DataAgent

REMEDIATION FIRST, THE REST IS NOISE

What Are You Actually Covered For?

A guide to fault coverage. The ways a Kubernetes and AWS estate really fails, which of those failures your current stack can see, where the gaps are, and how to prove coverage instead of claiming it.

CONTENTS

What is in this guide

Ten chapters, in the order you would actually work through the question. Each one answers a question you can ask out loud.

	Where this fits in the product	3
	Summary	4
	How the assessment works, end to end	7
01	The ways an estate actually fails	9
02	Reading your estate, read-only	11
03	Which failures can happen in your stack	12
04	What your past alerts already tell you	14
05	What your own alert rules would catch	16
06	Redundant, single path, or nothing	18
07	The failures nothing would catch	20
08	Proving coverage instead of claiming it	22
09	Covered is not the same as automatic	24
10	How to read a coverage report	25
	The verdict	27
	Appendix: the catalogue	28

WHERE THE NUMBERS COME FROM

Every figure in this guide is **counted** from the published fault catalogue and the classification ruleset that ship with the coverage scanner, or **measured** on a stated sample. Where something is modelled rather than measured, the caption says so. Nothing here is a projection.

WHERE THIS FITS

One product, and the part of it this book is about

> DataAgent is a loop, not a feature. It detects, models what is running and what changed, correlates, reasons to a root cause, decides whether a person must approve the fix, hands the fix to your own pipeline, verifies and learns. This book is about the first stage, and whether it would fire at all.

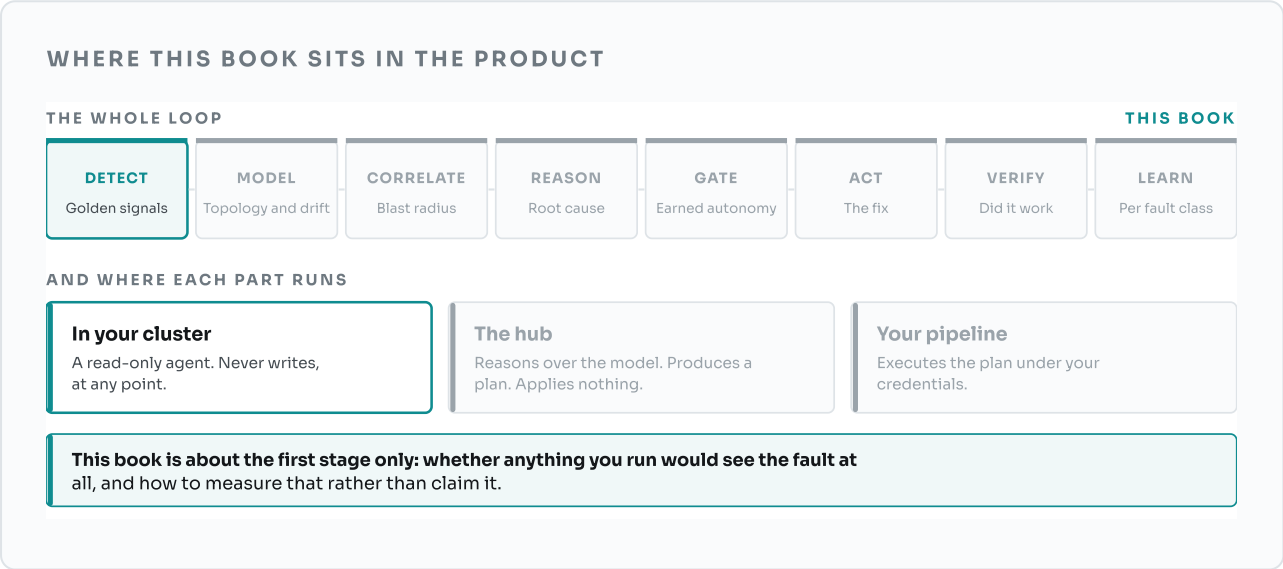


Figure 1. The same drawing appears throughout this series, with a different part lit up. The stages left pale here are not out of scope for the product, only for this book.

The stages to the right of Detect are the subject of the other guides. **Remediation First in Action** is the approval and the fix; **AI SRE for Kubernetes** is the model. This one stops at the question that comes before all of them: would you know?

SUMMARY

Most teams cannot name the failures they are covered for

Alert counts do not answer the question, because one cause produces hundreds of alerts and a cause nobody instrumented produces none. The unit that answers it is the fault class: a named way a system breaks, which you can count, test and prove.

TWO ESTATES, SAME ALERT VOLUME, DIFFERENT COVERAGE

What an alert count tells you

1,400

alerts last month

Which of them were the same cause? How many causes were there? Which failures produced no alert at all, because nothing was watching for them?

The count cannot say.

What the catalogue tells you

11

distinct faults seen, of 32 reproducible faults that can occur

Named, countable, and each either reproducible in a sandbox or not. The 21 that did not occur are the interesting part.

That is a coverage answer.

Figure 2. The left number is the one every team has. The right one is the one that answers the question. [MODEL] Alert volume illustrative; the 32 faults are counted from the published catalogue.

This guide is about getting the right-hand number for your own estate, without installing anything and without taking anyone's word for it.

The shape of the answer

Four questions, in order. Each one narrows the last, and each has a method rather than an opinion behind it.

