

RUNBOOKS · FUNDAMENTALS · DOWNLOAD

Your First Runbook: Template Kit

A template, a guide to every field with patterns for other kinds of action, a checklist, a test record sheet, and a complete worked runbook. From the article “Your First Runbook: The Minimum That Makes It Safe” on revisualized.com.

Joseph Tracy

https://revisualized.com/articles/your-first-runbook

Every service, server, and value in the examples is constructed, apart from systemd's default timeouts.

© 2026 Joseph Tracy. Published on revisualized.com.

Released 2026-11-28. Version 1.

Licensed under CC BY-NC 4.0, with workplace use permitted (see License).

Contents

How to use it	3
Words used in this kit	5
1. The template	8
2. Field guide	11
3. Author's checklist	21
4. Test record sheet	24
5. Worked example: RB-MEDIA-004	25
6. Why each line of RB-MEDIA-004 exists	30
7. Defects this runbook survived	35
8. The template, to write on	44
Sources	50
License	51

> Commands in this PDF are for reading. Copy the template and the worked runbook from the page on revisualized.com, where each line pastes as written; the three patterns at the end of section 2 appear only in this PDF, as models for writing and testing your own commands. This PDF wraps long lines to fit the page.

From the article "Your First Runbook: The Minimum That Makes It Safe" on revisualized.com, published 2026-11-28.

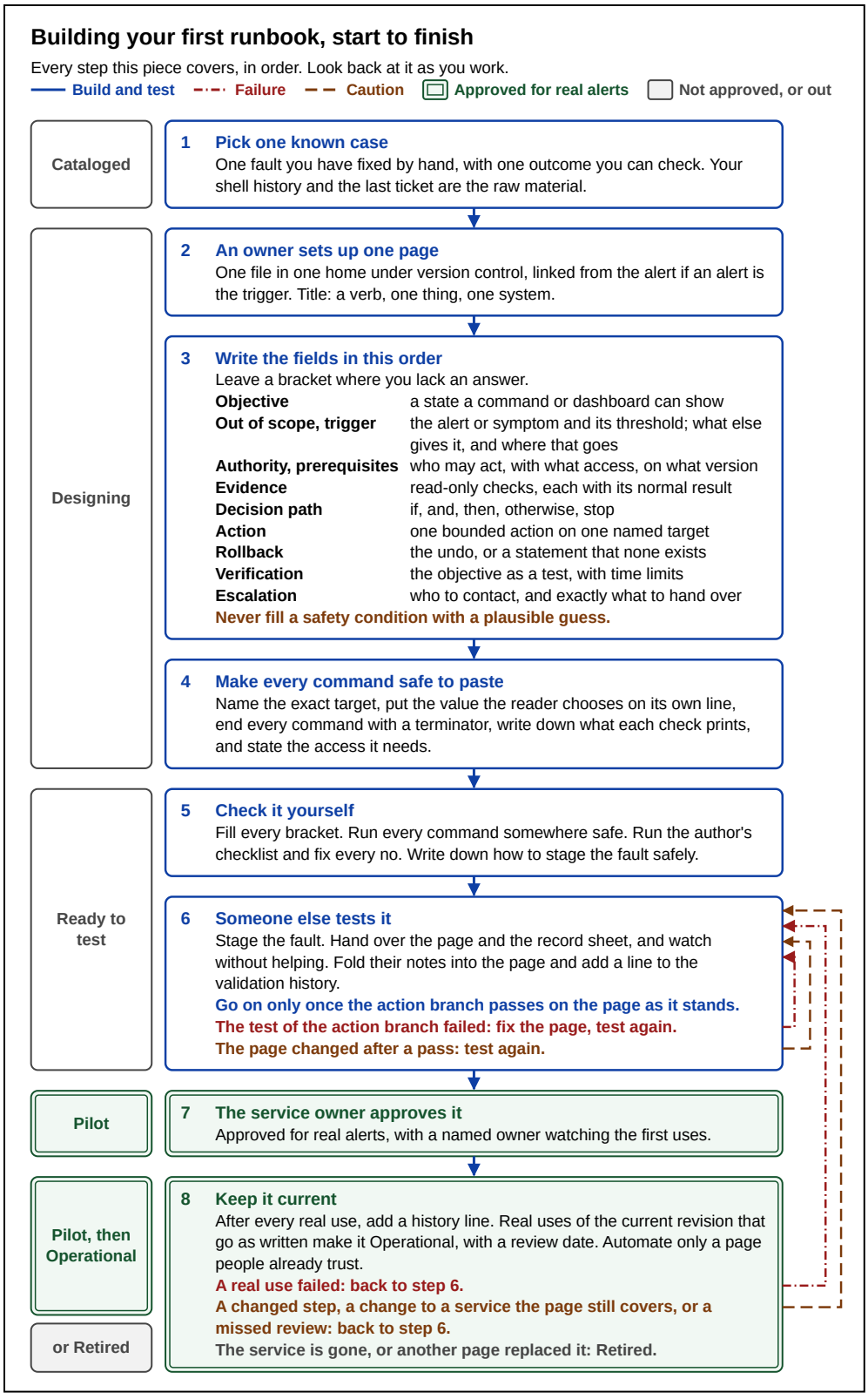
Use this manual to write a first runbook, check it, and record what happens when someone else follows it. It is for engineers who know the service they run and want to turn a recovery they have done before into a page someone else can follow. Every service, server, and value in the examples is constructed, apart from systemd's default timeouts.

To do this	Start with	Keep beside you
Write a first runbook	Section 2, the field guide	Section 8, the template to write on
Check a draft before anyone tests it	Section 3, the author's checklist	Section 7, the defects this runbook survived
Stage a fault and test a page	Section 3, staging a test	Section 4, the test record sheet
Write an action other than a restart	Section 2, the three patterns at its end	Section 3, the author's checklist
See a complete runbook and why it is built that way	Section 5, the worked example	Section 6, why each line exists
Tell what a page's status means	The lifecycle table under "Words used in this kit"	The page's validation history

How to use it

1. Copy the template into wherever your team keeps runbooks, as one file per runbook. Section 1 shows it; copy it from the article at https://revisualized.com/articles/your-first-runbook, where each line pastes as written.
2. Fill in the fields in this order: objective; out of scope and trigger; authority and prerequisites; evidence checks; decision path; action; rollback; verification; escalation. The other header lines can wait until the end.
3. Leave the brackets in any field you cannot fill yet, and note beside each who can supply the answer. A safety condition, something that must already be true for the action to be safe (a restore someone has tested, say), is never filled with a plausible guess. While you are filling the page in, its status is Designing.
4. Run the checklist in section 3. Fix every "no".
5. Hand the page and the record sheet in section 4 to someone who has never run it. Stage the fault somewhere safe, and watch without helping.
6. Fold their notes back into the page, add a line to the validation history, and set the status.

The worked runbook in section 5 shows the method on one constructed service that runs all the time. A scheduled job, a single unit, or a different fault needs its own checks, states, and time limits: copy the method, and work out every count, threshold, time limit, and cutoff from your own service.



Building a first runbook, start to finish: every step in order, with the page status each step leaves.

Copy commands and the template from the article on revisualized.com, at https://revisualized.com/articles/your-first-runbook. The three patterns at the end of section 2 appear only here, so the web page has no copy of their commands: use them as models to write and test commands for your own service. This PDF is for reading, writing on, and keeping beside a test, and it wraps long lines to fit the page, and a line break pasted into a shell splits the command in two.

Words used in this kit

Runbook. A documented procedure that reaches one known outcome for one known case. Its checks confirm that case is present, and any other result goes elsewhere. This kit's template is a runbook template.

Playbook. The document you open while the cause is still unknown. It gathers evidence, narrows the possibilities, and ends by pointing to a runbook or by escalating. A runbook stays a runbook when a playbook is written in front of it; it becomes one of the playbook's exits. During an incident you move from the playbook to a runbook, or to escalation when the evidence matches no known cause; the documents usually get written the other way round.

Standard operating procedure. Planned, recurring work with no alert behind it, such as a weekly audit. It often calls runbooks for its steps.

Automation script. Code that performs some of a runbook's steps, written after the runbook has been tested by people, with comments that trace each block back to a step.

Automation workflow. An event starts it, and it gathers evidence, checks a policy, takes an action, and verifies the result without a person driving each step.

Post-incident review. The look back after an event that decides what to change, often including the runbooks and playbooks that were used.

Ansible playbook. A YAML file of automation tasks that Ansible runs against hosts, usually changing them. In this kit's terms it is automation. Say "Ansible playbook" for it and keep "playbook" for the diagnostic document. A runbook step that runs one names the exact file and the one host, like any other action.

Some teams also call a communication handbook a playbook. Check which kind you have been handed before acting on it.

Finding and error. A finding is a result a check lists as abnormal, such as a count of zero; it decides the branch. An error is a line the check says it always prints that never printed, or any other line, result, or exit status its text leaves out; it sends the reader to escalation.

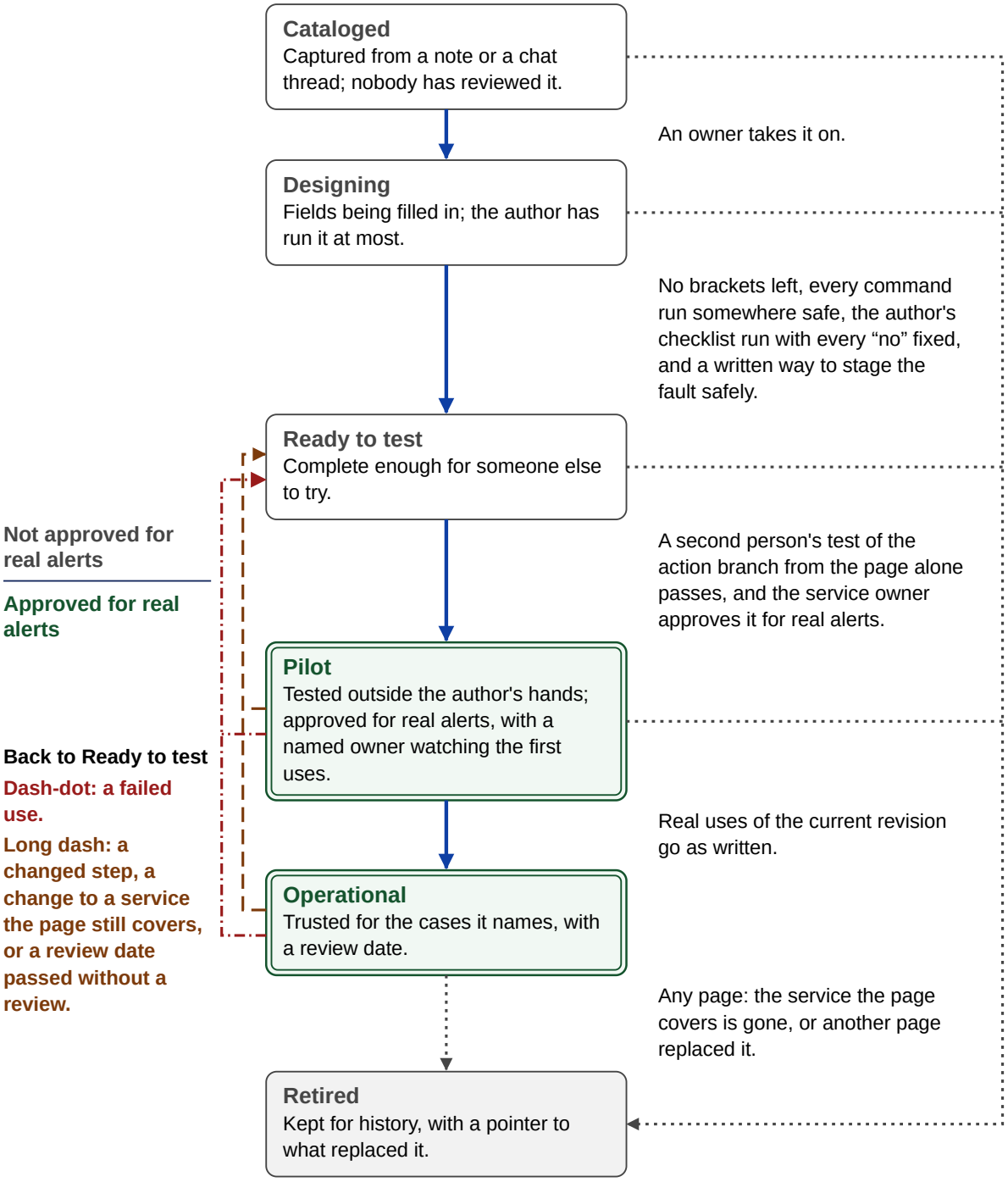
Lifecycle states. A page's status says how far it has come. These states come from a personal runbook standard:

State	What it means	What changes its state
Cataloged	Captured from a note or a chat thread; nobody has reviewed it	An owner takes it on
Designing	Fields are being filled in; the author has run it at most	No brackets are left, the author has run every command somewhere safe, the author's checklist has been run with every "no" fixed, and there is a written way to stage the fault safely
Ready to test	Complete enough for someone else to try	A second person's test of the action branch from the page alone passes, and the service owner approves it for real alerts
Pilot	Tested outside the author's hands; approved for real alerts, with a named owner watching the first uses	Real uses of the current revision go as written; a change to a step or to a service the page still covers, a failed use, or a review date passed without a review sends it back to Ready to test
Operational	Trusted for the cases it names, with a review date	A change to a step or to a service the page still covers, a failed use, or a review date passed without a review sends it back to Ready to test
Retired	The service the page covers is gone, or another page replaced it; kept for history, with a pointer to what replaced it	Nothing; it stays as a record

A passing test of a no-change branch shows the escalation works and leaves the action untested; the action branch needs its own passing test before Pilot.

