second part is a rule. When a transaction commits, Spanner takes its timestamp from the top of the interval, `latest`, a time no earlier than now, and then waits, before letting anyone see the commit, until `TT.now().earliest` has passed that timestamp. After the wait, the timestamp is certainly in the past everywhere. Any transaction that starts afterwards, on any machine, will read a clock whose interval lies entirely after it, and so get a larger timestamp. The figure shows one commit, and the example tests the rule against the alternative of stamping commits with the local clock.



Take the timestamp from the top of the uncertainty interval, then wait until the bottom of the interval has passed it. The wait, about twice epsilon, is the price of timestamps that agree with real time.

EXAMPLE

truetype.py · ten thousand causally ordered pairs of transactions on nodes with skewed clocks, stamped by local clocks and then with commit wait

"""Spanner's commit wait, from the paper's rules: why waiting out the clock uncertainty makes timestamps agree with real time, and what it costs."""

import random

EPS = 0.007 # TrueTime's uncertainty: every clock is within 7 ms of true time

class Node:

def __init__(self, offset):
 self.offset = offset # how far this node's clock is from true time

def clock(self, t):
 return t + self.offset

def tt_now(self, t): # TrueTime: an interval sure to contain true time
 c = self.clock(t)
 return c - EPS, c + EPS

def commit(node, t, wait):

"""Commit at true time t. Returns (timestamp, true time the commit becomes visible)."""
 earliest, latest = node.tt_now(t)

if not wait:
 return node.clock(t), t # stamp with the local clock, visible at once

ts = latest # a timestamp no earlier than true time
 while node.tt_now(t)[0] <= ts: # commit wait: until ts is surely in the past
 t += 0.0001
 return ts, t

def trial(rng, wait):

a, b = Node(rng.uniform(-EPS, EPS)), Node(rng.uniform(-EPS, EPS))
 t1 = rng.uniform(0, 100)
 ts1, visible = commit(a, t1, wait) # T1 on node a: revoke Bob's access to /payroll

```

t2 = visible + rng.uniform(0, 0.005)           # T2 starts after T1 is visible, on node b
ts2, done = commit(b, t2, wait)                 # T2: upload salaries.csv to /payroll
return ts2 <= ts1, visible - t1, (ts1, ts2)

for wait in (False, True):
    rng = random.Random(15)
    results = [trial(rng, wait) for _ in range(10_000)]
    bad = sum(r[0] for r in results)
    added = sum(r[1] for r in results) / len(results)
    label = "with commit wait " if wait else "local clock stamps "
    print(f"{label}: {bad:5} of 10,000 causal pairs got timestamps"
          f" in the wrong order; commit latency added {added * 1000:.1f} ms")
    if bad:
        ts1, ts2 = next(r[2] for r in results if r[0])
        snap = (ts1 + ts2) / 2
        print(f"    e.g. revoke stamped {ts1:.4f}, upload stamped {ts2:.4f}: a snapshot read at"
              f"{snap:.4f}"
              f"\n    sees salaries.csv in /payroll and Bob still allowed to read it")

print(f"\nthe wait is 2 x epsilon: {2 * EPS * 1000:.0f} ms at epsilon = {EPS * 1000:.0f} ms. "
      "Better clocks make every commit faster:")
for eps_ms in (7, 4, 1):
    print(f"    epsilon {eps_ms} ms -> commit wait {2 * eps_ms} ms")

```

OUTPUT

```

local clock stamps : 3446 of 10,000 causal pairs got timestamps in the wrong order; commit
    latency added 0.0 ms
    e.g. revoke stamped 73.6057, upload stamped 73.5931: a snapshot read at 73.5994
    sees salaries.csv in /payroll and Bob still allowed to read it
with commit wait   :    0 of 10,000 causal pairs got timestamps in the wrong order; commit
    latency added 14.0 ms

the wait is 2 x epsilon: 14 ms at epsilon = 7 ms. Better clocks make every commit faster:
    epsilon 7 ms -> commit wait 14 ms
    epsilon 4 ms -> commit wait 8 ms
    epsilon 1 ms -> commit wait 2 ms

```

What the example shows

Local clocks get the order wrong a third of the time. With clocks up to 7 ms off in either direction, and the second transaction starting within 5 ms of the first becoming visible, 3,446 of 10,000 causally ordered pairs received timestamps in the wrong order. The example prints one: the revoke stamped later than the upload, so a snapshot between them shows the salary file with Bob still allowed in. Real gaps between transactions are usually longer than 5 ms, so real error rates are lower, but any rate above zero breaks the guarantee.

Commit wait gets it right every time, for a fixed price. With the wait, none of the 10,000 pairs was misordered, and every commit took 14 ms longer, exactly twice epsilon. In the real system, much of that wait overlaps with the Paxos round that replicates the commit, so it costs less than it seems, but the trade is fundamental: certainty about order is bought with time, in proportion to how uncertain the clocks are.

Better clocks make every write faster. At 1 ms of uncertainty the wait falls to 2 ms. That's why Google put atomic clocks in its data centres, and why the idea has spread. Amazon's Time Sync Service and its open-source ClockBound daemon now give ordinary cloud

machines a clock with a stated error bound. Other databases make a different trade. CockroachDB, without special clocks, assumes a maximum offset and, instead of waiting at commit, restarts a read that encounters a value whose timestamp falls inside its uncertainty window. Reading Spanner closely is what lets you see that these are variations on one idea.

Reading the canon today

None of these systems is used today exactly as published, and that's not the point. Hardware changed, Bigtable's successors look different, and Raft has largely replaced the Paxos implementations of the earlier papers. What lasts is the method each paper shows: name the workload, state the assumptions, pick the weakest guarantee that serves the application, and measure. When you design something, try writing the first page of the paper you'd publish about it: the problem, the assumptions and the key idea. If you can't, the design isn't ready.

COMMON TRAPS

Reading the papers out of order, and missing what each one was answering.

Copying a design without its assumptions, such as GFS's append-only workload for a database.

Trusting a timestamp from a machine's clock to order events across machines.

Treating a clock's error as zero, when it's milliseconds and sometimes much more.

Reading only Google's line, or only Amazon's, and taking one set of trade-offs as the only one.

EXERCISES

EXERCISE 3.1 The assumption under each design

For Bigtable, Dynamo and Raft, name the assumption the design depends on, and a situation where it stops being true.

SOLUTION

Bigtable assumes a reliable lock service and file system underneath it: it uses Chubby to make sure only one master and one server per tablet are active, and GFS for durability. If Chubby is unavailable for long, Bigtable becomes unavailable too; the paper reports exactly this, as a small fraction of downtime. **Dynamo** assumes applications can

reconcile conflicting versions of an object, merging two shopping carts, say. For a bank balance, where merging two concurrent withdrawals produces money that doesn't exist, the assumption fails and Dynamo's model is the wrong one. **Raft** assumes a majority of servers can communicate and that messages are eventually delivered. In a partition that leaves no majority, it correctly refuses to make progress, so a cluster of two data centres with an even split can't elect a leader. Each answer names when the design is wrong, which is the point of reading assumptions first.

EXERCISE 3.2 When the clock lies

Change the example so one node's clock is 20 ms off while TrueTime still claims 7 ms of uncertainty. What happens, and what does Spanner do to prevent it?

SOLUTION

Commit wait stops protecting you. The guarantee depends on the interval really containing true time; with a node 20 ms off, its "latest" can be earlier than true time, so a timestamp taken there can still precede a transaction that commits later somewhere else, and misordered pairs reappear. In the example, give the first node an offset of 0.020, 20 ms fast, and about three quarters of the pairs come out in the wrong order despite the wait. Spanner defends the assumption rather than the algorithm: each machine's time daemon checks itself against several time masters, and a machine whose local clock drifts beyond its bound, or that can't reach enough masters, removes itself from service. The lesson generalises: when an algorithm's correctness rests on an assumption, the system needs a mechanism that detects the assumption failing and stops, not just an algorithm that's correct when it holds.

EXERCISE 3.3 Two ways to pay for order

Spanner waits at commit; CockroachDB restarts reads that see an uncertain value. Which workload suffers more under each approach?

SOLUTION

Commit wait taxes writes: every read-write transaction pays about twice epsilon, whether or not any reader would have been confused, so a write-heavy workload with small, frequent transactions pays most. Reads are free of it. CockroachDB's uncertainty restarts tax reads instead, and only some of them: a read restarts when it meets a value

written within the clock uncertainty window just before its own timestamp, which happens mostly when reads closely chase recent writes to the same keys, such as a hot counter read and written constantly. With a large maximum offset, those restarts become frequent. So Spanner's approach is predictable and suits workloads that can afford a few milliseconds per write; CockroachDB's is cheaper on average without special clocks but less predictable under contention. Both are ways of paying for the same thing, which is ignorance of what time it is.

CHECKLIST

- ☐ Can you place each of the seven papers in order and say what problem it answered?
- ☐ Can you name the central assumption of GFS, Dynamo and Spanner?
- ☐ Can you explain external consistency with an example of what goes wrong without it?
- ☐ Can you explain commit wait, and why it costs about twice epsilon?
- ☐ Have you done a first pass of every paper on the list?

GO DEEPER

PAPER Spanner: Google's Globally-Distributed Database Corbett et al., 2012

PAPER Dynamo: Amazon's Highly Available Key-value Store DeCandia et al., 2007

PAPER In Search of an Understandable Consensus Algorithm Ongaro & Ousterhout, 2014

Reading production code

3 exercises

The source of the tools you use every day is the most accurate documentation they have, and reading it is how level 2 becomes level 3. But nobody reads a large codebase from the first line. Map it first: its files, its entry points, its dependencies, its history. Then trace what actually runs for one real call, because the code you read and the code that runs are not always the same. And then fix something in it, which is the fastest way to understand any of it.

Why read the source

Documentation describes what the authors intended; source code describes what happens. When they disagree, the code wins, and the disagreements are exactly the cases you need to know about: the edge where a timeout isn't enforced, the default that's different from the one in the manual, the fast path that skips a check. Every expert you admire in an area has read the code at its centre. Linus Torvalds' "use the source, Luke" is a joke, and also the most practical advice in this volume.

Production code teaches something papers can't: what a design looks like after ten years of bug reports. The paper describes the idea; the code shows the idea plus every special case that real users found. Reading the special cases, and the commit messages that explain them, is how you learn which parts of a design are fragile.

Map before you read

Approach a codebase the way you'd approach a city: get a map before walking the streets. Four questions, each answered with a tool rather than by reading.

- **What's here, and how big is it?** List the files and count their lines. A thousand-line file in a project of hundred-line files is where the complexity lives.
- **Where does it start?** Find the entry points: `main`, the public API, the request handler, the `__all__` of a Python package. Everything else is reached from these.
- **Who depends on whom?** Draw the import graph. A module everything imports is the foundation; one that imports everything is the glue.
- **What changes, and why?** `git log --format= --name-only | sort | uniq -c | sort -rn | head` lists the most-changed files, which are usually the ones that matter and the ones that hurt. `git log -L` and `git blame` show why a line is the way it is.

The tests are part of the map. A project's test suite is a list of the behaviours its authors cared enough to pin down, written as runnable examples. Reading the tests for a function before the function is often faster than reading the function.

Trace what runs

A map shows what could run. A trace shows what does, for one concrete input. Debuggers, profilers and logging all give traces; in Python, `sys.setprofile` lets a few lines of code record every function call, which is what the example uses on the standard library's `json` package. Tracing first, then reading only the functions on the path, turns a codebase of thousands of lines into a few dozen that matter for your question.

EXAMPLE

codemap.py · the json package mapped with the ast module, then one call to `json.dumps` traced, with and without its C accelerator

```
"""Reading production code: map a real package before reading it, then trace what actually
    runs.
The package is the standard library's json, as shipped with this Python."""
import ast
import json
import pathlib
import sys

pkg = pathlib.Path(json.__file__).parent
print(f"json package, Python {sys.version_info.major}.{sys.version_info.minor}")

# 1. The map: files, sizes, what each defines, and who imports whom.
print("\nmodule      lines  defines")
imports = {}
for f in sorted(pkg.glob("*.py")):
    tree = ast.parse(f.read_text(encoding="utf-8"))
    names = [n.name for n in tree.body if isinstance(n, (ast.FunctionDef, ast.ClassDef))]
    public = [n for n in names if not n.startswith("_")]
    deps = set()
    for n in ast.walk(tree):
        if isinstance(n, ast.ImportFrom) and n.level:                # from .decoder import ...
            deps.add((n.module or "").split(".")[0])
        elif isinstance(n, ast.ImportFrom) and n.module == "json":  # from json import scanner
            deps.update(a.name for a in n.names)
        elif isinstance(n, ast.ImportFrom) and (n.module or "").startswith("json."):
            deps.add(n.module.split(".")[1])
    imports[f.stem] = sorted(deps - {"", f.stem})
    lines = len(f.read_text(encoding="utf-8").splitlines())
    print(f"{f.stem:12} {lines:6} {'', '.join(public) or '(only private helpers)'}")
print("\nimports within the package:")
for mod, deps in imports.items():
    if deps:
        print(f" {mod} -> {'', '.join(deps)}")
print("the public entry points (json.__all__):", "", ".join(json.__all__)")

# 2. The trace: which of those functions actually run for one call, and in what order.
def trace(call):
    seen, depth = [], [0]
    def profiler(frame, event, arg):
        if pathlib.Path(frame.f_code.co_filename).parent != pkg:
            return
        if event == "call":
            module = pathlib.Path(frame.f_code.co_filename).stem
            seen.append(" " * depth[0] + f"{module}.{frame.f_code.co_qualname}")
            depth[0] += 1
        elif event == "return":
            depth[0] -= 1
    sys.setprofile(profiler)
```

```

try:
    return call(), seen
finally:
    sys.setprofile(None)

value = {"id": 7, "tags": ["a", "b"], "price": 9.5, "ok": None}
fast, calls = trace(lambda: json.dumps(value))
print(f"\njson.dumps(value): {len(calls)} Python-level calls")
for line in calls:
    print(" " + line)
print(f" the encoding itself ran in C: encoder.c_make_encoder comes from module "
      f"{json.encoder.c_make_encoder.__module__!r}")

# Switch the C accelerator off, as CPython's own tests do, and the Python you were reading
runs.
saved, json.encoder.c_make_encoder = json.encoder.c_make_encoder, None
try:
    slow, calls = trace(lambda: json.dumps(value))
finally:
    json.encoder.c_make_encoder = saved
names = sorted({c.strip() for c in calls})
print(f"\nthe same call with the accelerator off: {len(calls)} Python-level calls, "
      f"{len(names)} distinct functions; same output: {slow == fast}")
for line in calls[:8]:
    print(" " + line)
print(f" ... and {len(calls) - 8} more")

```

OUTPUT

json package, Python 3.14

module	lines	defines
__init__	365	dump, dumps, detect_encoding, load, loads
__main__	20	(only private helpers)
decoder	364	JSONDecodeError, py_scanstring, JSONObject, JSONArray, JSONDecoder
encoder	461	py_encode_basestring, py_encode_basestring_ascii, JSONEncoder
scanner	73	py_make_scanner
tool	122	main

imports within the package:

```

__init__ -> decoder, encoder
decoder -> scanner

```

the public entry points (json.__all__): dump, dumps, load, loads, JSONDecoder, JSONDecodeError, JSONEncoder

json.dumps(value): 3 Python-level calls

```

__init__.dumps
encoder.JSONEncoder.encode
encoder.JSONEncoder.iterencode
the encoding itself ran in C: encoder.c_make_encoder comes from module '_json'

```

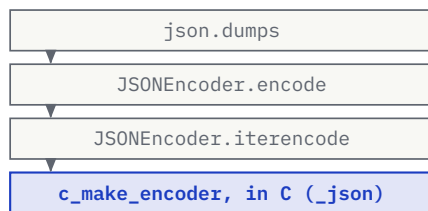
the same call with the accelerator off: 49 Python-level calls, 8 distinct functions; same output: True

```

__init__.dumps
encoder.JSONEncoder.encode
encoder.JSONEncoder.iterencode
encoder._make_iterencode
encoder._make_iterencode.<locals>._iterencode
encoder._make_iterencode.<locals>._iterencode_dict
encoder._make_iterencode.<locals>._iterencode
encoder._make_iterencode.<locals>._iterencode_dict
... and 41 more

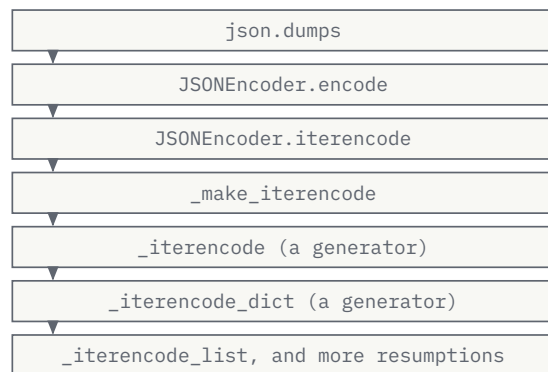
```

DEFAULT: WITH THE ACCELERATOR



3 Python-level calls; the rest is C

ACCELERATOR SWITCHED OFF



49 Python-level calls, same output

The same call, two paths. By default the encoding happens in C, and most of encoder.py's Python never runs; switch the accelerator off and that Python does the work. Both produce the same output, which is what CPython's tests check.

What the example shows

The map comes first and costs nothing. Six modules, about 1,400 lines in all, with the work split between `encoder.py` and `decoder.py`, and `decoder.py` leaning on `scanner.py`. The seven names in `__all__` are the whole public surface. That's enough to know where to look for any question about the package, before reading a single function body.

The code you read isn't always the code that runs. The trace of `json.dumps` found only three Python calls, then nothing: the encoding itself ran in C, in the `_json` module, which `encoder.py` imports if it can. Someone reading `_make_iterencode`, the most interesting function in the file, would be reading code that, in an ordinary run, never executes. Switch the accelerator off, as CPython's own tests do, and the same call made 49 Python-level calls through eight distinct functions, with identical output. The count is high because each time a generator resumes it counts as a call, and the pure-Python encoder is a tree of nested generators.

This pattern, a readable implementation with a fast one beside it, is common: Python's `heapq`, `datetime` and `decimal` do it too, as do many libraries in other languages. When a fast path exists, the slow one is usually the specification, and the tests check that the two agree. Knowing which one runs is part of reading the code.

Reading strategies that work

Start from a question. "Where does this library decide to retry?" gives you a search term, a trace to run and a place to stop. "Understand this library" gives you nothing. **Follow one request end to end.** For a web framework, a database or a message broker, pick one operation and trace it from the socket to the disk and back; most of the architecture is on

that path. **Keep a map document** as you go: the modules, their roles, the path you traced, the surprises. It's your notes for the next time, and, cleaned up, it's documentation the project probably lacks.

Read the history of the lines that puzzle you. A strange condition in a function is usually a bug fix, and `git log -S 'that condition'` finds the commit that added it, whose message, linked issue or review thread explains the case it handles. That's the special-case knowledge that makes level 3.

Then fix something

The syllabus asks you to read the source of a tool you use daily "and fix something in one". Fixing is the fastest way in because it forces every step: reproduce the bug, find the code responsible, understand it well enough to change it safely, write a test that fails before and passes after, and explain the change to a maintainer who knows the code better than you do. Start with a bug you've actually hit, or a "good first issue" in a project you use. A one-line fix with a good test and a clear explanation teaches more about a codebase than a week of reading. Chapter 5 is about the rest of that journey.

COMMON TRAPS

Reading from the first line of the first file, without a map or a question.

Reading code that never runs, such as the pure-Python fallback beside a C accelerator.

Skipping the tests, the clearest statement of what the code is meant to do.

Ignoring the history, when the commit message explains the strange line.

Reading without ever changing anything, so understanding is never tested.

EXERCISES

EXERCISE 4.1 Map another package

Point the example's mapping half at the standard library's `http` package. What are its modules, which is the largest, and where would you start reading to find out how `http.server` parses a request line?

SOLUTION

Replace `json` with `http` in the import and the `pathlib.Path(...).parent` line. The package is small: `__init__.py` with the `HTTPStatus` enum, `client.py`, `cookiejar.py`, `cookies.py`

and `server.py`, with `cookiejar.py` and `client.py` the largest. For request parsing, the map points straight at `server.py`'s `BaseHTTPRequestHandler`, and within it `parse_request`, which splits the request line into method, path and version and rejects malformed ones. A trace confirms it: run a tiny `HTTPServer` in a thread, make one request with `urllib`, and profile the handler; `handle_one_request` calls `parse_request`, then the `do_GET` method. Two minutes of mapping and one trace replace an hour of reading.

EXERCISE 4.2 Where the change happens

In a Git repository you work in, list the ten most-changed files of the last year. What do you learn, and what would you do with it?

SOLUTION

```
git log --since=1.year --format=%f | grep . | sort | uniq -c | sort -rn | head
```

The list usually shows three kinds of file. Some are configuration and dependency files that change with every release, which you can ignore. Some are the genuine centre of the product, the module every feature touches; that's where to read first and where a new engineer most needs a map. And some are files that change often because they're fragile: the same function patched again and again. Those are candidates for the refactoring *The Engineer's Craft* describes, and combining change frequency with complexity, a long, deeply nested file that changes weekly, is the method Adam Tornhill calls finding hotspots. Bring the list to your team: it's evidence, not opinion, about where the code hurts.

EXERCISE 4.3 How both paths are tested

CPython ships two JSON encoders, one in C and one in Python. Find out how its test suite makes sure both are tested and give the same results.

SOLUTION

In CPython's source, `Lib/test/test_json/__init__.py` imports the package twice with `import_helper.import_fresh_module`: once with `_json` blocked, giving a pure-Python `pyjson`, and once with `_json` fresh, giving `cjson`. It then defines two mixin classes, `PyTest` and `CTest`, that point a test at one or the other, and most test classes are

written once and inherited by both, so every test runs against both implementations. If the C accelerator isn't available, the C half is skipped. It's a pattern worth copying in your own code whenever you keep a fast path beside a simple one: one set of tests, run against both, so they can't drift apart. And it's the kind of thing you only learn by reading the tests, because the documentation doesn't mention it.

CHECKLIST

- ☐ Can you map an unfamiliar codebase, files, entry points, imports and history, in under an hour?
- ☐ Can you trace which functions run for one real call, and read only those?
- ☐ Do you check whether a fast path means the code you're reading doesn't run?
- ☐ Do you read the tests and the commit history of puzzling code?
- ☐ Have you fixed something, with a test, in a tool you use every day?

GO DEEPER

BOOK [The Architecture of Open Source Applications](#) ed. Brown & Wilson

GUIDE [Python Developer's Guide](#) the CPython core team

DOCS [git log](#), including -S and -L [Git](#)

PART II

Working in the open

Four chapters on the work where taste is built and tested in front of other people. Open source, where trust is earned one reviewable patch at a time. Design documents, which must survive sceptical senior engineers. Trade-offs, argued honestly from both sides with the uncertain numbers in view. And writing in public, which compounds more than any certificate. Each is a skill that improves only with an audience, and each chapter shows the mechanical checks that catch the most common failures before the audience does.

