Documentation & Runbooks

A Complete 10-Chapter Technical Support Course

Topics covered:

KB articles, runbooks & tickets · writing for a reader under pressure

Structuring a runbook · capturing tribal knowledge

Documenting without creating a security liability · fighting silent decay

Testing a runbook · organizing & making documentation findable

Capstone: three fresh tasks — promotion, capture, and audit

Exercises: 30 hands-on scenarios with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

"We Wrote It Down" Isn't the Same as "Someone Can Follow It"

Documentation & Runbooks

Chapter 1 · "We Wrote It Down" Isn't the Same as "Someone Can Follow It"

Every course in this subject so far has assumed good documentation already exists somewhere to consult — a ladder to follow, a procedure to reference, a runbook to reach for. This course exists because that assumption is often false, and because writing something down is only the first of two very different achievements. `backup1` spent an entire course establishing that "we have backups" isn't the same claim as "we can recover." This course makes the identical argument about documentation: "we wrote it down" isn't the same claim as "someone under pressure, unfamiliar with the system, can actually follow it and succeed."

What Every Prior Chapter in This Subject Quietly Assumed

Each Technical Support course so far has assumed a piece of documentation exists somewhere, ready to be used. This course exists because writing that documentation, and making it actually work, is a real skill nobody else in this subject has taught:

COURSE	WHAT IT QUIETLY ASSUMES
log1	Logs exist and are trustworthy — not that anyone wrote down how to interpret them
netdiag1 / perfdiag1 / appdiag1	A diagnostic ladder exists in the reader's head — not that it's captured anywhere for the next person
incident1	A ticket captures what happened for *this* incident — a one-off record, not reusable knowledge
remote1	You know how to reach a system — not that anyone wrote the procedure down for a new hire to follow safely
secsupport1	You can recognize an attack — not that the actual response procedure is documented anywhere
backup1	A restore procedure exists to follow — and, per its own core lesson, existing isn't the same as working

This course is specifically about producing the thing every other course assumes is already sitting there, ready and correct, when it's actually needed.

Not a Sysadmin-Architecture Course

This course isn't about designing systems or deciding what tooling an organization should use — that's a separate discipline. It's about capturing and communicating knowledge about systems that already exist, so the next person who needs that knowledge doesn't have to rediscover it under pressure.

The Central Claim of This Whole Course

"We wrote it down" confirms only that words exist in a document somewhere. It confirms nothing about whether those words are complete, accurate, still current, or genuinely followable by someone who isn't already the expert who wrote them. Those are different claims, and — exactly as `backup1` argued for backups — only one of them is actually proven by someone genuinely unfamiliar with the system successfully following the document, under real conditions.

> **THE ONE IDEA TO CARRY THROUGH THIS WHOLE COURSE**
> A document existing and a document working are two different claims. Chapter 8 covers the only thing that actually proves the second one — and it's the exact same discipline `backup1` built around test restores, applied to documentation instead of data.

What This Course Actually Covers

- **Choosing and structuring the right document** (Chapters 2–4) — KB article vs. runbook vs. ticket, writing for a reader under pressure, and a runbook's own proper shape
- **Capturing and protecting knowledge** (Chapters 5–6) — getting tribal knowledge out of people's heads before it's lost, and doing so without creating a security liability
- **Keeping documentation actually trustworthy** (Chapters 7–8) — fighting silent decay over time, and testing a runbook the way `backup1` tests a backup
- **Making it usable** (Chapter 9) — organizing documentation so the right person can actually find it when they need it

An Outage That Isn't Resolved Yet

An on-call technician, alone during a real outage at 3am, pulls up an existing runbook for restarting a specific service and follows it step by step. It fails partway through — step 4 references a server hostname that was decommissioned eight months ago during a migration nobody updated the runbook for. The outage takes twice as long to resolve as it should have. Nothing about this is resolved in this chapter — it's deliberately left open. Chapter 8 comes back to it directly, once the material on testing a runbook has actually been covered.

WHAT THIS COURSE WON'T GIVE YOU

A single one-size-fits-all template to copy and reuse everywhere. Templates matter far less than genuinely understanding the reader's own situation — Chapter 3 covers that directly, and it's the foundation everything else in this course builds on.

Hands-On Exercises

EXERCISE 1

Explain the parallel this chapter draws between its own central claim and `backup1`'s central claim, and why the same underlying logic applies to both documentation and backups.

View solution

EXERCISE 2

Using the comparison table, identify the shared assumption across the other seven Technical Support courses this new course exists to question, and explain why it can't always be trusted.

View solution

EXERCISE 3

Explain why the 3am outage scenario is left deliberately unresolved in this chapter, and name the specific chapter that returns to it.

View solution

CHAPTER 1 QUICK REFERENCE

- This course covers **producing** documentation, not consuming documentation someone else already wrote
- Core claim: writing something down proves it exists, not that it's accurate, current, or followable under pressure
- Four areas ahead: choosing/structuring documents, capturing knowledge safely, keeping documentation current, making it findable
- **Core idea:** only someone genuinely unfamiliar successfully following a document proves it actually works — the same test-restore logic `backup1` applied to backups
- Next: Chapter 2, KB articles, runbooks, and tickets as three genuinely different documents

CHAPTER 2 OF 10

KB Articles, Runbooks & Tickets: Three Different Documents for Three Different Purposes

Documentation & Runbooks

Chapter 2 · KB Articles, Runbooks & Tickets: Three Different Documents for Three Different Purposes

A support technician produces three genuinely different kinds of documents, and conflating them is one of the most common, most quietly damaging mistakes in this whole subject. Each answers a different question, for a different reader, at a different point in time — knowing which one you're actually writing before you start is the first real skill this course teaches.