A runbook's status, from Cataloged to Retired

States and moves from the lifecycle table. Solid arrows move forward, dashed arrows go back to Ready to test, and dotted arrows go out to Retired. A double border marks approval for real alerts.



A runbook's status from Cataloged to Retired. Only Pilot and Operational pages are approved for real alerts; a failed use, a changed step, a change to a service the page still covers, or a missed review sends either one back to Ready to test.

1. The template

```
[RB-identifier]  [Verb] [one thing] on [one system]

Owner:         [role that keeps this page current]; service owner: [role]
Status:        [Cataloged | Designing | Ready to test | Pilot | Operational | Retired],
               page revision [n]; last tested by someone other than the
               author: [not yet | who, where, YYYY-MM-DD UTC]
Next review:   [YYYY-MM-DD], or after any change to [service]
Lives at:      [repository and path of the authoritative copy]
Applies to:    [servers, service, instances, versions]
Out of scope:  [cases that look similar and need something else, with
               where to go instead]

1. Objective
[The end state, written as something a command or a dashboard can show.]

2. Trigger
[The alert or symptom that sends a reader here, with its threshold.]

3. Authority and prerequisites
- Who may run this without asking, and what needs approval: [ ]
- Access needed, check by check (sudo rule or group): [ ]
- The shell, and the settings its checks need (errexit off, say): [ ]
- What the action does to work in progress, and how to confirm it: [ ]
- Where to record commands and output: [ ]

4. Evidence (read-only; run all of it before deciding, unless a stop applies)
A check ran without an error when it printed each line its text says
it always prints, every other line it printed is one its text
describes, and its result as a whole, exit statuses included, is one
listed there. An abnormal result is a finding. Anything else, a
missing line included: Branch C.
[The checks below show the known cause this page fixes is present.]
E1  [What it checks]
    [command; after a pipeline, print each exit status by name;]
    Prints: [the lines it always prints; each other kind it can print]
    Normal: [ ]  Abnormal: [ ]
E2  [What it checks]
    [command; after a pipeline, print each exit status by name;]
    Prints: [the lines it always prints; each other kind it can print]
    Normal: [ ]  Abnormal: [ ]

5. Decision path
Branch A: [name of the bounded case]
  If every check ran without an error,
    and [observable condition],
    and [scope or risk condition],
    then go to section 6.
Branch B: [shared or larger problem]
  Otherwise, if every check ran without an error,
    and at least one of these holds:
      [condition];
      [condition];
    then escalate with [evidence]. Make no change.
Branch C: anything else
  Otherwise, escalate with the evidence collected. Make no change.
Stop at any point if [harm condition]; escalate with what you have.

6. Action ([branch] only)
[Checks to run again right before the action, because their results
```

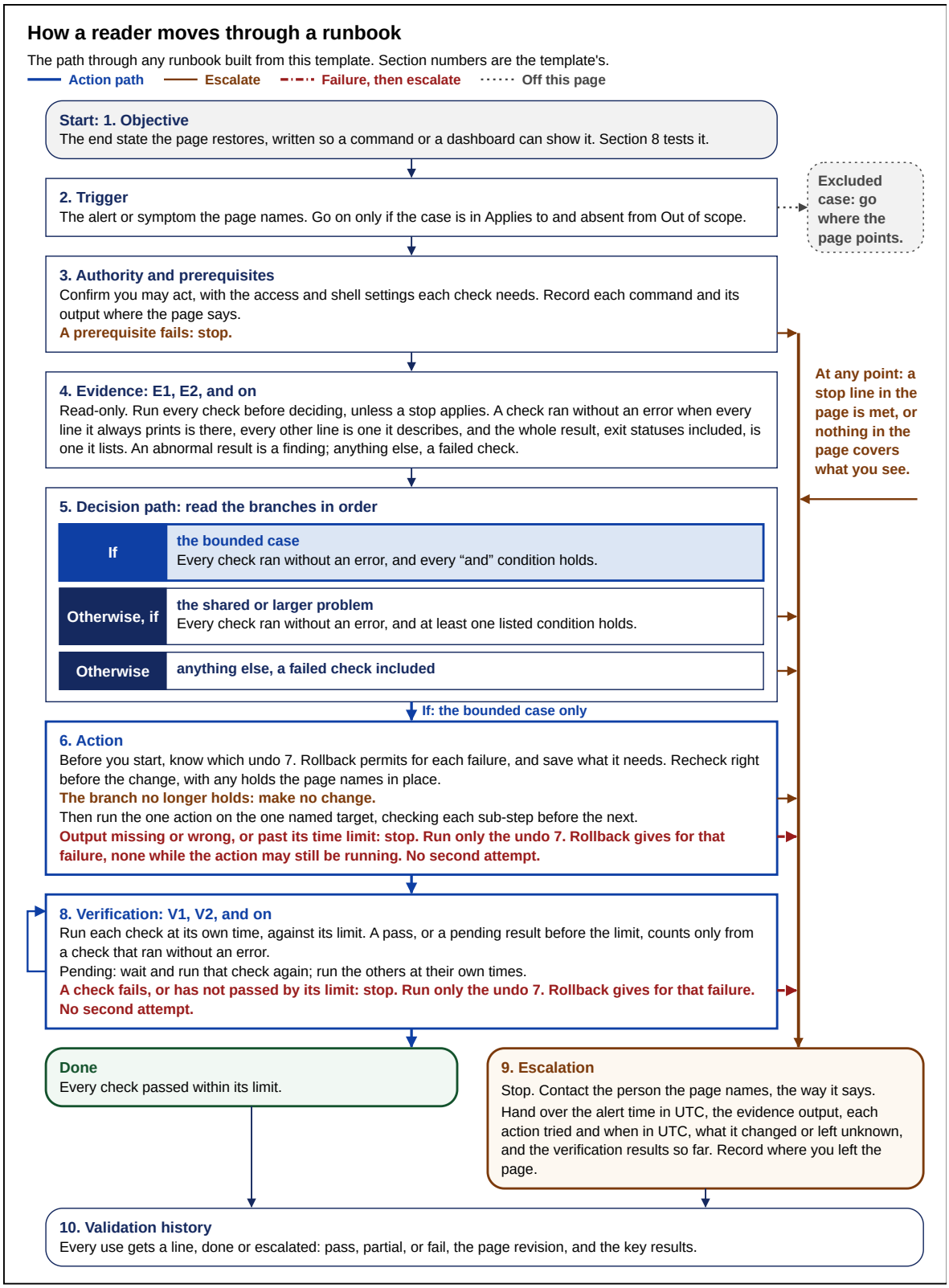
```
 can change; how the page keeps other writers or runs off meanwhile;
 and what to do if the branch no longer holds.]
Set [value the reader chooses] first.
    [variable=value;]
    [the action: one command, or a set sequence of numbered sub-steps
     with a check after each;]
Expected: [what must print, and within how long]. If a line it must
print is missing, anything else prints, or it runs past that: stop.
Run only the undo section 7 gives for that failure, and none while the
action may still be running. Make no second attempt; escalate.

7. Rollback
[How to undo each step that changed something: what to save before
 it, which failures permit the undo, who may do it, and the check
 that shows the undo worked.]
[Or: This cannot be undone. Section 5 limits it by: (conditions), and
 a check in section 3 or 4 proves that (what must already be true,
 such as a backup restored in a test, or the old primary fenced so it
 cannot accept writes) holds. A file listing falls short.]

8. Verification
V1  [Check of one part of section 1, with a time limit]
    [command; after a pipeline, print each exit status by name;]
    Prints: [the lines it always prints; each other kind it can print]
    Pass: [value]. Still pending before the limit: [value].
[V2 and on, until every part of section 1 has a check.]
A pass or a pending result counts only for a check that ran without
an error, by section 4's rule. If verification fails: stop, run only
the undo section 7 gives for that failure, make no second attempt, and
escalate.

9. Escalation
[Role, and how to reach them.] Hand over: [alert time in UTC, evidence
output, each action tried and when in UTC, what it changed or left
unknown, verification results].

10. Validation history
[YYYY-MM-DD]  [environment]  [who ran it]  [pass, partial, or fail]
            Page revision [n], [service version]; case [staged fault,
            or what the real alert showed]; [key results, times in
            UTC]; [page change, or "No page change"]
[YYYY-MM-DD]  [environment]  [approver]  approved  Revision [n] for
            [what it is approved for]; [who watches the first uses]
```



How a reader moves through a runbook built from this template. Only the bounded case reaches the action. An excluded case goes where the page points; any other stop or failure leads to escalation. Done and escalation both end in the validation history.

2. Field guide

Each entry says what goes in the field, shows a strong and a weak version, and gives the question a reviewer will ask about it. The strong examples come from the worked runbook in section 5. Three patterns at the end of this section cover actions the worked runbook never needed.

Title and identifier

What goes here: a stable identifier, then a verb, one thing, and one system.

- Strong: `RB-MEDIA-004 Restart one stalled thumbnail-worker instance on media-01`
- Weak: `Thumbnail worker stuck`

Reviewer question: from the title and trigger together, can a reader tell whether this page is for their alert? A symptom title attracts every case that looks like the symptom.

Owner and status

What goes here: the role that keeps the page current, the service owner, the lifecycle state, the page revision, and who other than the author last tested it, where, and when, in UTC. Until someone has, that part says "not yet".

- Strong: `Pilot, page revision 2 (retested in staging by a second operator, 2026-11-05 UTC; approved for real alerts by the media platform lead, 2026-11-06 UTC, who watches the first uses)`
- Weak: `Status: done`

Reviewer question: does the status say who other than the author last tested it, on which revision of the page, and, for Pilot, who approved real use and when? The approval goes in the validation history too. A validation date covers the page as it stood that day. After a step changes, or a service the page still covers changes, earlier tests no longer cover the page, and a Pilot or Operational page goes back to Ready to test; when the service is gone, the page is Retired.

Next review

What goes here: the date of the next scheduled review, and the changes that bring it forward.

- Strong: `2027-02-01, or after any change to thumbnail-worker`
- Weak: `Review occasionally.`

Reviewer question: does the page tell its owner when it needs review, including a change that comes before the date? A change to a service the page still covers sends a Pilot or Operational page back to Ready to test.

Lives at

What goes here: the repository and path of the authoritative copy.

- Strong: `operations-runbooks repository, media/RB-MEDIA-004.md (the authoritative copy); alert THUMBNAIL-QUEUE-AGE links here by identifier`
- Weak: nothing, with copies in three chat threads

Reviewer question: if this page is wrong, where does the fix go, and who sees the change?

Applies to and out of scope

What goes here: the servers, service, instances, and versions the page covers; then every known case that produces the same alert but needs a different response, each with where to go instead.

- Strong: `Out of scope: an instance in the failed state (use RB-MEDIA-002); slow instances, two or more quiet ones, disk use at 85 percent or more, one quiet instance holding no file, or a backlog after a large upload with all three busy (escalate, by Branch B or C as section 5 decides)`
- Weak: no out-of-scope line

A runbook checks enough to establish its own case and that acting is safe; the other causes of its alert belong to other pages. A result that calls for a different investigation or remedy sends the reader elsewhere.

Reviewer question: what else produces this alert, does the page say where those cases go, and does evidence that fits no action branch escalate?

1. Objective

What goes here: the end state, written so a command or a dashboard can show it. A service can be running and answering while a fault waits on disk, such as a settings file it will fail to load at its next start. When a fault like that can follow the action, the objective names the stored state as well as the running one, and verification checks both.

- Strong: `No file in /srv/thumbnails/incoming is older than 15 minutes, and all three instances are logging completed jobs.`
- Weak: `Fix the worker.`

Reviewer question: could two people looking at the same screen disagree about whether the objective holds? At the exact threshold, do the objective, the alert, and the verification command agree?

2. Trigger

What goes here: the alert or symptom that sends a reader to this page, with its threshold.

- Strong: `Alert THUMBNAIL-QUEUE-AGE: oldest file in /srv/thumbnails/incoming is older than 15 minutes.`
- Weak: `When thumbnails are slow.`

Reviewer question: does the trigger fire for every case this page covers? Write down any case it misses as a gap in the alert.