Open source, from patch to reviewer

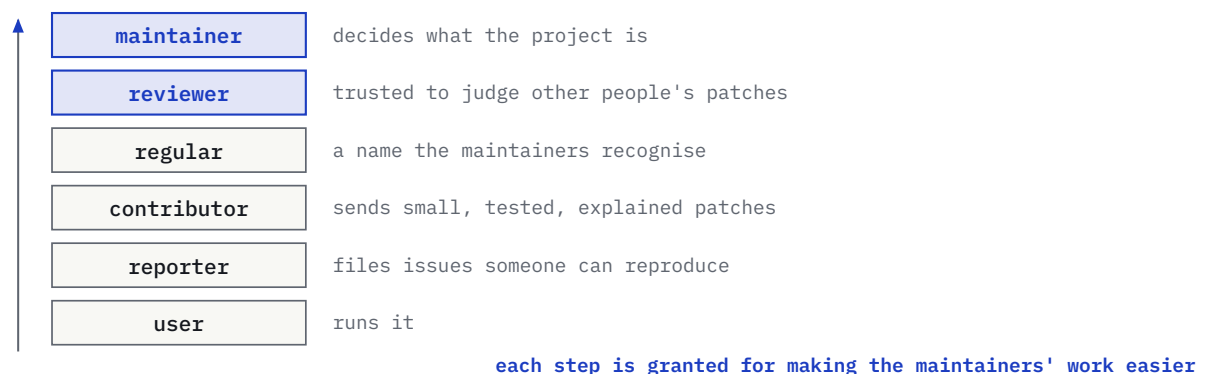
3 exercises

Contributing to open source is less about code than about attention: a maintainer has little of it, and every patch spends some. The contributors who become trusted are the ones whose patches cost the least to accept: small, tested, explained, and in the project's style. This chapter looks at a project from the maintainer's side, runs the checks a careful maintainer runs on two incoming patches, and maps the path from first patch to reviewer.

A ladder of trust

Every open-source project has a rough ladder, whether or not it's written down. **Users** run the software. **Reporters** file issues good enough to act on. **Contributors** send patches. **Regulars** send patches often enough that maintainers recognise their names. **Reviewers** are trusted to judge other people's patches, and **maintainers** decide what the project is. Each step up is granted, not claimed, and it's granted for one reason: the person has reliably made the maintainers' work easier.

The syllabus asks you to "contribute to open source until you are trusted with review, not just patches". That's a statement about the top of the ladder, and it takes months, sometimes years, in one project. It's worth it because review is where the deepest learning happens: you see every proposed change to a system you care about, and the maintainers' comments on your reviews teach you their taste. It's the apprenticeship the guild in this volume's title refers to.



Each rung is granted by the people above it, for one thing: making their work easier. The patches you send are how they judge it.

Choosing a project

Contribute to something you use, so the bugs you find are real and your motivation survives the slow parts. Before your first patch, check that the project is alive and welcoming. Are pull requests answered within weeks, not months? Is there a `CONTRIBUTING` file, a code of conduct, a label such as "good first issue"? Read a few recent reviews: are the maintainers patient with newcomers, or curt? A large, famous project is not necessarily the best place to start. A mid-sized one where your patch will be read by a human this week teaches more.

Start below the code. Reproduce someone else's bug report and add the missing details. Answer a question in the issue tracker. Improve a confusing paragraph of the documentation. These cost maintainers almost nothing to accept, and they teach you how the project works and who decides what.

The maintainer's view

Most open-source projects are maintained by a few people, often in their spare time, with far more incoming work than time. A patch, however well meant, is a request for their attention: to read it, understand it, check it, and then maintain the result for years. The patches that get merged quickly are the ones that make that easy. Before reading a single line of the diff, a careful maintainer asks a series of quick questions, and the example automates them: does it apply to the current main branch, do the tests pass, does it add a test, does it keep the public interface, is it small enough to review, and does its message explain why? Some projects also require a sign-off line, the Developer Certificate of Origin, in which the author certifies they have the right to submit the code.

The example sets up a small project, a `slugify` function with an open issue: accented letters are dropped, so "Café Société" becomes `caf-soci-t`. Two contributors send patches, created with `git format-patch` and applied in a fresh clone with `git am`, which is how many projects, Git and the Linux kernel among them, take patches by email.

EXAMPLE

`review.py` · a small project, an open issue, two patches, and the eight checks a maintainer runs before reading either diff

```
"""The maintainer's side of open source: two patches arrive for a small project, and each gets
the
checks a careful maintainer runs before reading a single line of the diff."""
import ast, os, pathlib, re, subprocess, sys, tempfile

BASE = {"PATH": os.environ["PATH"], "HOME": "/home/maintainer", "LC_ALL": "C",
        "GIT_CONFIG_NOSYSTEM": "1", "GIT_CONFIG_GLOBAL": os.devnull}
PEOPLE = {"mira": ("Mira Okafor", "mira@example.org"), "theo": ("Theo Lind",
        "theo@example.org"),
        "sam": ("Sam Reyes", "sam@example.org")}
clock = [1780000000]

def git(who, *args, cwd):
    clock[0] += 60
    name, email = PEOPLE[who]
    env = dict(BASE, GIT_AUTHOR_NAME=name, GIT_AUTHOR_EMAIL=email, GIT_COMMITTER_NAME=name,
        GIT_COMMITTER_EMAIL=email, GIT_AUTHOR_DATE=f"@{clock[0]} +0000",
```

```

        GIT_COMMITTER_DATE=f"@{clock[0]} +0000")
    r = subprocess.run(["git", *args], cwd=cwd, env=env, capture_output=True, text=True)
    return r

SLUG = '''import re

def slugify(title):
    """'Hello, World!' -> 'hello-world'"""
    return "-".join(re.findall(r"[a-z0-9]+", title.lower()))
'''

TESTS = '''from slug import slugify

assert slugify("Hello, World!") == "hello-world"
assert slugify(" many spaces ") == "many-spaces"
'''

def commit(who, repo, files, message):
    for name, text in files.items():
        (repo / name).write_text(text)
    git(who, "add", *files, cwd=repo)
    git(who, "commit", "-q", "-m", message, cwd=repo)

def public_names(source):
    return {n.name for n in ast.parse(source).body
            if isinstance(n, (ast.FunctionDef, ast.ClassDef)) and not n.name.startswith("_")}

def review(gh, patch):
    """Apply the patch to a fresh clone of main and report on it."""
    work = gh / ("review-" + patch.stem)
    git("mira", "clone", "-q", str(gh / "upstream.git"), str(work), cwd=gh)
    before = public_names((work / "slug.py").read_text())
    applied = git("mira", "am", "-q", "--3way", str(patch), cwd=work).returncode == 0
    if not applied:
        return [("applies cleanly", False, "conflicts with main")]
    message = git("mira", "log", "-1", "--format=%B", cwd=work).stdout.strip()
    subject, _, body = message.partition("\n")
    stat = git("mira", "diff", "--numstat", "HEAD~1", cwd=work).stdout.split("\n")
    files = {}
    for line in filter(None, stat):
        # "added<TAB>removed<TAB>path"
        added, removed, path = line.split("\t")
        files[path] = (int(added), int(removed))
    tests = subprocess.run([sys.executable, "test_slug.py"], cwd=work, capture_output=True,
                           text=True)
    lost = before - public_names((work / "slug.py").read_text())
    first = subject.split()[0].lower()
    return [
        ("applies cleanly", True, ""),
        ("tests pass", tests.returncode == 0,
         re.sub(r" \(/.*\)$", "", tests.stderr.strip().splitlines()[-1]) if tests.returncode
         else ""),
        ("adds or changes a test", "test_slug.py" in files, ""),
        ("keeps the public API", not lost, f"removes {', '.join(sorted(lost))}" if lost else
         ""),
        ("small enough to review", sum(a + d for a, d in files.values()) <= 40,
         f"{sum(a for a, _ in files.values())} lines added, {sum(d for _, d in files.values())}
         removed"),
        ("subject: short, imperative, no full stop", len(subject) <= 50 and not
         subject.endswith("."),
         and not re.search(r"(ed|ing|es)$", first), repr(subject)),
        ("body says why", len(body.split()) >= 12, f"{len(body.split())} words"),
        ("signed off (DCO)", "Signed-off-by: " in message, ""),
    ]

with tempfile.TemporaryDirectory() as tmp:
    gh = pathlib.Path(tmp).resolve()

```

```

git("mira", "init", "-q", "--bare", "-b", "main", "upstream.git", cwd=gh)
git("mira", "clone", "-q", "upstream.git", "mira", cwd=gh)
git("mira", "switch", "-q", "-c", "main", cwd=gh / "mira")
commit("mira", gh / "mira", {"slug.py": SLUG, "test_slug.py": TESTS}, "Add slugify")
git("mira", "push", "-q", "origin", "main", cwd=gh / "mira")

# Issue 12: "Café Soci      becomes "caf-soci-t". Two people send patches.
for who in ("theo", "sam"):
    git(who, "clone", "-q", "upstream.git", who, cwd=gh)
    commit("theo", gh / "theo", {
        "slug.py": SLUG.replace("import re\n", "import re\nimport unicodedata\n").replace(
            '    return "-".join(re.findall(r"[a-z0-9]+', title.lower()))',
            '    folded = unicodedata.normalize("NFKD", title).encode("ascii",
                "ignore").decode()\n'
            '    return "-".join(re.findall(r"[a-z0-9]+', folded.lower()))'),
        "test_slug.py": TESTS + 'assert slugify("Caf   Soci     ") == "cafe-societe"    # issue
            12\n'},
        "Fold accents before making a slug\n\nAccented letters were dropped, so \"Caf  
            Soci     \" became \"
        \"caf-soci-t\" (issue 12). Decompose the title with NFKD and keep the ASCII part, so
            it \"
        \"becomes \"cafe-societe\".\n\nSigned-off-by: Theo Lind <theo@example.org>")
    commit("sam", gh / "sam", {
        "slug.py": "import re, unicodedata\n\n
            def make_slug(t):\n"
            "    t = unicodedata.normalize('NFKD', t)\n"
            "    t = t.encode('ascii', 'ignore').decode('ascii')\n"
            "    t = t.lower().strip()\n"
            "    t = re.sub(r'^[a-z0-9]+', '-', t)\n"
            "    return t\n",
        "fixed the accents bug.")
    for who in ("theo", "sam"):
        out = git(who, "format-patch", "-1", "-o", str(gh / "inbox"), cwd=gh /
            who).stdout.strip()
        patch = pathlib.Path(out)
        print(f"{PEOPLE[who][0]}s patch, {patch.name}")
        for check, ok, note in review(gh, patch):
            print(f" {ok} ' if ok else 'FAIL' {check:42} {note}\".rstrip())

```

OUTPUT

```
Theo Lind's patch, 0001-Fold-accent-before-making-a-slug.patch:
ok    applies cleanly
ok    tests pass
ok    adds or changes a test
ok    keeps the public API
ok    small enough to review          4 lines added, 1 removed
ok    subject: short, imperative, no full stop  'Fold accents before making a slug'
ok    body says why                    29 words
ok    signed off (DCO)

Sam Reyes's patch, 0001-fixed-the-accent-bug.patch:
ok    applies cleanly
FAIL  tests pass                      ImportError: cannot import name 'slugify'
      from 'slug'
FAIL  adds or changes a test
FAIL  keeps the public API            removes slugify
ok    small enough to review          7 lines added, 5 removed
FAIL  subject: short, imperative, no full stop  'fixed the accents bug.'
FAIL  body says why                   0 words
FAIL  signed off (DCO)
```

What the example shows

Theo's patch costs almost nothing to accept. Five changed lines, a regression test that names the issue, a subject in the imperative mood under fifty characters, a body that says what was wrong and why the fix works, and a sign-off. A maintainer can read it in a minute, see the test, and merge.

Sam's patch might even work, and it doesn't matter. It also handles accents. But it renames `slugify` to `make_slug`, which breaks every existing caller, so the existing tests fail on `import`. It adds no test, so nothing proves it fixes the issue. Its message, "fixed the accents bug.", says nothing a maintainer needs, and it has no sign-off. Each of these is fixable, and a kind maintainer will say so, but each round trip costs days. Contributors who send patches like Theo's get merged in hours and noticed; contributors who send patches like Sam's get asked for changes, and often drift away.

None of the checks looked at whether the code was good. They are the cheap, mechanical checks, and that's the point: by passing them, you spend the maintainer's attention only on the part that needs a human, the design.

From patches to review

The step from regular contributor to reviewer comes from doing reviewer's work before you're asked. Read other people's pull requests and leave useful comments: a failing case you tried, a question about a design choice, a pointer to a related issue. Triage new issues: reproduce them, label them, find the duplicates. Answer questions from users. Be consistent over months; maintainers notice who shows up every week. When your review comments start matching what the maintainers would have said, they'll start asking you to review, and eventually give you the permission to approve.

A good review, there as at work, is specific, kind and prioritised. Say what must change before merging and what's merely a suggestion; Conventional Comments' labels, such as *issue*, *suggestion* and *nitpick*, make the difference explicit. Explain why, so the contributor learns something. And review the design before the details: a perfect variable name in the wrong approach is wasted effort for everyone.

Being a good citizen

Maintainer burnout is the commonest way open-source projects die. Every issue that says "this is broken, fix it" without a reproduction, every complaint about the pace of a volunteer project, every unreviewable thousand-line patch, adds to it. Be the opposite. Say thank you. Close your own stale pull requests. If your company depends on a project, say so, and ask whether it can sponsor the maintainers or give engineers time to help. A project where several companies' engineers share the maintenance survives the departure of any one person; a project with a single exhausted maintainer, however popular, does not.

COMMON TRAPS

A first contribution that's a large feature, unannounced, in a project you don't know yet.

Mixing a fix with reformatting or renaming, so the real change is buried in the diff.

No test, so nothing proves the fix, and nothing stops the bug returning.

A message that says what, not why, or nothing at all.

Treating maintainers as a support desk, when most are volunteers with little time.

EXERCISES

EXERCISE 5.1 Rescue Sam's patch

Rewrite Sam's commit so that it passes all eight checks, keeping Sam's approach to the fix.

SOLUTION

Keep the name `slugify`, so existing callers and tests work, and make the change only inside it. Sam's version also behaves differently from the original on leading and trailing punctuation, "--hi--" becomes `-hi-`, so add `.strip("-")` to keep the old behaviour. Add the regression test from the issue, `assert slugify("Café Société") == "cafe-societe"`. Then write a message: a subject such as "Fold accents before making a slug", a body that says accented letters were dropped and that NFKD decomposition keeps their ASCII base letters, a reference to issue 12, and a `Signed-off-by` line, added with `git commit -s`. If Sam genuinely wanted a new name, that's a separate proposal: open an issue first, and if it's accepted, add `make_slug` alongside `slugify` and deprecate the old name over a release or two, rather than breaking every caller in a bug fix.

EXERCISE 5.2 Triage

Three new issues arrive: "doesn't work", with no details; "slugify('ß') returns empty string"; and "please add support for Chinese titles". How would you respond to each, as a helpful triager?

SOLUTION

For "doesn't work", ask kindly for what's needed to act: the version, the exact input, the output they got and the output they expected, and label it as needing information; close it after a few weeks if nothing comes. For "ß", reproduce it first: NFKD doesn't decompose "ß" into ASCII, so

it's dropped and the slug is empty. Confirm it in a comment with the one-line reproduction, label it as a bug, and note the design question it raises: should "ß" become "ss", and what about other letters with no ASCII decomposition? That's exactly the kind of comment that saves a maintainer an hour. For Chinese titles, recognise it as a feature request with a real design question, transliteration or keeping the characters, and label it so; link any related discussion, and leave the decision to the maintainers rather than promising anything. In each case you've moved the issue forward without needing any special permission, which is what triagers do.

EXERCISE 5.3 Twelve weeks

Plan twelve weeks, at three hours a week, to go from user to regular contributor in an open-source project you depend on.

SOLUTION

Weeks 1 and 2: read the contributing guide, build the project from source, run its test suite, and read the last twenty merged pull requests and their reviews, noting what the maintainers ask for. Weeks 3 and 4: triage, reproducing three open bug reports and adding what you find. Weeks 5 to 7: your first patch, a small bug you've hit or a "good first issue", with a test and a clear message; respond quickly to the review. Weeks 8 and 9: a second patch in the same area, so you start to know one part of the code well. Weeks 10 to 12: review two other contributors' pull requests in that area, carefully and kindly, and write up a map of the part of the codebase you now know, offering it as documentation. By the end you're a name the maintainers recognise, in one part of the code, which is how the next rung starts.

CHECKLIST

- ☐ Have you chosen a project you use, and checked that it's alive and welcoming?
- ☐ Do your patches apply cleanly, pass the tests, add a test and explain why?
- ☐ Do you keep fixes separate from refactors and renames?
- ☐ Do you triage issues and review others' pull requests, not only send your own?
- ☐ Can you write review comments that are specific, kind and prioritised?

GO DEEPER

[OUTRE](#) [How to Contribute to Open Source](#) [GitHub Open Source Guides](#)

GUIDE **How to Contribute to Open Source** GitHub's Open Source Guides

BOOK **Producing Open Source Software** Karl Fogel

BOOK **Pro Git: Contributing to a Project** Chacon & Straub

Design documents

3 exercises

A design document is how a senior engineer changes a system before writing the code: by thinking it through in writing, and letting sceptical colleagues find the flaws while they're cheap. A good one says what it proposes in its first paragraph, puts numbers on its goals, admits what it won't do, shows the alternatives it rejected and why, and says how it will roll out and roll back. Most weak documents fail the same few checks, and this chapter automates them.

What a design document is for

A design document does three jobs. It's a tool for **thinking**: writing the design down forces the vague parts into the open, and many designs change, or die, while the document is being written. It's a tool for **agreement**: the people whose systems the change touches can object before a line of code depends on the design, when changing it costs an afternoon rather than a quarter. And it's a **record**: a year later, when someone asks why the system works this way, the document answers, alternatives and all.

Not every change needs one. A design document earns its cost when a change is hard to reverse, crosses team boundaries, or has several plausible approaches. A one-week change inside your own service, with an obvious design, needs a good pull request description instead. The test is simple: if someone could reasonably say "why didn't you just....?" after it's built, write the document first.

The shape of a good one

Templates vary between companies, but the strong ones share a shape, because each section answers a question a reviewer will otherwise ask.

SECTION	THE REVIEWER'S QUESTION IT ANSWERS
First paragraph	What are you proposing, in two sentences?
Context	Why now, and what's the evidence?
Goals and non-goals	How will we know it worked? What's out of scope?
Proposal	How does it work?
Alternatives considered	Why not the obvious others, or doing nothing?
Risks	What could go wrong, and what then?
Rollout	How does it ship, and how do we undo it?
Open questions	What's still undecided?

Every section exists because a reviewer will ask its question anyway. Answer them in the document and the review discusses the design instead.

Two sections do most of the work and are most often missing. **Non-goals** prevent the scope argument: "personalised ranking is a separate project" stops a reviewer asking why the design doesn't handle it, and stops the project growing to include it. **Alternatives considered** is where a document earns trust. A proposal with no alternatives reads as the first idea its author had; one that shows two alternatives, each tested or priced, and explains why it lost, reads as a decision. Always include doing nothing, with its cost, because it's the alternative a sceptical manager is weighing.

Writing for sceptical readers

The syllabus asks for documents that "survive contact with sceptical senior engineers". Sceptics aren't obstacles; they're the cheapest testing your design will ever get. Write for them. Put the proposal in the first paragraph, so nobody has to read four pages to find out what you want. Replace adjectives with numbers: not "search is slow" but "p95 is 2.4 seconds at 40 queries a second". State your assumptions where they can be attacked. And anticipate the obvious objection in the text: if everyone will ask "why not just add an index?", answer it in the alternatives section before they do.

Keep it short. A design document that takes an hour to read gets skimmed; one that takes fifteen minutes gets read. Move detail into appendices, and link to the prototype rather than describing it. Diagrams help when they show the change, the new box and the arrows that move, not the whole system.

A first-pass reviewer

Most weak documents fail the same mechanical checks, long before anyone argues with the design: no stated proposal, goals without numbers, adjectives where measurements should be, no alternatives, risks with no mitigation, a rollout with no way back. The example runs

those checks on two drafts of the same document, a search migration, the first as a first draft often looks and the second after revision.

EXAMPLE

designdoc.py · the questions a sceptical reviewer asks first, run on a weak draft and a revised one

```
"""A first-pass reviewer for design documents: the questions a sceptical senior engineer asks before arguing with the substance. It doesn't judge the design; it finds what's missing."""
```

```
import re
import textwrap
```

```
WEAK = """# Search improvements
```

```
## Background
```

```
Search is slow and users don't like it. We should make it much faster and more scalable.
```

```
## Proposal
```

```
We will move search to Elasticsearch, which is robust and should be easy to run.
```

```
This will significantly improve the experience and make search seamless.
```

```
"""
```

```
STRONG = """# Move product search to a dedicated index
```

```
We propose moving product search from Postgres full-text queries to a dedicated OpenSearch index, fed from the existing change stream. Prototypes bring p95 latency from 2.4 s to 240 ms.
```

```
## Context
```

```
Search p95 is 2.4 s at today's 40 queries a second, and support logs about ten complaints a week.
```

```
The slow queries are prefix and typo-tolerant matches, which Postgres serves with sequential scans.
```

```
## Goals
```

- p95 search latency under 300 ms at 120 queries a second, three times today's peak.
- New and changed products searchable within 60 s of the write.

```
## Non-goals
```

- Personalised ranking. It is a separate project, and nothing here prevents it.
- Replacing Postgres as the source of truth for products.

```
## Proposal
```

```
A consumer reads product changes from the existing change stream and writes them to the index. Search requests go to the index; product pages still read from Postgres.
```

```
## Alternatives considered
```

- Tune Postgres with trigram indexes: p95 fell to 900 ms in a test, short of the goal.
- A managed search service: meets the goal, but at \$4,100 a month against \$900 for self-hosting.
- Do nothing: complaints are growing about 8% a month with traffic.

```
## Risks
```

- The index falls behind the database. Mitigation: alert when lag exceeds 60 s; fall back to Postgres.
- Cost grows with the catalogue. Mitigation: a 30-day review of node sizes after launch.

```
## Rollout
```

```
Shadow traffic for a week and compare results, then 5%, 50% and 100% behind a flag, with the flag as the rollback.
```

```
## Open questions
```

- Who owns the index on call: the search team or the platform team?

```

"""

REQUIRED = ["context", "goals", "non-goals", "proposal", "alternatives", "risks", "rollout"]
VAGUE = (r"\b(fast(er)?|slow|scalable|robust|easy|simple|significant(ly)?|seamless(ly)?"
        r"|better|improve[sd]?)\b)")

def sections(doc):
    parts, current = {"_lead": []}, "_lead"
    for line in doc.splitlines():
        if line.startswith("## "):
            current = line[3:].strip().lower().split()[0]
            parts[current] = []
        elif not line.startswith("# "):
            parts[current].append(line)
    return {k: "\n".join(v).strip() for k, v in parts.items()}

def review(doc):
    s, notes = sections(doc), []
    missing = [r for r in REQUIRED if r not in s]
    if missing:
        notes.append("missing sections: " + ", ".join(missing))
    lead = s["_lead"] or s.get("background", "")
    if not re.search(r"\b(we propose|we recommend|this document proposes)\b", lead, re.I):
        notes.append("the first paragraph doesn't say what is being proposed")
    for goal in re.findall(r"^(.*)$", s.get("goals", ""), re.M):
        if not re.search(r"\d", goal):
            notes.append(f"goal without a number: {goal!r}")
    prose = " ".join(line for line in doc.splitlines() if not line.startswith("#"))
    for sentence in re.split(r"(?<=[!?])\s+", prose):
        words = dict.fromkeys(m.group(0).lower() for m in re.finditer(VAGUE, sentence, re.I))
        if words and not re.search(r"\d", sentence):
            quote = textwrap.shorten(sentence.strip(), 64, placeholder=" ...")
            notes.append(f'{', '.join(words)} with no number beside it: {quote!r}')
    alternatives = re.findall(r"^(.*)", s.get("alternatives", ""), re.M)
    if len(alternatives) < 2:
        notes.append(f'{len(alternatives)} alternatives considered; reviewers will ask about at least two, "
            "and about doing nothing")
    for risk in re.findall(r"^(.*)$", s.get("risks", ""), re.M):
        if "mitigation" not in risk.lower():
            notes.append(f"risk without a mitigation: {risk!r}")
    if "rollout" in s and not re.search(r"rollback|roll back|revert|flag", s["rollout"], re.I):
        notes.append("the rollout has no way back")
    if "open" not in s:
        notes.append("no open questions: either nothing is unsettled, or it isn't written down")
    return notes

for name, doc in [("weak draft", WEAK), ("revised draft", STRONG)]:
    notes = review(doc)
    print(f'{name}: {len(doc.split())} words, {len(notes)} review note{s' * (len(notes) != 1)}s')
    for n in notes:
        print("  - " + n)
    print()

```

OUTPUT

```

weak draft: 49 words, 8 review notes
  - missing sections: context, goals, non-goals, alternatives, risks, rollout
  - the first paragraph doesn't say what is being proposed
  - slow with no number beside it: "Search is slow and users don't like it."
  - faster, scalable with no number beside it: 'We should make it much faster and more

```

```
scalable.'
- robust, easy with no number beside it: 'We will move search to Elasticsearch, which is
  robust and ...'
- significantly, improve, seamless with no number beside it: 'This will significantly improve
  the experience and make ...'
- 0 alternatives considered; reviewers will ask about at least two, and about doing nothing
- no open questions: either nothing is unsettled, or it isn't written down

revised draft: 284 words, 1 review note
- slow with no number beside it: 'The slow queries are prefix and typo-tolerant matches,
  which ...'
```

What the example shows

The weak draft fails before the design is discussed. Forty-nine words produce eight notes: six missing sections, no stated proposal, and four sentences whose only content is adjectives: slow, faster, scalable, robust, easy, significantly, seamless. A reviewer handed this document has nothing to agree or disagree with. "Make search much faster" can't be wrong, which is exactly why it isn't useful.

