OWASP TOP 10 FOR AGENTIC APPLICATIONS · 2026 · FACILITATOR GUIDE

Cascading Agent Failures

In a web of connected agents, one small failure doesn't stay small — it ripples outward until the whole system is down.

ASI08:2026 | SEVERITY: ELEVATED | RISK 08 OF 10

WHAT THIS DOCUMENT IS

The companion to the Cascading Agent Failures slide deck. Each section maps to a slide and gives you the background to explain it, worked examples, and links to go deeper. Present from the slides; read this to answer the questions the slides provoke.

HOW EXAMPLES ARE LABELLED

Attack traces in this guide are **ILLUSTRATIVE RECONSTRUCTIONS** written to teach the mechanism — they are not transcripts of real incidents. Real events are collected in **DOCUMENTED INCIDENTS AND RESEARCH**, each with a verified source link and a label showing whether it was a confirmed incident, a disclosed vulnerability, a researcher demonstration, or a research paper. The distinction is deliberate: do not present an illustration as a breach.

AUDIENCE

Employees and developers who build, ship, or work alongside AI agents. Developer-specific material is marked; everything else is for everyone.

01 · Executive summary

The only risk on the list that needs no attacker at all. Ordinary bugs, plus tight coupling and well-intentioned retries, are sufficient to take down an entire agent fleet — which is why this is an availability and safety problem as much as a security one.

> **The risk in one sentence.** Small agent failures propagate through connected systems and cause large-scale impact — errors spreading through tool sequences, one agent's failure taking down its dependents, resource depletion spreading, or security failures propagating along trust relationships.

By the end of this module the audience should be able to:

- Explain how local failures become system-wide outages.
- Recognise retry storms and resource-depletion cascades.
- Apply circuit breakers, bulkheads, and fallbacks.
- Test blast radius by failing one component on purpose.

02 · Background — why this risk exists

SLIDES 4–6

Distributed systems have known these dynamics for years: retry storms, thundering herds, and metastable failure, where a system stays broken after the original trigger is gone because the retries themselves sustain the load. Agent systems import all of it and add two accelerants — agents retry on semantic failures (a malformed answer, an unsatisfying result), not just transport errors, and they fan out to many tools at once. The classic finance analogues, the 2012 Knight Capital loss and the 2010 Flash Crash, are worth citing because they show automation converting a small fault into a systemic one at machine speed.

Why it matters now

Orchestration frameworks make fan-out trivial and safety unglamorous. Defaults are commonly unbounded retries, no circuit breaker, and a shared tool endpoint for every agent — three ingredients that reliably convert one bad output into a fleet-wide stall.

What changes when the system is agentic

Classic LLM or application	Agentic system
A crashed service returns errors until it restarts, affecting its own callers.	Agents retry, fan out, and depend on shared tools, so one failure triggers retry storms and resource exhaustion that cascade across the whole orchestration.

03 · Explaining it to a non-technical audience

SLIDE 5 — THE ANALOGY

One tripped breaker, the whole block dark. A single overloaded circuit trips, so its neighbours pick up the load, overload, and trip too. Without isolation, one fault marches across the grid until the entire block goes dark — each step 'helping,' each step spreading it.

In the analogy	Maps to	Why it works
THE BREAKER	**A failing agent**	One component errors, stalls, or floods.
THE GRID	**Connected fleet**	Dependents retry and absorb load, spreading the fault.
THE FIX	**Isolate the fault**	Circuit breakers and bulkheads stop one failure from taking the rest.

> PRESENTER TIP
> Use the tripped-breaker analogy, then note that every step in a cascade is a component 'helping'. Cascades are made of reasonable local decisions.

04 · How the attack unfolds

SLIDE 7 — THE KILL CHAIN

Walk the stages in order. The teaching point is that no single stage looks alarming on its own; the compromise is the sequence.

Stage	What happens
01 · FAIL	One agent errors, stalls, or emits bad output.
02 · RETRY	Dependents retry aggressively, adding load.
03 · SATURATE	Shared tools and queues fill and slow.
04 · SPREAD	More agents time out and fail in turn.
05 · COLLAPSE	The orchestration wedges or goes down.

05 · Attack vector 1 — Error propagation through tool sequences

SLIDE 8

HOW IT WORKS A bad output from one step is consumed by the next, corrupting or crashing every downstream agent in the chain.

What it looks like in practice:

- Malformed JSON breaks the parser.
- Bad data poisons later computations.
- No validation between steps.

Illustrative trace — a reconstruction of the mechanism, not a real transcript:

```
agentA ▸ returns { malformed
agentB ▸ parse error → retry
agentC,D depend on B → stalled
✗ chain-wide failure
```

06 · Attack vector 2 — Retry storms & resource depletion

SLIDE 9

HOW IT WORKS Every agent retries the same failing dependency at once, multiplying calls until the shared resource is exhausted.

What it looks like in practice:

- Synchronised retries with no backoff.
- Shared API rate-limit blown.
- Connection/thread pool exhausted.