3. Authority and prerequisites

What goes here: who may act without asking and what needs approval, and, for an action allowed only so often, what counts as one use, an attempt that stopped partway included; the access each check needs; what the action does to work in progress and how to confirm it; where to record commands and output (long output as a short extract, with where the full copy is saved).

- Strong: `sudo, which E2 (and V3, its rerun), section 6, and V1 use as written. Without a sudo rule, sudo prints an error (the tested one names the sudoers file), and in E2 and V1 the count shows 0. ... E2 and V1 describe neither line, so the check failed.`
- Weak: `Needs root.`

Reviewer question: if the reader lacks one piece of access, which check gives a wrong answer instead of an error? Does the page say where to find each record it asks the reader to check, such as the ticket and the on-call handoff?

4. Evidence

What goes here: read-only checks, numbered E1, E2, and so on, each with its command, each kind of line it can print, and which of those are normal. Above them, the rule for "ran without an error": every line the check says it always prints printed, every other line printed is one the check describes, and the result as a whole, exit statuses included, is one the check lists. Lines that are alternatives, such as one line per held file, are described without being required. A mix of lines and statuses missing from the check's list is an error even when each piece appears somewhere in its text. An abnormal result is a finding; anything else is an error.

A command can exit 0 and still have failed at its work, such as a job that finds a lock from an earlier run and exits. Name the evidence that shows the work happened (a log line, an output file), and keep it apart from the exit status. After any command whose output alone cannot tell a finding from a failure, pipelines included, print its exit status by name, on the next command, before anything else resets it. The status line is part of the check: if it never printed, the check failed, whatever the line before it showed. Name the shell settings the checks rely on as well: with bash's errexit option on, a command written as a plain step, as every check here is, that exits nonzero on purpose, such as `grep -c` counting nothing, ends the shell before its status line prints. Inside an `if` test, the same command leaves the shell running; the sandbox

showed both. When two checks must describe the same thing, such as the file a log line names and the file in a folder, say how the reader matches them. Until someone has run a check, the output written beside it is a prediction: the author runs each command somewhere safe before the status leaves Designing.

Three things a check cannot show by itself. A check that relies on something being absent proves absence only where it can see: an empty result from the wrong folder, an unreadable one, or one whose parent folder is gone looks the same as nothing there. A check shows the moment or the time window it measured: when another person or process can change the answer before the action, the page needs a recheck right before it and, when the change could do harm, a way to keep the other writer off (6. Action). And some checks show the action is safe, while others only show it is likely to work: decide which conditions must hold before the action, and make each one either a section 3 prerequisite that stops the page when it fails or a condition of the branch that permits the action; the failures the page can catch afterwards belong in verification.

- Strong: `E3 Files held in a working folder longer than 10 minutes ... -cmin +10 ...; echo "find exit $?"; Normal: only the line "find exit 0".`
- Strong: after `journalctl ... | grep -c '^completed job '`, the line `printf 'instance %s: journalctl exit %s, grep exit %s\n' "${worker}" "${PIPESTATUS[0]}" "${PIPESTATUS[1]}";` with `Abnormal: a count of 0, with journalctl exit 0 and grep exit 1.`
- Weak: a command with no normal result beside it
- Weak: `Any error a check prints means it failed.` A line on standard error is easy to miss, and with bash's pipefail option off, its default, a pipeline's status is its last command's, so a journalctl failure in front of `grep -c` prints a count of 0 and no status at all.

Reviewer question: can every check run without changing anything, and does every one say what normal looks like? Does it describe each kind of line it can print, so anything else is an error? After a pipeline, does a line name each command's exit status? Does each check measure the thing its title names (time held in a folder, say, as opposed to time since upload)? Do the checks together show the known cause this page fixes, and every condition the action needs to be safe?

5. Decision path

What goes here: branches in the if, and, then, otherwise, stop form. Each condition names the output of a numbered check or a named prerequisite. One branch covers the bounded case, one covers the shared or larger problem, and one covers anything else.

- Strong: `If every check ran without an error, and E2 shows zero for exactly one instance, and E2 shows 20 or more for each of the other two, ...` and then `Otherwise, if ...` for each later branch, with any list of alternatives on its own lines under `and at least one of these holds:`
- Weak: `If the worker seems stuck, restart it.`

Reviewer question: is there a combination of check results that reaches no branch, or two? Writing each later branch as "Otherwise, if" makes two impossible. Where a branch mixes "and" with "or", could a reader group the conditions two ways?

6. Action

What goes here: one state-changing step for one branch, with the value the reader chooses on its own line and the values it may take, the exact target, a terminator on every command, and what must print. A missing line stops the reader as surely as a wrong one. Keep the name that holds the target for the target alone: a loop that checks every instance needs its own name, or it overwrites the one verification reads (entry 14).

When something can change the answer between the check and the change, run the check again right before the first change, and say what to do when the branch no longer holds. A recheck gives a newer reading, and the answer can still change after it. Two kinds of change need different answers. Work that someone or something starts (another operator, a deploy tool, a scheduler, a person starting a job by hand) can be held off: when it could do harm, name every source, keep each one off with a method the service owner approves (the ticket assigned to you under a team rule, deploys paused, a timer stopped), and say how the reader confirms each hold took effect and how they release it. Release only a hold the page took; when a hold's state is unknown, say so when you escalate. A source that cannot be kept off is a reason to stop and escalate, however clean the checks look. A fault that starts on its own, such as a peer that stalls, cannot be held off: the recheck, the stop line, and verification are what can catch it, and the page says what each one can miss (entry 10). Patterns 1 and 2 at the end of this section show holds.

An action that can run for a long time, such as a job that `systemctl start` waits on, needs its own cutoff: name it, say how the reader watches the time while the command holds their terminal, say the work may still be running past it, and say what to report, and to whom, when it passes. Reaching the cutoff gives no authority to stop the work. In a sequence, check the time again before any sub-step whose remaining work could run past an outside deadline; if it no longer fits, stop and report what the earlier sub-steps already changed.

- Strong: `instance=2;` then `sudo systemctl restart "thumbnail-worker@${instance}.service" && restart_time="$(date -u '+%Y-%m-%d %H:%M:%S.%6N UTC')" && echo "instance=${instance}; restart_time='${restart_time}';";`
- Weak: `sudo systemctl restart thumbnail-worker@*;`

Reviewer question: could this command touch anything other than the one target the branch names, and what happens if the reader types a value outside the ones the page allows? If the action is a set sequence (restore a file, reload, start), does each sub-step have a check, does the page stop at the first one that fails, and does any step that cannot be undone come as late as the operation allows? Does the page say what state each sub-step leaves, and what to report if the next one fails? Does any later step, rollback included, need a file or value that an earlier step moved, replaced, or removed?

7. Rollback

What goes here: how to undo the action and check that the undo worked, or a plain statement that it cannot be undone, followed by the conditions in section 5 that limit it. For a sequence, say for each sub-step that changed something what can be undone, who may do it, and what the reader reports when they may not. Removing a leftover lock cannot be undone, and stopping a job the page then started may belong to another team: the page names that team and what to hand them. For an action that cannot be undone, name what must be true for it to be safe (the process named in a lock file is gone, say, as in pattern 1) and the read-only check that proves it. When that answer can change before the action, the check runs again right before it. An undo that puts an old file back puts back whatever was wrong with it too, so say which failures permit the rollback, save what it needs before the action changes anything, and check the saved copy before relying on it (pattern 3). An undo can fail too: say what the reader does then, and let no reload or restart follow an undo that did not check out. The checks that show an undo worked look for the old state, so they belong in this section; section 8's checks of the new state can fail after a rollback that worked. A rollback that works can still leave the alert's cause in place; the page then escalates.

- Strong: `A restart cannot be undone. Section 5 limits it: one instance, once, and only one that has completed nothing in 10 minutes.`
- Weak: an empty section

Reviewer question: if the action makes things worse, what does the reader do next? For an action that cannot be undone, what must already be true for it to be safe (a backup restored in a test before a delete, the old primary fenced before a failover), and which evidence check proves it? A backup file in a listing and a primary that stops answering health checks both fall short of proof.

8. Verification

What goes here: checks of the outcome from section 1, each with a time limit, and what to do when one fails. Each check says what a pass is, what a result still pending before the limit looks like, and which results are errors; a passing or pending result counts only for a check that ran without an error. Say whether a limit applies to the work finishing, the reader confirming it, or both. A late look at current state, such as a file that is there now, leaves open when the work finished. A recorded event tied to its own run, such as the log line written when the file is renamed into place, can show when it finished, even when the reader looks late. Checks with different limits run at their own times; a later check runs on time even while an earlier one is pending. Together they show each result within its window, which falls short of showing that every result still holds afterwards.

Set each limit from the longest normal duration of the work, plus a margin. When an outside deadline applies, such as a scheduled reboot or another team's import, work back from it to the latest time the action may start: the deadline, minus the longest normal run, minus the time verification needs, minus the time to escalate. With a 07:00 UTC import, a 50-minute longest run, 5 minutes of checks, and 10 minutes to escalate, the latest start is 05:55 UTC. That is the latest start of the long step: sub-steps before it, such as removing a lock, must be done by then, so allow time for them too. The longest normal run is a planning number, and the work can take longer. Write the allowances down, check the time again right before the

action, and if the latest start has passed, escalate without starting. Say when the team that needs the result hears that it may be late, and say plainly when the deadline has been missed: those are different messages.

A check that counts events needs a starting time chosen for what it measures. Events that show work after the action count from a time recorded once the action succeeds. Events the action itself logs count from a second time recorded just before it starts, while the time recorded after it succeeds still shows that it ran; names such as `action_start` and `action_returned` keep the two apart. When other runs can log the same message, a time window alone cannot say which run logged it: the page needs another way to tie the event to its own run, or a way to keep other runs off. A check of current state, such as the files waiting in a folder, runs after the action and looks at everything there.

A command that returned, a service that looks healthy, and work that finished are different results. When an output file can exist after a failed or conflicting run, say why the reader can trust that file before reporting success.

- Strong: `V1 Within 5 minutes of the restart, the restarted instance logs a completed job` (counted after the restart time section 6 printed once the restart returned), `V2 ... the alert clears`, and `V3 Fifteen minutes after the restart, rerun E2 ... all three show 20 or more`
- Weak: `Check that the restart command succeeded.`

Reviewer question: do the checks together cover every part of section 1, or could they all pass while part of the objective still fails?

9. Escalation

What goes here: who to contact and how, and exactly what to hand them.

- Strong: `Hand over: the alert time in UTC, E1 to E4 output, any restart run with the time written before it (UTC) and what it printed, and the V1 to V3 results so far.`
- Weak: `Ask the media team.`

Reviewer question: could the person receiving the escalation start work without asking the reader a single question?

10. Validation history

What goes here: one line per test or real use, and one per approval. A test or real-use line gives the date, environment, who ran it, and the result, then, in this order: the page revision and service version; the case (the fault staged, or what the real alert showed); the key results, with times in UTC; and the page change, or "No page change". An approval line gives the date, who approved, the page revision, what it is approved for, and who watches the first uses. A real use records what happened during an actual event; the independent test of the current page that the lifecycle asks for before Pilot is still needed.

- Strong: `2026-11-05 staging second operator pass Page revision 2, worker 2.3.1; same fault. Restart 14:02 UTC; V1 3 at 14:05; V3 24 22 23 at 14:17; V2 0 and alert clear at 14:21. No page change.` after a first run that led to a change, then the approval line
- Weak: `Tested.`

Reviewer question: when was this page last tested by someone other than its author, on which revision of the page, and against which version of the service? A change after the last run means that run no longer covers the page, and a Pilot or Operational page goes back to Ready to test.

Three patterns for other kinds of action

The worked runbook restarts one process. These three patterns cover actions it never needed. Their services, paths, lock files, deploy tool, and file checker are constructed; the shell behavior each one relies on was tested in the drafting sandbox (bash 5.2, procps-ng 4.0.4, GNU coreutils 9.4, GNU diffutils 3.10). Copy the shape, and write and test your own commands.

Like the worked runbook, the patterns assume bash with errexit off. Several commands here are expected to exit nonzero, and the status line after each must still print. Run the errexit check from the worked runbook's section 3 before trying them, apply its rules for sudo's prompt and errors, and treat a status line that never printed as a failed check.

1. Prove the process named in a lock file is gone. A constructed nightly import writes its process number to `/var/lib/import/import.lock` and removes the file when it finishes. Killed partway, it leaves the file behind, and the next run refuses to start. Removing the lock while an import still runs lets two of them write at once, so the page proves the old run is gone first:

```
pid="$(cat /var/lib/import/import.lock)"; echo "cat exit $?; pid='${pid}'";
ps -p "${pid}" -o pid=,comm=; echo "ps exit $?";
```

Always prints the cat line, then the ps line last. Normal for this branch (the old import is gone): `cat exit 0` with only digits between the quotes, then only `ps exit 1`. Abnormal (leave the lock alone): a line naming that number and a program before `ps exit 0`. Some process holds the number, the import or another program that reused it, and either way the page escalates. Anything else is an error. In the sandbox, ps also exited 1 for an empty lock file and for one holding `12ab`, after printing a usage message; the pid line, and the rule that an undescribed line is an error, separate those from a finished import.