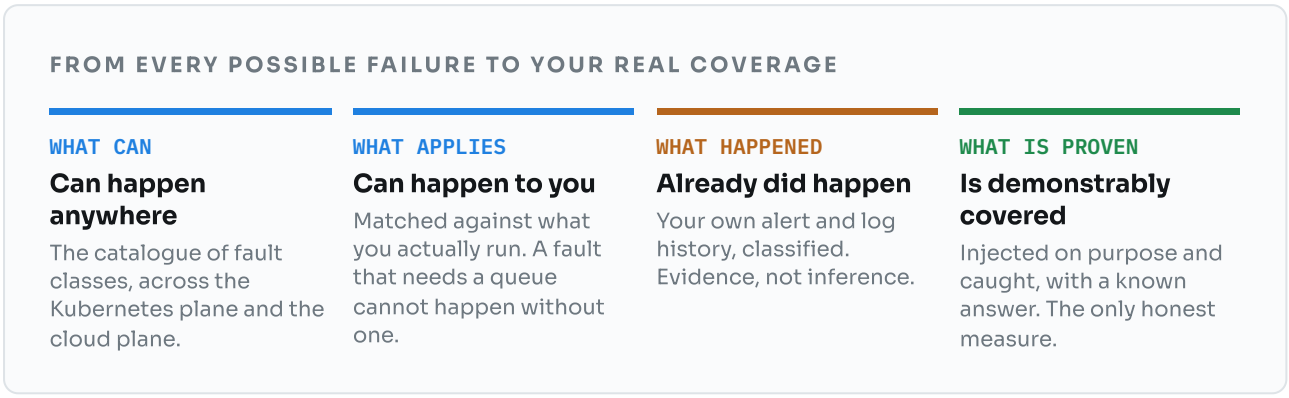


Figure 3. Most tools stop at the second step and call it coverage. The gap between step two and step four is where incidents live.

And one honest admission in the middle

Any scan of your history will find signals it cannot classify. The useful behaviour is to surface them as candidate new fault classes rather than force them into the nearest bucket. Chapter seven is about that.

The verdict, up front

You can answer this question in an afternoon, for free, without giving anyone access to your environment. That is the whole argument and it does not depend on buying anything.

Start here, no credentials

Describe your stack. Get back the failures that apply to it and a cost model. Nothing connects to your environment at all.

Then, read-only

Point it at the observability backend you already run. It issues ordinary read queries, classifies what it finds, and leaves.

Only an abstracted report leaves your environment. No agent, no writes, no changes.

Figure 4. Two modes, and the first needs nothing from you but a description of what you run.

> **Coverage you cannot reproduce is coverage you are taking on trust.**

If you read one chapter, read chapter eight. Everything before it describes what coverage means; chapter eight is the part that separates a measurement from a claim, and it is the question to put to any vendor who tells you they cover your failures.

Five stages, and nothing is written to your estate

> Coverage is a question about your estate, not about a product, so the answer has to be built from your estate's own record. The assessment reads what is already there, keys every finding to a fault class, and reports the difference between what can happen, what has happened, and what your alert rules claim they would catch.