The revised draft gives the reviewer something to argue with. It states the proposal and its headline number in the first paragraph, gives a measurable goal three times today's peak load, rules two things out of scope, prices two alternatives and the cost of doing nothing, gives each risk a mitigation, and has a rollout whose rollback is a flag. Its one note is a false alarm: "the slow queries are prefix matches" uses "slow" to identify which queries, not to make a claim. That's worth remembering. A check like this is a checklist, not a judge. It finds what's missing; it can't tell you whether the design is good. That's what the sceptical humans are for.

Running the review

Send the document to the right readers, not everyone: the owners of every system it touches, someone who will use the result, and at least one person you expect to disagree. Give them a few days to comment in the document, where each objection gets a thread and an answer. Hold a meeting only if threads remain unresolved, and spend it only on those. Record the decision and its date at the top, so readers know whether they're reading a proposal or a plan.

Decisions outlive their documents, and documents go stale as designs change during building. Two habits help. Update the document when the design changes, or mark it superseded with a link to its replacement. And for the individual decisions inside it, keep short architecture decision records, the format Michael Nygard proposed: a title, the context, the decision and its consequences, one decision per file, numbered, never edited after acceptance, only superseded. A folder of them is the history of why your system is the way it is.

Surviving contact

The hardest part of a design review isn't technical. When a reviewer finds a real flaw, the right response is thanks, because they just saved you weeks. Separate the design from yourself: the goal is the best design, not your design. Change your mind visibly when the evidence says so, and say what changed it; colleagues trust engineers who can be persuaded far more than ones who can't. And when a decision goes against you after a fair hearing, commit to it fully. "Disagree and commit" works only if the commitment is real, and nothing damages a team's trust faster than someone quietly building the design they lost.

COMMON TRAPS

Burying the proposal on page four, after the history of the problem.

Goals without numbers, so nobody can tell later whether it worked.

No alternatives, or straw-man alternatives built only to lose.

A rollout with no rollback, for a change that will need one.

Defending the design instead of testing it, and treating sceptics as opponents.

EXERCISES

EXERCISE 6.1 Goals with numbers

The weak draft's only goal is "make it much faster and more scalable". Rewrite it as two or three goals a reviewer could check after launch.

SOLUTION

- p95 search latency under 300 ms, measured over a week at peak traffic.
- Sustain 120 queries a second, three times today's peak, with p95 still under 300 ms in a load test.
- Product changes appear in search results within 60 seconds of the write.

Each goal has a metric, a threshold and a way to measure it, so after launch it's simply true or false. "Faster" became a latency at a percentile; "scalable" became a throughput with the latency still holding, which is what scalability actually means for a user. The freshness goal wasn't in the original at all; writing measurable goals often exposes requirements nobody had stated, and freshness is exactly the property a search index can quietly break.

EXERCISE 6.2 The alternatives section

Write the alternatives section the weak draft lacks, for a proposal to move search to Elasticsearch.

SOLUTION

```
## Alternatives considered
- Do nothing. Complaints grow with traffic, about 8% a month; at that rate,
  p95 passes 4 s within a year.
- Tune the existing Postgres queries with trigram indexes. A one-day test cut p95
  to 900 ms: a real improvement, but three times the goal, and it doesn't help
  typo tolerance.
- A managed search service. Meets every goal with the least operational work, at
  about $4,100 a month against $900 for running our own; worth revisiting if the
  search index's on-call load turns out higher than we expect.
```

Each alternative is real, measured or priced, and says why it lost, and the last one even says what would make it win. That's what turns "we will move to Elasticsearch" into a decision a reviewer can evaluate. Note that the cheapest alternative, tuning Postgres, was actually tried. A one-day experiment that rules out the obvious alternative is worth more to a document than any amount of argument.

EXERCISE 6.3 Answer the sceptic

A senior reviewer comments on the revised draft: "This is a lot of new infrastructure. Have you profiled the slow queries? I suspect an index fixes most of it." Write the reply.

SOLUTION

Thank them, and answer with evidence: "Good question, and you're partly right. We profiled the twenty slowest query shapes last month. Eight were missing an index; we added those and p95 for exact matches fell to 180 ms, which is already shipped. The rest are prefix and typo-tolerant matches, and the best index we found, trigram, brings them to 900 ms, still three times the goal. The numbers are in the alternatives section and the profile is linked there. If you see something we missed in it, I'd much rather find out now." The reply concedes what's true, shows the work, points at the evidence, and invites a further challenge. It treats the reviewer as a collaborator testing the design, which is what they are. And if the profile hadn't been done, the right reply is "no, and we should; I'll do it before we go further."

CHECKLIST

- ☐ Does your document say what it proposes in its first paragraph?
- ☐ Are its goals measurable, and its non-goals stated?
- ☐ Does it show real alternatives, including doing nothing, and why each lost?
- ☐ Does every risk have a mitigation, and the rollout a way back?
- ☐ Do you send it to someone you expect to disagree, and thank them when they do?

GO DEEPER

ESSAY Design Docs at Google [Malte Uhl](#)

ESSAY Documenting Architecture Decisions [Michael Nygard](#)

COURSE Technical Writing Google for Developers

Trade-offs, argued both ways

3 exercises

Taste in trade-offs means being able to argue both sides honestly, and then knowing which fact would settle the argument. Write the uncertain inputs as ranges, not single guesses, run the arithmetic over many possible futures, and look for the assumption that swings the answer. That assumption, not the conclusion, is what the debate should be about. And spend that care where a decision is hard to reverse; most aren't.

Every option costs something

An engineering decision with no downside isn't a decision; it's a chore. The real ones trade something for something: latency for consistency, flexibility for simplicity, money for operational burden, speed now for change later. Juniors argue for the option they like. Seniors can state the best case for each option, including the ones they'd vote against, and say under what conditions each would win. The syllabus calls this "taste in trade-offs, which means being able to argue both sides honestly", and it's a learnable skill with a method.

The method starts by noticing that most trade-off arguments are disguised disagreements about facts. Two engineers arguing "managed service or run our own" usually agree on the costs once they're written down. What they disagree about is a number neither has stated: how fast traffic will grow, how much time running the system would take. Make those numbers explicit, as ranges, and the argument changes from opinion to evidence.

Ranges, not points

A single estimate, "it'll take a quarter of an engineer to run", hides how unsure you are. A range, "somewhere between 15% and 60% of an engineer", states it, and invites the right question: what would narrow it? Once every uncertain input is a range, the arithmetic is simple: draw one value of each input from its range, compute the cost of each option, and repeat thousands of times. Each repetition is one possible future. The result isn't one number but a distribution: how often each option wins, and by how much.

This is called Monte Carlo simulation, and it sounds grander than it is. It's twenty lines of Python and a few honest ranges. Douglas Hubbard's *How to Measure Anything* argues that the ranges themselves are the valuable part: people asked for a 90% range on a quantity they "can't estimate" usually give a perfectly usable one, and the act of giving it shows which inputs are genuinely uncertain.

Find the assumption that decides it

Not every uncertainty matters equally. Some inputs, however unsure, barely move the answer; one or two decide it. To find them, split the simulated futures by each input, the half where it was low and the half where it was high, and see how much the answer changes between the halves. The input with the biggest swing is the one to argue about, and the one worth paying to measure. Hubbard calls this the value of information: measure what could change the decision, not what's easy to measure.

The example applies this to a familiar choice: three years of running your own event-streaming cluster against a managed service, with traffic growth, the share of an engineer spent operating it, the number of serious incidents and their cost all given as ranges.

EXAMPLE

tradeoff.py · three years of running our own cluster against a managed service, over twenty thousand possible futures, and which assumption decides it

```
"""Arguing a trade-off both ways: three years of running our own event streaming cluster
against a
managed service, with the uncertain inputs as ranges instead of single guesses. The question
isn't
only which is cheaper, but which assumption decides it."""
import random
import statistics

ENGINEER_MONTH = 16_000          # loaded cost of one engineer for a month
rng = random.Random(42)

def sample():
    """One possible future: every uncertain input drawn from its range."""
    return {
        "growth": rng.uniform(1.0, 3.0),          # traffic multiplier per year
        "ops_fte": rng.uniform(0.15, 0.6),        # share of an engineer spent running it
                                                    ourselves
        "incidents": rng.uniform(1, 6),           # serious incidents a year if we run it
        "incident_cost": rng.lognormvariate(9.9, 0.6), # median about $20,000 each
    }

def three_years(f):
    traffic = [f["growth"] ** (m / 12) for m in range(36)] # 1.0 = today's traffic
    own = (2 * ENGINEER_MONTH                               # setting it up
           + sum(1_800 + 900 * t for t in traffic)          # machines and storage
           + 36 * f["ops_fte"] * ENGINEER_MONTH            # running it
           + 3 * f["incidents"] * f["incident_cost"])
    managed = (0.5 * ENGINEER_MONTH                         # migrating to it
              + sum(2_600 + 2_400 * t for t in traffic)    # priced by throughput
              + 36 * 0.05 * ENGINEER_MONTH                # a little of someone's
                  time                                       #
              + 3 * 1 * f["incident_cost"])                # the provider's
                  outages hit us too
    return own, managed

futures = [sample() for _ in range(20_000)]
costs = [three_years(f) for f in futures]
pct = lambda xs, p: sorted(xs)[int(p * (len(xs) - 1))]
for i, name in enumerate(("run our own", "managed")):
    xs = [c[i] for c in costs]
    k = lambda p: f"${pct(xs, p) / 1e3:,.0f}k"
```

```

print(f"{name:12} 3-year cost: median {k(.5)} (80% between {k(.1)} "
      f"and {k(.9)})")
managed_wins = [c[1] < c[0] for c in costs]
print(f"managed is cheaper in {sum(managed_wins) / len(costs):.0%} of 20,000 simulated
      futures")

# Which assumption decides it? Split the futures on each input, and see how far the answer
# swings.
print("\nhow much each assumption moves the answer (share of futures where managed is
      cheaper):")
swings = []
for key in futures[0]:
    cut = statistics.median(f[key] for f in futures)
    low = [w for f, w in zip(futures, managed_wins) if f[key] <= cut]
    high = [w for f, w in zip(futures, managed_wins) if f[key] > cut]
    lo, hi = sum(low) / len(low), sum(high) / len(high)
    swings.append((abs(hi - lo), key, lo, hi))
for swing, key, lo, hi in sorted(swings, reverse=True):
    print(f" {key:14} low half {lo:4.0%} high half {hi:4.0%} swing {swing * 100:3.0f}
          points")

# The two that matter most, as a table: what would have to be true for running our own to win?
print("\nwhich is cheaper, by traffic growth and time spent running it (other inputs at their
      medians):")
print(f" {'':19}" + "".join(f"{'f'}{x} FTE':>10}" for x in (0.2, 0.3, 0.4, 0.5)))
for growth in (1.0, 1.5, 2.0, 2.5, 3.0):
    row = []
    for fte in (0.2, 0.3, 0.4, 0.5):
        future = {"growth": growth, "ops_fte": fte, "incidents": 3.5, "incident_cost": 20_000}
        own, managed = three_years(future)
        row.append("managed" if managed < own else "own")
    print(f" {'f'traffic x{growth:.1f}/year':19}" + "".join(f"{'r':>10}" for r in row))

```

OUTPUT

```

run our own 3-year cost: median $638k (80% between $445k and $955k)
managed      3-year cost: median $488k (80% between $306k and $765k)
managed is cheaper in 79% of 20,000 simulated futures

```

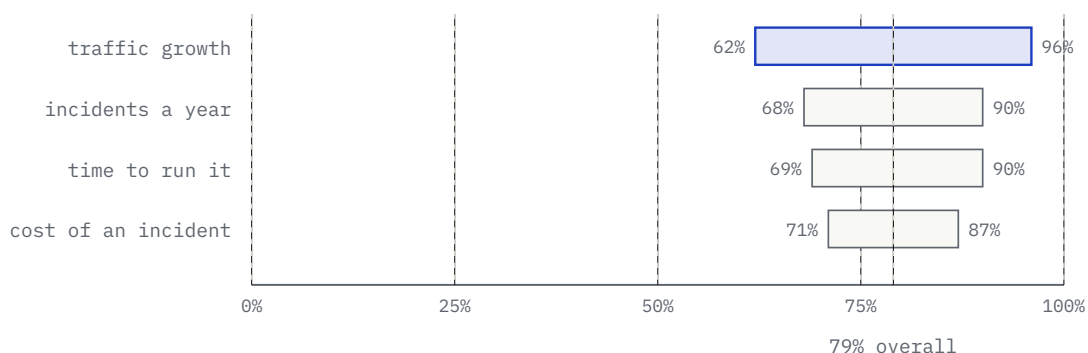
how much each assumption moves the answer (share of futures where managed is cheaper):

growth	low half	96%	high half	62%	swing	34 points
incidents	low half	68%	high half	90%	swing	22 points
ops_fte	low half	69%	high half	90%	swing	21 points
incident_cost	low half	71%	high half	87%	swing	16 points

which is cheaper, by traffic growth and time spent running it (other inputs at their medians):

	0.2 FTE	0.3 FTE	0.4 FTE	0.5 FTE
traffic x1.0/year	managed	managed	managed	managed
traffic x1.5/year	managed	managed	managed	managed
traffic x2.0/year	managed	managed	managed	managed
traffic x2.5/year	own	managed	managed	managed
traffic x3.0/year	own	own	own	own

SHARE OF FUTURES WHERE MANAGED IS CHEAPER



A tornado chart: the widest bar is the assumption that moves the answer most. Here it's traffic growth, so that's the forecast to scrutinise before deciding.

What the example shows

The answer is **"managed, probably"**. The managed service is cheaper in 79% of the simulated futures, with a median three-year cost of \$488,000 against \$638,000. That's a useful answer, and a more honest one than a single number, because it admits a one-in-five chance of being wrong.

Growth decides it. When traffic growth was in the low half of its range, managed won 96% of the time; in the high half, only 62%. No other input swung the answer by more than 22 points. The table then shows the boundary: running your own wins only if traffic grows two and a half times a year or more and operating it takes little time, or at three times a year. So the right conversation isn't "managed or our own?" in the abstract. It's "how confident are we that traffic will more than double every year for three years?" That's a question the product and finance teams can help answer, which turns an engineering argument into a shared one.

The model is simple on purpose. It leaves out lock-in, the cost of migrating away later, and the learning that running your own brings. Leaving things out is fine if you say so, and the omissions are a good place for the other side's case to start.

Arguing the other side

Two techniques make it practical. The first is to **steelman**: write the strongest case for the option you're against, as its best advocate would, before arguing against it. If you can't write a paragraph your opponent would sign, you don't yet understand the trade-off. The second is the **pre-mortem**, from the psychologist Gary Klein: imagine it's a year later and your chosen option has failed badly, then write down why. People who struggle to criticise a plan they've just agreed to find it easy to explain a failure that has "already happened", and the reasons they give are the risks to plan for.

One-way and two-way doors

In his 2015 letter to shareholders, Jeff Bezos distinguished two kinds of decision. Some are one-way doors: hard or impossible to reverse, such as choosing a database for a system that will hold years of data, or publishing an API others will build on. These deserve slow, careful analysis. Most decisions are two-way doors: if you're wrong, you walk back through. Treating every decision as a one-way door is how organisations become slow. Part of taste is knowing which kind you face. The managed-service choice is closer to a two-way door than it looks, if you keep your code behind an interface and your data exportable, and making a decision more reversible is often cheaper than analysing it more carefully.

COMMON TRAPS

Arguing conclusions when the disagreement is really about an unstated number.

Single-point estimates that hide how uncertain each input is.

Measuring what's easy instead of the assumption that decides the answer.

Straw-manning the other option, so the comparison proves nothing.

Treating every decision as irreversible, and analysing two-way doors for weeks.

EXERCISES

EXERCISE 7.1 If growth were known

Product says traffic will grow between 1.0 and 1.5 times a year. Change the growth range in the example. What happens to the answer and to the swings, and what does that tell you about the decision?

SOLUTION

```
run our own 3-year cost: median $562k (80% between $388k and $876k)
managed     3-year cost: median $317k (80% between $269k and $389k)
managed is cheaper in 99% of 20,000 simulated futures
(every assumption now swings the answer by 1 point or less)
```

With growth pinned to the slow end, the managed service wins in 99% of futures, and no other assumption matters: operating time and incidents, which looked important before, now swing the answer by a point. Narrowing the one input that decided the question settled it. That's the value of information in practice: one conversation with product, about one number, was worth more than any further engineering analysis. If they'd said 2.5 to 3.5 times a year instead,

managed would win in only a third of futures, and running your own would become the default. Either way, the forecast, not the technology, made the decision.

EXERCISE 7.2 Steelman the loser

Write the strongest one-paragraph case for running your own cluster, as someone who believes in it would.

SOLUTION

"Event streaming is becoming the backbone of our platform, and we'll depend on it for a decade. The managed service's price scales with throughput, so our bill grows with our success, and our traffic doubled last year. Running it ourselves costs more now and less later, and the crossover comes sooner than the model's median suggests if growth continues. It also builds expertise we'll need anyway: when something goes wrong in the managed service at 3 a.m., we're waiting on someone else's support queue, unable to see inside it. And the model leaves out lock-in: the longer we use a provider's proprietary features, the more expensive leaving becomes. We should pay the learning cost while we're small." A good steelman uses the model's own blind spots, growth, lock-in and control, rather than ignoring the model. Writing it shows exactly which assumptions the managed option depends on, and those become the questions to check before committing.

EXERCISE 7.3 A pre-mortem

It's two years later, and the move to the managed service is widely seen as a mistake. Write the three most likely reasons, and what you'd do now about each.

SOLUTION

First, the bill grew faster than the model said, because traffic tripled after a new product launch. Now: set a cost alert at twice the projected monthly bill, and agree in advance the traffic level at which we'd re-run the analysis. Second, we built on the provider's proprietary connectors, and leaving would take a quarter. Now: put a thin interface between our services and the stream, use only the open protocol, and test an export once a year. Third, a provider outage took us down for a day, and we couldn't do anything but wait. Now: decide which consumers must keep working during an outage, buffer their input locally, and read the provider's incident history before signing. Each reason

became a cheap precaution taken today, which is what a pre-mortem is for. None of them argues for not choosing the managed service; they make the choice safer and more reversible.

CHECKLIST

- ☐ Do you write uncertain inputs as ranges, and say how sure you are?
- ☐ Can you find which assumption swings a decision, and focus the argument there?
- ☐ Can you steelman the option you're against?
- ☐ Do you run pre-mortems on important decisions?
- ☐ Do you tell one-way doors from two-way doors, and make decisions more reversible when you can?

GO DEEPER

BOOK

How to Measure Anything Douglas Hubbard

ESSAY

Performing a Project Premortem Gary Klein, Harvard Business Review

LETTER

2015 Letter to Shareholders Jeff Bezos

Writing in public

3 exercises

A body of written work compounds more than any certificate: every post that explains something well keeps working, for readers you'll never meet, for years. Write about what you learned the hard way, in your area, regularly. And write so it gets read: lead with the point, prefer concrete facts to adjectives, cut the hedges and the throat-clearing, and say who did what. Most of those habits can be checked mechanically, and the example checks them.

Why write where others can read it

Writing in public does four things at once. It **clarifies**: an explanation you've had to write down, for readers who can't interrupt, is one you actually understand, and the gaps show while you write. It **compounds**: a good post on why your migration locked a table answers the same question for hundreds of engineers who search for it later, without any further effort from you. It **builds a reputation** in your area that no job title gives: when people in your field know your name from your writing, opportunities find you. And it **helps people**, which is reason enough.

The syllabus asks for "a written archive others cite". That's a high bar on purpose. It doesn't mean a viral post. It means years of steady, useful writing in one area, until other people link to it when they explain the subject. That's achievable for anyone who writes something worthwhile every month or so and keeps going.

What to write

The best material is whatever cost you the most to learn. The incident that took a day to diagnose, the migration that went wrong, the library behaviour that wasn't documented, the paper you finally understood by building it: each is a post that saves someone else the same day. Explanations of your area, written at the level of a capable engineer who doesn't know it yet, are always needed. Notes on what you read, the one-page summaries chapter 2 suggested, are publishable with a little polish. And small posts count. Simon Willison's habit of publishing short "today I learned" notes, hundreds of them, built one of the most useful archives in the field. A post doesn't need to be long; it needs to be true, specific and findable.

Check what you can share. Anything about your employer's systems needs their permission, and many companies have a review process for engineering posts. Incidents can usually be described without customer data or confidential numbers, and the lesson rarely depends on them.

Writing that gets read

Readers decide in a sentence or two whether to keep going. Put the most important thing first, the way a news story does: what happened, or what the reader will learn. Use specific facts instead of adjectives: "locked the orders table for eleven minutes" rather than "caused some issues". Keep sentences short enough to read in one breath. Say who did what: "we ran the migration", not "the migration was run", because the passive voice hides the actor, and in technical writing the actor is usually the point. Cut the hedges, "I think", "perhaps", "quite", "basically", which make a writer sound unsure without making them more accurate. And cut the throat-clearing, "In this post I want to talk about", which spends the reader's patience before saying anything.

These are habits, so they can be checked. The example measures a draft opening of a blog post about a migration that went wrong, and its rewrite: sentence length, a readability score, and the hedges, throat-clearing and passive constructions in each.