Proving the old run is gone says nothing about the next one. A timer, a person starting an import by hand, or another service can start one between this check and the removal. Name every way an import can start; before removing the lock, keep each one off with a method the service owner approves (a timer paused, a hold the scheduler offers), then run the check again. A way to start an import that cannot be kept off is a reason to stop and escalate, however clean the check looks. Say how to release each hold, and when.

2. A sequence that keeps other writers off, checks its precondition again, and stops cleanly. A constructed service, example-api, reads `/etc/example-api/example-api.conf` only at start and on reload. A constructed deploy tool, example-deploy, can also replace that file. It keeps a plain folder,

`/run/deploy`, holds `/run/deploy/example-api.lock` while it works, and has pause and resume commands the service owner approves for on-call use: a pause that exits 0 is a hold this operation owns, and resume releases it. The action installs an approved file and reloads. Each sub-step says what a failure leaves, because the right next move depends on it. The undo steps below apply once the failed command has returned; if it may still be running, or its state is unknown, stop and escalate without starting the undo:

```
6.1  Pause deploys:
       sudo example-deploy pause example-api; echo "pause exit $?";
     Normal: only the line "pause exit 0". Anything else: resume
     nothing, stop, and escalate with the full output; say whether the
     pause exited 0, so a hold may be in place, or its state is
     unknown. After a normal pause, check that no deploy is running:
       ls -d /run/deploy /run/deploy/example-api.lock; echo "ls exit $?";
     Normal: the line "/run/deploy" and the line
     "ls: cannot access '/run/deploy/example-api.lock': No such file
     or directory", in either order; then "ls exit 2". Anything else:
     resume deploys (6.6), stop, and escalate. The page has not
     changed the live file.
6.2  Save the live file, as in pattern 3. If a check fails: resume
     deploys (6.6), stop, and escalate. The page has not changed the
     live file; rely on no saved copy.
6.3  Install the approved file:
       sudo cp /srv/approved/example-api.conf /etc/example-api/example-api.conf; echo "cp exit $?";
     Normal: "cp exit 0". If cp returned anything else, the live file
     may be partly written: put the saved copy back (pattern 3), then
     stop and escalate. Skip 6.4.
6.4  Check the installed file:
       sudo example-api --check-file /etc/example-api/example-api.conf; echo "check exit $?";
     Normal: the checker's success line, then "check exit 0". If the
     checker returned anything else: put the saved copy back (pattern
     3), then stop and escalate.
6.5  Reload:
       sudo systemctl reload example-api.service; echo "reload exit $?";
     Normal: "reload exit 0", then section 8's checks. Anything else:
     stop and escalate; report that the new file is on disk, what the
     reload printed, and whether the running state is known.
6.6  Resume deploys when section 8 passes, or after a stop in 6.1's
     ls check or in 6.2:
       sudo example-deploy resume example-api; echo "resume exit $?";
     Normal: "resume exit 0". Anything else: escalate, say deploys may
     still be paused, and say how far the page got. After a stop in
     6.3 to 6.5, or a section 8 check that fails, leave deploys paused
     and say so when you escalate; whoever takes over decides.
```

Pausing deploys before the check is what gives the check its meaning: a check run again shows the lock was absent at that moment, and without the pause a deploy can start a second later. When no approved way to hold the deploy tool off exists, the page stops before 6.2 and escalates. 6.1 lists `/run/deploy` as well as the lock, because ls prints the same "No such file or directory" when the folder itself is gone, and a missing folder can mean the deploy tool itself is broken. ls stands in for `test -e` for a similar reason: in the sandbox, run as a user who could not read the lock's folder, `test -e` returned 1, the same status an absent lock gives, while ls printed "Permission denied".

The undo in 6.3 and 6.4 needs the copy 6.2 saved, which is why 6.2 checks that copy before 6.3 changes the live file. The page has not reloaded by 6.4. If the service can start or reload on its own during these steps (a crash and an automatic restart, say), find out whether it loaded the new file, or its running state

changed, before choosing what to put back, and say so when you escalate.

3. A file-level rollback with a check that the undo worked. Saving the old file is part of the action. The copy goes to a name reserved for this operation, in `/var/backups/example-api`, a plain folder in this model, and it is checked before anything relies on it:

```
Save (step 6.2 above):
  saved=/var/backups/example-api/example-api.conf.INC-1234;
  sudo ls -d /var/backups/example-api "${saved}"; echo "ls exit $?";
Normal: the line "/var/backups/example-api" and the line "ls: cannot
access '/var/backups/example-api/example-api.conf.INC-1234': No such
file or directory", in either order, then "ls exit 2". Anything else:
6.2 has failed; follow its failure line.
  sudo cp -p /etc/example-api/example-api.conf "${saved}"; echo "cp exit $?";
  sudo cmp /etc/example-api/example-api.conf "${saved}"; echo "cmp exit $?";
  sudo stat -c '%a %U:%G %n' /etc/example-api/example-api.conf "${saved}"; echo "stat exit $?";
Normal: "cp exit 0", "cmp exit 0", two stat lines with the same mode,
owner, and group, then "stat exit 0". Anything else: 6.2 has failed;
follow its failure line, and treat the saved copy as unusable.
Put back (6.3 or 6.4 above):
  sudo cp -p "${saved}" /etc/example-api/example-api.conf; echo "cp exit $?";
  then the same cmp and stat lines.
Normal: the same results. Anything else: stop and escalate, with no
reload; report which commands ran and whether the live file's state is
known.
```

The ticket number in the name (INC-1234 here) keeps one operation's copy from replacing another's, and the ls line before the copy checks that nothing holds the name yet. That matters: in the sandbox, cp wrote through a link that already held the name, into the file the link pointed at, and a link with no file behind it still made ls print the name and exit 0. The cp status matters on its own as well. When the saved name was a second name for the live file itself, cp refused with "are the same file" and exit 1, while cmp still exited 0 and stat showed the same mode and owner: no separate copy existed.

The cmp and stat lines only read, so they run even after a failed copy, and the three results are judged together: nothing relies on the copy unless all three are normal. cmp prints nothing and exits 0 when the files match. When they differ it prints the first difference and exits 1, and when a file is missing it prints an error and exits 2; the sandbox showed all three. The stat line covers what cmp leaves out: a copy with the right contents and the wrong mode or owner can stop a service from reading it.

After a put-back that checks out, what the running service needs depends on what it loaded. If it never loaded the new file and its running state stayed the same while that file was on disk, the file on disk is all that changed, and no reload is needed. If it loaded it, or its running state changed (an automatic restart that failed on the new file, say), the page's section 7 says how to return the running service, and section 7's own checks show the undo worked, since section 8's checks of the new state can fail after a rollback that worked. Two cautions belong on the page. cp writes over the live file in place, so a program that reads it during the copy can see part of it; if yours can, ask the service owner which method to use. And putting the old file back puts back whatever was wrong with it, so section 7 says which failures permit the rollback.

3. Author's checklist

Run this before the status leaves Designing. Every "no" names a field to fix.

- [ ] The title names a verb, one thing, and one system, and the identifier is unique.
- [ ] The objective names a state a command or a dashboard can show.
- [ ] Every known case that produces the same alert but needs something else is out of scope, with a pointer, and evidence that fits no action branch escalates.
- [ ] Every evidence check is read-only, says what normal looks like, and measures what its title names; the page lists the access each check needs.
- [ ] The evidence checks show the known cause this page fixes is present.
- [ ] Every evidence and verification check describes the lines it always prints and each other kind of line it can print, normal and abnormal, so anything else is an error, and each normal and abnormal result is listed as a whole, statuses included. After a pipeline, or any command whose output alone cannot tell a finding from a failure, a line prints each exit status by name; a status line that never printed counts as a failed check. The page names the shell it needs and the settings its checks rely on, such as errexit off.
- [ ] Every branch condition refers to the output of a numbered check or a named prerequisite, and every combination of results reaches exactly one branch. A branch with alternatives lists them on separate lines under one shared condition.
- [ ] Each branch that changes something has one action, with its target named exactly and no glob (wildcard pattern); a branch that escalates makes no change to the service. An action can be a set sequence of numbered sub-steps, each with a check, that stops at the first failed check, with any step that cannot be undone as late as the operation allows; two different remedies on one judgment are two actions. Each action and sub-step says what must print, and a missing line stops the page as a wrong one does.
- [ ] Values the reader chooses sit on their own line, before the command that uses them, and the page says which values are allowed and what any other value does.
- [ ] Every command line on the page ends with `;` or `&&`.
- [ ] Any automation a step runs, an Ansible playbook included, is named exactly and limited to the one target the branch names.
- [ ] Verification reuses the target the action set, instead of setting its own, and no later loop or check overwrites the name that holds it. A check that counts events uses a starting time, in UTC, that fits what it measures: one recorded after the action succeeds for later work, or one recorded just before it starts for events the action itself logs; a check of current state runs after the action.
- [ ] The rollback section says how to undo the action, what it saves first, which failures permit it, who may do it, how to check the undo worked, and what happens if the undo fails; for a sequence, it says this for each sub-step that changed something. Or it says plainly that the action cannot be undone.

- [ ] Any check whose answer another person or process can change before the action runs again right before the first change, and the page says what to do when the branch no longer holds.
- [ ] Every source of conflicting work (a person, a timer, a deploy tool, another service) is named. Where that work could do harm, the page keeps each one off before the first change, with a method the service owner approves, says how each hold is confirmed and released, and releases only holds it took; or it stops and escalates. A fault that can start on its own has a recheck, a stop line, or a verification check, and the page says what they can miss.
- [ ] For an action that cannot be undone, the page names what must already be true for it to be safe, and a prerequisite or evidence check proves it: a restore that was tested, a primary that is fenced. A file listing or an unanswered health check falls short of proof. When that proof can change before the action, the page checks it again right before the step.
- [ ] Verification checks every part of the objective, each with a time limit set from the work's longest normal duration, and says what pending means and what to do when any check fails. Before an outside deadline, the page names the latest start, worked out from the longest normal run, the checks, and the time to escalate.
- [ ] Escalation names a role and lists what to hand over.
- [ ] The page lives in one place, under version control, and the alert links to it when an alert is the trigger.
- [ ] A safe way to stage the fault for a tester exists, and it is written down, with how to undo the staged fault if the test ends before the action.
- [ ] The author has run every command on the page somewhere safe, and each check printed what its text says it prints.
- [ ] No brackets are left, and every field was filled from something checked: the service, its owner, or a test. A fact nobody has yet stays bracketed, with who can supply it. Until all of this holds, the status stays Designing.

Staging a test

A tester needs a fault to find. Before handing a page over, write down:

- Where the fault goes: a staging server, or another place where nothing real depends on the service.
- How to make it: the command or step that creates the fault, and who may run it.
- Which branch it should produce, and the evidence that shows it did. Check that evidence yourself before the tester starts; a staged fault that produces some other case tests some other branch.
- Which names change: if staging uses different hosts, paths, or units, each substitution, written down before the tester starts.
- How to put staging back after each way the test can end (the action finished, the action stopped partway, or the test ended before it), the check that shows the service is back to normal, and who watches an action still running when the test ends.
- What stops the test early, and where the tester's commands and output go.

For the worked runbook, on a staging server only, keep test uploads arriving fast enough that all three instances are needed, and freeze one instance while it holds a file in its working folder:

```
sudo systemctl kill --signal=SIGSTOP thumbnail-worker@2.service;
```

After 10 minutes, run E1 to E4 and confirm the whole Branch A case before handing it over: every check runs without an error, E1 shows all three active and running, the frozen instance's E2 count is 0 and the other two show 20 or more, E3 lists its file, E4 shows under 85 percent, and the staging record shows no restart in the last hour or of that instance in the last 24 hours. Branch A can hold before any waiting file reaches the alert's 15 minutes, so if the test is meant to include the alert, confirm that it fired as well. If the freeze landed between jobs, E3 lists nothing: resume the instance and freeze it again. If the test ends before the restart runs, resume the instance and confirm it logs completed jobs again:

```
sudo systemctl kill --signal=SIGCONT thumbnail-worker@2.service;
```

4. Test record sheet

For the tester. Fill it in as you work, from the page alone, with no help from the author. Note each place you hesitated, guessed, asked, or left the page, pass or fail; the author decides what each means.