Illustrative trace — a reconstruction of the mechanism, not a real transcript:

```
dep down → 20 agents retry ×5
→ 100s of calls/sec
shared API: 429 · pool exhausted
✗ resource depletion cascade
```

07 · Attack vector 3 — Trust-relationship propagation

SLIDE 10

HOW IT WORKS A compromised or failing agent is still trusted by its peers, so its bad state or actions flow along the trust graph.

What it looks like in practice:

- Peers accept a failing agent's output.
- A security failure spreads via trust.
- No isolation between trust zones.

Illustrative trace — a reconstruction of the mechanism, not a real transcript:

```
agentX compromised (still trusted)
peers ▸ accept its results
bad state spreads across zone
✗ failure follows the trust graph
```

08 · Documented incidents and research

USE THESE WHEN SOMEONE ASKS "HAS THIS ACTUALLY HAPPENED?"

Every item below was verified against the linked source. Use the labels to be precise about what each one is — overstating a demonstration as a breach damages your credibility with a technical audience.

> CONFIRMED INCIDENT **Knight Capital — an automated router that could not tell it was done**
>
> 2012-08-01 · REGULATOR ENFORCEMENT RELEASE
>
> A partial deployment reactivated defective, long-dormant code in Knight Capital's equity router, which then could not recognise that orders had been filled. In 45 minutes it sent over 4 million orders to fill just 212 customer orders, and the firm lost more than $460 million. Crucially for this module: an internal system had sent 97 automated warning emails before the market opened, and nobody acted on them. The SEC's first enforcement of the Market Access Rule followed.
>
> **Source:** U.S. Securities and Exchange Commission (Press Release 2013-222)
>
> https://www.sec.gov/newsroom/press-releases/2013-222

CONFIRMED INCIDENT **AWS DynamoDB — a textbook retry storm on a metadata service**

2015-09-20 · VENDOR POSTMORTEM

After a brief network disruption, storage servers could not renew their membership in time, became unavailable, and retried. In Amazon's own words, 'unavailable servers continued to retry requests for membership data, maintaining high load on the metadata service.' The positive feedback loop pulled healthy servers in too; error rates reached about 55% and engineers had to pause the metadata service entirely to break the cycle.

Source: Amazon Web Services (official postmortem)

https://aws.amazon.com/message/5467D2/

CONFIRMED INCIDENT **AWS US-EAST-1 — backoff that existed but did not engage**

2021-12-07 · VENDOR POSTMORTEM

An automated scaling activity triggered a surge of connection attempts; the resulting latency and errors produced, in AWS's words, 'even more connection attempts and retries.' Clients had tested their back-off behaviour, but 'a latent issue prevented these clients from adequately backing off during this event.' Monitoring was itself impaired, so operators worked with reduced visibility. Use this against 'we have retries with backoff' as a complete answer.

Source: Amazon Web Services (official postmortem)

https://aws.amazon.com/message/12721/

CONFIRMED INCIDENT **Google Cloud — recovery that caused a second outage**

2025-06-12 · VENDOR INCIDENT REPORT

An automated quota-policy update with blank fields hit an unprotected code path, crashing Service Control binaries in every region at once. The recovery then inflicted a second failure: as tasks restarted they 'created a herd effect on the underlying infrastructure', and Service Control 'did not have the appropriate randomized exponential backoff implemented to avoid this' — adding roughly two hours forty minutes to recovery in one region.

Source: Google Cloud (official incident report)

https://status.cloud.google.com/incidents/ow5i3PPK96RduMcb1SsW

CONFIRMED INCIDENT **Cloudflare — an automated pipeline distributing a bad artifact every five minutes**

2025-11-18 · VENDOR POSTMORTEM

A permissions change caused a query to return duplicate metadata, so a feature-file generator produced a file exceeding a hard limit and the proxy panicked. Because an automated pipeline regenerated and distributed that file every five minutes, the fault propagated network-wide, taking down Workers KV, Access, the dashboard and Turnstile. Automation converted one bad artifact into a global outage.

Source: Cloudflare (official postmortem)

https://blog.cloudflare.com/18-november-2025-outage/

RESEARCH PAPER **Why do multi-agent LLM systems fail? (the MAST taxonomy)**

2025-03 · ARXIV:2503.13657 · PEER-REVIEWED RESEARCH PAPER

An empirical failure taxonomy built from more than 1,600 annotated execution traces across seven multi-agent frameworks, defining 14 failure modes in three clusters: system design issues, inter-agent misalignment, and task verification. The closest thing to a canonical, evidence-based account of how multi-agent LLM systems actually break.

Source: Cemri et al. (UC Berkeley and collaborators)

https://arxiv.org/abs/2503.13657

09 · Worked scenario — The malformed reply that wedged everything

SLIDE 11

Illustrative. Constructed to show the mechanism end to end.

One agent returns malformed JSON. Downstream agents retry infinitely, saturating the tool API and taking the whole orchestration down.