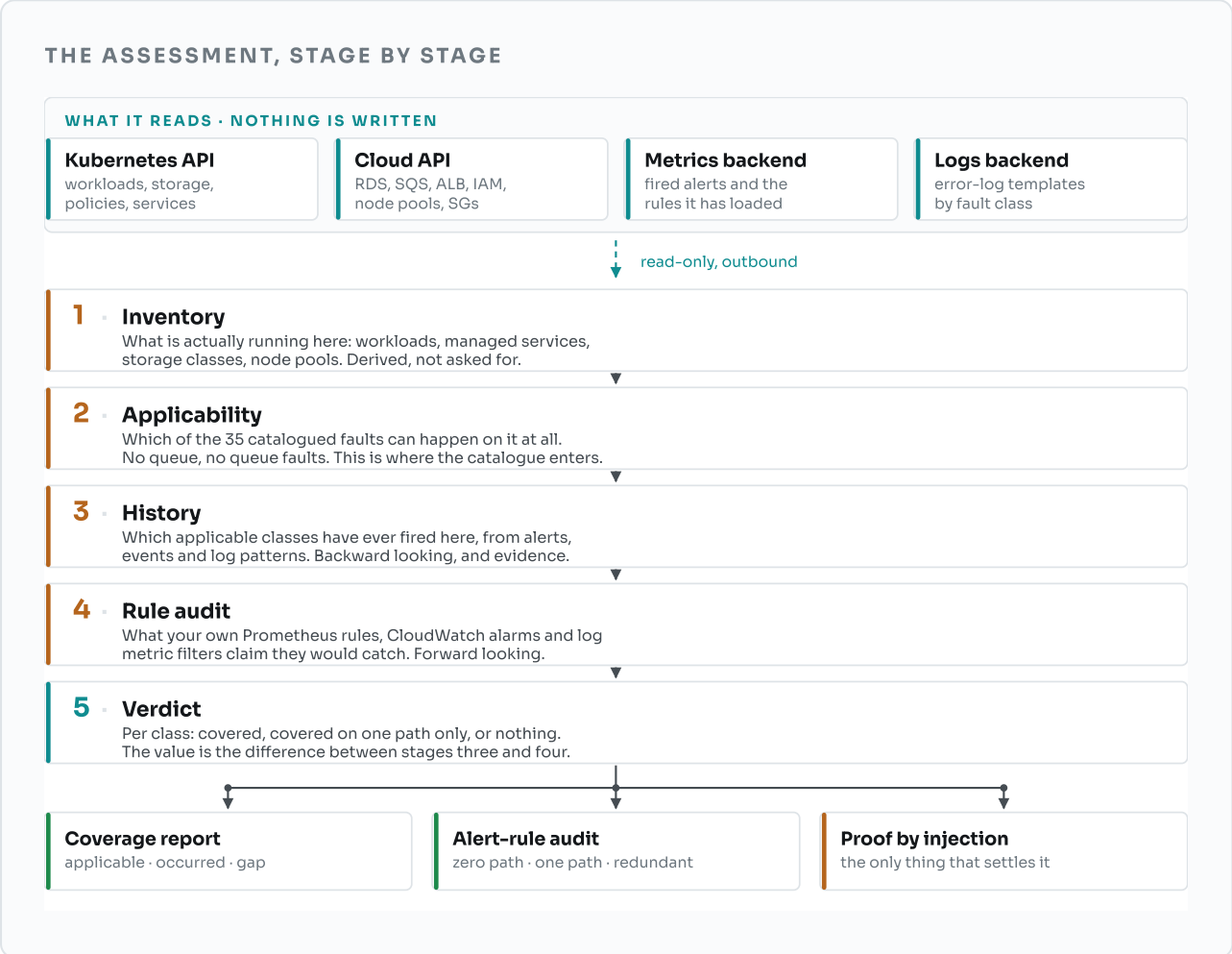


Figure 5. Read-only throughout. Stage two is where the catalogue enters: a fault applies only when every token it requires is present in your stack. Stage five is the only stage that returns a verdict, and it is a verdict about a class, never about a rule.

Stages three and four are the pair that matters. Three is backward looking: what has already broken here. Four is forward looking: what would page someone if it broke tomorrow. Either on its own misleads, because a class that has never fired looks like a gap and a class with a rule against it looks covered. The answer is the difference.

Where each part runs

Three parts, and the boundary between them is the point. A collector runs where your data already is. A coverage engine keys what comes back to the fault catalogue.

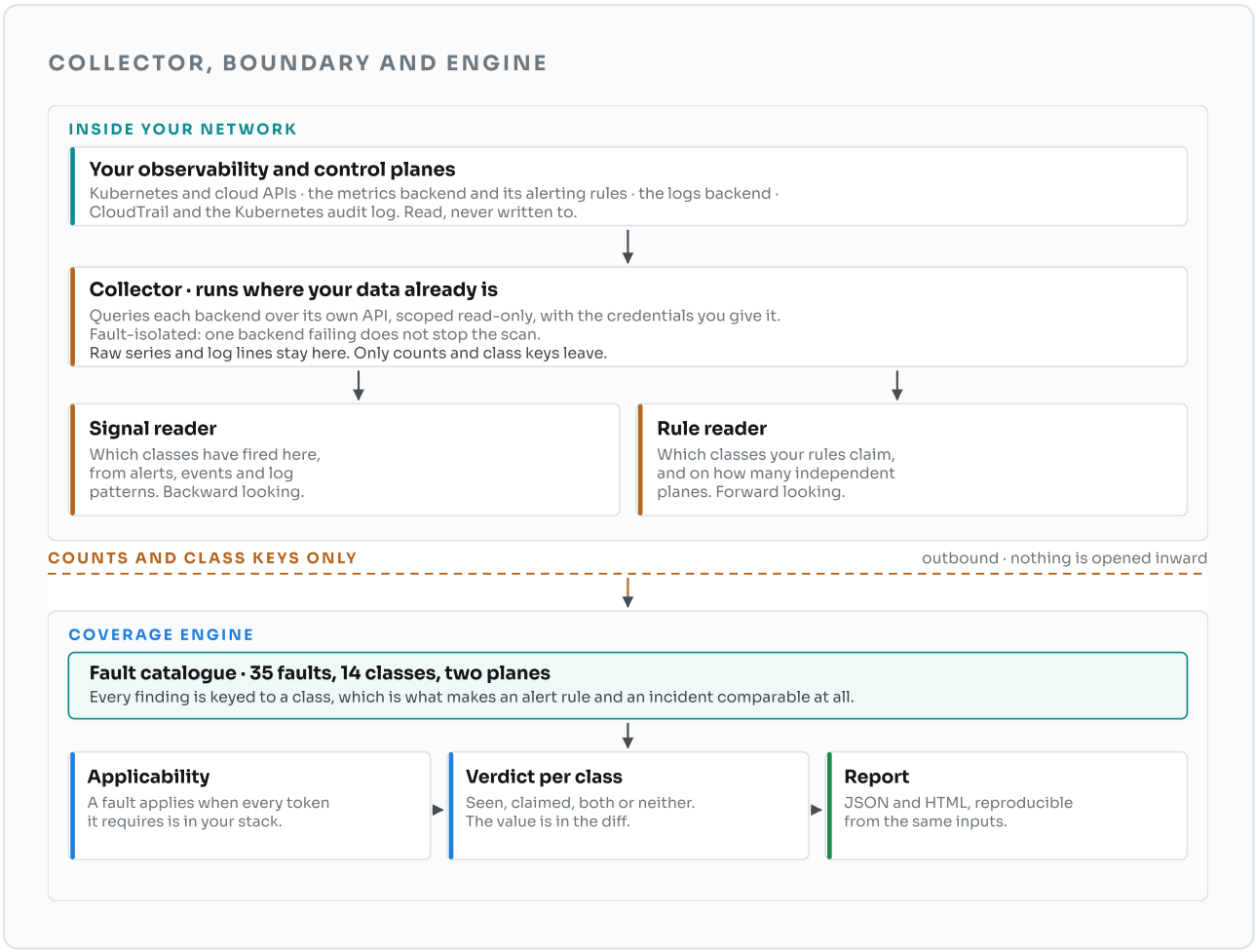


Figure 6. The collector reads and never writes. Raw series and log lines stay inside your network. Only findings leave, outbound, so nothing has to be opened inward.

Why everything is keyed to a class

A rule named **HighDBErrors** and an incident titled **checkout 503s** cannot be compared as strings. Keyed to the same class they can, which is what lets the assessment say a class has fired four times with no rule watching for it, or that five rules claim a class and all five read one log pipeline, which is one detection path rather than five.