EXAMPLE

prose.py · the mechanical habits that make technical writing hard to read, measured on a draft opening and its rewrite

```
"""A prose checker for writing in public: the mechanical habits that make technical writing
    hard to
    read, measured on the opening of a blog post before and after editing."""
import re
import statistics

BEFORE = """In this post I want to talk a little bit about something that I think is quite
    important,
    which is the way that database migrations are handled by many teams. It is important to note
    that
    mistakes were made by us in the past, and it could perhaps be argued that some of them were
    really
    avoidable. Basically, the migration was run during the day and the table was locked by it for
    eleven minutes, which was obviously not ideal for our users."""

AFTER = """A migration locked our orders table for eleven minutes on a Tuesday afternoon.
    Nobody could
    check out. The fix took one line, and the lesson took longer: on a large table, adding a column
    with
    a default rewrote every row while holding the lock. Here is what we changed, and the three
    rules we
    now follow for every migration."""

HEDGES = (r"\b(i think|perhaps|maybe|might|could|somewhat|quite|very|really|just|basically"
           r"|obviously|a little bit)\b")
OPENERS = r"\b(in this post|it is important to note|i want to talk|needless to say|at the end
           of the day)\b"
PASSIVE = r"\b(?:is|are|was|were|be|been|being)\s+(?:\w+ly\s+)?
           (\w+ed|made|run|done|held|taken|written)\b"

def syllables(word):
    groups = re.findall(r"[aeiouy]+", word.lower())
    n = len(groups) - (word.lower().endswith("e") and len(groups) > 1)
    return max(1, n)

def check(text):
```

```

text = " ".join(text.split())
sentences = [s for s in re.split(r"(?<=[.!?])\s+", text) if s]
words = re.findall(r"[A-Za-z']+", text)
lengths = [len(re.findall(r"[A-Za-z']+", s)) for s in sentences]
flesch = 206.835 - 1.015 * len(words) / len(sentences) - 84.6 * sum(map(syllables, words))
/ len(words)
return {
    "sentences": len(sentences),
    "words per sentence": round(statistics.mean(lengths), 1),
    "longest sentence": max(lengths),
    "reading ease": round(flesch),
    "hedges": [m.group(0) for m in re.finditer(HEDGES, text, re.I)],
    "throat-clearing": [m.group(0) for m in re.finditer(OPENERS, text, re.I)],
    "passive": [m.group(0) for m in re.finditer(PASSIVE, text, re.I)],
    "first sentence": sentences[0],
}

before, after = check(BEFORE), check(AFTER)
print(f"{'':22}{'before':>10}{'after':>10}")
for key in ("sentences", "words per sentence", "longest sentence", "reading ease"):
    print(f"{'key':22}{before[key]:>10}{after[key]:>10}")
for key in ("hedges", "throat-clearing", "passive"):
    print(f"{'key':22}{len(before[key]):>10}{len(after[key]):>10}")
print("\nflagged in the first draft:")
for key in ("throat-clearing", "hedges", "passive"):
    print(f"  {key}: " + " ".join(before[key]))
print("\nwhat each version's first sentence promises the reader:")
print("  before:", before["first sentence"][:88] + " ...")
print("  after: ", after["first sentence"])

```

OUTPUT

	before	after
sentences	3	4
words per sentence	27.7	14.8
longest sentence	30	27
reading ease	58	66
hedges	8	1
throat-clearing	3	0
passive	5	0

flagged in the first draft:

throat-clearing: In this post; I want to talk; It is important to note
hedges: a little bit; I think; quite; could; perhaps; really; Basically; obviously
passive: are handled; were made; be argued; was run; was locked

what each version's first sentence promises the reader:

before: In this post I want to talk a little bit about something that I think is quite
important ...
after: A migration locked our orders table for eleven minutes on a Tuesday afternoon.



The same story, told twice. The rewrite halves the sentence length and loses almost every hedge and passive, and its first sentence says what happened.

What the example shows

The draft spends its opening on nothing. Its first sentence, "In this post I want to talk a little bit about something that I think is quite important", contains three hedges and a throat-clearing phrase, and tells the reader nothing. Five passive constructions, "mistakes were made by us", "the table was locked by it", keep the events at arm's length. Its sentences average 28 words.

The rewrite starts with the event. "A migration locked our orders table for eleven minutes on a Tuesday afternoon." The reader knows what happened, how bad it was and when, in one sentence, and the next, "Nobody could check out", says why it mattered. Sentences average 15 words, the reading-ease score rises from 58 to 66, and the passives are gone. The one "hedge" left is a false alarm: "nobody could check out" uses "could" literally. Like the design-document reviewer in chapter 6, this checker finds candidates, and a human decides.

The readability formula, Rudolf Flesch's reading ease from 1948, is crude. It counts only sentence and word length, and its syllable counting here is a heuristic. Use it to compare drafts of the same text, never as a target in itself.

Publishing, and keeping going

Publish where you own the archive. A blog on your own domain, even a simple static one, keeps working when platforms change their rules or disappear, and every link to it builds on the same address. Cross-post to community sites if you like, but keep the canonical copy at home. A company engineering blog is a fine second outlet, with a review process that protects you as well as them.

Consistency beats brilliance. One useful post a month, for three years, is thirty-six posts in your area, which is more than most experts have written. Talks multiply the effect: a post that's been well received is most of a conference talk, and a talk is most of a post. Find one

reader who'll tell you honestly where a draft loses them, and edit before publishing: most first drafts improve by losing a third of their words, as *The Engineer's Craft* argues for internal documents too.

COMMON TRAPS

Waiting until you're an expert to write, when explaining what you just learned is the most useful writing there is.

Opening with throat-clearing, so readers leave before the point.

Adjectives instead of facts, "significant", "robust", "slow".

Publishing only on a platform you don't control, so the archive can vanish.

Writing once, brilliantly, then never again, when the value is in the archive.

EXERCISES

EXERCISE 8.1 Rewrite a paragraph

Rewrite this opening so it says what happened in the first sentence:

"Recently, it was decided that our caching layer should be looked at, as there had been some concerns raised about performance, which were thought to be quite significant."

SOLUTION

"Our product pages got 40% slower in March, and the cache was the reason: a deploy had cut its hit rate from 94% to 61%. Here's how we found it, and the alert we added so it can't happen quietly again." The original had four passives, "it was decided", "be looked at", "concerns raised", "were thought", and no facts. The rewrite names the event, gives the numbers, and promises the reader what they'll learn. Notice that writing it required facts the original didn't contain, how much slower, why, what changed. That's typical: vague prose often hides that the writer hasn't yet found out the specifics, and rewriting it is how you discover what you still need to know.

EXERCISE 8.2 Titles that promise something

Improve these titles: "Some thoughts on databases", "Our migration journey" and "Caching: a deep dive".

SOLUTION

Some thoughts on databases	-> Why our Postgres table grew to 40 GB when the data was 2
Our migration journey	-> The migration that locked checkout for eleven minutes
Caching: a deep dive	-> How one deploy cut our cache hit rate from 94% to 61%

Each new title makes a specific promise, contains a fact, and tells a reader whether the post is for them. They're also what people search for: someone whose table is mysteriously large will find the first, and nobody searches for "some thoughts". A useful test is whether the title could only belong to your post. "Caching: a deep dive" could head ten thousand articles; the rewrite heads one.

EXERCISE 8.3 A year of writing

Plan twelve posts for the next year in your chosen area, with a sentence each.

SOLUTION

Draw from four sources, three posts each. **Hard lessons:** the incident you diagnosed this quarter; the migration that taught you about locks; the performance problem that turned out to be somewhere else. **Explanations:** how your database's query planner estimates rows; what fsync really guarantees; why your area's most common advice is sometimes wrong. **Reading notes:** one paper a quarter from your list, summarised with what you built to understand it. **Tools:** the small script that saved you hours, the checklist you use in reviews, the dashboard query everyone copies. Put them on a calendar, one in the first week of each month, and write the easiest first so the habit starts. If one month slips, publish two short posts the next rather than skipping; the habit is the thing being built, and at the end of the year the twelve posts are the start of an archive.

CHECKLIST

- ☐ Do you have somewhere you own to publish, and something on it?
- ☐ Does your first sentence say what happened or what the reader will learn?
- ☐ Do you replace adjectives with facts, and cut hedges and throat-clearing?
- ☐ Do you have a reader who'll tell you where a draft loses them?

☐ Do you publish on a regular rhythm, in your area?

GO DEEPER

COURSE Technical Writing Google for Developers

ESSAY What to blog about Simon Willison

BOOK On Writing Well William Zinsser

PART III

Leading

Four chapters on leading work you can't do alone. Estimating honestly, with ranges and probabilities instead of a single hopeful date. Sequencing a project so its unknowns surface early, while they're cheap. Communicating its state to the people who depend on it, each in the form they need. And mentoring, the deliberate work of making other engineers better, which is the fastest way to find the holes in your own understanding. The syllabus asks you to "lead a project end to end: scoping, sequencing, risk, stakeholders, landing it". This part covers each of those words.

Estimating honestly

3 exercises

A plan built by adding up each task's most likely duration is very unlikely to be met, because it ignores how much worse things can go than they can go well, and because waiting for parallel work makes every join later. Estimate each task as a range, combine the ranges along the real dependencies, and talk about dates as probabilities. Commit to a date you're 85% likely to meet, find the tasks that decide it, and update the forecast as you learn.

Why estimates fail

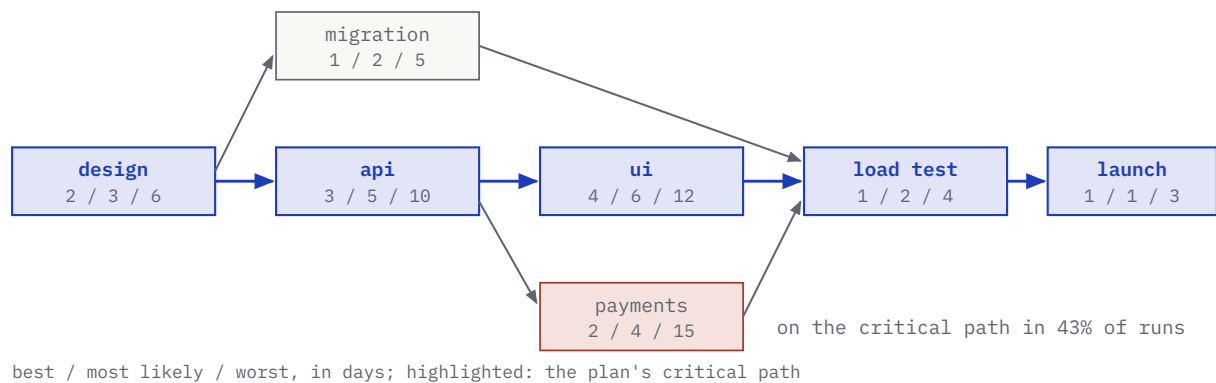
Engineers are often told they're bad at estimating, and the evidence agrees: projects routinely take far longer than planned. In 1979, Daniel Kahneman and Amos Tversky named the **planning fallacy**: people estimate their own tasks by imagining how they'll go, a best-case story, and ignore how similar tasks actually went. But optimism is only part of it. Even an honest estimator, who gives the true most likely duration for every task, produces a plan that's usually late, for two mechanical reasons.

The first is **skew**. A task estimated at five days might finish in four if everything goes well, but in ten if the third-party sandbox is broken or the requirements turn out to be ambiguous. Durations can't go much below their most likely value, and can go far above it, so the average is later than the most likely. Add up the most likely values and you've added up numbers that are each below their task's average. The second is **merge bias**. When a task waits for two others, it waits for the later of the two, and the later of two uncertain things is later than either is likely to be on its own. Every join in a plan pushes the end date out.

Ranges, and the graph

The cure is to estimate each task as three numbers, best case, most likely and worst case, and to combine them along the dependencies rather than adding them up. That combination is hard by hand and easy by simulation: draw a duration for every task from its range, compute when the project finishes given what waits for what, and repeat thousands of times. The result is a distribution of end dates, from which you read percentiles: **P50**, the date you're as likely to beat as to miss, and **P85**, the date you'll meet five times in six.

The simulation also says which tasks matter. In each run, the **critical path** is the chain of dependent tasks that determined the end date. A task that's on the critical path in most runs decides the schedule, and it's where to spend attention; a task that's rarely on it can slip without consequence. The example builds a seven-task project, the launch of a checkout feature, and does all of this.



The plan's critical path runs through the UI. But the payments integration, with its long worst case, decides the end date in 43% of simulated runs, and the plan never mentions it.

EXAMPLE

schedule.py · a seven-task launch simulated twenty thousand times: what the plan's date means, which tasks decide the end, and merge bias

"""Estimating a project as a graph of uncertain tasks: what the plan's date really means, which tasks decide the end date, and why parallel work finishes later than its parts suggest."""

`import random`

`# task: (best case, most likely, worst case) in working days, and what it waits for`

```
TASKS = {
    "design": ((2, 3, 6), []),
    "migration": ((1, 2, 5), ["design"]),
    "api": ((3, 5, 10), ["design"]),
    "ui": ((4, 6, 12), ["api"]),
    "payments": ((2, 4, 15), ["api"]),      # a third party's sandbox: usually fine,
                                          sometimes not
    "load test": ((1, 2, 4), ["ui", "payments", "migration"]),
    "launch": ((1, 1, 3), ["load test"]),
}
```

```
def finish_times(duration):
    done = {}
    for task, (_, deps) in TASKS.items():      # listed in dependency order
        start = max((done[d] for d in deps), default=0)
        done[task] = start + duration[task]
    return done
```

```
def critical_path(duration, done):
    path, task = [], "launch"
    while task:
        path.append(task)
        deps = TASKS[task][1]
        task = max(deps, key=lambda d: done[d]) if deps else None
    return path[::-1]
```

```
likely = {t: est[1] for t, (est, _) in TASKS.items()}
plan = finish_times(likely)["launch"]
print(f"the plan, using every 'most likely' estimate: {plan} days "
      f"({' -> '.join(critical_path(likely, finish_times(likely))})")
```

```

rng = random.Random(9)
runs, ends, critical = 20_000, [], {t: 0 for t in TASKS}
for _ in range(runs):
    d = {t: rng.triangular(lo, hi, mode) for t, ((lo, mode, hi), _) in TASKS.items()}
    done = finish_times(d)
    ends.append(done["launch"])
    for t in critical_path(d, done):
        critical[t] += 1
ends.sort()
p = lambda q: ends[int(q * (runs - 1))]
print(f"chance of finishing by day {plan}: {sum(e <= plan for e in ends) / runs:.0%}")
print(f"P50 {p(.5):.1f} days    P85 {p(.85):.1f} days    P95 {p(.95):.1f} days")

print("\nhow often each task is on the critical path:")
for t, n in sorted(critical.items(), key=lambda kv: -kv[1]):
    print(f" {t:10} {n / runs:5.0%}")

# Merge bias: waiting for the later of two parallel tasks, each likely to take 6 days.
pairs = [max(rng.triangular(4, 12, 6), rng.triangular(4, 12, 6)) for _ in range(runs)]
single = sorted(rng.triangular(4, 12, 6) for _ in range(runs))
print(f"\none 6-day task: median {single[runs // 2]:.1f} days; "
      f"waiting for the later of two: median {sorted(pairs)[runs // 2]:.1f} days")

print("\nBrooks's law: pairs of people who must stay in sync")
for n in (3, 5, 8, 12):
    print(f" {n:2} people: {n * (n - 1) // 2:3} channels")

```

OUTPUT

```

the plan, using every 'most likely' estimate: 17 days (design -> api -> ui -> load test ->
launch)
chance of finishing by day 17: 2%
P50 22.0 days    P85 25.2 days    P95 27.0 days

how often each task is on the critical path:
design          100%
api             100%
load test      100%
launch         100%
ui              57%
payments       43%
migration       0%

one 6-day task: median 7.1 days; waiting for the later of two: median 8.2 days

Brooks's law: pairs of people who must stay in sync
3 people:      3 channels
5 people:     10 channels
8 people:     28 channels
12 people:    66 channels

```

What the example shows

The plan's date is a 2% chance. Adding the most likely durations along the longest chain gives seventeen days, the kind of number that goes in a slide. In the simulation, the project finished by day 17 in only 2% of runs. The median was 22 days and P85 was 25.2. Nobody was pessimistic or careless: every task's most likely value was honest. The plan was wrong because of skew and joins, and it would have been wrong for anyone who made it this way.

The task that decides the date isn't on the plan's critical path. Design, the API, the load test and the launch are on the critical path in every run, as they must be. But the UI was on it in only 57% of runs, and the payments integration, absent from the planned path, decided the end date in 43%. Its most likely four days hide a worst case of fifteen, a third-party sandbox that's usually fine and sometimes not. That's where a lead should look first: start it earlier, prototype against the sandbox in week one, or find a fallback.

Waiting for two things takes longer. One task estimated at six days, best four and worst twelve, had a median of 7.1 days. Waiting for the later of two such tasks had a median of 8.2. Every join in a plan quietly adds time like this, and a plan with many parallel streams merging into one launch adds a lot.

Talking about dates

Steve McConnell's *Software Estimation* separates three things that conversations about dates mix up. An **estimate** is a prediction: what you think will happen, with its uncertainty. A **target** is a wish: when the business would like it. A **commitment** is a promise: the date you agree to be held to. Most estimation arguments are a target being presented as an estimate, or an estimate being heard as a commitment. Name which one you're discussing.

Give estimates as ranges with confidence: "most likely around day 22; 85% confident of day 25". Commit at your P85, not your P50, because a P50 commitment is missed half the time, and repeated misses destroy trust faster than conservative dates. Update the forecast every week as tasks finish and their ranges collapse to actual values. Early in a project the range is wide; it should narrow as you go, and if it doesn't, the project has a problem worth raising now. And when the date you can honestly commit to isn't the target, say so early and offer choices: a later date, less scope, or a different approach. Leaders are trusted for bad news delivered early.

Brooks's law

Fred Brooks's *The Mythical Man-Month*, from 1975, states the most famous law of software management: adding people to a late project makes it later. New people need weeks to become productive, and someone who knows the project has to teach them, so the team slows down before it speeds up. And communication grows faster than headcount: with n people there are $n(n-1)/2$ pairs who may need to stay in sync, three channels for three people, sixty-six for twelve. Brooks called his book's subject "the bearing of a child takes nine months, no matter how many women are assigned". Some work divides well; much doesn't. When a project is late, cutting scope or removing a blocker usually helps more than adding people.

Learning from your own record

The best correction for the planning fallacy is your own history. Bent Flyvbjerg's **reference class forecasting**, used for large infrastructure projects, starts from how long similar past projects actually took, not from the details of this one. You can do the same at a small scale: keep a record of each task's estimate and its actual duration. After a few months, the ratio of

actual to estimated, perhaps 1.4 on average, with a long tail, tells you how to adjust, and its spread tells you how wide your ranges should be. Teams that keep this record estimate better within a year, not because they become less optimistic but because they calibrate.

COMMON TRAPS

Adding up most-likely durations and calling the sum a date.

Committing at P50, and missing half of your commitments.

Ignoring the task with the long worst case, because its most likely value is small.

Treating a target as an estimate, or an estimate as a commitment.

Adding people to a late project, instead of cutting scope or removing a blocker.

EXERCISES

EXERCISE 9.1 The date to commit to

Product asks for the launch "in three weeks", fifteen working days.
Using the example's results, write the reply.

SOLUTION

"We can't commit to fifteen days. Even our plan's most optimistic chain is seventeen, and we'd meet that only about one time in fifty. Our honest forecast is a median of 22 working days, and we're 85% confident of 25, so 25 is the date I'd commit to. If three weeks matters, here are the options: we could ship without the load test and launch prep, which saves about three days and adds risk we'd need to agree on; or we could cut the new UI and launch the API for internal use first. The biggest single uncertainty is the payments sandbox; if we prototype against it this week, I expect the range to narrow, and I'll send an updated forecast on Friday." The reply separates the target from the estimate, gives the estimate with its confidence, offers real choices for meeting the target, and promises an update when uncertainty shrinks.

EXERCISE 9.2 De-risk the decider

A two-day spike against the payments sandbox finds no surprises, cutting the task's worst case from 15 days to 8. Change the example and describe what happens.

SOLUTION

chance of finishing by day 17: 4%
P50 20.9 days P85 23.7 days P95 25.3 days
on the critical path: ui 90%, payments 10%

P85 improves by a day and a half, from 25.2 to 23.7, from narrowing one task's range, without making anything faster. And the critical path changes character: payments now decides the date in only 10% of runs, and the UI in 90%, so attention should move to the UI. That's the pattern of a well-led project: find the task whose uncertainty decides the date, spend a little to shrink it, and re-run the forecast to see where the risk went. Notice also what didn't change much: the chance of the plan's seventeen days is still only 4%. Reducing uncertainty makes dates more predictable; it doesn't make optimistic plans come true.

EXERCISE 9.3 Your own multiplier

Design a lightweight record for your team's estimates and actuals, and say what you'd compute from it after three months.

SOLUTION

Add three fields to each task in your tracker: the best, likely and worst estimate in days, recorded when the task is started, and the actual duration, recorded when it's done. Nothing else; the habit has to survive busy weeks. After three months, compute for each task where its actual fell relative to its range: below best, between best and likely, between likely and worst, or beyond worst. If estimates are calibrated, about half the actuals land above the likely value and only a few beyond the worst. If a third land beyond the worst, your ranges are too narrow; widen the worst cases. Also compute the median ratio of actual to likely: a value of 1.3 says your "likely" is consistently optimistic by about 30%, which you can now correct for, honestly and with evidence. Share the results with the team without blame; the goal is calibration, not accountability.

CHECKLIST

- ☐ Do you estimate tasks as ranges, and combine them along the real dependencies?
- ☐ Can you explain why a sum of most-likely durations is usually late?
- ☐ Do you commit at P85, and update the forecast as tasks finish?
- ☐ Do you find the tasks that decide the end date, and de-risk them first?
- ☐ Do you keep a record of estimates against actuals, and use it?

GO DEEPER

BOOK [The Mythical Man-Month](#) Fred Brooks

BOOK [Software Estimation: Demystifying the Black Art](#) Steve McConnell

ESSAY [Estimating software projects](#) Jacob Kaplan-Moss

Leading a project

3 exercises

Leading a project end to end means owning its outcome, not its code: what's in scope, in what order the work happens, what could kill it, and whether it really lands. Most of the skill is in the order. Do the work that could change the plan first, while little has been built on top of it, and do the comfortable work later. The same project, sequenced risk first, discovers its bad news weeks earlier and throws away far less.

What leading means

A technical lead on a project is rarely the person who writes the most code. They're the person who makes sure the project succeeds: that it's solving the right problem, that the plan is realistic, that the risks are known and handled, that the people who depend on it know what's happening, and that it actually finishes. Tanya Reilly's *The Staff Engineer's Path* describes this as holding the whole project in your head, so that nothing important is nobody's job. The syllabus lists the parts: scoping, sequencing, risk, stakeholders and landing it. This chapter covers all but stakeholders, which have chapter 11.

Scoping

Start by writing down what "done" means, in a sentence a stakeholder would agree with: "customers can refund an order from the admin screen, and the money reaches their card within a week". Then find the smallest version of the project that achieves it, and list everything else as explicitly out of scope, in the design document's non-goals. Every project grows during its life, and most of the growth is reasonable on its own; the scope list is what lets you say "that's a good idea for phase two" without an argument.

Keep a **cut list** from the start: the parts you'd drop first if the project ran late, agreed with the stakeholders in advance. Agreeing to cuts when nobody is under pressure is easy. Agreeing to them in the last week, with the date slipping, is a negotiation nobody wins.

Sequencing

Most project plans order work by what's comfortable or what's logically first: build the data model, then the screens, then the reports, and leave the integrations and the hard questions until the pieces they connect exist. That ordering has a hidden cost. The hard questions, will the bank's API support refunds, will the risk team accept rules instead of a model, are the ones whose answers can change the plan, and every week they wait, more work is built on assumptions that might be wrong.

The alternative is to **sequence by risk**: of the tasks you can start, do first the one with the most uncertainty per day of work, so bad news arrives while it's still cheap. Two techniques sharpen it. A **spike** is a short, throwaway experiment that answers one question, two days of prototype calls against the bank's sandbox, before committing to the work that depends on the answer. A **walking skeleton**, Alistair Cockburn's term, is the thinnest end-to-end version of the system, every component present but barely functional, built first so that integration problems appear in week one instead of the last week. The example measures what sequencing is worth.