Three Documents, Three Questions

KB Article

Answers: "*What is this, and why does it behave this way?*"

Explanatory, conceptual, durable — background and context, not necessarily a walkthrough.

Runbook

Answers: "*What do I do, step by step, to accomplish this?*"

Procedural, durable, reusable — a precise sequence of actions. Chapter 4 covers its own proper shape.

Ticket

Answers: "*What happened this specific time?*"

A one-off historical record for a single incident — `incident1`'s own territory, not a reusable procedure.

Why Conflating Them Actually Hurts

- **A runbook padded with background explanation** becomes hard to follow step by step under real pressure — the "why" belongs in a linked KB article, not woven into the steps themselves
- **A ticket written as if it were a reusable procedure** wastes effort generalizing something that may never recur in exactly that form — and if it does recur, it belongs promoted into a real runbook, not left buried in ticket history where nobody will find it again

- **A KB article with no procedural content**, when the reader actually needed concrete steps, leaves them with theory and no clear action to take

When One Should Become Another

A single ticket documenting one incident is exactly the right document the first time something happens. Once the same problem recurs — the specific pattern `incident1``s own material teaches recognizing — that recurring pattern deserves promotion into a genuine runbook, not another one-off ticket that repeats the same investigation from scratch. Likewise, if a runbook's own background section keeps growing, that's a sign it should be split out into its own linked KB article, leaving the runbook itself lean and purely procedural.

Comparing the Three Directly

	KB ARTICLE	RUNBOOK	TICKET
Purpose	Explain	Guide action	Record history
Scope	General, durable	General, durable	One specific instance
Reusable?	Yes	Yes	No — but may reveal a pattern worth promoting
Shape	Prose, diagrams, context	Numbered steps, expected outputs	Timeline, what was done, resolution

Worked Example: From Ticket to Runbook to KB Article

A specific service crashes under load for the first time — a ticket records what happened and how it was resolved. It happens again, and a third time — the recurring pattern is now promoted into a genuine runbook: the exact steps to mitigate and restart the service safely. Separately, a linked KB article explains *why* the service crashes under load in the first place — the underlying architectural cause, useful context for understanding the problem, but not itself a set of actionable steps. Three documents, three purposes, each doing its own job.

> **DON'T MAKE ONE DOCUMENT TRY TO DO ALL THREE JOBS**
>
> Cramming explanation, procedure, and incident-specific detail into a single document usually means it serves none of the three purposes well — a reader looking for quick steps has to wade through background, and a reader looking for context has to wade through procedural minutiae that doesn't apply to their situation.

Hands-On Exercises

EXERCISE 1

Using the worked example, explain why the crashing service eventually needed all three document types, rather than one thorough document covering everything.

View solution

EXERCISE 2

Explain why a runbook padded with background explanation is described as harder to follow under pressure, even though the added information is accurate and relevant.

View solution

EXERCISE 3

Explain what specifically should trigger promoting a ticket-documented issue into a real runbook, and why writing a runbook after the very first occurrence would usually be premature.

View solution

CHAPTER 2 QUICK REFERENCE

- **KB article:** what is this and why — explanatory, durable, not necessarily procedural
- **Runbook:** step-by-step actions to accomplish a specific task — durable, reusable, procedural
- **Ticket:** what happened this specific time — one-off, historical, not reusable on its own
- A recurring pattern in tickets deserves promotion into a runbook; a growing background section deserves its own linked KB article
- Don't make one document try to serve all three purposes at once
- Next: Chapter 3, writing for the reader at 3am — clarity under pressure

CHAPTER 3 OF 10

Writing for the 3am Reader: Clarity Under Pressure

Documentation & Runbooks

Chapter 3 · Writing for the 3am Reader: Clarity Under Pressure

Chapter 2 distinguished which document you're writing. This chapter covers how to actually write it — specifically for the reader this whole course keeps returning to: someone stressed, tired, possibly unfamiliar with the system, and quite possibly not the document's own author, trying to resolve a real problem right now.

The Core Design Principle: Assume the Reader Isn't You

Not your knowledge, not your context, not your calm state of mind. Write as if for a stranger under pressure — not a future version of yourself who remembers writing it and already knows what every step means.

Concrete Techniques

- **Short, imperative steps** — "Run `X`," not "You might want to consider running `X`"
- **One action per step** — never bundle multiple actions into a single numbered instruction
- **State the expected result after each step** — so the reader can confirm they're actually on track before moving forward, rather than discovering something went wrong three steps later
- **No unexplained jargon or internal-only abbreviations** — define anything non-obvious on first use, or avoid it entirely
- **Concrete over vague** — "restart the `nginx` service," not "restart the web server," if there's any ambiguity about which service that actually means
- **Never assume unstated context** — "obviously you'd check the logs first" isn't obvious to a stressed stranger; say it explicitly

Why This Matters Specifically at 3am

Stress and fatigue measurably reduce reading comprehension and increase the tendency to skip steps or misread instructions. A document that reads as perfectly clear to a calm, attentive author reviewing their own work may fail completely for a stressed reader encountering it cold. This isn't a claim that Chapter 1's own runbook was badly written in the first place — its failure was a different problem (Chapter 8's own territory) —

but it's exactly why writing quality matters more, not less, in documents meant to be used under these specific conditions.