WHAT IT DELIBERATELY DOES NOT DO

It installs no agent, writes to none of your systems, and copies your telemetry nowhere. It also does not treat a rule as proof. A rule becomes a verdict only when a fault is injected and the rule fires, which is the last step in Figure 3 and the subject of chapter eight.

The ways a Kubernetes and AWS estate actually fails

> Production fails in a narrow, knowable set of ways, and the list is shorter than most teams expect. The published catalogue holds 32 reproducible faults grouped into 14 classes, split across the Kubernetes plane and the cloud plane.

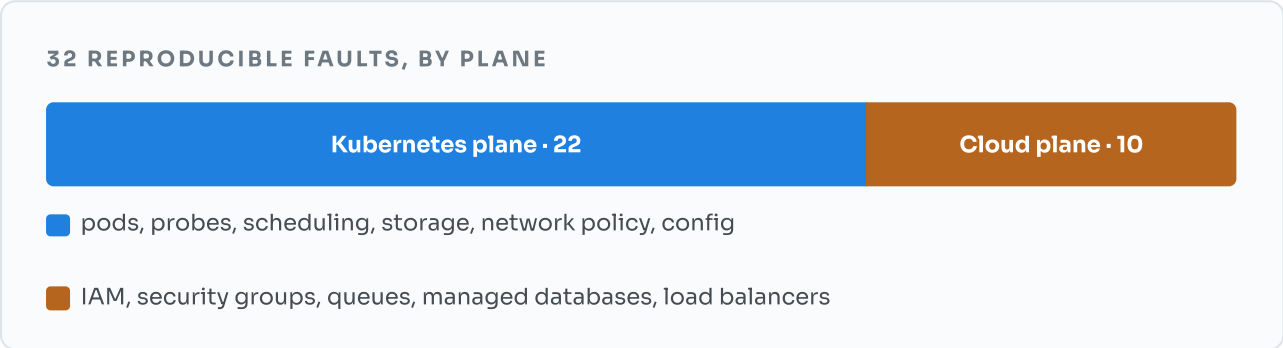


Figure 7. Counted from the published catalogue. Roughly a third of what breaks a Kubernetes application is not in Kubernetes at all, which is the first reason cluster-only monitoring misses things.

The classes, by how many faults sit under them

FAULT CLASS	FAULTS	PLANE	EXAMPLE
Dependency unavailable	7	both	security group blocks the database, DNS fails, managed database fails over
Resource exhaustion	5	Kubernetes	CPU throttling, heap exhaustion, connection pool exhausted, autoscaler stuck
IAM permission drift	5	cloud	a policy change revokes a permission a service depended on
Crash loop	3	Kubernetes	image that does not exist, probe that fails, init container that cannot finish
Storage failure	2	Kubernetes	volume never binds, stateful pod stuck on its volume
Service failure	2	both	service with zero ready endpoints, queue filling with poison messages

plus eight single-fault classes, listed in the appendix

Why the class matters more than the alert

One failing dependency produces alerts on every service that touches it. Count alerts and you get a number that tracks how wide your estate is, not how often it breaks. Count fault classes and you get something that tracks reality: one cause, counted once, whatever it touched on the way out.

> **A fault class is the unit that survives the alert storm.**

And it is the unit everything else attaches to

- **Detection** is per class. A detector either catches that shape of failure or it does not, and you can test which.
- **Root cause** is per class. The evidence that resolves a crash loop is not the evidence that resolves an IAM denial.
- **Remediation** is per class. The fix for a bad image is a rollback; there is no generic fix.
- **Trust** is per class, and this is the one teams skip. Chapter nine explains why.

So coverage is not one number. It is a column: for each class that can happen to you, can it be detected, explained, fixed, and has that been shown rather than asserted.

QUESTIONS THIS PAGE ANSWERS

Why not just count alerts?

Because alert volume measures the width of your estate. One failing database can raise hundreds of alerts across every service that depends on it, and a failure nobody instrumented raises none at all.

Is more coverage always better?

Not if it is unproven. A long list of claimed fault classes is worth less than a shorter list where each one has been injected and caught.

How to find out what is running, read-only

> You cannot say which failures apply until you know what you run, and most teams do not have that written down. A read-only scan derives it: platform, cluster size, runtimes and the cloud services actually in use, without installing an agent.

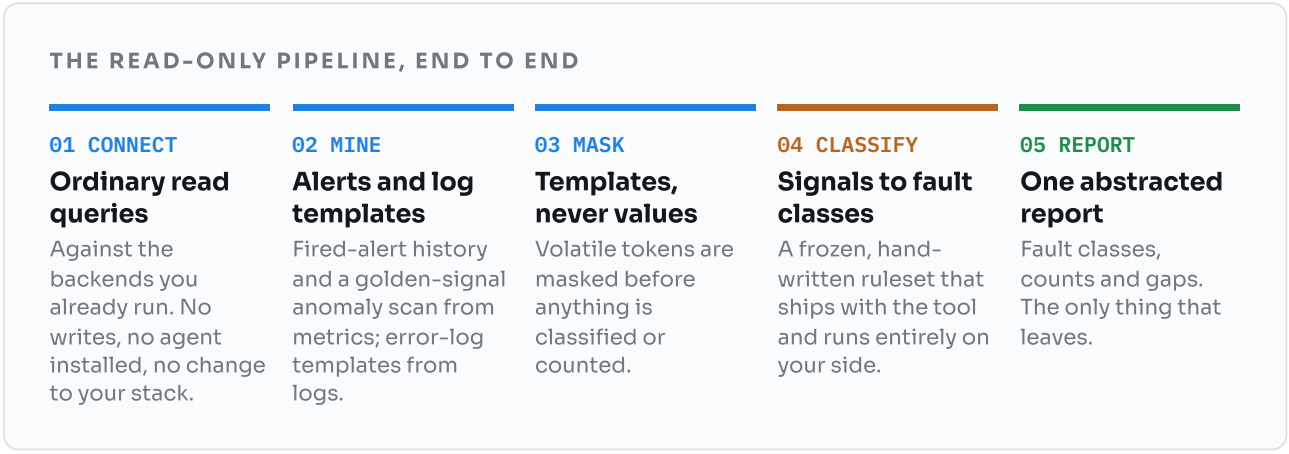


Figure 8. Each stage narrows raw telemetry towards a classification. The direction of travel is important: nothing is gathered up and shipped out for analysis somewhere else.