Runbook identifier and title:

Page version or commit tested: | **Same as the current page:** yes / no

Tester: | **Author present:** yes / no (watching only)

Environment: | **Date and start time (UTC):**

How the fault was staged: | **Author confirmed the case first:** yes / no

Outside deadline (UTC), if any: | **Stopped early:** no / yes (when, why)

Action started (UTC): | **Returned (UTC), or still running:**

Rollback run: yes (result) / no / none exists | **Staged fault undone:** yes / n/a

Changed by another person or process during the test, and what:

Section	Hesitated, guessed, asked, or left the page (what, why)
Header, 1. Objective	
2. Trigger	
3. Prerequisites	Each check's output and verdict; what the ticket and handoff showed.
4. Evidence	For each check, E1 on: what it printed, and finding, normal, or error.
5. Decision path	Branch reached:
6. Action	
7. Rollback	
8. Verification	Each check's time (UTC), result, and part of section 1; for an alert, its state as well as any count.
9. Escalation	Who, how, and when (UTC); what was handed over.

Result: pass / partial / fail | **End time (UTC):**

Pass means every check on the path the page took ran and is recorded above; the staged fault showed in them; the page reached the branch or stop the fault calls for; the tester used the current page alone and kept to its cutoffs, stops, and escalations; and, for an action, section 3 was normal, the action and each sub-step printed what the page expects, on the target the evidence named, and every verification check passed in time, covering all of section 1. A no-change branch, or a stop the fault calls for, also needs no change to the service and the escalation recorded above. Partial: the current page alone, its stops kept, every check run without an error and recorded, the fault's action branch reached and still shown by any recheck, and a stop before acting. Neither rests on a change by another person or a process that helped or left the cause uncertain. A test that ends with the action still running is a fail; record who watches it. Anything else is a fail.

Changes the page needs:

Validation history line to add:

5. Worked example: RB-MEDIA-004

A complete runbook for a constructed service. Three instances of `thumbnail-worker` run on one server. Uploaded images land in an incoming folder; an instance claims one by moving it into its own working folder, writes a thumbnail to the done folder, and logs a line that starts "completed job" and names the file. An alert fires when the oldest file in the incoming folder is more than 15 minutes old.

```
RB-MEDIA-004  Restart one stalled thumbnail-worker instance on media-01

Owner:         media service on-call (role); service owner: media platform lead
Status:        Pilot, page revision 2 (retested in staging by a second
               operator, 2026-11-05 UTC; approved for real alerts by the
               media platform lead, 2026-11-06 UTC, who watches the first
               uses)
Next review:   2027-02-01, or after any change to thumbnail-worker
Lives at:      operations-runbooks repository, media/RB-MEDIA-004.md
               (the authoritative copy); alert THUMBNAIL-QUEUE-AGE links
               here by identifier
Applies to:    media-01, thumbnail-worker@1 to @3, worker version 2.3 or later
Out of scope:  other servers or services; an instance in the failed state
               (use RB-MEDIA-002); slow instances, two or more quiet
               ones, disk use at 85 percent or more, one quiet
               instance holding no file,
               or a backlog after a large upload with all three busy
               (escalate, by Branch B or C as section 5 decides;
               nothing here to restart)

1. Objective
No file in /srv/thumbnails/incoming is older than 15 minutes, and all
three instances are logging completed jobs.

2. Trigger
Alert THUMBNAIL-QUEUE-AGE: oldest file in /srv/thumbnails/incoming is
older than 15 minutes.

3. Authority and prerequisites
- The on-call operator may restart one instance, once per hour, with no
  change record. A restart counts once its command is run, even if it
  fails. Before section 6, check the ticket and the on-call
  handoff for a restart of any instance in the last hour, and of the
  instance E2 shows at zero in the last 24 hours, dating each restart
  by the time written in the ticket just before it; if there was either,
  or you cannot tell, the service owner decides. Anything beyond that
  needs the owner. The team lets only the ticket's assignee act on this
  alert: assign the ticket to yourself before E1 and check that it
  shows you; if it shows someone else, they decide, and you make no
  change. When you escalate, assign the ticket to the person you hand
  it to.
- A bash shell on media-01 with errexit off. E1, E3, E4, and V2 need
  no sudo: the on-call account can read the unit state, the working
  and incoming folders, and the disk use they check, and can see the
  current state of THUMBNAIL-QUEUE-AGE. The exit status lines read
  bash's PIPESTATUS; with errexit on, grep's exit 1 on a count of 0
  (E2, V1, V3) ends the shell before its status line prints.
  Check it before E1:
    [[ -o errexit ]] && echo "errexit on" || echo "errexit off";
  Normal: "errexit off". If it prints "errexit on", run
  set +o errexit; and check again.
- sudo, which E2 (and V3, its rerun), section 6, and V1 use as
```

```
  written. Without a sudo rule, sudo prints an error (the tested one
  names the sudoers file), and in E2 and V1 the count shows 0. Run
  without sudo by a reader who lacks access to the system journal,
  journalctl prints a notice such as
  "No journal files were opened due to insufficient permissions."
  and the count shows 0. E2 and V1 describe neither line, so the check
  failed. sudo may ask for your password, or print its first-use
  notice, before a command runs; that is sudo talking, and no part of
  any check's output. An error from sudo is still an error.
- All three units run the same program on the default stop and start
  timeouts, 90 seconds each. Check every instance section 6 might
  restart:
    for worker in 1 2 3; do systemctl show "thumbnail-worker@${worker}.service"
    --property=Id,ExecStart,TimeoutStartUSec,TimeoutStopUSec; done;
  Prints four lines per instance. Normal: each ExecStart line names
  path=/opt/media/bin/thumbnail-worker, and
  TimeoutStartUSec and TimeoutStopUSec are both 1min 30s. Anything
  else: stop and escalate.
- Worker version 2.3 or later, which moves any file left in its working
  folder back to incoming when it starts. When it starts after a
  restart, it therefore requeues the job the instance was holding.
  Check the program the units run, by the path the check above printed:
    /opt/media/bin/thumbnail-worker --version; echo "version exit $?";
  Normal: one line naming version 2.3 or later, then "version exit 0".
  An earlier version, any other line, or any other status: stop and
  escalate, because an older worker can lose the held job.
- Paste each command and its output into the incident ticket as you go;
  every check here prints a few lines. For longer output, paste an
  extract and say where the full copy is saved.

4. Evidence (read-only; run all of it before deciding, unless a stop applies)
A check ran without an error when it printed each line its text below
says it always prints, every other line it printed is one its text
describes, and its result as a whole, exit statuses included, is one
listed there. An abnormal result, such as a count of 0 or a listed
file, is a finding: it decides the branch. A missing line, any other
line (journalctl's permission notice, a "No such file or directory"),
or any other result means that check failed (Branch C).
E1  Instance state
    for worker in 1 2 3; do systemctl show "thumbnail-worker@${worker}.service"
    --property=Id,ActiveState,SubState; done;
    Always prints Id=, ActiveState=, and SubState= for each instance.
    Normal: ActiveState=active and SubState=running for all three.
    Abnormal: any other state, such as ActiveState=failed.
E2  Completed jobs per instance, last 10 minutes
    for worker in 1 2 3; do printf 'instance %s: ' "${worker}"; sudo journalctl
    --unit="thumbnail-worker@${worker}.service" --since='10 minutes ago' --no-pager --output=cat |
    grep -c '^completed job '; printf 'instance %s: journalctl exit %s, grep exit %s\n' "${worker}"
    "${PIPESTATUS[0]}" "${PIPESTATUS[1]}"; done;
    Always prints two lines per instance: the count, then both
    exit statuses.
    Normal: 20 or more per instance while the queue has work, and
    both statuses 0 (journalctl exit 0, grep exit 0). Abnormal: a count
    of 0, with journalctl exit 0 and grep exit 1, or a count from 1 to
    19, with both statuses 0. Each count covers the 10 minutes before
    its own query: run the loop in one go, and again if interrupted.
E3  Files held in a working folder longer than 10 minutes
    find /srv/thumbnails/working -type f -cmin +10 -printf '%h %f %CY-%Cm-%Cd %CH:%CM %Cz\n'; echo
    "find exit $?";
    Always prints "find exit" and its status last; a held file adds a
    line before it.
    Normal: only the line "find exit 0". Abnormal: one or more lines
    naming a held file, then "find exit 0". A listed folder ends in the
```

```
    instance number (/srv/thumbnails/working/2 is instance 2). The move
    into the folder sets the change time that -cmin reads. An instance
    holds one file at a time, and a job takes about 20 seconds and never
    more than two minutes.
E4  Disk space
    df -h /srv/thumbnails;
    Always prints a header line and one line for the filesystem. The
    incoming, working, and done folders are plain folders on it.
    Normal: under 85 percent used. Abnormal: 85 percent or more.

5. Decision path
Branch A: one instance stalled
  If every check ran without an error,
    and E2 shows zero for exactly one instance,
    and E2 shows 20 or more for each of the other two,
    and E1 shows all three active and running,
    and E3 lists a file in that instance's working folder,
    and E4 shows under 85 percent used,
    and section 3's restart check found no restart in the last hour
      and none of that instance in the last 24 hours,
    then go to section 6 for that instance.
Branch B: slow or quiet instances, or the disk at its limit
  Otherwise, if every check ran without an error,
    and no instance is in the failed state,
    and at least one of these holds:
      E2 shows zero for two or three instances;
      E2 shows a count from 1 to 19 for any instance;
      E4 shows 85 percent or more used;
    then escalate with E1 to E4. Make no change.
Branch C: anything else
  Otherwise (a check ran with an error, an instance is in the failed
  state, or the evidence fits neither branch): escalate with the output.
  Make no change.
Stop at any point if a second instance goes quiet or any instance is
in the failed state, and escalate with the evidence collected so far.

6. Action (Branch A only)
Right before the restart, run E1 to E4 again, check that the ticket
still shows you, and repeat section 3's restart check in the ticket
and the on-call handoff. If the ticket shows someone else, Branch A
no longer holds, or it now points at a different instance, make no
change: escalate with both sets of output.
Set instance to the number the second E2 run showed at zero. The 2
below belongs to this example. Use only 1, 2, or 3: for any other
number, systemd starts a new, fourth instance. Just before you run the
second line, write the UTC time from a clock outside this shell in the
ticket; if anything delays the line, write a new time. The 3 minutes
below count from that time, any sudo prompt included.
    instance=2;
    sudo systemctl restart "thumbnail-worker@${instance}.service" && restart_time="$(date -u
    '+%Y-%m-%d %H:%M:%S.%6N UTC')" && echo "instance=${instance}; restart_time='${restart_time}';";
Expected, after any sudo prompt: one line, such as
instance=2; restart_time='2026-11-20 03:14:08.512340 UTC';
If that line is missing, anything else prints, or the restart has
not returned after 3 minutes, stop. No second restart. Escalate, and
say whether the restart returned; one that has not may still be
running: 3 minutes is this page's cutoff, set from the two timeouts
section 3 checked, and systemd can take longer than their sum.
Copy the line into the ticket. In a new shell, run section 3's errexit
check, then paste the line to set both values again.

7. Rollback
A restart cannot be undone. Section 5 limits it: one instance, once, and
```

```
only one that has completed nothing in 10 minutes. It holds one file at
a time (E3), so the restart interrupts at most one job: the one it
holds at that moment. A peer that stalls after E2 ran can show up in
V3. On startup the worker returns the held file to incoming for another
attempt. If the job runs again, its output name comes from its input
name, so the rerun overwrites its own earlier output and creates no
duplicate.

8. Verification
Run this section only after section 6 printed its line. V1 to V3 count
from restart_time, which that line holds.
V1  Within 5 minutes of the restart, the restarted instance logs a
    completed job. Uses instance and restart_time from section 6; in a
    new shell, run section 3's errexit check and paste the line section
    6 printed first. Always prints the count, then both exit statuses;
    a "Failed to parse timestamp" line means restart_time is empty or
    malformed.
    sudo journalctl --unit="thumbnail-worker@${instance}.service" --since="${restart_time}"
    --no-pager --output=cat | grep -c '^completed job '; printf 'journalctl exit %s, grep exit %s\n'
    "${PIPESTATUS[0]}" "${PIPESTATUS[1]}";
    Pass: 1 or more before the 5 minutes are up, with journalctl exit
    0 and grep exit 0. A job can take two minutes, so a 0 with
    journalctl exit 0 and grep exit 1 before then means wait and run
    it again.
V2  Within 30 minutes, the alert clears and this prints 0, then
    "find exit 0, wc exit 0":
    find /srv/thumbnails/incoming -maxdepth 1 -type f -mmin +15 | wc -l; printf 'find exit %s, wc
    exit %s\n' "${PIPESTATUS[0]}" "${PIPESTATUS[1]}";
    Always prints the count, then both exit statuses. Before the 30
    minutes are up, find exit 0 and wc exit 0 with a count above 0, or
    with the alert still active, means wait and run it again.
V3  Fifteen minutes after the restart, rerun E2, so its 10-minute window
    starts after the restart; it prints E2's lines. Pass, before the 30
    minutes are up: all three show 20 or more, and every status line
    shows journalctl exit 0, grep exit 0. Its loop leaves instance
    alone, so V1 still names the restarted instance.
Run each check at its own time, V3 included, even while V2 is still
pending. A wait applies only to a check that ran without an error.
If any check fails or ran with an error, or a second instance goes
quiet: stop. No second restart. Escalate.
If the same instance stalls again within the 24 hours section 3
counts, escalate even though V1 to V3 passed.

9. Escalation
Media platform lead, through the media service on-call rota. Hand over:
the alert time in UTC, E1 to E4 output, any restart run with the time
written before it (UTC) and what it printed, and the V1 to V3 results
so far.

10. Validation history
2026-11-02  staging  second operator  pass  Page revision 1, worker
            2.3.1; staged @2 frozen with SIGSTOP. Restart 10:12 UTC; V1
            2 at 10:15; V3 23 21 24 at 10:27; V2 0 and alert clear at
            10:33. Ran the page alone and asked how to confirm the
            requeue. Version check added to section 3.
2026-11-05  staging  second operator  pass  Page revision 2, worker
            2.3.1; same fault. Restart 14:02 UTC; V1 3 at 14:05; V3 24
            22 23 at 14:17; V2 0 and alert clear at 14:21. No page change.
2026-11-06  production  media platform lead  approved  Revision 2 for
            real alerts; the lead watches the first uses.
```

What to copy and what to leave

Copy the habits: one known case, with the cases that look like it written down and sent elsewhere; checks that say what they print, exit statuses included; branches whose conditions name check results; one action, on the target the evidence named, under a name nothing else overwrites; a time the action records, which verification reuses; and a verification check, with a time limit, for every part of the objective.