EXAMPLE

sequence.py · one project of eight tasks, three of them hiding serious unknowns, run in three orders twenty thousand times

```
"""Sequencing a project: the same work in three orders. Some tasks hide an unknown that, if it
turns out badly, forces a change of plan and throws away part of the work already done."""
import math
import random

# task: (days of work, chance its unknown turns out badly)
TASKS = {
    "data model": (5, 0.05),
    "admin screens": (8, 0.02),
    "reporting": (6, 0.05),
    "email templates": (3, 0.02),
    "bank API integration": (7, 0.35), # does the bank's API support refunds at all?
    "fraud rules": (6, 0.20), # will the risk team accept rules instead of a model?
    "migration of old data": (5, 0.15),
    "load and failover test": (4, 0.10),
}

REWORK = 0.5 # share of the work already done that a change of plan throws away

NEEDS = {"migration of old data": ["data model"], "reporting": ["data model"],
         "load and failover test": [t for t in TASKS if t != "load and failover test"]}

def riskiest_first():
    """Of the tasks whose prerequisites are done, always take the most risk per day of work."""
    order = []
    while len(order) < len(TASKS):
        ready = [t for t in TASKS if t not in order and all(d in order for d in NEEDS.get(t, []))]
        order.append(max(ready, key=lambda t: (TASKS[t][1] / TASKS[t][0], t)))
    return order

ORDERS = {
    "comfortable first": ["data model", "email templates", "admin screens", "reporting",
                        "migration of old data", "fraud rules", "bank API integration",
                        "load and failover test"],
    "riskiest first": riskiest_first(),
}

def run(order, rng, spike=False):
    """Days wasted, and the day the first bad surprise surfaced (None if there was none)."""
    day, wasted, first = 0, 0.0, None
    if spike: # two days of prototypes against the two
               riskiest unknowns
        day += 2
    for t in ("bank API integration", "fraud rules"):
```

```

        if rng.random() < TASKS[t][1]:
            first = first or day          # found before anything is built on it:
            nothing to redo
    for t in order:
        days, p = TASKS[t]
        day += days
        if not spike or t not in ("bank API integration", "fraud rules"):
            if rng.random() < p:
                first = first or day
                wasted += REWORK * (day - days)  # the plan changes; half the earlier work is
                redone
    return wasted, first

rng = random.Random(10)
print(f"{sum(d for d, _ in TASKS.values())} days of work; chance at least one unknown turns out
badly: "
      f"{1 - math.prod(1 - p for _, p in TASKS.values()):.0%}\n")
print(f"{'order':40}{'days wasted':>12}{'first surprise':>16}")
plans = [(name, order, False) for name, order in ORDERS.items()]
plans.append(("riskiest first, after a 2-day spike", ORDERS["riskiest first"], True))
for name, order, spike in plans:
    results = [run(order, rng, spike) for _ in range(20_000)]
    wasted = sum(w for w, _ in results) / len(results)
    firsts = sorted(f for _, f in results if f is not None)
    print(f"{name:40}{wasted:>12.1f}{ 'day ' + str(firsts[len(firsts) // 2]):>16}")
print("\n(days wasted: the average over 20,000 runs; first surprise: the median day it
surfaced)")
print("riskiest-first order:", " -> ".join(ORDERS["riskiest first"]))

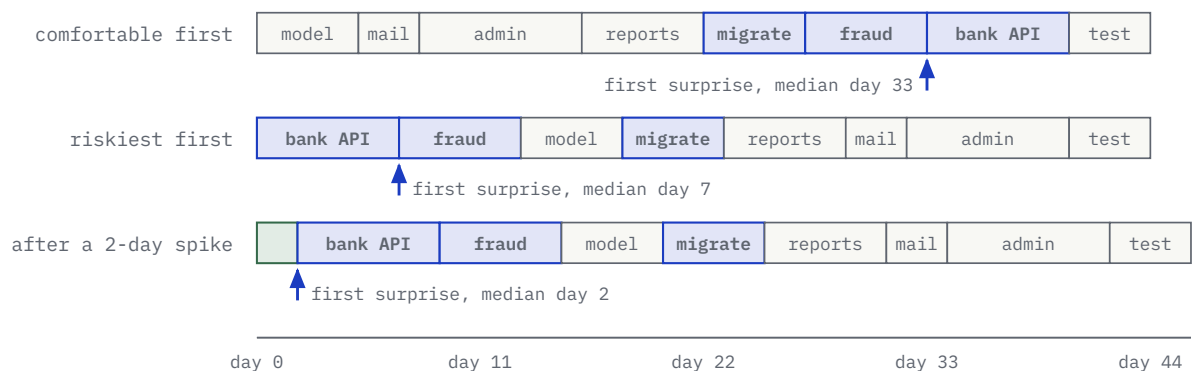
```

OUTPUT

44 days of work; chance at least one unknown turns out badly: 66%

order	days wasted	first surprise
comfortable first	12.5	day 33
riskiest first	5.5	day 7
riskiest first, after a 2-day spike	5.3	day 2

(days wasted: the average over 20,000 runs; first surprise: the median day it surfaced)
riskiest-first order: bank API integration -> fraud rules -> data model -> migration of old
data -> reporting -> email templates -> admin screens -> load and failover test



Same work, different order. Put the tasks that could change the plan first, and bad news arrives in week one or two instead of week seven, before most of the work that it would invalidate exists.

What the example shows

Something will probably go wrong. The eight tasks each have a chance of revealing a problem that changes the plan, mostly small, but the bank integration has a 35% chance and the fraud rules 20%. Together, at least one goes badly 66% of the time. That's realistic for a project with external dependencies, and it means the plan should assume a surprise, not hope to avoid one.

The order decides what the surprise costs. Comfortable first, the median first surprise came on day 33 of 44, after most of the work was done, and on average 12.5 days of work were thrown away. Riskiest first, respecting the real dependencies, it came on day 7, and the average waste fell to 5.5 days. That's a week of work saved per project, from nothing but the order of the same tasks, plus five extra weeks to react to bad news while there's still room to.

A spike buys time, more than it saves work. Two days of prototypes against the two riskiest unknowns moved the first surprise from day 7 to day 2 while saving little more work. What it bought was knowing early. On a real project, knowing in week one that the bank doesn't support partial refunds means the product decision, cut the feature or change the design, happens while the team is still flexible, which is usually worth more than the days.

Managing risk, on purpose

Keep a **risk register**: a short list, reviewed weekly, of what could go wrong. Each entry has a description, a rough likelihood and impact, an owner, the mitigation, and a **trigger**, the observable sign that the risk is happening. "Bank sandbox doesn't support refunds; likely; high impact; owner Ana; mitigation: spike in week 1, fallback of manual refunds; trigger: the refund endpoint returns an error after a day of trying." Ten lines like that, kept current, are worth more than a hundred-row spreadsheet nobody reads.

For a risky project, decide **kill criteria** before starting: the results that would mean stopping or changing course. "If the spike shows the bank can't support refunds through its API, we stop and propose manual refunds instead." Decided in advance, a kill criterion is a plan. Decided afterwards, under the weight of the work already done, it's a fight against the sunk-cost fallacy, which usually wins.

Landing it

Projects die in their last ten percent. The feature works, the team's attention moves on, and what's left, migrating the last customers, deleting the old code path and its feature flag, writing the runbook, handing over to the team that will own it, never quite happens. The old and new systems then run side by side for years, and the project's real benefit, simplicity, never arrives. A project is done when the old path is gone, the documentation is written, the on-call team knows how to run it, and the success measure from the scope has been checked. Plan that work explicitly, with dates, from the start.

Then close it properly. Tell everyone who depended on it that it's finished and what changed. Hold a short retrospective, what to repeat and what to change, while memories are fresh. And thank the people who did the work, by name, where their managers can see it. Landing a project well is what makes people want to work on your next one.

COMMON TRAPS

Ordering work by comfort, so the questions that could change the plan are answered last.

No agreed cut list, so scope negotiations happen in the final week.

A risk register that's written once and never read again.

No kill criteria, so a failing project continues on sunk cost.

Declaring victory at launch, with the old path, the flag and the handover still undone.

EXERCISES

EXERCISE 10.1 When rework is cheap

Predict what happens to the three orders' waste if a change of plan throws away only 10% of the work done, rather than 50%. Then check it by changing **REWORK**. What does this tell you about when sequencing matters?

SOLUTION

Every order's waste shrinks to a fifth, since waste is proportional to **REWORK**: about 2.5 days comfortable first and 1.1 days riskiest first. The first-surprise days don't change at all, because the order of the tasks hasn't changed. So the work saved by sequencing scales with how much a surprise invalidates. When the components are independent and a change of plan touches little else, order hardly matters for waste. When everything is built on an early assumption, a data model, an integration, an API contract, it matters a lot. But the time to react to bad news still favours risky-first, whatever the rework rate: learning on day 7 rather than day 33 leaves time to change course, which is often the bigger benefit and the one the waste figure doesn't measure.

EXERCISE 10.2 Kill criteria

Write kill criteria for the refunds project, to be agreed with stakeholders before work starts.

SOLUTION

We stop, or return to stakeholders with a new proposal, if:

1. The week-1 spike shows the bank's API can't issue refunds to cards.
-> Propose manual refunds through the bank's portal, with an admin-screen record.
2. The risk team won't accept rule-based fraud checks by the end of week 2.
-> Pause until a model-based approach is scoped; don't build rules they'll reject.
3. The P85 forecast passes the end of the quarter at any weekly review.
-> Cut to the agreed minimum: full refunds only, no partial refunds, no reporting.

Each criterion is observable, has a date or a trigger, and says what happens next, so it's a decision made in advance rather than a vague warning. The third is the most important and the least common: a schedule kill criterion connects the forecast from chapter 9 to an action, so a slipping date triggers the cut list automatically instead of a debate. Agree these with the stakeholders in the kickoff, and write them in the design document.

EXERCISE 10.3 A walking skeleton

Describe the walking skeleton for the refunds project: what does it contain, and what would you learn from building it in the first week?

SOLUTION

One button on the admin screen that refunds a hard-coded test order in full, through every real component: the admin screen calls the refund API, which writes a refund record, calls the bank's sandbox, waits for its webhook, updates the record and sends the customer an email from the real template. No validation, no partial refunds, no fraud rules, no reporting. Building it in week one tests every integration point at once: authentication to the bank, the webhook arriving through your network, the email sender, the deployment pipeline for the new service. Problems found there would otherwise have surfaced only when the polished components were connected near the end. After that, each week fleshes out one part of the skeleton, and the system works end to end the whole time, which also makes the status report in chapter 11 much easier to write honestly.

CHECKLIST

- ☐ Is "done" written down in a sentence stakeholders agree with, with a cut list?

- ☐ Is the work sequenced so the questions that could change the plan are answered first?
- ☐ Do you use spikes and a walking skeleton to surface problems early?
- ☐ Do you keep a short risk register with triggers, and agreed kill criteria?
- ☐ Does your plan include landing: the old path removed, documentation, handover?

GO DEEPER

BOOK The Staff Engineer's Path Tanya Reilly

GUIDE Staff Engineer: guides Will Larson

ESSAY Performing a Project Premortem Gary Klein

Status, stakeholders and influence

3 exercises

A project that's going well but communicated badly looks like a project in trouble, and one in trouble but communicated well can still be saved. Know who depends on the project and what each of them needs. Write status updates that put the bottom line first, take their colour from the forecast rather than from mood, and ask for decisions by a date. And since a lead rarely has authority over the people a project needs, learn to move them by persuasion, relationships and clear writing.

Who needs to know

A project's stakeholders are everyone who's affected by it, decides something about it, or must do something for it: the sponsor who funds it, the product lead who decides its scope, the partner teams whose systems it touches, the support team who'll answer questions about it, the engineers who build it. They don't all need the same thing. A useful exercise at the start is to list them and sort them by two questions, the grid that management texts attribute to Aubrey Mendelow: how much influence do they have over the project, and how much does it affect them? The sponsor, high on both, needs regular, short updates and early warning. A partner team that must change an API needs to know exactly what you need and by when. Someone affected but with little say needs to be told before the change reaches them, not after.

Status updates that get read

Most status updates are activity logs: "this week we continued working on the integration and had several meetings". They tell the reader that people were busy, which they assumed, and nothing they need: is it on track, what's the risk, what do you need from me? A useful update answers those in its first two lines, in the order a busy reader needs them. Military writing calls this **BLUF**, bottom line up front.

The status colour deserves particular care, because it's the one thing everyone reads. Green, amber and red should come from the forecast, not from the lead's mood or fear of looking bad. One workable rule: **green** if the P85 forecast meets the committed date, **amber** if only the median does, and **red** if the median misses it. A rule makes the colour honest and removes the temptation to report "green" until the week the date is missed, the pattern known as the watermelon project: green outside, red inside. The example writes a week's update from the project's own data, for three audiences.

EXAMPLE

```
status.py · one week's status written from the project's data: the colour from the
forecast, and the bottom line first for each audience
```

```

"""A status update written from the project's own data: the colour comes from the forecast, not
from mood, and each audience gets the bottom line first."""
import random

TODAY, COMMITTED = 12, 30          # working days since kickoff; the date we promised
REMAINING = {                     # task: (best, most likely, worst) days left
    "refunds in the bank API": (3, 5, 12),
    "fraud rules sign-off": (1, 2, 6),
    "admin screens": (4, 5, 8),
    "load and failover test": (2, 3, 5),
}
DONE = ["data model", "migration of old data", "email templates"]
BLOCKERS = [("bank API: refunds endpoint returns 501 in sandbox", "the bank's integration
    team",
    "ticket open 6 days")]
DECISIONS = [("ship without partial refunds if the bank can't support them by day 20",
    "product lead", "day 18")]

def forecast(remaining, start=TODAY, seed=11):
    rng = random.Random(seed)
    return sorted(start + sum(rng.triangular(lo, hi, mode) for lo, mode, hi in
        remaining.values())
        for _ in range(10_000))

ends = forecast(REMAINING)
p50, p85 = ends[len(ends) // 2], ends[int(len(ends) * 0.85)]
stuck = forecast(REMAINING, start=TODAY + 5)          # if the blocker holds the work up
    another week
stuck_p50 = stuck[len(stuck) // 2]
on_time = sum(e <= COMMITTED for e in ends) / len(ends)
colour = "GREEN" if p85 <= COMMITTED else ("AMBER" if p50 <= COMMITTED else "RED")

print("=== the status update nobody reads ===")
print("This week we continued working on the bank integration and made good progress on the
    admin")
print("screens. We had several meetings with the bank. Next week we will keep going.\n")

print("=== for the leadership summary: two lines ===")
print(f"{colour}: {on_time:.0%} likely to ship by day {COMMITTED} as committed "
    f"(median forecast day {p50:.1f}).")
what, who, when = DECISIONS[0]
print(f"Need: a decision from the {who} by {when} on whether to {what}.\n")

print("=== for partner teams: what we need from you ===")
for what, who, age in BLOCKERS:
    print(f"- {who}: {what} ({age}). If it holds us up another week, "
        f"the median forecast moves from day {p50:.1f} to day {stuck_p50:.1f}.")
print()

print("=== for the team: the detail ===")
print(f"done: {' '.join(DONE)}")
for task, (lo, mode, hi) in REMAINING.items():
    print(f"open: {task:26} {mode} days likely, {lo} to {hi}")
widest = max(REMAINING, key=lambda t: REMAINING[t][2] - REMAINING[t][0])
print(f"forecast: P50 day {p50:.1f}, P85 day {p85:.1f}; committed day {COMMITTED}. "
    f"The widest range is {widest}, so it's the task to finish first.")

```

OUTPUT

```

=== the status update nobody reads ===
This week we continued working on the bank integration and made good progress on the admin
screens. We had several meetings with the bank. Next week we will keep going.

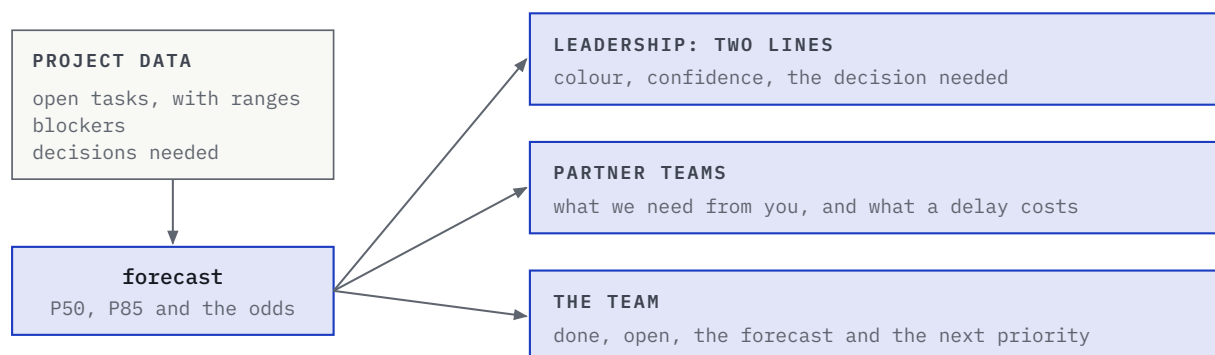
=== for the leadership summary: two lines ===

```

RED: 43% likely to ship by day 30 as committed (median forecast day 30.5).
 Need: a decision from the product lead by day 18 on whether to ship without partial refunds if the bank can't support them by day 20.

=== for partner teams: what we need from you ===
 - the bank's integration team: bank API: refunds endpoint returns 501 in sandbox (ticket open 6 days). If it holds us up another week, the median forecast moves from day 30.5 to day 35.5.

=== for the team: the detail ===
 done: data model, migration of old data, email templates
 open: refunds in the bank API 5 days likely, 3 to 12
 open: fraud rules sign-off 2 days likely, 1 to 6
 open: admin screens 5 days likely, 4 to 8
 open: load and failover test 3 days likely, 2 to 5
 forecast: P50 day 30.5, P85 day 33.4; committed day 30. The widest range is refunds in the bank API, so it's the task to finish first.



One set of facts, three readers. The colour and the dates come from the forecast, so the update is honest by construction; each audience gets its own bottom line first.

What the example shows

The colour is red, and that's useful. The project is only 43% likely to ship by its committed day 30, and its median forecast, day 30.5, is just past it. Under the rule, that's red, even though the project is close. A red that early, with three weeks of work left and a clear cause, isn't a failure; it's the update doing its job, a warning while there's still time to act. The same project reported as green "because we're nearly there" would become a surprise in three weeks, which is the thing stakeholders never forgive.

Every audience gets its bottom line, and an ask. Leadership gets two lines: the colour with its probability, and a decision needed from the product lead by day 18, whether to ship without partial refunds if the bank can't support them by day 20. Asking for a decision by a date, before the date at which it matters, is what turns a status report from information into action. Partner teams get the blocker and its cost: if the bank's refunds endpoint holds the work up another week, the median moves from day 30.5 to day 35.5. That's five days for five days, because the blocked task is on the critical path. Saying it in days makes the cost concrete to a team that has other priorities. The team gets the detail, and the reason to finish the bank integration first: it has the widest range.

Influence without authority

A technical lead usually can't tell anyone outside their own team what to do, and often not inside it either. Projects need other teams to change their systems, review designs, and give up some of their own priorities. The way this happens is persuasion, and persuasion is a skill with a few reliable parts.

Build the relationship before you need it. The partner team you've helped, reviewed for, or simply talked to, will find time for your request; the one that first hears from you when you need something will put it in a queue. **Understand what they want.** Every team has goals and pressures; a request framed in terms of their goals, "this also removes the manual reconciliation your on-call does every week", moves faster than one framed in yours. **Write it down.** A short, clear proposal, the problem, the request, the cost and the deadline, can be forwarded to the person who actually decides, and survives being forwarded. And **agree before the meeting.** Toyota's practice of *nemawashi*, talking a proposal through with each person affected before the formal decision, means the meeting confirms an agreement instead of starting an argument.

When persuasion fails and the project is genuinely blocked, **escalate well:** as a request for a decision, not a complaint. Say what's blocked, what it costs, the options and what you recommend, and copy the other party, so nothing is said behind their back. A good escalation is one the other team would describe as fair. It's how a lead uses the management chain without spending trust.

Meetings, briefly

A meeting is an expensive way to share information and a good way to make a decision. Hold decision meetings with a written pre-read, so the time goes on the disagreement rather than on explaining the background. Finish by reading back the decisions, the owners and the dates, and send them in writing within the hour. Most "we agreed on this" disputes come from meetings whose decisions weren't written down.

COMMON TRAPS

Status updates that are activity logs, with no bottom line and no ask.

Colour from mood, green until the week the date is missed.

Asking for a decision without a deadline, or after the date it matters.

Meeting partner teams for the first time when you need something.

Escalating as a complaint, behind the other team's back, instead of as a request for a decision.

EXERCISE 11.1 Rewrite the update

Rewrite this update for leadership, in two lines: "Good progress this week. The team has been heads-down on the migration and we're working through some issues with the vendor. Should be fine, will keep you posted."

SOLUTION

You can't, yet, and noticing that is the point: the original contains no facts. To write two honest lines you need the forecast and the vendor issue's cost, so first find them out. Then, for example: "AMBER: 70% likely to finish the migration by 14 November as committed; median forecast 12 November. Need: the vendor to restore API access to our staging account by Wednesday, or the median moves to 19 November; I've asked their account manager and will escalate on Wednesday if it isn't fixed." Each sentence is checkable: a colour from a rule, a probability, a date, a specific blocker with its cost, and what you're doing about it. "Should be fine" became a number, and "keep you posted" became a date for the next action.

EXERCISE 11.2 An escalation

The bank's integration team still hasn't fixed the refunds endpoint after two weeks. Write the escalation to your director, copying the bank's contact.

SOLUTION

Subject: Decision needed by Friday: refunds launch blocked on the bank's sandbox

The refunds endpoint in the bank's sandbox has returned 501 since 3 October (their ticket 4471). Our launch is committed for 14 November; every week this stays open moves our forecast by a week, and we're now amber.

Options:

1. Ask the bank's account team to prioritise the ticket. (Recommended; I've attached the reproduction their engineers asked for.)
2. Launch full refunds through the bank's portal, recorded in our admin screen, and add API refunds when the endpoint works. Two days of work, no date risk.
3. Move the launch date.

I recommend 1 and 2 together: ask for the fix, and build the fallback now so the date doesn't depend on it. Copying Priya at the bank, who has been helpful and is waiting on her own platform team.

The subject says what's needed and by when. The body states the facts and their cost in dates, gives real options with a recommendation, and is fair to the other party, who is copied and credited. The director can decide in two minutes, and nobody at the bank has reason to feel ambushed. Notice that the recommended plan removes the dependency rather than only pushing on it: good escalations usually include a way to stop needing the escalation.

EXERCISE 11.3 A stakeholder map

List the stakeholders of the refunds project, sort them by influence and by how much the project affects them, and say what each needs from you and how often.

SOLUTION

High influence, much affected:

product lead	the forecast and decisions with deadlines; weekly, two lines
finance director	how refunds are reconciled and reported; at the design review, and before launch

High influence, little affected:

engineering director	colour and risks; weekly summary, early warning of red
security team	the fraud rules' design; review in week 2, sign-off before launch

Little influence, much affected:

support team	what customers will see and how to answer; training before launch
the bank's team	exactly what we need and by when; whenever blocked, with a reproduction
on-call engineers	runbook and alerts; handover in the last week

The map shows who'd be surprised if you forgot them, which is its real purpose. The support team and on-call engineers have little say in the project but will live with its results, so they need early, practical information; the security and finance teams can stop the launch, so their reviews belong in the plan with dates, not as a last-week surprise. Revisit the map at each phase, because stakeholders change as a project moves from design to launch.

CHECKLIST

- ☐ Do you know your project's stakeholders, and what each needs from you?
- ☐ Does your status update put the bottom line first, with an ask and a date?
- ☐ Does the colour come from the forecast, by a stated rule?
- ☐ Do you build relationships with partner teams before you need them?

- ☐ Do you escalate as a request for a decision, with options and the other party copied?

GO DEEPER

BOOK The Staff Engineer's Path Tanya Reilly

ESSAY Work on what matters Will Larson

ESSAY How to Write Email with Military Precision Kabir Sehgal, Harvard Business Review

Mentoring

3 exercises

Mentoring is the most leveraged work a senior engineer does, and it's a skill, not a personality trait. Start from what the other person actually knows, which is often less solid than what they've used. Teach at the edge of that knowledge, in an order where each idea rests on ones already secure. Give feedback that's specific enough to act on. And sponsor as well as advise: use your standing to give people the opportunities that make them grow.