What it derives without being told

Platform and cluster size, down to nodes, pods and namespaces with per-namespace minimum, average and maximum. Runtimes in use. Which cloud services are actually attached, rather than which ones someone once wrote into a diagram. That inventory is what chapter three matches against.

QUESTIONS THIS PAGE ANSWERS

Does this need access to my cluster?

No. It reads the observability backend you already run, with ordinary read queries. There is also a mode that needs no credentials at all, covered in chapter three.

What leaves my environment?

Only findings leave, in one abstracted report: fault classes, counts and gaps. Raw telemetry stays where it was born, and log values are masked before anything is counted.

Do I have to install an agent?

No. The scan runs as a tool inside your environment. It opens nothing inward, and only the abstracted report goes out.

Which failures can actually happen in your stack

A fault that needs a message queue cannot happen to you if you do not run one, so the catalogue has to be filtered against your real inventory. You can get this list with no credentials at all, by describing your stack rather than connecting anything.

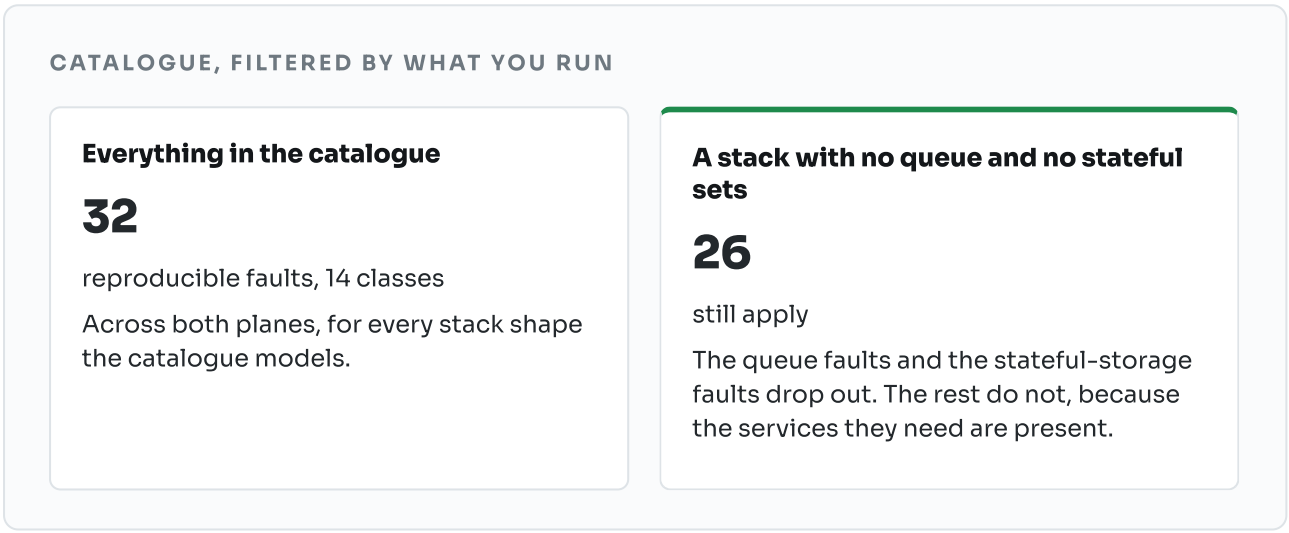


Figure 9. [MODEL] Illustrative filtering on a stack without queues or stateful sets, using the real catalogue. The point is the method, not the number: your own figure comes from your own inventory.

This is the cheapest useful artefact in the whole exercise

Most teams have never seen the list written down. Not the alerts they happen to have configured, but the failures that are possible given what they run.

Each fault on the list has its own page with the exact error text as the tooling prints it, the symptoms, what usually changed, the manual fix and how to verify it. Those pages are useful whether or not you ever automate anything.

QUESTIONS THIS PAGE ANSWERS

Can I get this without connecting anything?

Yes. Describe your stack and the applicable faults come back with a cost model. Nothing touches your environment.

How is a fault decided to apply or not?

By whether the resources it needs are present. A queue consumer stall needs a queue; a stateful volume fault needs a stateful set.

Where do I see the detail for one fault?

Each applicable fault links to its own page, which carries the exact error text, the cause, the manual fix and the verification step.

What your past alerts already tell you

> Your own alert and log history is the strongest evidence of what breaks in your estate, and almost nobody mines it. Classified against the catalogue, it turns a year of noise into a ranked list of the failures you actually have.

THE LIFE OF ONE LOG LINE

MASKED, GROUPED, THEN MATCHED

```
raw      2026-07-20 10:10:03 UTC:10.0.11.23(46156):...[3735]:LOG: duration: 2645.169 ms
mask     <ts> UTC:<ip>(<n>):...[<n>]:LOG: duration: <n>.<n> ms
group    3,593 identical lines collapse into 1 template
match    ordered rules, first match wins, anchored on the :LOG: prefix
result   slow query  ·  confidence high  ·  reproducible
```

Figure 10. Values that change become placeholders, so thousands of lines fold into a handful of shapes. The masking is what makes counting meaningful.