One Clear Path, Fallbacks Kept Separate

Avoid branching, conditional prose in the main procedure — "if this doesn't work, you could try either A or B depending on..." embedded mid-step slows down the common case for every reader, including the vast majority who won't need the fallback at all. State one clear primary path, and keep troubleshooting or fallback steps clearly separated out, not interleaved with the steps most readers will actually follow.

Before and After

SITUATION	VAGUE	CLEAR
Restarting a service	"Restart the service if needed"	"Run `systemctl restart nginx`. Expected result: no error output, and `systemctl status nginx` shows 'active (running)'."
Multiple actions in one step	"Check the logs, and if you see errors, escalate"	Split into two numbered steps — one for checking, one for escalating, each with its own expected result
Assumed context	"Obviously, confirm the backup exists first"	"Confirm a backup exists for today's date before continuing. See Step 1."

"OBVIOUSLY" AND "SIMPLY" ARE WORTH CUTTING FROM EVERY DRAFT

If a step needs the word "simply" to sound easy, that's often a sign it's actually doing multiple things at once, or quietly skipping a prerequisite the reader hasn't been told about yet. Both words are a signal to go back and look more closely at the step they're attached to.

Hands-On Exercises

EXERCISE 1

Explain why stating the expected result after each step matters, specifically in terms of when a mistake gets caught versus when it doesn't.

View solution

EXERCISE 2

Explain why embedding conditional branches ("if this doesn't work, try A or B") directly in the main procedure is described as slowing down the common case, even for readers who never need the fallback at all.

View solution

EXERCISE 3

Explain why "simply" in a runbook step is treated as a warning sign rather than a harmless filler word.

View solution

CHAPTER 3 QUICK REFERENCE

- Assume the reader isn't you — not your knowledge, not your context, not your calm state of mind
- Short imperative steps, one action per step, an explicit expected result after each one
- No unexplained jargon, no vague references, no assumed unstated context
- One clear primary path — keep troubleshooting and fallbacks separated out, not interleaved
- Cut "obviously" and "simply" — both are signals a step needs a closer look
- Next: Chapter 4, structuring a runbook properly — preconditions, steps, verification, rollback

CHAPTER 4 OF 10

Structuring a Runbook: Preconditions, Steps, Verification & Rollback

Documentation & Runbooks

Chapter 4 · Structuring a Runbook: Preconditions, Steps, Verification & Rollback

Chapter 3 covered how to write each individual step. This chapter covers the shape the runbook itself needs as a whole — four structural sections that, together, make a runbook genuinely complete and safe to use, not just a list of individually well-written instructions.

The Four Sections

1. Preconditions

What must be true before starting — required access, tools, and confirming you're actually on the correct system.

2. Steps

The numbered procedure itself, following Chapter 3's own writing principles — one action and an expected result per step.

3. Verification

An explicit, end-to-end check confirming the actual goal was achieved — not just that the last step ran without error.

4. Rollback

What to do if something goes wrong partway through, or the procedure completes without actually fixing the problem.

Preconditions Deserve Real Weight

Preconditions are easy to write as a single throwaway line, but skipping them properly is a common way a runbook goes wrong before step 1 even happens. This connects directly to `remote1``'s own material on working safely on a system you don't own — confirming you're actually on the correct system, and that you have the access and authorization the procedure needs, belongs here explicitly, not left as an unstated assumption.

Verification Is Different From a Per-Step Expected Result

Chapter 3's per-step expected results confirm each individual action worked. Verification is a separate, final check confirming the whole procedure achieved its actual goal — every step can complete without error while the underlying problem remains unresolved. A runbook without this final check can report success while leaving the real issue untouched.

Rollback: The Most Neglected Section

People write the happy-path steps carefully and then skip planning for failure — even though a runbook is disproportionately likely to be used exactly when something is already going wrong, which makes hitting a genuine failure partway through more likely, not less, compared to routine documentation used under calm conditions.

> **NO ROLLBACK SECTION IS AN UNSTATED ASSUMPTION OF ITS OWN**
>
> A runbook without a rollback section implicitly assumes the procedure will always succeed — exactly the kind of assumption Chapter 1's own outage scenario demonstrates can't be trusted. Every runbook needs an explicit answer to "what do I do if this goes wrong partway through," not silence on the question.

Worked Example: A Runbook Skeleton

```
RUNBOOK: Restarting a Stuck Background Worker

PRECONDITIONS
- Confirm you are connected to the correct production host (check hostname
  matches the current, current-as-of-today worker server — see Chapter 1)
- Confirm you have sudo access to the worker service account

STEPS
1. Run: systemctl status worker.service
   Expected: shows "active (running)" but with a queue depth above 500
2. Run: systemctl restart worker.service
   Expected: no error output
3. Run: systemctl status worker.service
   Expected: shows "active (running)" with queue depth actively decreasing

VERIFICATION
- Confirm queue depth has dropped below 100 within 5 minutes of restart
- Confirm no new errors appear in worker.log during that window

ROLLBACK
- If queue depth does not drop after 10 minutes, stop the worker service
  and escalate per incident1's own escalation process — do not attempt
  a second restart without investigating further
```

Hands-On Exercises

EXERCISE 1

Explain why verification is described as genuinely different from a per-step expected result, and give an example of a runbook that could pass every per-step check while still failing verification.

View solution

EXERCISE 2

Explain why rollback is described as more likely to be needed in a runbook than in routine documentation, using this chapter's own reasoning about when runbooks actually get used.