Why mentor deliberately

An engineer who makes three colleagues 20% more effective has added more than most engineers add by their own work, and the effect compounds: those colleagues mentor others. That's the leverage the syllabus means by "the ability to move other engineers". There's a second reason, which the syllabus also names: "teaching is the fastest way to find the holes in your own model". Explaining something to a sharp junior engineer, who asks why at each step, exposes every place where your understanding is a memorised rule rather than a reason. Richard Feynman is often credited with the test: if you can't explain it simply, you don't understand it well enough.

Most mentoring happens by accident: a junior engineer absorbs what they can from code reviews and overheard conversations. Deliberate mentoring does better, because it starts from a diagnosis, follows a plan, and checks whether the plan is working.

Mentor, coach, sponsor

Lara Hogan, writing about engineering management, distinguishes three roles that are often blurred. A **mentor** gives advice from experience: "here's how I'd approach this". A **coach** asks questions that help the other person find their own answer: "what have you tried, and what would you try next?" A **sponsor** uses their standing to create opportunities: recommending someone for a stretch project, naming them in a meeting they're not in, putting their work in front of people who decide promotions. Each has its place. Advice is fastest for facts and conventions; coaching builds judgement, which can't be handed over; and sponsorship is what changes careers, especially for people whose work tends to be overlooked. A good mentor moves between all three and knows which one the moment calls for.

Start from what they really know

People describe what they know by what they've used. A mentee who says they know query planning may mean they've read `EXPLAIN` output and added indexes when told to. That's real, useful knowledge, and it may rest on nothing: they may not know what a B-tree is, how a page is read from disk, or where the planner's row estimates come from. Knowledge like that is **fragile**. It works for the cases they've seen and fails, confusingly, on the first case that depends on the missing foundation.

So start with a diagnosis, done kindly: ask them to explain, not to recall. "Walk me through what happens when this query runs" reveals far more than "do you know indexes?". Then map what you learn against what each idea builds on. The example does this for a mentee learning database internals: a small graph of concepts and what each needs, the four the mentee says they know, and three questions. What is fragile, what are they ready to learn now, and in what order should they learn their way to crash recovery?

EXAMPLE

curriculum.py · a mentee's knowledge against a graph of what each concept builds on: what's fragile, what's ready to learn, and a teaching order to a goal

```
"""Mentoring as planning: what someone knows, what they're ready to learn next, and where their
knowledge is fragile because it rests on something they don't yet have."""
from graphlib import TopologicalSorter

# concept: the concepts it builds on
NEEDS = {
    "files and fsync": [],
    "pages and the buffer pool": ["files and fsync"],
    "B-trees": ["pages and the buffer pool"],
    "indexes": ["B-trees"],
    "query plans": ["indexes", "statistics"],
    "statistics": [],
    "transactions": [],
    "write-ahead log": ["files and fsync", "transactions"],
    "crash recovery": ["write-ahead log", "pages and the buffer pool"],
    "MVCC": ["transactions", "pages and the buffer pool"],
    "isolation anomalies": ["MVCC"],
    "replication": ["write-ahead log"],
}

# What a mentee says they know, after a conversation: what they've used, not what they can
explain.
KNOWS = {"transactions", "indexes", "query plans", "replication"}

order = list(TopologicalSorter(NEEDS).static_order())

def ancestors(concept):
    seen, todo = set(), list(NEEDS[concept])
    while todo:
        c = todo.pop()
        if c not in seen:
            seen.add(c)
            todo.extend(NEEDS[c])
    return seen

print("fragile: known, but resting on something not yet known")
for c in [c for c in order if c in KNOWS]:
    gaps = sorted(ancestors(c) - KNOWS, key=order.index)
    if gaps:
        print(f" {c:16} rests on: {' '.join(gaps)}")
```

```

ready = [c for c in order if c not in KNOWS and all(d in KNOWS for d in NEEDS[c])]
print("\nready to learn now (everything it needs is known):", ", ".join(ready))

goal = "crash recovery"
path = [c for c in order if (c in ancestors(goal) or c == goal) and c not in KNOWS]
print(f"\nto reach {goal!r}, in an order that never needs what hasn't been taught:")
for step, c in enumerate(path, 1):
    print(f"    {step}. {c}")
print(f"\nshortcut check: of {len(path)} steps, the first is {path[0]!r}: "
      f"ask them to explain it before teaching anything that rests on it")

```

OUTPUT

```

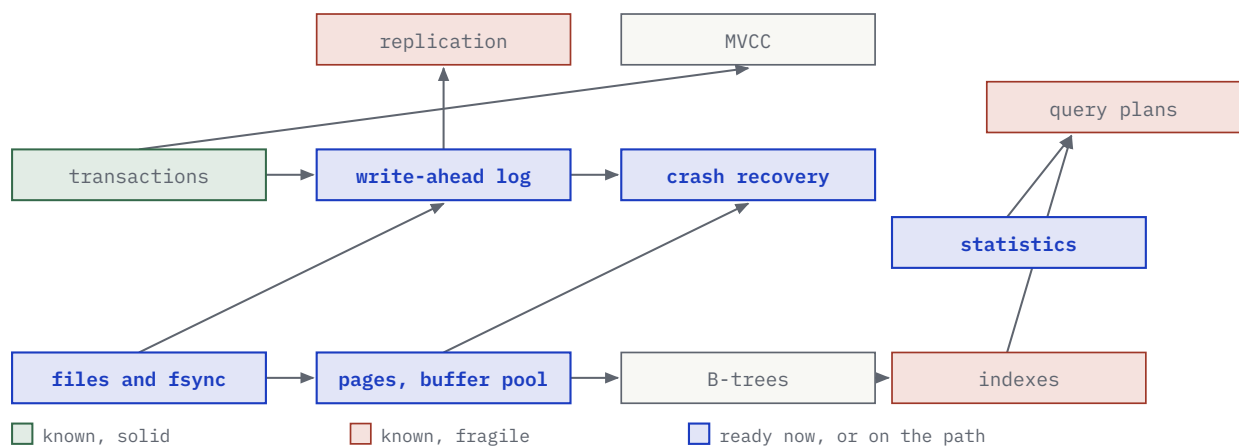
fragile: known, but resting on something not yet known
  replication      rests on: files and fsync, write-ahead log
  indexes          rests on: files and fsync, pages and the buffer pool, B-trees
  query plans      rests on: files and fsync, statistics, pages and the buffer pool, B-trees

ready to learn now (everything it needs is known): files and fsync, statistics

to reach 'crash recovery', in an order that never needs what hasn't been taught:
  1. files and fsync
  2. pages and the buffer pool
  3. write-ahead log
  4. crash recovery

shortcut check: of 4 steps, the first is 'files and fsync': ask them to explain it before
teaching anything that rests on it

```



What the mentee knows, against what it rests on. Three of their four known concepts are fragile; the frontier, what they're ready to learn now, is two concepts at the bottom of the graph; and the path to the goal starts there too.

What the example shows

Three of the four things they know are fragile. Indexes rest on B-trees, pages and the buffer pool, and files and fsync, none of which they know; query plans rest on those and on statistics; replication rests on the write-ahead log. Only transactions stand on their own. That's typical of engineers who learned on the job: competent at the top of the stack, with gaps underneath. It isn't a criticism; it's a map of where their next bug will confuse them.

The frontier is at the bottom. The concepts they're ready to learn now, everything they need already known, are files and fsync, and statistics. Teaching anything else first would mean explaining an idea in terms of others they don't have, which is how explanations lose people. The path to their goal, crash recovery, is four steps, and starts with files and fsync. The example's last line is the practical advice: before teaching anything on the path, ask them to explain its first step. If they can, you're further along than the map says; if they can't, you've found the place to begin.

Teaching at the edge

Lev Vygotsky called it the zone of proximal development: what a learner can't yet do alone but can do with help. Teaching works there and nowhere else. Below it, they're bored; above it, they're lost. The example's frontier is that zone, computed. In practice, three techniques work well. **Worked examples, then faded guidance:** show a complete solution, then give a similar problem with part of the solution, then one with none. **Pairing, then reversing:** you drive while they watch and ask; then they drive while you watch and ask. **Questions before answers:** when they bring a problem, ask what they've tried and what they think is happening before saying what you think. The answer they reach themselves is the one they'll remember, and the question shows them how you think, which is what they most need to learn.

Code review is the mentoring most engineers already do, and it's easy to do it badly. Explain why, not just what: "this query runs once per order, so a page of 100 orders makes 101 queries" teaches; "use select_related" doesn't. Separate what must change from suggestions and nitpicks, and say which is which. And comment on what's good, specifically, because people repeat what's noticed.

Feedback that helps

Feedback is useful when it's specific, timely, about behaviour rather than personality, and private if it's critical. The **situation-behaviour-impact** model, from the Center for Creative Leadership, gives it a shape: describe the situation, the specific behaviour, and its impact. "In yesterday's incident, you posted updates in the channel every ten minutes, and support could answer customers without asking us" is feedback someone can repeat. "Great job in the incident" is a pleasantry. The same shape works for criticism, and keeps it about something the person can change.

Sponsorship

Advice helps people grow; opportunities make them grow faster. Give your mentee the stretch work: the design document for the next project, the lead on a small incident review, the talk at the team meeting. Back them while they do it, and let them get the credit, in public. Recommend them for things when they're not in the room, by name. And give away the work you'd enjoy doing yourself, because the interesting work is exactly what someone growing needs. The sign that mentoring has worked isn't that they ask you fewer questions; it's that they start answering other people's.

COMMON TRAPS

Taking "I know it" at face value, instead of asking for an explanation.

Teaching above the frontier, in terms of ideas the learner doesn't have yet.

Answering every question at once, so the learner never builds their own judgement.

Feedback too vague to act on, praise or criticism without a specific behaviour.

Mentoring without sponsoring, advice in private and no opportunities in public.

EXERCISES

EXERCISE 12.1 A diagnostic question

Write one question that would reveal whether someone's knowledge of indexes is solid or fragile, and say what a fragile answer sounds like.

SOLUTION

"We have an index on `(customer_id, created_at)`. Which of these two queries can use it well, and why: one filtering on `customer_id`, one filtering on `created_at`?" A solid answer explains that the index is a B-tree sorted by `customer_id` first, so the first query can descend straight to one customer's entries, while the second has no efficient way in, because rows for a given date are scattered across every customer. A fragile answer gets the right result by rule, "the leftmost column must be in the filter", but can't explain it, or gets it wrong, "both, because both columns are indexed". The question works because it can't be answered from a remembered rule alone; it needs the structure underneath, which is exactly what the map said might be missing.

EXERCISE 12.2 Feedback with a shape

Rewrite these two pieces of feedback using situation, behaviour and impact: "Your design doc was really good" and "You need to communicate better".

SOLUTION

"In your search design document last week, you priced the managed service and tested trigram indexes before proposing the new index. When the director asked why not the cheaper option, the answer was already in the document, and the review took twenty minutes instead of a second meeting." And: "In the refunds project, the bank's sandbox was blocked for six days before the team heard about it, at the stand-up on Thursday. We lost a week we could have used for the fallback. Next time something blocks you for more than a day, could you post it in the channel that day?" Both name a situation and a behaviour the person can repeat or change, and say why it mattered. The second also ends with a specific request, which turns criticism into something the person can act on tomorrow.

EXERCISE 12.3 Six weeks to crash recovery

Using the example's path, plan six weeks of mentoring, an hour a week, to get the mentee to explain crash recovery.

SOLUTION

Week 1: files and fsync. Have them write a small program that writes a file, kills itself before and after `fsync`, and checks what survived; ask them to explain the results. Week 2: pages and the buffer pool. Read the relevant chapter of *Database Internals* together, then ask them to explain why a database reads 8 KB when you ask for one row. Week 3: the write-ahead log. Give them the rule, log before data, and ask them to argue why it's necessary, before showing the answer. Week 4: crash recovery. Walk through a crash with a half-written page and a log, and have them replay the log by hand. Week 5: reverse roles. They explain crash recovery to you, or better, to another junior engineer, while you only ask questions. Week 6: revisit the fragile concepts that now have foundations, replication on top of the log, and ask them to explain why a replica that applies the log in order converges. Each week starts with a question, ends with them explaining, and moves only when the step is secure.

CHECKLIST

- ☐ Do you diagnose what someone knows by asking them to explain, not to recall?
- ☐ Can you find their frontier, and teach in an order that never skips a foundation?
- ☐ Do you ask questions before giving answers?
- ☐ Is your feedback specific: a situation, a behaviour and its impact?

- ☐ Do you sponsor as well as mentor: stretch work, public credit, recommendations?

GO DEEPER

ESSAY What does sponsorship look like? Lara Hogan

BOOK The Manager's Path Camille Fournier · its first chapter, on being mentored and mentoring

GUIDE Code review developer guide Google's engineering practices

PART IV

The long game

Four chapters on what makes the rest last. The business: what your company sells, what it costs, and where engineering moves the numbers, so you can choose work by its value. Measurement: the four delivery metrics that research ties to better outcomes, computed from your own history. Finishing: why limiting work in progress makes everything faster, and the other boring virtues that make people trust you. And a career that compounds: the written record, the body of work and the sustainable pace that turn years into expertise.

Learning the business

3 exercises

Every engineering choice is also a business choice, and the engineers who are trusted with the biggest ones can say what their work is worth in the business's own units. Learn a handful of numbers: revenue, margin, what a customer is worth, what drives churn, what infrastructure costs per unit of work. Then price projects in money, write the assumption beside each number, and rank them by value per week of effort. It won't be precise. It will be arguable, which is far better than unspoken.

Why an engineer needs the numbers

The syllabus asks you to "learn the business: what your company sells, what it costs, where engineering moves the number". There's a practical reason. Senior engineers spend much of their time choosing what to work on, and arguing for it. "This would be cleaner" persuades other engineers. "This saves \$110,000 a year and pays for itself in six weeks" persuades everyone, including the people who decide budgets. The second argument needs the numbers, and engineers who have them get a different kind of hearing.

The numbers also change what you think good engineering is. Reliability matters because downtime makes customers leave, not because uptime is a virtue. Performance matters where it changes conversion or cost, and hardly at all elsewhere. Once you see which numbers your work moves, the trade-offs in chapter 7 get much easier to argue.

The handful of numbers

A product business, at its simplest, runs on a few quantities. **Revenue**, and how it's earned: subscriptions, usage, transactions, advertising. **Gross margin**: the share of revenue left after the costs of serving customers, which for a software product are mostly infrastructure and support. **Churn**: the share of customers who leave each month. **Lifetime value**: what a customer is worth over their whole time as a customer, roughly the margin they bring each month divided by the churn rate. **Conversion**: the share of trials or visitors who become paying customers. And for an engineer, **unit cost**: what one request, one invoice or one gigabyte costs to serve.

You can learn your company's versions of these in a week if you ask. Finance teams are usually delighted that an engineer wants to understand the model. Public companies publish most of them in their annual reports. And the pricing page, the support queue and an hour with a salesperson teach the parts no spreadsheet shows: what customers actually buy, and what makes them angry.

Pricing engineering work

With those numbers, an engineering project can be priced in the same currency as every other business decision. The example does it for a small invoicing product, 4,000 paying teams at \$49 a month, and three candidate projects: making checkout faster, caching invoice PDFs, and fixing the bulk-invoice failures that exit surveys blame for some churn. Each project's value comes from one assumption, written beside it in the code, where anyone can see it and argue with it.

EXAMPLE

economics.py · the unit economics of a small invoicing product, an outage priced two ways, and three projects ranked by value per week of effort

```
"""Learning the business: the unit economics of a small invoicing product, and three
engineering
projects priced in the same currency, so they can be compared and argued about."""

CUSTOMERS, PRICE = 4_000, 49          # paying teams, dollars a month each
INFRA = 38_000                        # cloud bill, dollars a month
SUPPORT = 22_000                      # support staff, dollars a month
REQUESTS = 900e6                     # API requests a month
CHURN = 0.025                         # share of customers who leave each month
TRIALS, CONVERSION = 2_000, 0.18     # trial sign-ups a month, and the share who pay
ENGINEER_WEEK = 4_000                # loaded cost of one engineer for a week

revenue = CUSTOMERS * PRICE
margin = (revenue - INFRA - SUPPORT) / revenue
lifetime_value = PRICE * margin / CHURN      # what one customer is worth over their lifetime
print(f"revenue ${revenue:,.0f} a month; gross margin {margin:.0%}; "
      f"infrastructure ${INFRA / (REQUESTS / 1000):.3f} per thousand requests")
print(f"a customer stays {1 / CHURN:.0f} months on average and is worth ${lifetime_value:,.0f}
in margin")
hourly = revenue / (30 * 24)
print(f"customers pay ${hourly:,.0f} for each hour of service, so a 4-hour outage refunded in
full "
      f"costs ${4 * hourly:,.0f};\nif it also makes 1% of customers leave, it costs "
      f"${0.01 * CUSTOMERS * lifetime_value:,.0f} in lifetime margin")

def annual_value(name):
    if name == "faster checkout":          # trial conversion 18% -> 19%, from an A/B test
        return TRIALS * 0.01 * 12 * lifetime_value / 2    # half the value lands within the
        year
    if name == "cache invoice PDFs":       # PDFs are 30% of the cloud bill; caching cuts
        that by 80%
        return INFRA * 0.30 * 0.80 * 12
    if name == "fix bulk-invoice failures": # exit surveys blame it for a tenth of churn
        kept = CUSTOMERS * CHURN * 0.10 * 12
        return kept * lifetime_value / 2

PROJECTS = {"faster checkout": 6, "cache invoice PDFs": 3,          # engineer-weeks of work
            "fix bulk-invoice failures": 8}
print(f"\n{'project':>28}{ 'value a year':>14}{ 'cost':>10}{ 'payback':>12}{ 'value/week':>14}")
rows = []
for name, weeks in PROJECTS.items():
    value, cost = annual_value(name), weeks * ENGINEER_WEEK
    rows.append((value / weeks, name, value, cost))
for per_week, name, value, cost in sorted(rows, reverse=True):
    print(f"{'name':>28}{value:>14,.0f}{cost:>10,.0f}{cost / (value / 52):>9.1f}
wk{per_week:>14,.0f}")
best = max(rows)[1]
print(f"\nDo {best!r} first: its value per week of effort is the highest, "
```

```

"so every week it waits costs the most.")
print("Each figure rests on one assumption written beside it; "
      "change the assumption and the ranking can change.")

```

OUTPUT

revenue \$196,000 a month; gross margin 69%; infrastructure \$0.042 per thousand requests
 a customer stays 40 months on average and is worth \$1,360 in margin
 customers pay \$272 for each hour of service, so a 4-hour outage refunded in full costs \$1,089;
 if it also makes 1% of customers leave, it costs \$54,400 in lifetime margin

project	value a year	cost	payback	value/week
cache invoice PDFs	109,440	12,000	5.7 wk	36,480
faster checkout	163,200	24,000	7.6 wk	27,200
fix bulk-invoice failures	81,600	32,000	20.4 wk	10,200

Do 'cache invoice PDFs' first: its value per week of effort is the highest, so every week it waits costs the most.
 Each figure rests on one assumption written beside it; change the assumption and the ranking can change.

VALUE PER WEEK OF ENGINEERING EFFORT



Ranked by value per week of effort, the smallest project comes first. Every week it waits costs the most, which is what the ranking measures.

What the example shows

A customer is worth about \$1,360. With a 69% gross margin and 2.5% of customers leaving each month, the average customer stays 40 months and brings \$1,360 of margin over that time. That single number makes the rest computable: anything that keeps a customer, or wins one, is worth about that much.

An outage costs less in refunds than in leaving. Customers pay \$272 for each hour of service, so a four-hour outage refunded in full costs \$1,089. If the same outage makes 1% of customers decide to leave, it costs \$54,400 in lifetime margin, fifty times as much. That's why reliability work is worth far more than its "lost minutes" suggest, and why the right way to argue for it is through churn, not uptime percentages.

The smallest project goes first. Faster checkout is worth the most per year, \$163,200, but takes six engineer-weeks. Caching invoice PDFs is worth less, \$109,440, but takes three, so its value per week of effort, \$36,480, is the highest, and it pays for itself in under six weeks. The bulk-invoice fix, at \$10,200 a week of effort, comes last despite fixing a real customer

pain. Ranking by value per week of effort is a simple form of what Don Reinertsen calls weighted shortest job first: since every week a valuable project waits is value lost, do first the work that delivers the most value per unit of time it occupies the team.

Every figure is an assumption. The conversion gain comes from one A/B test; the PDF share of the cloud bill from one month's invoice; the churn share from exit surveys. Writing each assumption beside its number, as the code does, is the point: it turns "I think checkout matters more" into "your A/B test may be optimistic, so let's check it", which is an argument that can be settled.

The cost of delay

Reinertsen's *The Principles of Product Development Flow* makes a claim worth remembering: if you quantify only one thing, quantify the cost of delay. It's the value lost for each week a piece of work isn't done, and it's what the ranking above really compares. Teams that don't quantify it make prioritisation decisions by volume and seniority, and routinely delay their most valuable work behind their loudest. Cost of delay also makes the case for small batches and finishing things, the subject of chapter 15: a project half done delivers nothing, so every week spent on two projects at once delays the value of both.

Speaking the business's language

Translate engineering work into the business's units: money, risk and time. Technical debt is interest: "our build takes twenty minutes, and with thirty engineers it costs about \$660,000 a year in waiting", rather than "the build is slow". Reliability is churn and revenue at risk. Security is the expected cost of a breach, and the deals lost without a compliance certificate. Performance is conversion and infrastructure cost. None of these needs to be precise. A rough number with its assumption stated is enough to start a real conversation, and it's how engineering work competes fairly for time with everything else a business could do.

COMMON TRAPS

Arguing for work by elegance to people who decide by value.

Pricing outages in lost minutes instead of customers who leave.

Ranking by total value, ignoring how long each project occupies the team.

Precise-looking numbers with hidden assumptions, which nobody can check.

Never asking finance for the model, and guessing at the numbers that matter.

EXERCISE 13.1 The A/B test was optimistic

A re-run of the A/B test shows faster checkout raises conversion by half a point, not one. What happens to its value and to the ranking?

SOLUTION

Its value halves, from \$163,200 a year to \$81,600, and its value per week of effort from \$27,200 to \$13,600. The ranking doesn't change: caching PDFs at \$36,480 a week still goes first, checkout second, the bulk-invoice fix third at \$10,200. But checkout's lead over the bulk fix has shrunk from nearly three times its value to a third more, so one more change in an assumption, say the exit surveys undercounting the bulk-invoice failures, could reorder them. That's the useful result. It shows which comparison is close, and therefore which assumption is worth checking before committing six weeks of work. It's chapter 7's method applied to prioritisation: find the assumption that decides it.

EXERCISE 13.2 The slow build, in money

Thirty engineers each run the full build about six times a day, and it takes twenty minutes. Estimate what it costs a year, and what cutting it to eight minutes would be worth, stating your assumptions.

SOLUTION

Thirty engineers, six runs a day and twenty minutes a run is 3,600 minutes, sixty hours of waiting a day. Assume half of it is genuinely lost, since people do other things during some builds but lose focus switching back: thirty hours a day. Over about 220 working days that's 6,600 hours a year, and at a loaded cost of \$100 an hour, about \$660,000 a year. Cutting the build to eight minutes removes 60% of the wait, worth about \$400,000 a year. Even if the "half lost" assumption is generous and the true figure is a quarter, it's \$200,000 a year, enough to justify several engineer-months of build work. State each assumption, the runs a day, the share lost and the hourly cost, so that someone can change the one they doubt and see whether the conclusion survives. It usually does, which is why build speed is one of the easiest engineering investments to justify.