Masking beats clustering, and it was measured

99%

of lines classified by masking plus ordered rules, into 16 log-pattern classes

90%

by a standard clustering algorithm on the same lines, with one wrong class

3,800

real error lines in the benchmark

Measured on 3,800 real error lines. The clustering algorithm lost because its wildcards erased the exact tokens the rules key on, including the database log prefixes that tell a slow query apart from an out-of-memory kill. Clustering can shorten a report after classification. It should not do the classifying.

QUESTIONS THIS PAGE ANSWERS

Does this read my log contents?

It reads them to build templates, in your environment, and the values are masked before anything is counted. Raw log lines stay where they are; only findings reach the report.

Why rules instead of machine learning?

Because every mapping is readable and reviewable, and because on a measured benchmark it classified 99% of lines against 90% for clustering, without inventing a wrong class.

What about alerts rather than logs?

Fired-alert names are matched the same way, by exact name and by suffix patterns for generated per-service alerts, alongside a golden-signal anomaly scan.

What your own alert rules would actually catch

Which classes have never fired here and what would page someone if one fired tomorrow are two different questions. The second is answerable today, read-only, from artefacts every estate already has: Prometheus rules, CloudWatch alarms and log metric filters.



Figure 11. [MODEL] One estate, 13 applicable classes, illustrative split. The value is in the diff, not in either column. Neither is proof until a fault is injected.

Read the red band first. A class with no rule and nothing observed is not invisible, because the cluster's own record would very likely still show it. It means nothing of yours is watching for it specifically, so nobody is woken up.

QUESTIONS THIS PAGE ANSWERS

Does an alert rule mean the failure is covered?

No. A rule can carry the wrong threshold or the wrong labels and look healthy until the day it fails to wake anyone.

Where do the rules come from?

Prometheus rule definitions and the Prometheus API, CloudWatch alarms, and log metric filters. All read-only, all artefacts you already have.

What if a rule cannot be classified?

It lands in an explicit unmapped bucket and is counted and listed, never guessed into a class and never silently dropped.

Every mapping says where it came from

A rule is keyed to a fault class in one of three ways, and which one is recorded against the row rather than hidden behind a single confidence score.

LABELLED

The rule says so

It carries an explicit fault-class label. Authoritative, no inference involved.

MAPPED

Its metric is known

The expression matches a curated signal map that is versioned alongside the catalogue.

INFERRED

Read from the text

Classified from the rule name, expression and annotations, and marked as inferred on the row.

UNMAPPED

Listed, not hidden

Counted and shown with what it looks like, so you can label it in one pass and the next run stops guessing.

Figure 12. The fastest way to make the audit more accurate is to label a rule with its fault class, because a labelled rule overrides every inference.

> **A rule existing is a hypothesis. It is not detection until something breaks and it fires.**

Redundant, single path, or nothing

Count independent detection paths per fault class, not rules. Five keyword rules on the same log pipeline are one path, because one dead pipeline removes all five.

THREE VERDICTS, AND TWO FLAGS THAT RIDE ALONGSIDE

ZERO PATH

Nothing covers it

Applicable here, and no rule anywhere claims it. The strongest finding in the whole audit.

SINGLE PATH

One plane only

One path, or several that all sit on the same signal plane. Plane diversity is what counts, not rule count.

REDUNDANT

Two or more planes

Not a defect. It is also where consolidation savings sit.

Flag: fails open

No staleness guard. The rule compares a number against a limit with nothing to catch the number not arriving at all.

If the thing sending it dies, the alert stays green instead of firing. Ranked as severe as a missing rule when the class is single path.

Flag: unproven

No injection has demonstrated the class is caught here.

Combined with single path, this is the top of the queue for what to break on purpose next.

Figure 13. The two flags matter more than the verdict. A redundant class whose paths all fail open is worse than a single-path class that pages on a dead feed.

QUESTIONS THIS PAGE ANSWERS

Why count paths instead of rules?

Because rules on the same plane share a failure mode. Five log-keyword rules all stop working when one log pipeline dies, so they are one path, not five.

What does fails open mean?

The rule has no guard for its signal disappearing. It compares a number to a threshold, so if the exporter dies the number stops arriving and the alert stays green.

Is redundancy a problem?

Not for reliability. It is where consolidation savings live, which is why the same audit feeds the cost view.

What one pass found on a real estate

A hand-run comparison of one team's own database rules against the catalogue, in a single pass, with no incident, no agent install and no write access. **Audited**, August 2026.

- **One class with three independent paths.** Connection exhaustion was covered by a cloud alarm, a Prometheus rule and an application log keyword. Triple redundancy on one class while others had none.
- **Two classes hanging on a single path.** Database failover and a blocked security group were detectable only from application log keywords. No replica-lag, uptime or failover metric existed anywhere, so one empty log group took both to zero detection.
- **Rules that fail open.** The Prometheus rules had no staleness guard, so a dead exporter or a revoked permission would silently stop all three from ever firing. The cloud alarm sitting next to them was configured to page on a missing feed. Same estate, opposite failure semantics.
- **A discovery scope that excluded the estate itself.** An exporter discovered targets by a tag that existed nowhere in that account, so its database coverage applied to none of its own databases.

> **None of these were visible from either inventory alone.**

That last point is the argument for doing this at all. The rule list looks complete. The incident history looks quiet. The gap only appears when you put them side by side.

07 · THE GAPS

The failures nothing in your stack would catch

> The valuable output of a coverage scan is not the list of things it found, it is the list of things nothing would have told you about. A scan that cannot admit a gap is marketing.