Leave the numbers. Three instances, 20 completions in 10 minutes, 85 percent, the 5, 15, and 30 minute limits, and the 3-minute cutoff belong to this constructed service. Work out your own from your service, as the field guide describes.

6. Why each line of RB-MEDIA-004 exists

The worked runbook is printed in full in section 5. This section walks through it in order and says what each line is for, and so what leaving it out would let back in. A change the page asks for, such as the instance number, is part of following it. A reader who cannot follow a line as written stops there, records where and why in the ticket, and escalates. That use counts as a failed use: the page goes back to Ready to test, its owner works out what went wrong, and the page moves on again only as the lifecycle table says. The article's section "When the reader leaves the page" gives examples from this runbook. Where a line was added or changed because of a defect, the note gives the entry number in section 7, "Defects this runbook survived". Read the runbook and these notes side by side: the runbook shows what the page says, and the notes show what each line protects against.

Header

`RB-MEDIA-004 Restart one stalled thumbnail-worker instance on media-01`: a stable identifier, then a verb, one thing, and one system. The title tells a reader what the page changes and where; the title and the trigger together tell them whether the page is for their alert.

`Owner:` names a role, so the page outlives any one person, and the service owner, whose approval the page requires for anything beyond the on-call operator's stated authority.

`Status:` carries the lifecycle state, the page revision, who other than the author last tested it, where, and when, and for Pilot, who approved real use and who watches it. A validation date covers one revision of the page.

`Next review:` gives a date and an event. For a page in Pilot or Operational, a change to the worker sends it back to Ready to test before the date arrives.

`Lives at:` names the one authoritative copy and says the alert links to it by identifier, so a reader at 03:10 can go straight to the maintained copy.

`Applies to:` limits the page to media-01, instances 1 to 3, and worker 2.3 or later. The version limit is a safety condition: in the constructed model a 2.3 or later worker moves the held job back to incoming when it starts, so once the worker starts again, the job the restart interrupted goes back to the queue instead of being lost. The rollback section's sentence about the rerun depends on that requeue.

`Out of scope:` lists each known case that raises the same alert and needs something else, with where it goes. A case nobody listed goes where its evidence sends it. Evidence that fits Branch B or C escalates it; evidence that meets every Branch A condition reaches the restart. If the restart leaves the problem in place, a failed V1 to V3 or a repeat stall within 24 hours sends it to escalation; a problem neither one shows stays unseen by this page.

Sections 1 and 2: Objective and Trigger

The objective has two parts, the queue and all three instances logging completions, and each part has a verification check: V2 for the queue, V3 for the instances. The objective, the alert, and V2 use the same limit: a file older than 15 minutes.

The trigger names the alert and its threshold. The alert measures the oldest file waiting in the incoming folder, so a stalled instance raises it only once some waiting file passes 15 minutes. If the other two instances keep up with the uploads, no waiting file gets that old, and the alert stays silent while an instance is stalled. The article's paragraph on staging a test calls this a gap in the trigger, worth writing down.

Section 3: Authority and prerequisites

The restart-history bullet bounds the operator's authority: one restart an hour, and no second restart of the same instance within 24 hours, the limit section 8's 24-hour line also sets. It tells the reader where to look (the ticket and the on-call handoff) and who decides when they cannot tell. Branch A names this check, so an unknown answer can never reach the restart. Checking history alone leaves room for two people who answer the same alert to both reach section 6. In the constructed model the team lets only the ticket's assignee act on its alert, so the reader assigns the ticket to themselves before E1, checks that it shows them, and checks again right before the restart; anyone who finds it held by someone else makes no change. The assignment is a hold only because the team keeps that rule, and the reader releases it by assigning the ticket to whoever they escalate to. A restart counts against the hour once its command is run, even if it fails: a failed attempt leaves the instance in a state nobody has checked, so the next reader goes to the owner (a constructed policy; a service owner may draw the line elsewhere and should say where). Both limits count from the UTC time written in the ticket just before each restart, because that time exists even for an attempt that failed or never returned, and restart_time exists only for one that returned (also constructed).

The shell bullet names bash, because the status lines read `PIPESTATUS`, and checks errexit: with it on, grep's exit 1 on a count of 0 (E2, V1, V3) ends the shell before the status line prints, and V2's count of 0 leaves the shell running, since find and wc exit 0 (lab test, drafting sandbox, bash 5.2). The sudo bullet names the access each check needs as written: sudo for E2 (and V3, its rerun), section 6, and V1, and none for E1, E3, E4, and V2, which also needs a view of the alert's current state. It describes the error sudo prints without a rule, quotes the notice journalctl prints when run without sudo by a reader who lacks access to the system journal, and says either line means the check failed (entry 13). The count note covers E2 and V1; V3 is a rerun of E2 and counts the same way. Section 6 has no count. The sentences about sudo's password prompt and first-use notice place them outside every check's output, and the last sentence keeps an error from sudo counted as an error.

The timeouts loop checks all three instances, because section 6 may restart any one of them (entry 7). Like the loops in E1 and E2, it counts with `worker`, leaving `instance` for the target section 6 sets (entry 14). It also prints each ExecStart line, which the version check depends on (entry 6). "90 seconds each" describes the timeouts without claiming their sum bounds a restart (entry 8).

The version bullet states what the action does to work in progress in the constructed model (a 2.3 or later worker requeues the held job when it starts) and gives a way to confirm, on the server, that the program the units run meets that version (entry 6). The version check confirms the prerequisite; a staging test is where the held job's fate is observed.

The ticket bullet asks for each command and its output, with a short extract and the full copy's location in the ticket for anything long, so the escalation can start from the record.

Section 4: Evidence

The opening rule decides "ran without an error" by enumeration: the check printed every line its text says it always prints, every other line it printed is one its text describes, and its result as a whole, statuses included, is one its text lists. A missing line, or any other line or status, is an error, with one treatment, Branch C (entries 2 and 13). E1 and E2 always print a result for each of the three instances, so output missing an instance is an error. Alternatives, such as E3's held-file lines, are described without being required, so a check that prints one of them can still run without an error.

An instance in the failed state ends the page at E1, by section 5's stop line, away from the restart. In the rehearsal, a frozen instance still reported active and running, so E1 cannot show the stall; E2 and E3 show it.

E2 counts lines that start with `completed job`, per instance, and prints both statuses. For a run whose lines and statuses are all described, its normal and abnormal results cover every count: 0 with journalctl exit 0 and grep exit 1, 1 to 19, and 20 or more (entry 9 explains the floor). Each query's `--since='10 minutes ago'` is read when that query runs, so the three windows can start at different times; the page asks for the loop in one go, and a rerun if it is interrupted.

E3 always prints its status line last. One or more held-file lines, then find exit 0, is the finding; a held-file line beside any other status is an error. It reads change time, because a move leaves the modification time at the upload (entry 1). Its last sentences state the two model facts the page relies on: one file at a time, and jobs of about 20 seconds and never more than two minutes.

E4 checks the disk, which none of E1 to E3 measures. It looks at one filesystem, so the page states the model fact that makes one enough: the incoming, working, and done folders are plain folders on it. Branch A needs it under 85 percent. At 85 percent or more it is one of Branch B's alternatives, which counts only when Branch B's shared conditions hold (entry 11).

Section 5: Decision path

Branch A lists every evidence condition the restart needs, including "every check ran without an error", and the restart-history check. Section 3 has already stopped the page if a timeout, program path, or version check failed, so those need no repeat here.

Branch B starts with "Otherwise" and puts its alternatives on separate lines under one shared condition (entry 11). It escalates with the evidence and changes nothing.

Branch C takes everything left, including any check that ran with an error.

The stop line ends the page at any point if a second instance goes quiet or any instance is in the failed state, even partway through E1 to E4: the reader escalates with what they have instead of finishing the checks. Sections 3, 6, and 8 carry their own stop conditions.

Section 6: Action

The page runs E1 to E4 again, checks that the ticket still shows the reader, and repeats section 3's restart check in the ticket and the handoff, right before the restart. Between the first evidence run and the restart, a peer can stall or another operator can restart an instance; if the ticket shows someone else, or Branch A no longer holds or points at another instance, the reader changes nothing and escalates with both sets of output.

The instance number sits on its own line, with the instruction to use the number the second E2 run showed at zero and the allowed values (entries 5 and 12).

The restart line runs the restart, records the time only if the restart command succeeds, to the microsecond and in UTC, and prints a line that restores both values in a new shell (entries 3 and 4).

Just before running the restart line, the reader writes the UTC time from a clock outside the shell in the ticket, because a command that has not returned holds the shell, and the 3 minutes count from then, a sudo prompt included; a delay means a new time. The written time starts the cutoff and dates the attempt, and section 3's restart limits count from it; whether the line ran, and what it printed, goes in the ticket beside it, as section 9 asks. After any sudo prompt, the expected result is one line. That line missing, anything else, or no return within 3 minutes means stop, no second restart, escalate, say whether the restart returned, and say one that has not may still be running (entry 8). A missing line matters because a failed restart leaves any earlier restart_time in place (entry 4). In a new shell, the reader checks errexit again before pasting the line back, since a new shell can start with different settings.

Section 7: Rollback

A restart cannot be undone, so the page says so and names what limits it: one instance, once, and only one that completed nothing in 10 minutes. It cites E3's one file at a time, a fact of the constructed model, for the claim that the restart interrupts at most one job, the one the instance holds at that moment (normally the file E3 listed, unless the instance resumed after the recheck), and says why a rerun of the requeued file creates no duplicate. It promises the file another attempt and makes no claim that the attempt succeeds: a restarted worker can stall on the same input, which is what section 8's 24-hour line is for. It says a peer that stalls after E2 ran can show up in V3: V3 counts completions over a window, so a peer that stalls late in that window can still reach 20 (entry 10).

Section 8: Verification

V1 reuses the instance and the restart marker from section 6 (entry 5) and counts from that marker (entry 3). It passes on 1 or more completions before the 5 minutes are up, with journalctl exit 0 and grep exit 0. Before then, only a 0 with journalctl exit 0 and grep exit 1 means wait and run it again. A 0 beside journalctl

exit 1, such as the one an empty or malformed restart time gives (entry 5), is an error, and the reader escalates.

V2 checks the queue part of the objective. Within 30 minutes the alert must clear and find must list no incoming file older than 15 minutes (a count of 0 lines), with find exit 0 and wc exit 0, so a missing folder cannot pass (entry 2). Before the 30 minutes are up, find exit 0 and wc exit 0 with a count above 0, or with the alert still active, means wait and run it again. A wait applies only to a check that ran without an error, so a missing folder is an error at once, even with the alert active.

V3 checks the instances part of the objective. It reruns E2 fifteen minutes after restart_time, so each query's 10-minute window starts after the restart, and the three queries can start at slightly different times. It passes only if, before the 30 minutes are up, all three show 20 or more with every status line showing journalctl exit 0, grep exit 0 (entry 10). V3 runs at its own time even while V2 is still pending. The three checks show each result within its window; the page makes no claim that every result still holds after its check ran.

If any check fails, ran with an error, or a second instance goes quiet, the reader stops, makes no second restart, and escalates.