EXERCISE 13.3 Questions for finance

Write the five questions you'd ask your company's finance team to learn the numbers that matter for your area.

SOLUTION

1. How do we make money: what share is subscription, usage, transactions, services?
2. What's our gross margin, and how much of the cost of serving customers is infrastructure, and how much is support?
3. What's monthly churn, and what do we know about why customers leave?
4. What is a customer worth over their lifetime, and what does it cost to win one?
5. Which of these numbers is the company most worried about this year?

The first four give you the model; the fifth tells you where the attention and budget will go, which is where engineering work that moves that number will be valued most. Ask for the numbers as ranges if exact figures are confidential; for prioritisation, the order of magnitude is what matters. And offer something back: explain what drives the infrastructure bill, which finance teams often find opaque. A relationship with finance is one of the most useful a senior engineer can have.

CHECKLIST

- ☐ Do you know how your company makes money, its margin, churn and what a customer is worth?
- ☐ Do you know the unit cost of what your systems do?
- ☐ Can you price a project in money, with its assumption written beside the number?
- ☐ Do you rank work by value per week of effort, the cost of delay?
- ☐ Do you argue for reliability through churn, and for technical debt as interest?

GO DEEPER

BOOK The Principles of Product Development Flow Donald Reinertsen

ESSAY Cost of Delay Black Swan Farming

ESSAY Work on what matters Will Larson

Measuring delivery

3 exercises

A decade of research on software teams found four measures that predict both speed and stability: how often you deploy, how long a change takes to reach production, how often a deploy causes a failure, and how quickly you recover. They can be computed from data you already have, your Git history and your incident log. Use them to learn, as a team, never to rank individuals, and pair every measure of speed with one of stability, so that improving one can't quietly wreck the other.

Four keys

From 2014, the research programme now called DORA, led by Nicole Forsgren with Jez Humble and Gene Kim, surveyed tens of thousands of engineers about how their teams worked and how their organisations performed. The book that summarised it, *Accelerate*, published in 2018, identified four measures of software delivery that separated high performers from low:

- **Deployment frequency:** how often the team deploys to production.
- **Lead time for changes:** how long a committed change takes to reach production.
- **Change failure rate:** the share of deploys that cause a failure needing a fix.
- **Time to restore:** how long it takes to recover from such a failure. DORA's recent reports call it failed deployment recovery time.

The first two measure speed; the last two, stability. The research's most quoted finding is that they don't trade against each other. The teams that deployed most often also had the lowest failure rates and recovered fastest. Speed and stability come from the same practices: small changes, automated testing and deployment, and fast feedback.

Computing them from your own history

You don't need a vendor's dashboard to know your numbers. Deploys are usually tags or pipeline runs; commits carry timestamps; incidents are in a log or a channel. The example builds a Git repository with six weeks of history, ninety commits, three a day, with deploy tags: for three weeks the team releases once a week, on Friday, and for three weeks it releases every afternoon. An incident log records which deploys caused failures and how long recovery took. The four keys are then computed from the repository itself, using `git log` between consecutive deploy tags.

EXAMPLE

dora.py · the four DORA metrics computed from a Git history with deploy tags and an incident log, for weekly releases and then daily ones

```

"""Measuring delivery: the four DORA metrics computed from a real Git history with deploy tags
and
an incident log, for three weeks of weekly releases and three weeks of releasing every day."""
import os, pathlib, statistics, subprocess, tempfile

BASE = {"PATH": os.environ["PATH"], "HOME": "/home/team", "LC_ALL": "C", "GIT_CONFIG_NOSYSTEM":
"1",
        "GIT_CONFIG_GLOBAL": os.devnull, "GIT_AUTHOR_NAME": "Dev", "GIT_AUTHOR_EMAIL":
        "dev@example.org",
        "GIT_COMMITTER_NAME": "Dev", "GIT_COMMITTER_EMAIL": "dev@example.org"}
START = 1780876800                                # a Monday, 00:00 UTC
HOUR, DAY = 3600, 86400

def git(*args, cwd, when=None):
    env = dict(BASE)
    if when:
        env.update(GIT_AUTHOR_DATE=f"@{when} +0000", GIT_COMMITTER_DATE=f"@{when} +0000")
    r = subprocess.run(["git", *args], cwd=cwd, env=env, capture_output=True, text=True,
        check=True)
    return r.stdout.strip()

with tempfile.TemporaryDirectory() as tmp:
    repo = pathlib.Path(tmp)
    git("init", "-q", "-b", "main", cwd=repo)
    deploys, n = [], 0
    for day in range(42):
        week, weekday = divmod(day, 7)
        if weekday >= 5:
            continue                                # nobody works at the weekend
        for hour in (10, 14, 16):
            n += 1
            (repo / "app.txt").write_text(f"change {n}\n")
            git("add", "app.txt", cwd=repo)
            git("commit", "-q", "-m", f"Change {n}", cwd=repo, when=START + day * DAY + hour *
                HOUR)
        weekly = week < 3 and weekday == 4          # weeks 1-3: one big release on
            Friday
        daily = week >= 3                            # weeks 4-6: release every
            afternoon
        if weekly or daily:
            when = START + day * DAY + 17 * HOUR
            git("tag", f"deploy-{len(deploys) + 1}", cwd=repo)
            deploys.append((f"deploy-{len(deploys) + 1}", when, "weekly" if weekly else
                "daily"))

    # incidents: (the deploy that caused it, hours until service was restored)
    INCIDENTS = {"deploy-2": 6.0, "deploy-3": 3.5, "deploy-9": 0.5, "deploy-15": 0.75}

    print(f"{len(deploys)} deploys from {n} commits; incidents after {' '.join(INCIDENTS)}\n")
    print(f"{'':24}{ 'weekly releases':>18}{ 'daily releases':>18}")
    rows = {"weekly": {}, "daily": {}}
    previous = None
    for tag, when, kind in deploys:
        span = f"{previous}..{tag}" if previous else tag
        commit_times = [int(t) for t in git("log", "--format=%ct", span, cwd=repo).split()]
        r = rows[kind]
        r.setdefault("deploys", 0); r["deploys"] += 1
        r.setdefault("lead", []).extend((when - t) / HOUR for t in commit_times)
        r.setdefault("batch", []).append(len(commit_times))
        r.setdefault("failed", 0); r["failed"] += tag in INCIDENTS
        if tag in INCIDENTS:
            r.setdefault("restore", []).append(INCIDENTS[tag])
        previous = tag
    metrics = [
        ("deploys a week", lambda r: f"{r['deploys']} / 3:.1f"),
        ("changes per deploy", lambda r: f"{statistics.mean(r['batch']):.0f}"),
    ]

```

```

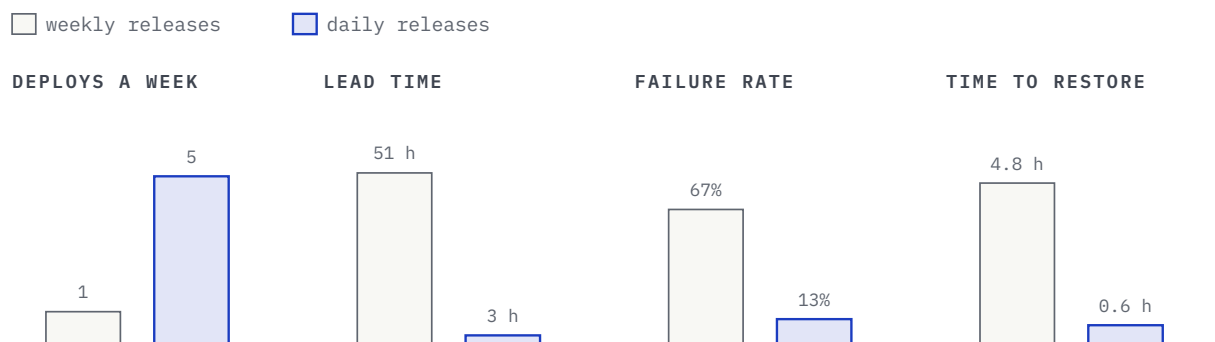
("lead time, median", lambda r: f"{statistics.median(r['lead']):.1f} h"),
("change failure rate", lambda r: f"{r['failed'] / r['deploys']:.0%}"),
("time to restore, median", lambda r: f"{statistics.median(r['restore']):.1f} h"),
]
for name, f in metrics:
    print(f"{name:24}{f(rows['weekly']):>18}{f(rows['daily']):>18}")

```

OUTPUT

18 deploys from 90 commits; incidents after deploy-2, deploy-3, deploy-9, deploy-15

	weekly releases	daily releases
deploys a week	1.0	5.0
changes per deploy	15	3
lead time, median	51.0 h	3.0 h
change failure rate	67%	13%
time to restore, median	4.8 h	0.6 h



Lead time and deploys follow from the schedule; the failure figures come from an invented log

The same team, the same work, released in smaller batches. Lead time falls because changes stop waiting for the weekly train; failures and recovery improve because each deploy contains less, and less to search when something breaks.

What the example shows

The arithmetic first. Weekly releases bundle fifteen changes into each deploy, and a change committed on Monday morning waits until Friday afternoon, so the median lead time is 51 hours. Releasing daily, three changes go out each afternoon and the median lead time is three hours. That part isn't an assumption: it follows directly from the release schedule and the commit times, and it's the same arithmetic in any team.

The stability numbers come from an invented incident log, so read them for their direction, not their size. Two of the three weekly releases caused incidents, against two of the fifteen daily ones, and the weekly failures took far longer to recover from: 4.8 hours at the median against 0.6. The mechanism behind that direction is real, and it's why the research found what it did. A deploy of fifteen changes has fifteen places a problem could hide; finding which one means bisecting a large batch under pressure, and rolling back

undoes all fifteen. A deploy of three changes fails less often, because there's less in it, and when it fails, the culprit is one of three changes written that day by people who still remember them.

Goodhart's law

The economist Charles Goodhart observed that a statistical regularity tends to collapse once it's used for control; the anthropologist Marilyn Strathern's paraphrase is the one everyone quotes: "when a measure becomes a target, it ceases to be a good measure". Delivery metrics are especially vulnerable. Set a target for deployment frequency and teams deploy empty changes. Rank engineers by commits and they split work into meaningless pieces. Punish change failures and failures stop being reported. Three rules keep the measures honest. Measure **teams**, **never individuals**, because software is delivered by teams and individual metrics are both unfair and gamed. **Pair speed with stability**, always reporting all four together, so that improving one at the expense of another is visible. And use them for **learning, not judgement**: the question each month is "what's slowing us down, and what would help?", not "who's behind?".

Learning from incidents

The incident log behind change failure rate is also the richest source of learning a team has, if incidents are examined well. A **blameless postmortem** asks how the failure happened, not who caused it, on the principle, from safety researchers such as Sidney Dekker, that people's actions made sense to them at the time with the information they had. Find out what that information was, and why it was misleading, and you'll find what to fix: a missing alert, a confusing dashboard, a deploy process that made the mistake easy. Blame finds a person to retrain and leaves the system that tripped them unchanged.

A useful postmortem has a timeline, the contributing factors, plural, since there's rarely one cause, what went well, and a short list of action items, each with an owner and a date. Then the part most teams skip: check, a month later, that the action items were done. The share of postmortem actions completed is itself a revealing measure of whether a team learns from its incidents or only documents them.

COMMON TRAPS

Measuring speed without stability, or the reverse.

Ranking individuals by commits, deploys or tickets closed.

Setting the metric as a target, and getting empty deploys.

Blaming a person for an incident, and leaving the system that tripped them.

Postmortem action items that are never done or never checked

EXERCISES

EXERCISE 14.1 Your own lead time

Your team tags each production deploy as `deploy-YYYYMMDD-N`. Write the commands that print, for the last deploy, how long each commit in it waited between being committed and being deployed.

SOLUTION

```
last=$(git tag --list 'deploy-*' --sort=-creatordate | sed -n 1p)
prev=$(git tag --list 'deploy-*' --sort=-creatordate | sed -n 2p)
deployed=$(git log -1 --format=%ct "$last")
git log --format='%ct %h %s' "$prev..$last" |
  while read -r committed hash subject; do
    printf '%5.1f h %s %s\n' "$(echo "($deployed - $committed) / 3600" | bc -l)"
      "$hash" "$subject"
  done
```

The two most recent deploy tags bound the range of commits in the last deploy, `git log prev..last` lists them with their commit times, and the tag's commit time stands for the deploy time. If tags are created after the commit they point to, use the tag's own date, `git for-each-ref --format='%(creatordate:unix)' refs/tags/$last`, instead. Two caveats make the number honest. Commit time is when the author committed, which may be long after they started the work, so this measures lead time from commit, as DORA defines it, not from idea. And a merged branch's commits keep their original times, so a branch that sat for a week shows up as a week of lead time, which is exactly the delay worth seeing.

EXERCISE 14.2 Gaming the target

A director sets a target: every team must deploy at least ten times a week. Predict three ways teams might hit the target without improving anything, and propose a better way to use the measure.

SOLUTION

Teams could deploy unchanged code, re-running the pipeline to count a deploy. They could split one change into ten tiny commits deployed separately, which hits the number while adding risk and noise. Or they could deploy to production behind flags that are never turned on, so deploys happen but nothing reaches users. Each hits the target and teaches the organisation nothing. A better use: report all four measures

for each team, monthly, without targets, and ask each team what's limiting its lead time. The answers, a slow review queue, a flaky test suite, a manual approval step, point at investments that improve every measure at once. If a target is unavoidable, make it about lead time, the measure that's hardest to game without genuinely shortening the path to production, and always show it beside the change failure rate.

EXERCISE 14.3 A postmortem outline

Write the outline of a blameless postmortem for the migration that locked the orders table for eleven minutes, from chapter 8.

SOLUTION

Summary: a migration adding a column with a default locked orders for 11 minutes at 14:02 on a Tuesday; checkout failed for every customer.

Timeline: 13:55 migration merged; 14:02 deploy starts migration; 14:03 checkout error alert; 14:06 on-call joins; 14:09 lock identified; 14:13 migration completes on its own; 14:15 recovery confirmed.

Contributing factors:

- On this database version, adding a column with a default rewrites the table.
- Migrations run automatically during deploys, at peak hours.
- The migration review checklist doesn't mention locking.
- Staging has 1,000 orders, so the rewrite took milliseconds there.

What went well: the alert fired within a minute; on-call found the lock in three.

Actions:

1. Add lock_timeout to every migration (owner: Ana, 1 Nov).
2. Add a locking question to the migration checklist (owner: Sam, 25 Oct).
3. Run large-table migrations outside peak hours, by default (owner: Lee, 8 Nov).
4. Review in a month that 1 to 3 are done (owner: the author).

Nothing in it names who wrote the migration, because that's not where the fix is: anyone could have written it, given the checklist, the staging data and the deploy process. The contributing factors are plural and each leads to an action, and the last action is the check that the others were done. That's the difference between a postmortem that changes the system and one that only records the event.

CHECKLIST

- ☐ Can you name the four keys and explain why speed and stability go together?
- ☐ Can you compute your team's lead time and deploy frequency from your own history?
- ☐ Do you report speed and stability together, for teams, never individuals?
- ☐ Do you use the measures for learning rather than targets?

☐ Are your postmortems blameless, with owned actions that are checked later?

GO DEEPER

GUIDE DORA's software delivery performance metrics DORA

BOOK Accelerate Forsgren, Humble & Kim

CHAPTER Postmortem Culture: Learning from Failure Google SRE book

Finishing things

3 exercises

The syllabus calls them the boring virtues: reliability, clear communication, finishing things. They're boring because they're not clever, and they're the reason some engineers are trusted with everything. Finishing has a mathematics, Little's law: at a given rate of completion, the more you have in progress, the longer each thing takes. Starting everything feels productive and makes everything slower. Limit what's in progress, finish before starting, and do what you said you'd do or say early that you won't.

The boring virtues

Ask engineers whom they'd most want on a difficult project and the answers are rarely the most brilliant people they know. They name the ones who do what they say they'll do, who say clearly what's happening, and who finish. Those virtues are boring in the sense that nobody writes papers about them, and they compound: every commitment kept makes the next request to you more likely, and every project finished frees attention for the next. An engineer with modest technical depth and these virtues is often more valuable than a brilliant one without them, and an engineer with both is the one the syllabus is trying to produce.

The previous chapters covered communication: design documents, status updates, writing in public. This chapter is about finishing, which turns out to have a precise mathematical basis, and about reliability, which is mostly a set of habits.

Little's law

In 1961, John Little proved a result that holds for almost any stable system where things arrive, wait, get worked on and leave: the average number of items in the system equals the rate at which they finish multiplied by the average time each spends inside. In symbols, $L = \lambda W$. It needs no assumptions about the distribution of arrivals or of work, which is why it applies to request queues in a server, cars on a road and tickets on a team's board alike.

Rearranged, it says something uncomfortable about how teams work: the time each item takes, W , equals the number in progress, L , divided by the rate of finishing, λ . If a team finishes three items a week and has twelve in progress, each takes four weeks on average. Have twenty-four in progress at the same rate, and each takes eight. Starting more work doesn't finish it faster; at a given throughput, it only makes every item take longer. And in practice it lowers throughput too, because people juggling several items lose time switching between them, a cost Gerald Weinberg estimated at about a fifth of a person's time for each additional project they carry.

Starting everything, or limiting it

The example runs a team of four for two thousand days, with the same stream of requests, 0.75 a day on average, each needing two to six days of focused work, in two ways. In the first, every request is started as soon as it arrives, and people divide their time between everything in progress, losing a little to each extra item they juggle. In the second, the team works on at most four items at once, one each, and the rest wait in a queue until someone is free. Then it checks Little's law for both.

EXAMPLE

flow.py · one team, the same arrivals, two policies: start everything, or limit work in progress to four, with Little's law checked for both

```
"""Finishing things: a team of four with the same arrivals, run two ways. One starts every
    request
as it comes in; the other limits work in progress. Little's law ties the numbers together."""
import random

PEOPLE, DAYS, WARMUP = 4, 2000, 200
SWITCH = 0.04          # each item juggled beyond one per person costs this share of their time

def simulate(wip_limit, seed=15):
    rng = random.Random(seed)
    waiting, active, finished = [], [], []          # items: [arrived, started, work left]
    in_progress_days = 0
    for day in range(DAYS):
        for _ in range(sum(rng.random() < 0.25 for _ in range(3))):      # 0.75 new requests a
            day                                                                # 2 to 6 days of
            waiting.append([day, None, rng.uniform(2, 6)])                  # 2 to 6 days of
            focused work
        while waiting and (wip_limit is None or len(active) < wip_limit):
            item = waiting.pop(0)
            item[1] = day
            active.append(item)
        if active:
            per_person = max(1, len(active) / PEOPLE)
            efficiency = 1 / (1 + SWITCH * (per_person - 1))              # context switching
            share = PEOPLE * efficiency / len(active) if len(active) > PEOPLE else efficiency
            for item in active:
                item[2] -= share
            done = [i for i in active if i[2] <= 0]
            active = [i for i in active if i[2] > 0]
            finished += [(i[0], i[1], day) for i in done if i[0] >= WARMUP]
        if day >= WARMUP:
            in_progress_days += len(active) + len(done)
    span = DAYS - WARMUP
    throughput = len(finished) / span
    wip = in_progress_days / span
    cycle = sum(end - start + 1 for _, start, end in finished) / len(finished)
    lead = sum(end - arrived + 1 for arrived, _, end in finished) / len(finished)
    return throughput, wip, cycle, lead

print(f"{'':28}{'start everything':>18}{'limit WIP to 4':>18}")
a, b = simulate(None), simulate(PEOPLE)
for i, name in enumerate(["finished per day", "average in progress", "cycle time (start-
>done)",
                        "lead time (asked->done)"]):
    unit = " " if i < 2 else " days"
```

```

print(f"{name:28}{a[i]:>13.2f}{unit:5}{b[i]:>13.2f}{unit:5}")
print("\nLittle's law, in progress = finished per day x cycle time:")
for name, (tp, wip, cycle, _) in [("start everything", a), ("limit WIP to 4", b)]:
    print(f"  {name:18} {wip:6.2f} vs {tp:.2f} x {cycle:5.2f} = {tp * cycle:6.2f}")

```

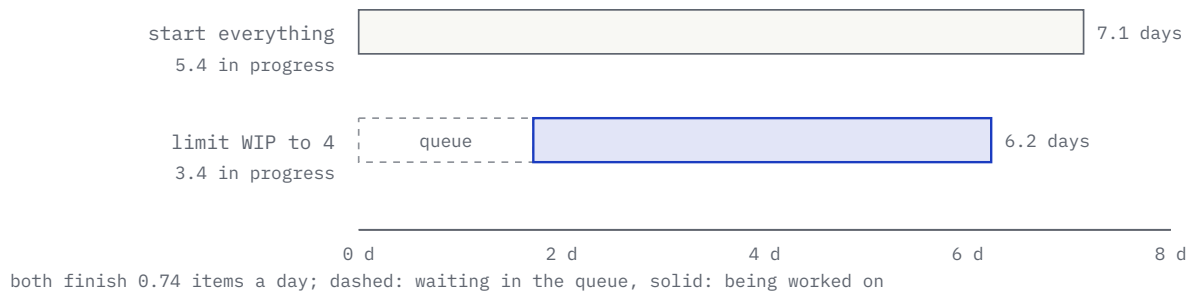
OUTPUT

	start everything	limit WIP to 4
finished per day	0.74	0.74
average in progress	5.35	3.36
cycle time (start->done)	7.14 days	4.51 days
lead time (asked->done)	7.14 days	6.23 days

Little's law, in progress = finished per day x cycle time:

start everything	5.35 vs 0.74 x 7.14 =	5.28
limit WIP to 4	3.36 vs 0.74 x 4.51 =	3.34

FROM ASKING TO DONE, ON AVERAGE



Same throughput, same requests. Starting everything removes the queue and makes each item take longer to do; limiting work in progress makes items wait a little before starting, and still finishes them sooner.

What the example shows

Both policies finish the same amount of work, 0.74 items a day, because at this load the team has enough capacity either way. The difference is entirely in how long each item takes.

Starting everything makes everything slower. With every request started at once, 5.4 items are in progress on average, people's attention is divided, and each item takes 7.1 days from start to finish. Limiting work in progress to four, 3.4 items are in progress on average and each takes 4.5 days. Little's law holds for both, to within the noise of the simulation: 5.35 against 0.74×7.14 , and 3.36 against 0.74×4.51 .

Waiting in a queue is faster than starting at once. The surprising number is lead time, from request to done. Under the work-in-progress limit, items wait in the queue for 1.7 days on average before anyone starts them, and they're still finished sooner, 6.2 days against 7.1. Starting a request immediately feels responsive to whoever asked, and it makes their request, and everyone else's, finish later. That's the case behind the Kanban slogan "stop starting, start finishing", and it applies to a person as much as to a team.

The example's numbers come from invented constants, and the switching cost is deliberately small. Exercise 15.1 shows what happens when the load rises, which is where the difference becomes dramatic.

Finishing, personally

The same law applies to one engineer's work. Five things in progress, a design document, two reviews, a bug and a migration, means each takes five times as long as it needs to, and the people waiting for each see slow progress on all of them. Keep one main thing, finish it, then take the next. Keep a "not now" list for everything else, which is honest about what's waiting rather than pretending everything is in progress. When an interruption comes, ask whether it's more important than the thing in hand; if it isn't, it goes on the list, with a date if the requester needs one.

Finishing also means finishing all of it. Chapter 10's last ten percent applies to every task: the pull request isn't done when it's approved but when it's deployed and verified; the bug isn't done when it's fixed but when the reporter knows; the document isn't done until the decision in it is recorded.