THREE HONEST OUTCOMES, AND THE RULE BEHIND THEM

MATCHED

A known fault class

With the evidence recorded, so you can see why it was classified that way.

LOW CONFIDENCE

Flagged for review

The rule was ambiguous or broad. A bare timeout also appears in healthy systems, so it is marked, not asserted.

NEW

A real fault, not modelled

Surfaced as a candidate class in its own right, never forced into the nearest bucket.

Figure 14. Anything matching no rule is reported as a candidate new fault class rather than dropped. A real failure nobody has modelled should never be hidden by the thing measuring coverage.

And the quality of the scan itself is reported

A malformed log line, a strange alert key or an odd API response is counted as unparsed and shown, not silently skipped. The report carries a scan-quality figure and a fragmentation figure, which is the share of templates that appeared exactly once. High fragmentation means the masking did not fold your log shapes well, and you should trust the counts less.

> **A coverage number without a scan-quality number beside it is an opinion.**

QUESTIONS THIS PAGE ANSWERS

What happens to signals that match nothing?

They are surfaced as candidate new fault classes with their evidence, so a real failure that nobody has modelled yet shows up as a gap rather than disappearing.

How do I know the scan itself worked?

The report carries a scan-quality strip and a fragmentation figure. Unparsed records are counted and shown rather than dropped.

Are the classifications authoritative?

No, and they do not claim to be. Every mapping is a reviewable suggestion with its evidence attached, and ambiguous ones are explicitly marked low confidence.

How to prove coverage instead of claiming it

> Coverage is only real if you can reproduce the failure and watch the system catch it. The honest measure is how many faults have been injected on purpose, with an answer known in advance, and caught.

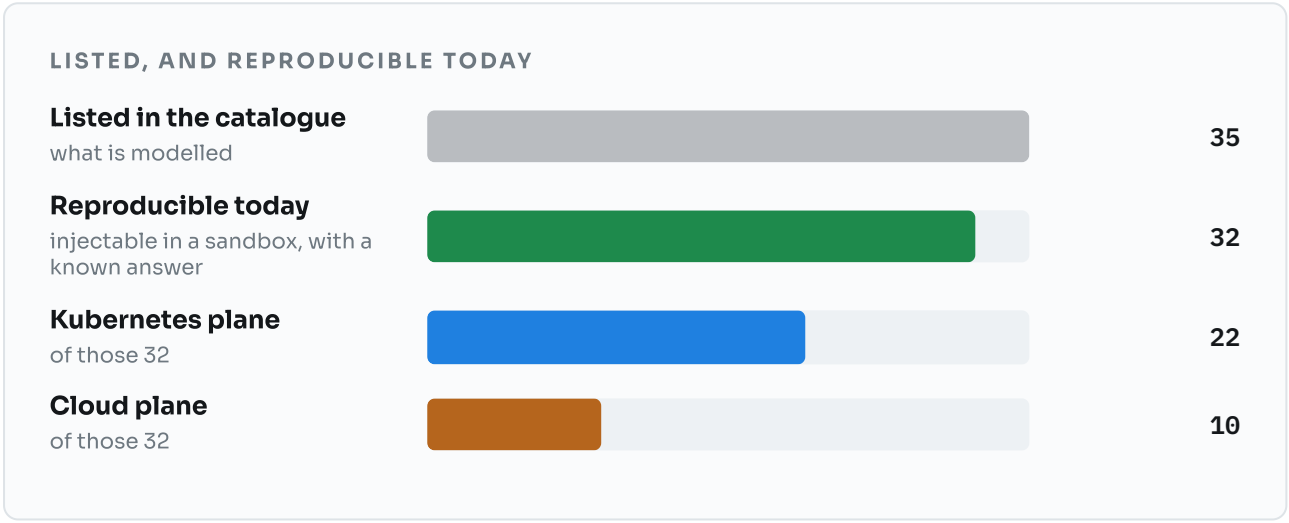


Figure 15. Counted from the injectable catalogue as it stands today. The gap between what is listed and what is reproducible is stated on purpose: a fault that cannot yet be injected is not claimed as proven.

What reproducing a fault actually involves

Inject a known fault into a live, resettable target. Run the whole detection and remediation path against it. Verify, then revert. Score the result against the answer you held before the run started, because you caused the fault.

That score is a measurement rather than a judgement, which is the distinction that matters. A model grading another model produces a number that moves; a fault with a known cause produces a number that means something.

> **If a vendor says they cover your failures, ask which ones they inject, and what the answer key is.**

QUESTIONS THIS PAGE ANSWERS

How many faults can actually be reproduced today?

32, across 22 in the Kubernetes plane and 10 in the cloud plane. That is counted from the injectable catalogue, not projected.

Why does injecting faults matter?

Because it is the only way to know the answer in advance. Without a known answer you are measuring agreement, not accuracy.

What about faults that cannot be injected yet?

They stay listed as modelled and are not counted as proven. The difference is reported rather than smoothed over.

09 · EARNING AUTONOMY

Covered is not the same as automatic

> A fault class being covered means it can be detected and explained. Letting a fix run without waiting for a person is a separate decision, made per class, on a record you can read.



Figure 16. Finer steps inside the three autonomy stages: recommend only, approve first, autonomous. Most conversations jump from step one to step five. The steps in the middle are where the trust is built.

Why the record is kept per class

Competence on one failure shape says nothing about another. A system that has rolled back two hundred verified bad deployments has earned something on bad deployments and nothing at all on database failover. Keeping one global autonomy switch is how teams end up turning the whole thing off after a single bad experience.

So the useful question to ask about any autonomous claim is not whether it can act. It is which classes it is allowed to act on, what the record is for each, and what happens when it is not confident.