The 24-hour line sends a repeat stall of the same instance within the 24 hours section 3 counts to escalation, even though V1 to V3 passed: a restart can clear one instance's symptom and leave its cause in place, and a recurrence is a problem for the service owner. Section 3's restart check carries the same limit forward, so a restart recorded in the ticket or handoff sends the next reader to the service owner instead of to a second restart.

Sections 9 and 10: Escalation and validation history

Escalation names a role and how to reach it, and lists what to hand over (the alert time, the evidence, any restart run with the time written before it and what it printed, the verification results so far), so the person receiving it has those without asking for them.

The history has two test lines and an approval line. Each test line records its key results with UTC times. The first test led to a page change (the version check), so the first test no longer covered the page, which stayed at Ready to test, and the second line records the retest of revision 2. The approval line names who approved real use and who watches it.

7. Defects this runbook survived

Use these cases to review a runbook of your own. Each starts with something the worked runbook once said, shows the case that broke it, and gives the fix and what settled it; its last line turns it into a question for your page. Entries 1 to 13 come from five rounds of outside review of the article's RB-MEDIA-004, in which 88 findings were applied and more were rejected after a test or a source showed them wrong; entry 14 came from a later review of this manual. This part keeps fourteen of the applied defects and four of the rejected claims.

By what goes wrong:

- Evidence that cannot be trusted: entries 1, 2, 6, and 13.
- An action that reaches too far or runs too long: entries 7, 8, 11, and 12.
- Verification of the wrong thing or from the wrong time: entries 3, 4, 5, 10, and 14.
- A model assumption that fails a healthy service: entry 9.

Each entry as a pattern to watch for in your own page:

Entry	Pattern to watch for	What the page does about it now
1	A time that only looks like the one you mean	E3 reads change time, which the move into the folder sets
2	A failed check that prints the passing value	A status line after each pipeline, and a rule that lists what each check prints
3	A time marker that counts work from before the action	The restart time, to the microsecond, recorded after the restart succeeds
4	A time with no zone, or taken before the action ran	One line restarts, records the time in UTC only on success, and prints it
5	Verification that sets its own target	V1 reuses the instance and time section 6 printed
6	A prerequisite checked on a different program	The version check runs the path each unit's ExecStart names
7	A prerequisite checked on one of several targets	The timeouts loop covers all three instances
8	A cutoff read as a promise about the system	3 minutes is this page's cutoff, and the escalation says the restart may still be running
9	A model number that fails a healthy service	The job time and the floor of 20 are stated as parts of the model
10	Verification that misses part of the objective	V3 checks all three instances, so each part of section 1 has a check
11	A branch that reads two ways	Alternatives on separate lines under one shared condition
12	A target value the reader can get wrong	The allowed values, and what any other number does
13	Missing access that looks like a finding	Section 3 names each access and the lines a missing one prints, and the rule counts them as errors
14	A later check that overwrites the target	The loops count with `worker`; only section 6 sets `instance`

Each entry gives what the page said, the case that broke it, the fix now on the page, and what settled it. The labels mean:

- Lab test: the command, or the behavior the page relies on, was tested on the lab workstation (systemd 259.5, GNU findutils 4.10, bash 5.3) or in a drafting sandbox (systemd 255 tools, GNU findutils 4.9.0, GNU coreutils 9.4, bash 5.2, sudo 1.9.15p5). Some tests ran the page's own commands with file paths swapped for test folders and unit names for temporary user units; others tested one behavior on its own, and the entry says which.
- Model: the result belongs to the constructed service, so no command tested it.
- Arithmetic, Reading, Model test: settled by a calculation, by a careful read of the page, or by running the page's rules over a model of its evidence.

The worked runbook's parts are named the way the page names them: E1 to E4 are its evidence checks, Branches A to C its decision path (A is the restart, B and C escalate without a change), section 6 the action, and V1 to V3 its verification checks.

1. Time held, measured from the upload

What the page said: E3 listed files held in a working folder for over 10 minutes with `-mmin +10`.

The case that broke it: an instance claims a file by moving it from the incoming folder into its own working folder. Modification time records when a file's contents last changed, and a move leaves it where the upload set it. A file that waited 12 minutes in incoming and was claimed a second ago is 12 minutes old by `-mmin`, so E3 listed a held file the instance had just picked up. E3 is one of the conditions Branch A acts on, so the check could report a stall that never happened.

The fix: E3 reads the change time with `-cmin +10`. Change time records when the file's status last changed, and the move sets it. E3 prints the change time it used, so the reader can see it.

Lab test: a file with its modification and access times set 20 minutes back, then moved into another folder. `-mmin +10` listed it, and `-cmin +10` left it out. Same result on findutils 4.9.0 and 4.10.

What the fix still assumes: anything that changes a file's status (its permissions, its owner) also resets the change time. The constructed worker moves the file and leaves it alone after that. On a service that touches held files, E3 could miss one, which sends the reader to Branch C (escalate), the safe direction.

Check your own page: Does each check measure the time or event your page cares about, such as time held, as opposed to a time that only looks like it?

2. A pipeline that prints its pass value when it fails

What the page said: V2 counted old files with `find ... | wc -l` and passed on 0.

The case that broke it: if the folder is missing (a typing slip, or the wrong server), find prints an error and exits 1, while wc counts zero lines, prints 0, and exits 0. With bash's `pipefail` option off, which is bash's default and which the page never turns on, the pipeline's own status is wc's 0. The reader sees 0, the pass

value, and find's error line is the only sign that the check failed.

The fix: each of the page's three pipelines (E2, V1, V2) is followed by a line that prints each command's exit status by name, read from bash's `PIPESTATUS`, which holds the exit status of each command in the last pipeline. Each check's normal result lists the statuses, and section 4 says a check ran without an error only when it printed every line its text says it always prints, every other line it printed is one its text describes, and its result as a whole, statuses included, is one its text lists. The count is what the check counts (completion lines for E2 and V1, the lines find printed for V2); the statuses say whether each command ran with a result the check lists.

Lab test: V2 as written, pointed at a missing folder, printed `0`, then `find exit 1, wc exit 0`, plus find's error line. With `pipefail` off, the pipeline's own status was 0; with it on, 1. A fresh bash started without startup files reported `pipefail` off. A separate check found that all three pipelines print their statuses this way.

Check your own page: Can a failed check print the same result as a passing one?

3. A restart time that counts work from before the restart

What the page said: an earlier revision recorded the restart time to the second.

The case that broke it: journalctl's `--since` with a whole-second time can include a line logged earlier in that same second, before the restart. V1 could count a completion from before the restart as one after it.

The fix: section 6 records the time to the microsecond (a millionth of a second), `date -u '+%Y-%m-%d %H:%M:%S.%6N UTC';`. The recorded time is a marker taken after the restart command succeeds, and V1 counts from that marker.

Lab test: on systemd 259.5, a line was logged with `systemd-cat`, then a whole-second marker and a microsecond marker were taken. The whole-second marker counted the line (1); the microsecond marker left it out (0). Two separate runs gave the same result.

Check your own page: Can the time your checks count from include anything that happened before the action?

4. A restart time with no zone, taken at the wrong moment

What the page said: section 6 set the restart time before running the restart, in local time with no zone, and printed nothing about the instance.

The case that broke it: if the restart failed or hung, the time was already set, and V1 would count from it. A time written without a zone names a different instant to a reader in another time zone. Every Local Timestamp in an Incident Record Needs an Offset covers why a time without its offset is ambiguous: https://revisualized.com/articles/every-local-timestamp-needs-an-offset

The fix: one line runs the restart, then sets the time only if the restart command succeeds (the `&&` between them), in UTC with the zone written out, then prints the instance and the time as a line the reader can paste into a new shell.

Lab tests: journalctl accepted the `UTC` suffix and counted the same lines as the same window in local time. The same digits without the suffix, read on a machine at -0700 (seven hours behind UTC), named another instant (0 lines against 207). Restarting a unit name with no unit file behind it, in a shell with no earlier restart time, left the time empty. The printed line, pasted into a fresh bash, set both values to what section 6 printed.

What the fix still assumes: a failed command skips the assignment and leaves any earlier value in place, so a shell that already holds a restart time keeps it. That is one reason section 6 says to stop, make no second restart, and escalate when the restart prints anything unexpected: the page sends the reader to escalation before V1.

Check your own page: Is the action's time set only after the action succeeds, and written with its zone? If the action itself logs events your checks need, does the page record a second time just before it?

5. Verification that checks a different instance

What the page said: V1 began with its own `instance=2;`.

The case that broke it: a reader who restarted instance 3 and pasted V1 unchanged would check instance 2. One completion from a healthy instance 2 after the restart time would pass V1, and the check meant to catch a failed fix would report success for the wrong instance.

The fix: V1 reuses the instance and restart time from section 6, and in a new shell the reader pastes the line section 6 printed. The checklist asks that verification reuse the target the action set.

Reading: the hard-coded case was settled by reading the page. Lab tests on the fix: with the instance unset, the V1 pipeline counted 0, below V1's pass threshold of 1 or more. With the restart time empty, journalctl printed `Failed to parse timestamp:` and the status line read `journalctl exit 1, grep exit 1`. A malformed time (a mistyped zone) gave the same error and statuses.

Check your own page: Does verification check the target the action changed?

6. A version check on the wrong program

What the page said: section 3 checked the worker version with `thumbnail-worker --version;`.

The case that broke it: a bare program name runs whatever the reader's shell resolves that name to, such as an alias, a function, or the first match on its search path, and that need not be the program the service units run. In the constructed model, a worker older than 2.3 can lose the held job on restart. An old copy named in the units and a new copy on the search path would pass the check while the restart still ran the old one.

The fix: section 3 prints each unit's ExecStart line (the command the unit starts). Its normal result requires every ExecStart line to name `/opt/media/bin/thumbnail-worker`, and the version check then runs that exact path.

Lab test: the section 3 loop printed four lines per instance, with an ExecStart line naming the program path. In the sandbox, a function and an alias with the program's name each ran in place of the copy on the search path. Model: the version behavior, and what an older worker does to a held job, belong to the constructed service. No command tested them.

Check your own page: Does each prerequisite check look at the program or file the action will actually use?

7. Timeouts checked on one instance

What the page said: section 3 checked the stop and start timeouts on instance 1.

The case that broke it: section 6 may restart instance 2 or 3, and checking instance 1 shows nothing about the settings of the instance actually restarted.

The fix: one loop over all three instances prints the identifier, ExecStart, and both timeouts.

Lab test: the loop printed both timeouts at `1min 30s` for each of three test instances, and the system manager's defaults read `1min 30s` for both.

Check your own page: Do the prerequisite checks cover every target the action might choose?

8. A cutoff read as a guarantee

What the page said: the 3-minute limit in section 6 added together the two 90-second timeouts.

The case that broke it: systemd applies the stop timeout to each ExecStop command (a command the unit runs to stop the service) and then again to the service itself, so a restart can run longer than the two timeouts added together. A reader told the sum is the most a restart can take may treat a slow restart as finished, or start a second one.

The fix: section 3 says "90 seconds each". Section 6 calls 3 minutes this page's cutoff, set from those timeouts, says systemd can take longer than their sum, and tells the reader to say the restart may still be running when they escalate.

Lab test: a test unit with 3-second start and stop timeouts, a stop command that runs for 30 seconds, a main process that ignores SIGTERM (the termination signal), and a 2-second start step. Its restart took 8,459 ms against 6 seconds of timeouts.

Check your own page: Does any limit on your page read as a promise about how long the system takes?

9. A job model that sends a healthy instance to escalation

What the page said: a job took about 30 seconds, and a normal instance logged 20 or more completions in 10 minutes.

The case that broke it: at 30 seconds a job, 20 jobs fill the 10-minute window exactly. One job that takes the full two minutes leaves 480 seconds for 16 more, so a healthy instance shows 17 completions. The page named any count from 1 to 19 as abnormal, which sent that instance to Branch B (escalate).

The fix: the model now says a job takes about 20 seconds. With one two-minute job and the rest at 20 seconds, 480 seconds holds 24 more jobs, 25 in all. The normal floor stays at 20, as a stated part of the model.

Arithmetic: the floor is a stated assumption, and the arithmetic only shows how much room it leaves. Three two-minute jobs in one window still drop a healthy instance to 15 (360 seconds for the long jobs, 240 seconds for 12 short ones). The page then escalates, which is the safe direction.

Check your own page: Which normal results rest on an assumption about the service, and what happens when a healthy service falls outside it?

10. Verification that misses part of the objective

What the page said: the objective named the queue and all three instances logging completed jobs; verification checked the restarted instance (V1) and the queue (V2).

The case that broke it: a second instance that stalls after the evidence was gathered leaves V1 and V2 passing while part of the objective fails.

The fix: V3 reruns E2 fifteen minutes after the restart and passes only if all three instances show 20 or more. The checklist asks for a verification check for every part of the objective.