Reliability

Reliability in a person means that what they say will happen does. It's built from small habits. Write down every commitment you make, in one list, so none is forgotten. When you can't keep one, say so as soon as you know, not at the deadline, and offer what you can do instead; a renegotiated commitment costs a little trust, a missed one costs a lot. Follow up: when you ask someone for something, note when you expect it and check then. Close loops: tell people when the thing they asked about is done. Be on time. None of these is hard, which is exactly why doing them consistently is noticed. Over a few years, reliability is what makes people give you the important work, and the important work is what builds a career.

COMMON TRAPS

Starting every request at once, to look responsive, and finishing everything later.

Five things in progress, each moving slowly, none finished.

"Done" that isn't: approved but not deployed, fixed but not communicated.

Missing a commitment silently, instead of renegotiating it early.

Commitments kept in your head, where some will be forgotten.

EXERCISE 15.1 A busier team

Raise the arrival rate to 0.9 a day, by changing the 0.25 in the arrivals line to 0.3. Predict, then run, what happens under each policy.

SOLUTION

	start everything	limit WIP to 4
finished per day	0.07	0.88
average in progress	723.16	4.00
cycle time (start->done)	275.40 days	4.50 days
lead time (asked->done)	275.40 days	47.54 days

Starting everything collapses. At 90% of capacity, the switching cost of juggling many items pushes the team's real capacity below the arrival rate, so work arrives faster than it's finished, the pile in progress grows, more switching makes the team slower still, and throughput falls to almost nothing. The "average in progress" figure isn't an average of anything stable: run the simulation twice as long and it roughly doubles. Little's law no longer balances, because it only applies to a stable system, and this one isn't. With the limit, the team keeps finishing 0.88 items a day, each still takes 4.5 days to do, and the queue absorbs the load: lead time grows to about seven weeks, which is a real problem, but a visible and manageable one. The limit turns overload from a collapse into a queue, and a queue tells you exactly how much work to stop accepting.

EXERCISE 15.2 Your own work in progress

List everything you currently have in progress, including reviews and promises. Apply Little's law to it, and decide what to finish, what to put on a "not now" list, and what to hand back.

SOLUTION

A typical list is longer than expected: two features, three reviews, a bug, a design document, a promise to look at someone's slow query, an interview loop. Suppose it's nine items, and you finish about three a week. Then by Little's law each takes about three weeks, and the people waiting for the review or the query see nothing for most of that time. Sort them. Finish the small, nearly-done ones first, the reviews and the slow query, this week, because each frees someone else. Pick one main item, the feature or the document, and work on it until it's done. Put the others on a "not now" list, and tell the people waiting when you

expect to start, which is far better for them than slow, invisible progress. Hand back anything you can't start for a month, so someone else can. Repeat the audit monthly; the list grows back.

EXERCISE 15.3 Renegotiating a commitment

You promised a partner team an API by Friday. On Tuesday you realise it won't be ready until the following Wednesday. Write the message.

SOLUTION

Hi Priya, a heads-up on the refunds API I promised for Friday: it won't be ready until next Wednesday. Pagination turned out to need a schema change I hadn't expected. Two options, whichever helps you more:

1. On Friday I can give you the endpoint without pagination, which works for up to 100 refunds, and add pagination on Wednesday.
2. Or the complete API on Wednesday.

Tell me which, and I'll make it happen. Sorry for the change; I wanted you to know today rather than on Friday.

It's sent on Tuesday, the moment you knew, which gives them three days to adjust their own plans. It gives the new date, the reason in one sentence, and options that let them choose what matters to them; often a partial delivery on the original date is what they actually need. It doesn't over-apologise or bury the news. A message like this costs a little trust, and builds more than it costs, because it shows that your dates mean something: when you say Friday, Friday is either true or you'll say so early.

CHECKLIST

- ☐ Can you state Little's law, and use it on a team's board or your own list?
- ☐ Does your team limit work in progress, and finish before starting?
- ☐ Do you have one main thing, and a "not now" list for the rest?
- ☐ Is "done" really done: deployed, verified and communicated?
- ☐ Do you write down your commitments, and renegotiate early when you can't keep one?

GO DEEPER

BOOK **The Principles of Product Development Flow** Donald Reinertsen

BOOK **Making Work Visible** David J. Anderson

BOOK MAKING WORK VISIBLE DOMINICA DEGRANDIS

PAPER A Proof for the Queuing Formula: $L = \lambda W$ John D. C. Little, 1961

A career that compounds

3 exercises

Skills, reputation, relationships and written work all compound, but only if they accumulate in one direction and you can see them. Keep a few lines a week about what you did and what it changed, because you won't remember, and the people deciding your next opportunity won't either. Work on problems that matter in your area. Build the body of work the stage asks for, slowly and with other people. And keep a pace you can sustain for decades, because the compounding only works if you're still going.

What compounds

Some things in a career grow on themselves. Depth in one area makes the next layer of it easier to learn. A reputation for reliable, clear work brings better projects, which build more reputation. Relationships with people who trust you bring opportunities you'd never have found. And written work, the archive from chapter 8, keeps finding new readers. Other things don't compound: a sequence of unrelated jobs, each started from scratch; work nobody saw or remembers; knowledge that left with the last project. The difference is mostly direction and visibility. Effort that adds to the same pile, and that someone can see, compounds; effort scattered across many piles, or invisible, doesn't.

Remembering what you did

Memory is a poor record of your own work. Six months later, the incident you resolved, the design you improved, the colleague you taught are a vague sense of having been busy. That matters when the moment comes to describe your work, in a performance review, a promotion case, an interview or a CV, because what you can't remember can't be counted. Julia Evans calls the cure a **brag document**: a running record of what you did and why it mattered. The name is half a joke; the practice is entirely serious.

Keep it simple enough to survive. A plain text file, a line or two each week, with the week, what you did, a tag for the kind of work, and, when you have one, the measured result. Five minutes on Friday afternoon is enough. The example takes three months of such a log and turns it into the summaries you'll need: what themes the work fell into, which have evidence, which weeks have nothing written down, and a first draft of the paragraph a review asks for.

EXAMPLE

worklog.py · three months of a plain-text work log turned into themes, the evidence for each, the gaps, and a first draft for a review

```
"""A career that compounds starts with remembering what you did. A plain-text work log, kept a few
```

```

lines a week, turned into the summary a review, a promotion case or a CV needs."""
import re
from collections import defaultdict

LOG = """
2026-W02 Wrote the design doc for moving search to a dedicated index #design
2026-W02 Reviewed 14 pull requests, mostly for the new hire #mentoring
2026-W03 Led the search migration kickoff with platform and support #leadership
2026-W04 Fixed the nightly export that silently dropped rows #reliability impact: 3,200 rows a
night
2026-W06 Search index live for 5% of traffic #design impact: p95 2.4 s -> 240 ms
2026-W07 Paired with Priya on her first on-call shift #mentoring
2026-W07 Wrote the postmortem for the 40-minute checkout outage #reliability
2026-W08 Search at 100% #design impact: support complaints 10 a week -> 1
2026-W09 Gave the internal talk on reading query plans #writing #mentoring
2026-W11 Published "What our search migration taught us" on our blog #writing impact: 4,100
readers
2026-W12 Added lag alerting to the change stream #reliability impact: caught 2 incidents
before users did
2026-W13 Priya led her first incident review on her own #mentoring
"""

entries = []
for line in LOG.strip().splitlines():
    week, text = line.split(None, 1)
    impact = re.search(r"impact:\s*(.*)$", text)
    themes = re.findall(r"#(\w+)", text)
    what = re.sub(r"\s*(#\w+|impact:.*)", "", text).strip()
    entries.append((week, what, themes, impact.group(1) if impact else None))

def entries_word(n):
    return f"{n:2} entr{'ies' if n != 1 else 'y'}"

by_theme = defaultdict(list)
for e in entries:
    for t in e[2]:
        by_theme[t].append(e)
print("themes, by how often they appear:")
for theme, es in sorted(by_theme.items(), key=lambda kv: (-len(kv[1]), kv[0])):
    measured = sum(1 for e in es if e[3])
    print(f" {theme:11} {entries_word(len(es))}, {measured} with a measured result")

weeks = sorted({int(w[-2:]) for w, *_ in entries})
gaps = [f"W{w:02}" for w in range(weeks[0], weeks[-1] + 1) if w not in weeks]
print(f"\nweeks with nothing written down: {'', '.join(gaps)} "
      f"({len(gaps)} of {weeks[-1] - weeks[0] + 1}; those are the ones you won't remember)")

print("\nthe paragraph for a review, written from the log, one sentence per theme:")
measured_first = lambda kv: (-sum(1 for e in kv[1] if e[3]), -len(kv[1]), kv[0])
for theme, es in sorted(by_theme.items(), key=measured_first):
    best = next((e for e in reversed(es) if e[3]), es[-1])
    result = f" ({best[3]})" if best[3] else ""
    print(f" {theme.capitalize():11} {best[1]}{result}; {entries_word(len(es))} in all.")

```

OUTPUT

```

themes, by how often they appear:
mentoring      4 entries, 0 with a measured result
design          3 entries, 2 with a measured result
reliability    3 entries, 2 with a measured result
writing        2 entries, 1 with a measured result
leadership     1 entry, 0 with a measured result

weeks with nothing written down: W05, W10 (2 of 12; those are the ones you won't remember)

```

the paragraph for a review, written from the log, one sentence per theme:

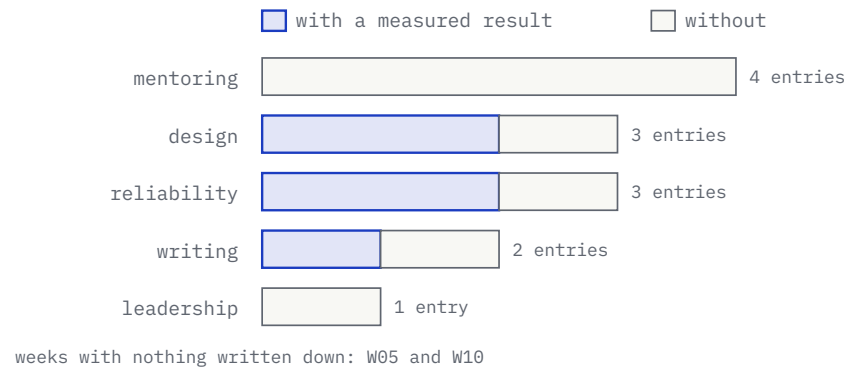
Design: Search at 100% (support complaints 10 a week -> 1); 3 entries in all.

Reliability: Added lag alerting to the change stream (caught 2 incidents before users did); 3 entries in all.

Writing: Published "What our search migration taught us" on our blog (4,100 readers); 2 entries in all.

Mentoring: Priya led her first incident review on her own; 4 entries in all.

Leadership: Led the search migration kickoff with platform and support; 1 entry in all.



Three months of work, by theme. The most frequent theme, mentoring, has no measured results; its evidence is an outcome, which is worth writing down as carefully as a number.

What the example shows

The log knows things you'd forget. Twelve entries over twelve weeks show a quarter that combined design work, reliability, writing, leadership and mentoring, with measured results for half of them: a search migration that cut p95 from 2.4 seconds to 240 ms and support complaints from ten a week to one, a fixed export recovering 3,200 rows a night, alerting that caught two incidents before users did, a post with 4,100 readers. Without the log, the review would mention the search migration and little else.

The most frequent theme has no numbers, and that's fine. Mentoring appears four times and never with a measured result. Mentoring rarely produces numbers; its evidence is outcomes in other people. The log's last mentoring entry, "Priya led her first incident review on her own", is the strongest possible evidence of the earlier three, and it's exactly the kind of thing that's forgotten. Write down outcomes like that as carefully as you'd write down a latency figure.

The gaps are the weeks you won't remember. Weeks 5 and 10 have no entries. Perhaps nothing happened; more likely, something did and wasn't written down. The draft paragraph at the end isn't finished prose, but it's a far better start than a blank page the night before a review, and it's assembled entirely from evidence.

Kinds of senior work

Senior engineers don't all do the same job. Will Larson's *Staff Engineer* describes four common shapes, which he calls archetypes. The **tech lead** guides one team's approach and execution. The **architect** owns the direction and quality of a critical area across teams. The **solver** goes deep on the hardest problems wherever they arise, one after another. The **right hand** extends a senior leader's reach across an organisation. Each needs the depth from Part I and the leadership from Part III, in different mixtures. Knowing which shape suits you, and which your organisation needs, makes the path less mysterious; the work log's themes are often a good first hint.

Working on important problems

In 1986, Richard Hamming gave a talk at Bell Labs called "You and Your Research", about why some scientists do great work and most don't. Its most famous moment is a question he used to ask colleagues at lunch: what are the most important problems in your field, and why aren't you working on them? Most people, he observed, could name the important problems and had a reason not to be working on any of them. The question translates directly to engineering. What are the problems in your area that, if solved, would matter most to your company, your field or the people who use your systems? Are any of them yours? You don't need to work on the most important problem every day. But a career that never touches one compounds slowly, whatever its effort.

The body of work

The stage's deliverable, described in the appendix, is "a body of work rather than a project": a maintained open-source contribution, a written archive others cite, and at least one system in production that other engineers depend on and you didn't build alone. Each part compounds in its own way. The contribution builds depth and a public record of judgement, through chapter 5's ladder. The archive builds reputation and keeps teaching, through chapter 8's habit. The production system builds the most valuable credibility of all, having built something real with other people that kept working. None of the three can be finished quickly, and none needs to be. Start all three, small, and let them grow together for years.

A pace you can sustain

Compounding needs time, and time needs you to still be doing the work. Burnout isn't dedication taken to its limit; it's the end of the compounding, often for years. Protect sleep, holidays and a life outside work, deliberately, as a professional obligation to your future self. Careers have seasons: some years are for intense learning or a demanding project, some for family or health, and a long career has room for both. The engineers who reach real mastery are rarely the ones who worked hardest for two years. They're the ones who kept going, steadily, in one direction, for twenty.

COMMON TRAPS

Relying on memory to describe a year of your own work

Relying on memory to describe a year of your own work.

Recording only numbers, and losing the outcomes in other people.

Scattering effort across unrelated areas, so nothing compounds.

Never asking Hamming's question, and never touching an important problem.

A pace you can't sustain, which ends the compounding entirely.

EXERCISES

EXERCISE 16.1 Start the log

Write the last four weeks of your work log from memory, then check it against your commits, calendar and chat history. What did memory leave out?

SOLUTION

```
git log --author="$(git config user.email)" --since=4.weeks --format='%ad %s' --
    date=short
gh pr list --author @me --state merged --search "merged:>=$(date -v-28d +%F)"
```

The first command lists your commits of the last four weeks; the second, if you use GitHub's command-line tool, your merged pull requests (on Linux, use `date -d '28 days ago' +%F`). Compare them, and your calendar, with what you wrote. Memory usually keeps the big, recent item and loses three kinds of work: the small fixes that each took an afternoon, the help you gave other people, reviews, debugging sessions, answered questions, and the meetings where something got decided because you'd prepared. Those last two are often the most valuable work a senior engineer does and the least visible. Add what you find, then keep the log going from now on, every Friday. Four weeks reconstructed from records is a lesson in why five minutes a week is cheaper.

EXERCISE 16.2 Which shape fits

Using your log, or the example's, decide which of Larson's four archetypes the work looks most like, and what you'd do to move towards another if you wanted to.

SOLUTION

The example's log looks like a tech lead growing towards an architect. The search migration, led from design document to full rollout with platform and support, is tech-lead work; the design documents, the postmortem and the public writing about the migration show influence beyond one team, the start of an architect's role in search. To move further towards architect, the next step would be owning the direction of search across teams: writing the multi-year plan for it, reviewing other teams' designs that touch it, and becoming the person consulted before search decisions rather than after. To move towards solver instead, the log would need more of the reliability entries, hard problems in unfamiliar parts of the system. The archetypes aren't boxes; they're a vocabulary for deciding, with your manager, what kind of work to seek next.

EXERCISE 16.3 Hamming's question

Write down the three most important problems in your area, at your company or in the field, and for each, why you aren't working on it.

SOLUTION

For an engineer whose area is databases at a growing company, perhaps: the primary database will run out of headroom in eighteen months and nobody owns the plan; restoring from backup has never been tested at full size; and the field is moving to separating storage from compute, which nobody on the team understands yet. The honest reasons not to be working on them are usually some of these: "it's not my job", "it's too big", "someone senior owns it", "I don't know enough yet". Each has a small first step. Write the one-page analysis of the headroom problem and send it to your manager. Run a full-size restore test on a weekend's staging environment. Read the two papers behind the storage-compute separation and write a summary for the team. Hamming's point was not that you should drop everything for the biggest problem, but that you should know what it is, and that small steps towards it, taken regularly, are how people end up doing the important work.

CHECKLIST

- ☐ Do you keep a work log, every week, with results and outcomes?
- ☐ Can you describe the last six months of your work, with evidence, in ten minutes?

- ☐ Do you know which kind of senior work suits you, and which your organisation needs?
- ☐ Can you name the important problems in your area, and is one of them yours?
- ☐ Have you started all three parts of the body of work, and is your pace sustainable?

GO DEEPER

ESSAY Get your work recognized: write a brag document Julia Evans

GUIDE Staff archetypes Will Larson

TALK You and Your Research Richard Hamming, 1986

The stage asks for "a body of work rather than a project": a maintained open-source contribution, a written archive others cite, and at least one system in production that other engineers depend on and that you didn't build alone. None of the three can be finished in a month, and they aren't meant to be. They're what years of the practices in this volume leave behind, and the lists below describe each when it's done. Start all three now, small, and review them every six months against these lists.

1. A maintained open-source contribution

Not a single patch, but a sustained part of a project that other people use. Chapter 5 is the model.

- 01 One project you depend on, where you've contributed over at least a year: fixes, features, documentation and, above all, reviews of other people's work.
- 02 A part of it you know well enough that maintainers ask for your review there, or give you the permission to approve changes.
- 03 Issues triaged, questions answered and newcomers helped, visibly and kindly, in the project's own channels.
- 04 Your contributions in your own words: a short page listing what you did, linked from your archive or CV, with the pull requests and reviews that show it.

2. A written archive others cite

Years of steady writing in your area, until other people link to it when they explain the subject. Chapter 8 is the model.

- 01 A site you own, with at least two dozen posts in your area, published on a regular rhythm.
- 02 A mix of hard lessons, explanations, reading notes and small tools, each with a specific title and its point in the first sentence.
- 03 At least one talk, at a meetup, a conference or your company's engineering

forum, made from posts that were well received.

- 04 Evidence that it's useful to others: links from other people's writing, questions from readers, a post that keeps being found by search.
-

3. A system others depend on

Something in production, built with other people, that other engineers rely on and that kept working. Parts III and IV are the model.

- 01 A design document that survived review, with alternatives, risks and a rollout, and the decision records that followed it.
 - 02 A project led end to end: scoped, sequenced risk first, forecast honestly, communicated to its stakeholders, and landed, with the old path removed.
 - 03 Engineers outside your team who depend on it, and documentation, runbooks and alerts good enough that they rarely need you.
 - 04 Its delivery and reliability measured, and at least one incident learned from in a blameless postmortem whose actions were done.
 - 05 Other people who could run it without you, because you taught them, which is the last test of having built something that lasts.
-

Keep the work log from chapter 16 throughout. In five years it's the record of how the body of work was built, and the evidence, for yourself as much as anyone, that it was.

THE SIXTEEN RULES

One per chapter, each linked back to it. Read them when choosing what to work on, and again when a project, a review or a conversation goes badly.

-
- 01 Choose an area to go deep in, write down why, and check the judgement your choice depends on.
-
- 02 Read papers in passes, with a question, and build the central idea small enough to run.
-
- 03 Find the assumption each design depends on; that's where it fails.
-
- 04 Map a codebase before reading it, trace what actually runs, then fix something.
-
- 05 Send patches that cost little to accept: small, tested, explained, in the project's style.
-
- 06 Propose in the first paragraph, put numbers on goals, and show the alternatives you rejected.
-
- 07 Write uncertain inputs as ranges, and argue about the assumption that decides it.
-
- 08 Lead with the point, replace adjectives with facts, and publish where you own the archive.
-
- 09 Estimate in ranges along the real dependencies, and commit at P85.
-
- 10 Sequence work so the questions that could change the plan are answered first.
-
- 11 Bottom line first, colour from the forecast, and every update ends with an ask and a date.
-
- 12 Diagnose before teaching, teach at the frontier, and sponsor as well as advise.
-
- 13 Price work in the business's units, with each assumption beside its number.

-
- 14 Measure speed and stability together, for teams, to learn rather than to judge.
-
- 15 Limit work in progress, finish before starting, and renegotiate commitments early.
-
- 16 Write down what you did every week, work on important problems, and keep a pace that lasts.
-

READING LIST

The books, papers and essays behind this volume, and the chapters each one deepens. If you read only two books, make them Tanya Reilly's *The Staff Engineer's Path*, the most useful book about senior technical work, and Fred Brooks's *The Mythical Man-Month*, fifty years old and still describing your project. For papers, start with the seven in chapter 3, in order.

WORK	AUTHOR	DEEPENS
The Staff Engineer's Path	Tanya Reilly	Ch. 1, 10–12, 16
Staff Engineer	Will Larson	Ch. 1, 11, 16
The Mythical Man-Month	Fred Brooks	Ch. 9–10
How to Read a Paper	S. Keshav	Ch. 2
Papers We Love	the Papers We Love community	Ch. 2–3
The Architecture of Open Source Applications	ed. Amy Brown & Greg Wilson	Ch. 4
Producing Open Source Software	Karl Fogel	Ch. 5
Design Docs at Google	Malte Ubl	Ch. 6
How to Measure Anything	Douglas Hubbard	Ch. 7, 9
Technical Writing	Google for Developers	Ch. 6, 8
Software Estimation	Steve McConnell	Ch. 9
The Manager's Path	Camille Fournier	Ch. 12
The Principles of Product Development Flow	Donald Reinertsen	Ch. 13, 15
Accelerate	Forsgren, Humble & Kim	Ch. 14
You and Your Research	Richard Hamming	Ch. 1, 16

Compiled October 2026 as the companion to stage 15, Mastery, of The Engineer's Syllabus, alongside *The Engineer's Whetstone, Primer, Workbench, Compass, Mill, Loom, Exchange, Press, Ledger, Craft, Shipyard, Fleet, Watch, Citadel* and *Foundry*, which completes the series: every stage now has a volume. It follows the practices the stage lists, and draws on the works in the reading list and on the original sources for its named results. These are Little (1961) on queues; Brooks (1975) on adding people to late projects; Kahneman and Tversky (1979) on the planning fallacy; Hamming (1986) on important problems; the Google File System (2003), MapReduce (2004), Bigtable and Chubby (2006), Dynamo (2007), Spanner (2012) and Raft (2014); Keshav (2007) on reading papers; Klein (2007) on pre-mortems; and the DORA research summarised in *Accelerate* (2018). The examples' numbers are invented, as each chapter says, so they show the shape of each argument rather than measure any real team.

Every example was executed before publication with Python 3.14.7 on macOS 26, using only the standard library, and the output printed under each one is what it printed. They're written for Python 3.12 or later and to print the same output on Linux: random choices use fixed seeds, Git runs with fixed names and dates and ignores your own settings, and anything else that varies between machines is kept out of the output. Chapter 4's map and trace describe the `json` package as shipped with Python 3.14, and change slightly between versions. Each example was also run a second time in a deliberately hostile environment, with a different time zone, temporary folder and home folder with its own Git settings, and printed exactly the same. Tested here: a decision matrix and its fragility; MapReduce with a combiner, re-execution and backup tasks; Spanner's commit wait against skewed clocks; a standard-library package mapped and traced with its accelerator on and off; two patches reviewed with `git am`; a design-document reviewer; a three-year cost comparison over twenty thousand futures; a prose checker; a task graph simulated for its percentiles and critical path; three sequencings of a risky project; a status update computed from a forecast; a prerequisite graph for mentoring; unit economics and a ranking by cost of delay; the four DORA keys from a Git history; Little's law under two policies; and a work log summarised.