QUESTIONS THIS PAGE ANSWERS

Does coverage mean it will fix things automatically?

No. Coverage means a failure can be detected and explained. Autonomy is earned per fault class: recommend only, then approve first, then autonomous, on a verified record. Even then, the change runs in your repository, on your runner, under your credentials.

Why per class rather than one setting?

Because competence does not transfer between failure shapes.

What happens when it is not confident?

It recommends and waits for a person, and says what it is unsure about. A system that cannot decline is not safe, it is lucky.

How to read a fault-coverage report

> The report has five parts and each answers a different question. Read them in order, but the gaps are the part that should change what you do next.

WHAT EACH PART OF THE REPORT IS FOR

PART	THE QUESTION IT ANSWERS	WHAT TO LOOK FOR
Observed stack	What do I actually run	Cluster size, runtimes, cloud services. Check it against what you believe you run.
Applicable faults	What could happen here	The filtered catalogue for your stack.
Observed faults	What already did happen	Ranked by evidence, each with its reproducibility rating and confidence.
Gaps	What would nothing have caught	Candidate new classes and unclassified residual. The most valuable page.
Scan quality	How much should I trust this	Unparsed share and fragmentation. High fragmentation means softer numbers.

Figure 17. Read gaps before observed faults. What you were not watching for is more actionable than what you already alerted on.

Two numbers that are easy to misread

A high observed-fault count is not bad news. It usually means your logging is good enough to classify. An estate with almost nothing observed and high fragmentation is the one to worry about, because it means the scan could not see clearly.

A low-confidence mapping is not a wrong mapping. It means the rule that matched is broad or ambiguous, and the evidence is attached so you can judge it yourself. Treat those as a review queue rather than a defect list.

QUESTIONS THIS PAGE ANSWERS

Which part of the report should I read first?

The gaps. What nothing would have caught is more actionable than what you already have alerts for.

Is a high fault count a bad sign?

Usually the opposite. It means the scan could classify what it saw. Low counts with high fragmentation mean the scan could not see clearly.

What do I do with low-confidence mappings?

Treat them as a review queue. The evidence is attached to each one so you can confirm or reject it yourself.

THE VERDICT

Get the list, then ask for the proof

Two steps, in order, and the first one costs you nothing and commits you to nothing.

First, get the list

Describe your stack and get back the failures that apply to it. No credentials, no access, no agent. It is useful on its own: it tells you which failures are possible. Read it against your current alerting and the gaps will be obvious.

Then, audit the rules you already have

Read your own alert rules against the same classes. It is read-only, it needs no incident, and it answers a question the incident history cannot: if one of these fired tomorrow, would anything page a human. Expect to find a class with three overlapping paths next to one with none, and rules that stay green when the thing feeding them dies.

Then, ask anyone selling you coverage for the proof

Not the list of fault classes they support. The list they **inject**, and the answer key they score against. A catalogue is a claim. A fault reproduced in a sandbox with a known cause, caught, fixed and verified, is a measurement. The difference is the entire subject of this guide.

And keep the two decisions separate

Covered is not automatic. Decide coverage first, per class, on evidence. Decide autonomy later, per class, on a record of verified fixes. Teams that collapse those two decisions into one end up trusting too much too early.

APPENDIX

The catalogue, as it stands today

32 reproducible faults in 14 classes. Counted from the published catalogue, not projected. Each one can be injected into a live, resettable target and replayed.

FAULT CLASS	N	K8S	CLOUD	WHAT SITS UNDER IT
Dependency unavailable	7	4	3	egress blocked, security group blocks the database, managed database failover, DNS failure
Resource exhaustion	5	5	0	CPU throttling, autoscaler cannot scale, connection pool exhausted, heap exhaustion
IAM permission drift	5	0	5	object storage denied, secrets access denied, queue read denied, node provisioner denied
Crash loop	3	3	0	image that does not exist, failing probe, init container failure
Storage failure	2	2	0	volume never binds, stateful pod stuck on its volume
Service failure	2	1	1	service with zero ready endpoints, queue filling with poison messages
Network partition	1	1	0	network policy blocks traffic between services
Scheduling failure	1	1	0	pod unschedulable for want of CPU
Bad deployment	1	1	0	disruption budget blocks the rollout
Config drift	1	1	0	config map changed out from under a running workload
RBAC denial	1	1	0	a role loses a permission a controller depended on
Certificate failure	1	1	0	certificate expires
External service failure	1	0	1	load balancer targets unhealthy
Capacity exhaustion	1	1	0	node provisioner cannot get capacity

How to read that table

Ten of the thirty-two sit in the cloud plane, in IAM, security groups, queues, managed databases and load balancers. None of those are visible to monitoring that stops at the cluster edge, and one of them, IAM permission drift, is tied for the largest class after dependency failures.

The two classes that appear in both planes are the ones worth reading twice. A dependency failure can originate in a network policy inside the cluster or in a security group outside it, and the symptom a service sees is identical. That is exactly the case where cluster-only tooling sends you looking in the wrong place.

> **A third of what breaks a Kubernetes application is not in Kubernetes.**

DataAgent

REMEDIATION FIRST, THE REST IS NOISE

Find out for free.

Start with a free, read-only assessment. It answers two questions: which failure classes you are covered for, and what your observability bill could right-size down to. Describe your stack with no credentials, then connect read-only when you are ready. Only an abstracted report ever leaves.

Also in this series: **DataAgent Autonomous SRE**, **Right-Sizing Your Observability**, **DataAgent Live Demo**, **Choosing an AI SRE for Kubernetes**, **AI SRE for Kubernetes** and **Remediation First in Action**.