Reading: an earlier revision ran V3 ten minutes after the restart, so its 10-minute window began at the restart itself. At fifteen minutes each query's window begins after the restart, about five minutes after it for the first query. V3 shows the instances working during that window; nothing on the page claims they keep working after it. It counts completions over the window, so a peer that stalls late in it can still pass: eight minutes of 20-second jobs is 24 completions.

Check your own page: Could every verification check pass while part of the objective fails?

11. A branch that reads two ways

What the page said: Branch B was one sentence: every check ran without an error and no instance failed and two zeros, or a count from 1 to 19, or a full disk.

The case that broke it: a reader can take "or a full disk" as standing on its own, outside the first two conditions. Then a check that printed an error beside a full disk reaches Branch B instead of Branch C.

The fix: Branch B lists its alternatives on separate lines under one shared condition, `and at least one of these holds:`, and starts with "Otherwise", as Branch C does.

Model test: a model of the evidence (zero counts on 0 to 3 instances, peer counts usual or low, instance states, a held file or none, the disk under 85 percent or at 85 percent and over, a check error or none) gives 192 combinations. The current rules send each to exactly one branch; the flat reading sends 96 combinations that include an error or a failed instance to Branch B. The model leaves out the restart-history check, which only narrows Branch A further. One branch per combination is half of the review; the other half is checking that each branch is the right one.

Check your own page: Can any alternative in a branch be read as standing outside the branch's shared condition?

12. A number that is wrong, or outside 1 to 3

What the page said: section 6 set `instance=2;` as "2 in this example".

The case that broke it: two slips. A reader who keeps the 2 when E2 showed instance 3 at zero targets the wrong instance, the same problem as entry 5: the restart can interrupt a working instance and leave the stall in place. A reader who types a number outside 1 to 3 (a 4, say) names an instance nobody has started. One reviewer said such a restart would exit 0 and do nothing.

The fix: section 6 tells the reader to set the instance to the number E2 showed at zero, says the 2 belongs to this example, names the allowed values, and says what any other number does.

Lab test: restarting the unused @4 instance of a template user unit (one unit file that starts an instance for any number after the @) started it, and its state read active. The reviewer's mechanism was wrong: the slip makes an unplanned change to the service instead of doing nothing.

Check your own page: What happens if the reader picks the wrong target, or types a value outside the allowed ones?

13. A silent zero from missing access

What the page said: section 3 said E2 and V1 needed sudo or membership in the systemd-journal group (a group whose members may read the system journal), while the commands themselves ran sudo.

The case that broke it: a reader in the journal group with no sudo rule takes the page at its word and runs E2 as written. sudo refuses with an error, journalctl never runs, and `grep -c` prints 0. A reader who drops the sudo and has no journal access gets the same 0: journalctl prints a notice on standard error and nothing on standard output. A count of 0 is also what a stalled instance gives, so a reader who looks only at the counts could take a failed check for a finding.

The fix: section 3 says E2 (and V3, its rerun), section 6, and V1 use sudo as written, describes sudo's error without a rule, and quotes journalctl's notice without journal access. Under section 4's rule, E2 describes neither line, so E2 failed and the reader escalates. After sudo's error, the status line also reads `journalctl`

`exit 1`, a status E2 never lists. That first status is sudo's own: journalctl never ran.

Lab tests: in the sandbox, a user with no sudo rule ran E2's pipeline for one instance, with the password supplied on standard input. sudo printed an error naming the user and the sudoers file, the count was 0, and the status line read `journalctl exit 1, grep exit 1`. On the lab workstation, as a user with no journal access, the system journal read printed nothing on standard output and `No journal files were opened due to insufficient permissions.` on standard error. In the sandbox, `grep -c` on empty input printed 0 and exited 1.

Check your own page: If the reader lacks one piece of access, does your page show an error, or a result that looks like a finding?

14. A loop that overwrites the target

What the page said: E1, E2, and the timeouts check in section 3 looped with `for instance in 1 2 3`, the same name section 6 sets to the instance it restarts and V1 reads.

The case that broke it: V3 reruns E2. After that loop finishes, `instance` holds 3, whichever instance was restarted. A reader who runs V1 again after V3, to recheck the restarted instance, counts instance 3's completions from instance 2's restart time, and a healthy instance 3 passes. In the order the page gives, V1 finishes before V3 starts, so the first V1 is safe; a later rerun is the one that goes wrong.

The fix: the loops count with `worker`. Only section 6 sets `instance`, and V3 says the rerun leaves it alone.

Lab test: in the sandbox (bash 5.2), with `instance` set to 2, a `for instance in 1 2 3` loop left it at 3; a `for worker in 1 2 3` loop left it at 2.

Check your own page: Can a later check overwrite a value that names what the action changed?

Claims the lab disproved

These came from reviewers. Each was run, and none held in the conditions named.

- GNU find rounds `-mmin` ages down to whole minutes, so `-mmin +10` needs 11 minutes. In the tests, a file 10.2 minutes old matched `-mmin +10` on findutils 4.9.0, one 10.5 minutes old matched on 4.10, and files 15.2 and 15.5 minutes old matched `-mmin +15`.
- find has no `%Cz` directive and prints a stray letter. In both test environments it printed a numeric zone: `-0700` on the lab workstation, `+0000` in the sandbox.
- `df -h` wraps a long filesystem name onto a second line, so E4's two-line description fails. On GNU coreutils 9.4, a filesystem with a 70-character source name printed a header and one line, the same as `df -hP`.
- A wrong instance number exits 0 and does nothing. Restarting the unused @4 instance started it (entry 12).

When a review claims what a command prints, ask for the command, the version and environment it ran in, and what it printed, or for the manual page that says so. Without one of those, the claim is a question for the author to test, in the environment the page names.

8. The template, to write on

These pages repeat the template one section at a time, with a ruled space to write in under each. Fill them in by hand while you work out a runbook, then type the result into the page your team keeps. The template in the article is the copy to paste from; these pages are for thinking on paper.

Leave a bracket in place when you cannot fill it yet. While you are filling the page in, its status is Designing; the lifecycle table under "Words used in this kit" says what moves it on from there.

Header

```
Header lines (work out Applies to and Out of scope first; the rest can wait):

[RB-identifier]  [Verb] [one thing] on [one system]

Owner:         [role that keeps this page current]; service owner: [role]
Status:        [Cataloged | Designing | Ready to test | Pilot | Operational | Retired],
               page revision [n]; last tested by someone other than the
               author: [not yet | who, where, YYYY-MM-DD UTC]
Next review:   [YYYY-MM-DD], or after any change to [service]
Lives at:      [repository and path of the authoritative copy]
Applies to:    [servers, service, instances, versions]
Out of scope:  [cases that look similar and need something else, with
               where to go instead]
```

Your draft

1. Objective

```
[The end state, written as something a command or a dashboard can show.]
```

Your draft

2. Trigger

```
[The alert or symptom that sends a reader here, with its threshold.]
```

Your draft

3. Authority and prerequisites

```
- Who may run this without asking, and what needs approval: [ ]
- Access needed, check by check (sudo rule or group): [ ]
- The shell, and the settings its checks need (errexit off, say): [ ]
- What the action does to work in progress, and how to confirm it: [ ]
- Where to record commands and output: [ ]
```

Your draft

4. Evidence (read-only; run all of it before deciding, unless a stop applies)

```
A check ran without an error when it printed each line its text says
it always prints, every other line it printed is one its text
describes, and its result as a whole, exit statuses included, is one
listed there. An abnormal result is a finding. Anything else, a
missing line included: Branch C.
[The checks below show the known cause this page fixes is present.]
E1  [What it checks]
    [command; after a pipeline, print each exit status by name;]
    Prints: [the lines it always prints; each other kind it can print]
    Normal: [ ]  Abnormal: [ ]
E2  [What it checks]
    [command; after a pipeline, print each exit status by name;]
    Prints: [the lines it always prints; each other kind it can print]
    Normal: [ ]  Abnormal: [ ]
```

Your draft

5. Decision path

```
Branch A: [name of the bounded case]
  If every check ran without an error,
    and [observable condition],
    and [scope or risk condition],
    then go to section 6.
```

```
Branch B: [shared or larger problem]
  Otherwise, if every check ran without an error,
    and at least one of these holds:
      [condition];
      [condition];
    then escalate with [evidence]. Make no change.
Branch C: anything else
  Otherwise, escalate with the evidence collected. Make no change.
Stop at any point if [harm condition]; escalate with what you have.
```

Your draft

6. Action ([branch] only)

```
[Checks to run again right before the action, because their results
 can change; how the page keeps other writers or runs off meanwhile;
 and what to do if the branch no longer holds.]
Set [value the reader chooses] first.
    [variable=value;]
    [the action: one command, or a set sequence of numbered sub-steps
     with a check after each;]
Expected: [what must print, and within how long]. If a line it must
print is missing, anything else prints, or it runs past that: stop.
Run only the undo section 7 gives for that failure, and none while the
action may still be running. Make no second attempt; escalate.
```

Your draft

7. Rollback

```
[How to undo each step that changed something: what to save before
 it, which failures permit the undo, who may do it, and the check
 that shows the undo worked.]
[Or: This cannot be undone. Section 5 limits it by: (conditions), and
 a check in section 3 or 4 proves that (what must already be true,
 such as a backup restored in a test, or the old primary fenced so it
 cannot accept writes) holds. A file listing falls short.]
```

Your draft

8. Verification

```
V1  [Check of one part of section 1, with a time limit]
    [command; after a pipeline, print each exit status by name;]
    Prints: [the lines it always prints; each other kind it can print]
    Pass: [value]. Still pending before the limit: [value].
[V2 and on, until every part of section 1 has a check.]
A pass or a pending result counts only for a check that ran without
an error, by section 4's rule. If verification fails: stop, run only
the undo section 7 gives for that failure, make no second attempt, and
escalate.
```

Your draft

9. Escalation

```
[Role, and how to reach them.] Hand over: [alert time in UTC, evidence
output, each action tried and when in UTC, what it changed or left
unknown, verification results].
```

Your draft

10. Validation history

```
[YYYY-MM-DD]  [environment]  [who ran it]  [pass, partial, or fail]
            Page revision [n], [service version]; case [staged fault,
            or what the real alert showed]; [key results, times in
            UTC]; [page change, or "No page change"]
[YYYY-MM-DD]  [environment]  [approver]  approved  Revision [n] for
            [what it is approved for]; [who watches the first uses]
```

Your draft

When the pages are full

Type the page into the place your team keeps runbooks. Run the author's checklist in section 3 and fix every "no", and write down how to stage the fault (section 3, "Staging a test"). Then hand the page and the record sheet in section 4 to someone who has never run it, and watch without helping.

Sources

- AWS Well-Architected Framework, Operational Excellence Pillar, OPS07-BP03 Use runbooks to perform procedures. https://docs.aws.amazon.com/wellarchitected/latest/operational-excellence-pillar/ops_ready_to_support_use_runbooks.html
- AWS Well-Architected Framework, Operational Excellence Pillar, OPS07-BP04 Use playbooks to investigate issues. https://docs.aws.amazon.com/wellarchitected/latest/operational-excellence-pillar/ops_ready_to_support_use_playbooks.html
- Ansible documentation, Ansible playbooks. https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html
- systemd-system.conf(5), systemd manual, DefaultTimeoutStartSec= and DefaultTimeoutStopSec=. https://man7.org/linux/man-pages/man5/systemd-system.conf.5.html
- systemctl(1), systemd manual. https://man7.org/linux/man-pages/man1/systemctl.1.html

- journalctl(1), systemd manual. https://man7.org/linux/man-pages/man1/journalctl.1.html
- systemd.kill(5), systemd manual. https://man7.org/linux/man-pages/man5/systemd.kill.5.html
- systemd.time(7), systemd manual. https://man7.org/linux/man-pages/man7/systemd.time.7.html
- find(1), GNU findutils (-cmin, -mmin, -printf). https://man7.org/linux/man-pages/man1/find.1.html
- bash(1), the set builtin (errexit, pipefail) and PIPESTATUS. https://man7.org/linux/man-pages/man1/bash.1.html
- ps(1), procps-ng (-p, -o). https://man7.org/linux/man-pages/man1/ps.1.html
- cmp(1), GNU diffutils. https://man7.org/linux/man-pages/man1/cmp.1.html
- cp(1), GNU coreutils (-p). https://man7.org/linux/man-pages/man1/cp.1.html
- stat(1), GNU coreutils (-c). https://man7.org/linux/man-pages/man1/stat.1.html
- ls(1), GNU coreutils (-d). https://man7.org/linux/man-pages/man1/ls.1.html

License

This work is licensed under the Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0): https://creativecommons.org/licenses/by-nc/4.0/

In addition to that license, you may use and adapt this document to write runbooks, labs, and procedures for your own team or employer.

Selling this document, or including it or an adaptation of it in paid training, courses, or products, requires the author's written permission.

Credit as: Your First Runbook: Template Kit, Joseph Tracy, revisualized.com, CC BY-NC 4.0.