View solution

EXERCISE 3

Explain why confirming preconditions is connected directly to `remote1`'s own material on working safely on a system you don't own, rather than treated as a separate, unrelated concern.

View solution

CHAPTER 4 QUICK REFERENCE

- **Preconditions:** required access, tools, and confirming you're on the correct system — not a throwaway line
- **Steps:** the numbered procedure, per Chapter 3's own writing principles
- **Verification:** an end-to-end check confirming the actual goal was achieved, separate from per-step results
- **Rollback:** what to do if something goes wrong — the most commonly neglected section, despite being disproportionately needed
- A runbook missing rollback implicitly assumes the procedure always succeeds
- Next: Chapter 5, capturing tribal knowledge before it walks out the door

CHAPTER 5 OF 10

Capturing Tribal Knowledge Before It Walks Out the Door

Documentation & Runbooks

Chapter 5 · Capturing Tribal Knowledge Before It Walks Out the Door

Chapters 2 through 4 assumed you already know what needs documenting. This chapter covers a harder problem: knowledge that only exists in one experienced person's head, never written down anywhere at all. This is the more extreme version of Chapter 1's own central claim — here there isn't even a flawed document to fall back on, only what one person happens to remember.

Why This Is a Real, Common Risk

An experienced technician often "just knows" how to handle a specific recurring quirk without ever having written it down — it works fine as long as they're around, and becomes a serious gap the moment they're not, whether that's a resignation, a transfer, or simply being unreachable during an incident that happens to hit while they're on leave.

Why Simply Asking Isn't Enough

People are notoriously bad at fully verbalizing procedural knowledge they perform automatically — steps that have become second nature get silently skipped when someone's asked to just describe what they do, not because they're hiding anything, but because they genuinely stop consciously noticing those steps.

Watch, Don't Just Ask

The single most effective technique is watching someone actually perform the task and writing down what they *actually* do — not what they said they'd do beforehand. The gap between the interview answer and the observed reality is often exactly where the missing documentation was hiding all along.

WORKED EXAMPLE: THE INTERVIEW-VS-OBSERVATION GAP

Asked how they handle a stuck worker process, an experienced technician says: "I just restart the service and check the logs." Watched actually doing it, they also check a specific configuration value first, and wait a specific amount of time before checking logs at all — details they never mentioned, because both had become fully automatic to them. Neither detail was a secret; neither was consciously remembered enough to say out loud.

Two Questions That Draw Out the Rest

- **"What would you do if X specific thing went wrong?"** — asked for several plausible failure variations, this draws out troubleshooting and rollback knowledge (Chapter 4's own territory), which tends to be the most tribal knowledge of all, since the failure paths are rehearsed far less often than the happy path
- **"How would you know if this actually worked?"** — draws out verification knowledge (also Chapter 4's own territory) that experienced people often check unconsciously without realizing they're doing it at all

Timing: This Is Proactive Work, Not an Emergency

Waiting until someone announces they're leaving is often too late to do this thoroughly — a rushed knowledge-transfer exercise days before someone's last day rarely reproduces the depth a calm, ongoing capture process would. This works best as a routine part of normal operations, not a crisis response.

Whose Knowledge to Prioritize

Start with the people who've been in a specific role or system the longest, and specifically anyone who's the *only* person who knows how to do a particular task — a genuine single point of failure in the team's own knowledge, worth identifying and addressing deliberately rather than discovering by accident when they're unavailable.

From Notes to a Real Document

Once captured, interview and observation notes are raw material, not a finished product — they still need to become an actual runbook or KB article, following Chapter 2's own document-type distinction and Chapters 3–4's own writing and structural principles, not left sitting as unstructured notes nobody else can easily use.

> **THIS ISN'T A ONE-TIME PROJECT**
>
> Capturing today's tribal knowledge doesn't prevent new tribal knowledge from quietly accumulating again as systems change — a related concern Chapter 7 covers directly for documentation that already exists. Treat this as an ongoing practice, not something to check off once and consider finished.

Hands-On Exercises

EXERCISE 1

Using the worked example, explain why the technician never mentioned the config-value check or the wait time during the interview, even though neither detail was intentionally withheld.

View solution

EXERCISE 2

Explain why "what would you do if X went wrong" is described as drawing out the most tribal knowledge of all, more so than questions about the normal happy path.

View solution

EXERCISE 3

Explain why waiting until someone announces they're leaving is described as often too late, rather than simply less convenient.

View solution

CHAPTER 5 QUICK REFERENCE

- Tribal knowledge is the extreme version of Chapter 1's own claim — here, nothing was ever written down at all
- Interviews alone are unreliable — people silently skip steps that have become automatic to them
- Watching someone actually perform a task reveals what they'd never think to mention
- Ask about failure paths and verification specifically — both tend to be the most tribal knowledge of all
- Do this proactively and ongoingly, not as a rushed exercise once someone's already leaving
- Prioritize single points of failure in the team's own knowledge
- Notes are raw material — turn them into a real runbook or KB article per Chapters 2–4
- Next: Chapter 6, documenting without creating a security liability

CHAPTER 6 OF 10

Documenting Without Creating a Security Liability

Documentation & Runbooks

Chapter 6 · Documenting Without Creating a Security Liability

Chapters 2 through 5 pushed toward documentation that's more complete, more detailed, and captures more of what an experienced person actually knows. That same completeness has a real cost: the more thoroughly useful a document is to a legitimate technician resolving an incident quickly, the more useful it can potentially be to someone who shouldn't have it at all. This chapter is about writing genuinely useful documentation without it becoming an attack tool sitting in plain sight.

The Core Rule: Never Embed Real Sensitive Values

Never write an actual password, a real API key, or a genuine sensitive configuration value directly into a runbook or KB article — even an internal one, even one you trust everyone currently reading it. Use a clear placeholder instead (`<YOUR_API_KEY>`), and reference a proper secrets vault or access-management process rather than the literal value.

Why "It's Internal Only" Isn't a Sufficient Excuse

This is `secsupport1``'s own least-exposure principle, applied specifically to documentation content itself, not just live data handling during a ticket. Internal documentation systems get compromised too — an insider misusing legitimate access (`secsupport1``'s own curiosity-access material), or a single compromised account with ordinary read access to the internal wiki. Documentation systems are frequently subject to less scrutiny than production systems themselves — the same reasoning `backup1` applied to backup infrastructure being an attractive, under-monitored target applies directly here.

Beyond Credentials: Attack-Blueprint-Level Detail

Some non-credential detail is still risky to document in full — exact firewall rule logic, or specific detection thresholds precise enough to let someone deliberately stay just under an alert threshold. A runbook can be specific enough to be genuinely useful to a legitimate reader without functioning as a literal blueprint for anyone else who happens to get hold of it.

Access Control on the Documentation Itself

Not every runbook needs to be visible to the entire organization. Documentation covering security-incident handling or disaster-recovery procedures deserves the same least-exposure access thinking `secsupport1` and `backup1` already applied to systems and backups — restrict access to whoever genuinely needs it, rather than defaulting to "everyone can see everything" purely because that's the more convenient default.

Reference, Don't Reveal

STEP	REVEALS THE SENSITIVE VALUE	REFERENCES IT INSTEAD
Authenticating to an API	"Use API key `sk_live_9f2...`"	"Retrieve the production API key from the secrets vault, field `prod-api-key`"
Accessing an elevated account	Documenting a shared admin password inline	"Confirm you have escalated access via the access-request process before continuing"

Both right-hand versions give a legitimate reader exactly what they need to proceed, without the document itself ever holding the actual sensitive value.

THIS TENSION HAS NO UNIVERSAL ANSWER

Writing a genuinely detailed, useful runbook while avoiding a security liability requires real judgment — what's genuinely needed for a legitimate reader to succeed, and what would also help an illegitimate reader succeed, aren't always the same list. There's no formula that resolves this automatically; it has to be weighed deliberately, document by document.

Hands-On Exercises

EXERCISE 1

Explain why this chapter draws a direct parallel between documentation systems and `backup1`'s own reasoning about backup infrastructure being under-scrutinized, rather than treating this as an unrelated new concern.

View solution

EXERCISE 2

Using the reference-don't-reveal table, explain why the "good" versions still give a legitimate reader everything they need, despite never containing the actual sensitive value.

View solution

EXERCISE 3

Explain why this chapter says there's no universal formula for balancing usefulness against security liability, using the specific example of detection-threshold detail.

View solution

CHAPTER 6 QUICK REFERENCE

- Never embed real credentials or sensitive values — use placeholders and reference a secrets vault instead
- "Internal only" isn't a sufficient excuse — insider misuse and documentation-system compromise are both real risks
- Some non-credential detail (exact firewall logic, detection thresholds) can still function as an attack blueprint if too precise
- Apply least-exposure access control to sensitive documentation itself, not just to systems and data
- Reference, don't reveal — point to where a value lives rather than writing it inline
- Balancing usefulness against liability requires judgment, not a fixed formula
- Next: Chapter 7, keeping documentation current — the silent decay problem

CHAPTER 7 OF 10

Keeping Documentation Current: The Silent Decay Problem

Documentation & Runbooks

Chapter 7 · Keeping Documentation Current: The Silent Decay Problem

Chapters 2 through 6 covered producing genuinely good, secure documentation. This chapter covers what happens to it afterward — because a document doesn't stay accurate forever, and the way it goes wrong is exactly the same shape `backup1` described for silent backup failures: nothing announces that a document has become wrong. It sits there, looking complete and authoritative, until someone follows it and it fails — precisely what happened in Chapter 1's own outage scenario.

The Direct Parallel to backup1

A document that was accurate when written is exactly like a backup job that completed successfully when it ran — both can silently stop being trustworthy as the underlying reality changes, with no alert, no error, and no visible signal that anything is now wrong.

What Causes Decay

- **Systems change** — servers get renamed or decommissioned (exactly Chapter 1's own scenario), software gets upgraded, procedures get updated — without the documentation being updated to match
- **Organizational change** — the person who wrote it leaves, and Chapter 5's own captured tribal knowledge can go stale too if nobody maintains it after the original author is gone
- **Nobody owns it** — no single person or team is actually responsible for keeping a specific document accurate, so it's genuinely nobody's job to notice when it's wrong

Why Decay Is Genuinely Silent

Unlike a broken link or an obvious formatting error, a runbook with an outdated hostname looks completely normal — it reads clearly, has all four sections Chapter 4 described, and follows every writing principle from Chapter 3. The only way to actually discover it's wrong is to try using it live (which is exactly the discovery method Chapter 1's own outage suffered through) or to deliberately audit it against current reality before that happens — Chapter 8's own testing practice.

Combating Decay

- **Explicit ownership** — every significant document has a named owner responsible for it, not left as an ownerless artifact nobody feels accountable for
- **A defined review cadence** — mirroring `backup1`'s own testing cadence, a periodic scheduled review rather than "whenever someone happens to notice"
- **Tying documentation updates into the change process itself** — when a system changes, updating anything that references it should be part of that change's own checklist, not a separate afterthought easy to forget
- **A visible "last reviewed" date** — an honest, immediate signal letting a reader judge how stale a document might be, even before discovering whether it's actually wrong

RESOLVING CHAPTER 1'S OWN OUTAGE, IN FULL

The runbook's hostname reference became wrong eight months before it was actually needed, during a server migration. The migration's own checklist never included "update the runbook" as a step — the documentation quietly drifted out of sync with reality, completely undetected, until the exact moment a real outage depended on it being correct.

A "LAST REVIEWED" DATE IS HONEST, BUT NOT SUFFICIENT ON ITS OWN

A document reviewed recently could still have been reviewed carelessly — a rubber-stamped date bump with no genuine scrutiny behind it. A review needs to actually verify accuracy against current reality, not simply update a timestamp to look current.

Hands-On Exercises

EXERCISE 1

Explain the specific parallel this chapter draws between documentation decay and `backup1`'s own silent backup failure, and why both are described as genuinely silent rather than merely easy to overlook.

View solution

EXERCISE 2

Using the resolution of Chapter 1's own outage, explain why tying documentation updates into the change process itself would have prevented the failure, when a "last reviewed" date alone might not have.

View solution

EXERCISE 3

Explain why a "last reviewed" date is described as honest but not sufficient on its own, and what specifically could make a recently-reviewed document still wrong.

View solution

CHAPTER 7 QUICK REFERENCE

- Documentation decay is the same silent-failure shape `backup1` described for backups — no alert, no visible signal, until someone depends on it
- Causes: systems change, people leave, nobody owns the document
- Decay is invisible on inspection — only actually using the document, or a deliberate audit, reveals it
- Countermeasures: named ownership, a defined review cadence, updates tied into the change process, a visible "last reviewed" date
- Chapter 1's outage fully explained: the migration checklist never included updating the runbook
- A recent review date doesn't guarantee genuine scrutiny happened
- Next: Chapter 8, testing a runbook like you'd test a backup

CHAPTER 8 OF 10

Testing a Runbook Like You'd Test a Backup

Documentation & Runbooks

Chapter 8 · Testing a Runbook Like You'd Test a Backup

Chapter 7 explained why documentation decays silently. This chapter delivers the actual practice that closes the gap — the exact discipline `backup1`'s own Chapter 5 built for backups, applied here to documentation: only someone genuinely attempting to follow a document proves it actually works.

Why the Author Testing Their Own Document Isn't Enough

The author already knows what they meant — which means they'll unconsciously fill in every gap and resolve every ambiguity a genuine stranger would trip over. This is the same underlying problem Chapter 5's own "watch, don't just ask" material identified: an expert's own automatic knowledge quietly contaminates their ability to notice what's actually missing from what they wrote down.

What Counts as a Genuine Test

- **Ideal:** someone unfamiliar with the specific procedure actually executes it, in a safe test environment when the real action would be risky or destructive
- **A reasonable substitute:** when live execution genuinely isn't safe or practical, a careful, skeptical walkthrough — checking that every referenced hostname, value, and tool actually still exists and is current — mirroring `backup1`'s own "sampling beats nothing" reasoning for backup sets too large to fully restore-test
- **What to confirm:** that preconditions are genuinely checkable as written, steps produce the stated expected results, verification actually confirms the real goal, and rollback is itself followable — Chapter 4's own four sections, each independently confirmed to hold up

Test Cadence, Tied to Chapter 7's Own Review Material

A test is a stronger, more rigorous form of the review Chapter 7 already established, not a separate unrelated task. Critical runbooks — the ones reached for during high-severity incidents — deserve more frequent, more rigorous testing than a rarely-used KB article ever needs.

Documenting Test Results

Each test should record the date, who tested it, what was found, and whether anything needed fixing — creating a real trail of evidence, echoing `log1`'s own foundational discipline, rather than an untracked, fading sense that a document "still seems fine."

RESOLVING CHAPTER 1'S OUTAGE COMPLETELY

Had that runbook been walked through by someone unfamiliar with the recent server migration, the outdated hostname reference would have been caught immediately — the precondition check itself, "confirm you are connected to the correct production host," would have failed the moment a tester actually tried to verify it against current reality. The exact gap that caused a real outage to take twice as long as it should have would have surfaced during a calm test, months before anyone genuinely needed the document to work.

A RUNBOOK THAT "SOUNDS RIGHT" TO AN EXPERT PROVES NOTHING

Confirming a document with its own author, or with anyone already expert in the system it describes, tests the wrong thing entirely. The whole point of testing is finding the gaps that are invisible to anyone who already knows the answer without needing the document at all.

Hands-On Exercises

EXERCISE 1

Explain why the author reviewing their own runbook is described as testing the wrong thing, using this chapter's own parallel to Chapter 5's "watch, don't just ask" material.

View solution

EXERCISE 2

Explain exactly how a genuine test would have caught Chapter 1's own outdated hostname before the real outage, and which specific structural section (from Chapter 4) is what would have surfaced the problem.

View solution

EXERCISE 3

Explain why this chapter describes a test as "a stronger, more rigorous form" of Chapter 7's own review, rather than as an entirely separate practice.

View solution

CHAPTER 8 QUICK REFERENCE

- Only someone genuinely unfamiliar with the system actually attempting the procedure proves a document works
- The author testing their own document doesn't count — their own knowledge fills gaps a real reader wouldn't be able to
- Ideal: real execution in a safe environment. Substitute: a skeptical walkthrough verifying every referenced value is still current
- Critical runbooks deserve more frequent, more rigorous testing than rarely-used documents
- Document each test's date, tester, findings, and fixes
- Chapter 1's outage is now fully resolved: a genuine test would have caught the outdated hostname via the precondition check itself, months before it mattered
- Next: Chapter 9, organizing and making documentation findable

CHAPTER 9 OF 10

Organizing & Making Documentation Findable

Documentation & Runbooks

Chapter 9 · Organizing & Making Documentation Findable

Chapters 1 through 8 covered producing documentation that's accurate, secure, and genuinely tested. This chapter covers the last practical problem: a technically excellent document nobody can actually find in the moment they need it fails its own purpose just as completely as a wrong one would — just through a different mechanism entirely.

Findability Is a Separate Problem From Quality

A perfectly accurate, well-tested runbook buried in the wrong folder, named inconsistently, or scattered across three slightly different copies fails a stressed reader just as thoroughly as an inaccurate one would. All the work from every prior chapter in this course is wasted if the reader simply can't locate the document at the moment it matters.

Consistent Naming and Location

A predictable naming and location scheme lets a reader guess where something likely lives, or search effectively, rather than needing to already know a specific document exists before they can find it. Consistency matters more than any particular convention — the value comes from a reader being able to predict the pattern, not from the pattern itself being perfect.

Tagging: Multiple Paths to the Same Document

Tag by system, by severity or criticality, and by document type (Chapter 2's own KB article/runbook/ticket distinction) — a reader searching under pressure might know the system name but not the specific problem, or the reverse, so more than one path to the same document genuinely helps.

Cross-Linking Related Documents

A runbook should link directly to its own supporting KB article — Chapter 2's own "split a growing background section into its own linked article" pattern — and related runbooks for the same system should

reference each other, so a reader who finds one has a clear path to the others they might also need.

The Duplicate-Copy Problem: A Sneaky Variant of Decay

Chapter 7 covered a document going stale where it lives. This is a related but distinct problem: once a document is updated, old copies pasted elsewhere — a chat message, a different wiki page, a printed page left on a desk — don't get updated along with it, and can be found and followed instead of the current version. The original document can be perfectly current while an outdated copy of it, still findable somewhere else, actively misleads whoever happens to find that copy first.

Deprecate, Don't Silently Delete or Silently Leave in Place

When a document is genuinely retired — the procedure no longer applies, the system it describes was decommissioned — mark it clearly as deprecated or archived. Silently deleting it loses potentially useful historical context; silently leaving it in place looking current means someone can find and follow it by accident, the same duplicate-copy problem above, but applied to the original document itself once it's no longer accurate.

WORKED EXAMPLE: TWO RUNBOOKS, ONE STRESSED READER

Two nearly identical runbooks exist for restarting the same service — one written 18 months ago, now subtly outdated, and one updated just last month. Both remain findable via search, with nothing distinguishing which is actually current. A stressed reader grabs the older one and runs into exactly the kind of gap Chapter 7 described. The fix isn't better search — it's clearly deprecating the old runbook and consolidating into one current, clearly-labeled document, rather than leaving two live copies for a reader to guess between.

"FINDABLE" ISN'T THE SAME AS "THE FIRST RESULT"

A good search and tagging system still needs a clear signal about which of several similar-looking results is actually the current, correct one. Making something findable solves half the problem; making it unambiguous which findable copy to trust solves the other half.

Hands-On Exercises

EXERCISE 1

Explain why findability is described as a genuinely separate problem from accuracy, rather than as one more dimension of a document's own quality.

View solution

EXERCISE 2

Explain how the duplicate-copy problem differs from Chapter 7's own decay problem, even though both result in a reader following outdated information.

View solution

EXERCISE 3

Using the two-runbooks worked example, explain why the actual fix was deprecating and consolidating rather than simply improving search so the current version ranks higher.

View solution

CHAPTER 9 QUICK REFERENCE

- Findability is separate from accuracy — an unfindable perfect document fails just as completely as a wrong one
- Consistent naming/location, tagging by multiple axes (system, severity, document type), and cross-linking all help
- Duplicate copies elsewhere don't update when the original does — a sneaky variant of Chapter 7's own decay problem
- Deprecate retired documents clearly — don't silently delete (loses context) or silently leave them looking current (misleads a reader)
- Findable and unambiguous are two different problems — solve both, not just one
- Next: Chapter 10, the capstone — three documentation tasks, start to finish

CHAPTER 10 OF 10

Capstone: Three Documentation Tasks, Start to Finish

Documentation & Runbooks

Chapter 10 · Capstone — Three Documentation Tasks, Start to Finish

Nine chapters built the toolkit — document types, writing for a stressed reader, structure, tribal-knowledge capture, security-conscious writing, decay, testing, and findability. This capstone runs three fresh tasks through that toolkit end to end, matching the three-scenario shape most of this subject's own courses use.

Task 1: Promoting a Recurring Ticket Into a Real Runbook

A specific API integration has failed intermittently three times over two months, each time handled as its own separate ticket.

RECOGNIZING THE PATTERN (CHAPTER 2)

Three tickets for the same underlying issue is exactly the recurrence trigger Chapter 2 named — this belongs promoted into a genuine runbook, not another one-off ticket repeating the same investigation.

WRITING IT FOR A STRESSED READER (CHAPTER 3)

Short imperative steps, one action each, an explicit expected result after every step — the background explanation of why the integration is flaky gets split into its own linked KB article rather than interrupting the procedure itself.

STRUCTURING IT PROPERLY (CHAPTER 4)

Preconditions confirm the correct environment and required access; steps walk through the fix; verification confirms the integration is genuinely working again, not just that the last command didn't error; rollback covers what to do if the fix doesn't resolve it.

AVOIDING A SECURITY LIABILITY (CHAPTER 6)

The runbook references the secrets vault field holding the integration's API key rather than pasting the actual key inline.

Task 2: Capturing a Departing Senior Technician's Knowledge

A senior technician who's the only person who really understands a quirky legacy subsystem is leaving in a few weeks.

WATCHING, NOT JUST ASKING (CHAPTER 5)

Rather than a single rushed interview, the technician is actually watched performing the task — surfacing a config check and a wait period they never thought to mention when simply asked to describe their process.

DRAWING OUT THE HARDEST PARTS (CHAPTER 5)

Specifically asking "what would you do if this failed a specific way" and "how would you know it actually worked" draws out the troubleshooting and verification knowledge that's the most tribal of all.

TURNING NOTES INTO A REAL DOCUMENT (CHAPTER 2)

The raw notes become an actual runbook, not left as unstructured interview transcripts nobody else can easily use.

MAKING IT FINDABLE AFTERWARD (CHAPTER 9)

Tagged by system and document type, and cross-linked to the related KB article covering why the subsystem behaves the way it does — so the next person facing this quirk can actually locate it.

Task 3: Auditing Runbooks Before a Disaster-Recovery Review

A scheduled review of the organization's critical runbooks, ahead of an upcoming disaster-recovery exercise.

FINDING DECAY (CHAPTER 7)

One runbook references a hostname that was quietly retired during a migration months ago — exactly the silent decay pattern Chapter 7 described, caught here by a deliberate audit rather than a real emergency.

A GENUINE TEST, NOT A SELF-REVIEW (CHAPTER 8)

Someone new to the team, unfamiliar with the specific procedures, actually walks through each critical runbook — precisely the kind of test that catches what an author or an existing expert never would.

A STRAY DUPLICATE, FOUND AND DEPRECATED (CHAPTER 9)

An outdated copy of one runbook turns up pasted into an old chat channel from over a year ago — clearly marked as deprecated so nobody stumbles onto it and follows it by accident.

CLOSING IT OUT

Every reviewed runbook gets a named owner and an updated "last reviewed" date — genuinely earned this time, not a rubber-stamped bump.

Chapter Attribution

TECHNIQUE USED ABOVE	SOURCE CHAPTER
"Documentation exists" ≠ "documentation works" as the motivating premise (Task 3)	Chapter 1
Recognizing a recurring ticket pattern; turning raw notes into a real document (Tasks 1 and 2)	Chapter 2
Writing short, imperative, verifiable steps for a stressed reader (Task 1)	Chapter 3
Preconditions, steps, verification, and rollback as a complete structure (Task 1)	Chapter 4
Watching rather than only asking; drawing out failure-path and verification knowledge (Task 2)	Chapter 5
Referencing a secrets vault instead of embedding a real API key (Task 1)	Chapter 6
Recognizing an outdated hostname reference during a proactive audit (Task 3)	Chapter 7
A genuinely unfamiliar tester walking through critical runbooks (Task 3)	Chapter 8
Tagging and cross-linking for findability (Task 2); finding and deprecating a stray duplicate copy (Task 3)	Chapter 9

Honest Scope Note

WHAT THIS COURSE DELIBERATELY DOESN'T COVER

- No specific documentation-platform tutorials (Confluence, Notion, and similar) — the underlying discipline transfers, specific tools change too often to document here
- No formal technical-writing style-guide depth (grammar standards, tone guidelines beyond clarity itself) — a genuinely separate, deeper discipline
- No legal or regulatory documentation-retention requirements — organization- and industry-specific, out of scope here
- No video or screen-recording documentation formats — this course is specifically about written documents
- No substitute for an organization's own actual documentation tooling and platform choice

Hands-On Exercises

EXERCISE 1

Explain why Task 1's runbook needed both Chapter 3's writing principles and Chapter 4's structural sections — what would have been missing if only one of the two had been applied.

View solution

EXERCISE 2

Explain why Task 2 needed Chapter 9's own findability material in addition to Chapter 5's knowledge-capture material, even though the knowledge itself was captured correctly either way.

View solution

EXERCISE 3

Explain why Task 3 used a genuinely unfamiliar tester rather than having each runbook's own original author re-review it, given that the goal was simply confirming the documents were still accurate.

View solution

CHAPTER 10 QUICK REFERENCE — COURSE COMPLETE

- Task 1: a recurring ticket pattern promoted into a properly written, properly structured, security-conscious runbook

- Task 2: a departing technician's tribal knowledge captured by watching, not just asking, and made findable afterward
- Task 3: a proactive audit catching silent decay and a stray duplicate copy before a real disaster-recovery review needed them to work
- The recurring theme across all ten chapters: **"we wrote it down" and "someone can follow it" are two different claims, and only genuine testing by an unfamiliar reader proves the second one**
- This closes *Documentation & Runbooks*, 10/10 chapters — the ninth complete course under the Technical Support subject