1. **SETUP — Tight coupling.** Many agents share a tool API and depend on each other's output.
2. **FAULT — Bad output.** One agent emits malformed JSON that its consumers can't parse.
3. **STORM — Infinite retries.** Consumers retry with no backoff, flooding the shared API.
4. **IMPACT — System wedged.** The API rate-limits everyone; the orchestration stalls.

WHY THIS MATTERS

Cascading failure is an availability and safety risk: even without a malicious actor, ordinary bugs plus tight coupling and naive retries can take down an entire agent system.

10 · Common misconceptions

EXPECT THESE AS QUESTIONS

What people say	What to say back
"Retries make the system more reliable."	Unbounded, synchronised retries are an amplifier. Reliability comes from backoff, jitter, and a breaker that stops the calls.
"This is an SRE concern, not a security one."	Availability is a security property, and cascades are a denial-of-service an attacker can deliberately trigger.
"It failed once and recovered, so we are fine."	Metastable failures recover only when load drops. The same fault under production load may not clear on its own.

11 · Containment — the three layers

SLIDE 13

Bound every call with a timeout and put a circuit breaker in front of every dependency so repeated failure stops the calls rather than multiplying them. Retry with exponential backoff and jitter, capped at a small number of attempts — jitter matters because synchronised retries are what create the storm. Bulkhead agents and resource pools so one saturated dependency cannot drain the shared budget, and cap fan-out explicitly. Validate outputs between pipeline stages so malformed data fails fast and locally. Finally, rehearse it: fail a dependency deliberately and measure how far the blast travels.

LAYER 01 — CIRCUIT BREAKERS

- Trip a breaker when a dependency fails; stop calling it.
- Time out fast instead of hanging.
- Back off exponentially with jitter on retries.

LAYER 02 — ISOLATE & BULKHEAD

- Partition agents so one can't sink the fleet.
- Separate resource pools per zone.
- Cap fan-out and concurrent calls.

LAYER 03 — FALLBACK & MONITOR

- Provide safe fallbacks and graceful degradation.
- Validate outputs between steps.
- Monitor to detect and contain cascades early.

12 · For developers — secure code patterns

SLIDE 14

VULNERABLE

```
# retry forever, no isolation
while True:
    try: call(dep)
    except: continue   # storm
```

HARDENED

```
@breaker(fail_max=5, reset=30)
@timeout(3)
def call_dep(): ...
retry(call_dep, backoff=expo, jitter=True,
      max=3, fallback=safe_default)
```

Failing dependencies trip a **circuit breaker**, calls **time out and back off with jitter**, agents are **bulkheaded**, and there's always a **safe fallback**.

Where the controls live

Point	Control
BREAKER	**Circuit breakers.** Trip on repeated failure and stop hammering a dead dependency.
TIMEOUT	**Fast timeouts + backoff.** Bound every call; retry with exponential backoff and jitter.
BULKHEAD	**Isolate resources.** Separate pools and cap fan-out so one fault can't drain everything.
FALLBACK	**Graceful degradation.** Provide safe defaults and validate outputs between steps.

13 · For every employee — your role

SLIDE 16

Non-technical staff are not bystanders here. Three behaviours matter more than any technical understanding of the mechanism:

- **Spot it.** If an agent does something nobody asked for — moves data, changes records, contacts someone — treat it as suspicious, however confident it sounds.
- **Don't feed it.** Keep secrets, credentials, and bulk personal records out of agents. Be wary of outside documents an agent will read and act on.
- **Report it fast.** Agents act at machine speed, so the value of a report decays in minutes. An early, slightly wrong report beats a late, well-researched one.

14 · Detection — what to watch and log

SLIDE 17

These signals are only available if the underlying events are logged. Confirm the telemetry exists before relying on the alert.

Signal	What to watch for
RETRY	**Retry spike.** A sudden surge of retries against one dependency.
QUEUE	**Queue growth.** Task queues backing up faster than they drain.
SATURATION	**Resource exhaustion.** Pools, rate limits, or connections maxing out.
CASCADE	**Correlated errors.** Multiple agents failing in quick succession along dependencies.

15 · Red-team drill — how to test your own agent

SLIDE 18

Run these against a non-production agent with the same configuration as production. A probe only passes if it fails safely *every* time, not most of the time.

Probe	Test and pass criterion
PROBE 1	**Bad output.** Inject malformed data from one agent. Pass = consumers validate and isolate, no chain failure.
PROBE 2	**Kill a dependency.** Take a shared tool offline. Pass = breakers trip and the fleet degrades gracefully.
PROBE 3	**Exhaust a resource.** Drive load at a shared pool. Pass = bulkheads and caps contain it.
PROBE 4	**Blast radius.** Fail one component and measure spread. Pass = impact stays local.

Ship-ready checklist

Control	Done when
Circuit breakers.	Repeated failures trip a breaker that stops calling the dependency.
Timeouts & backoff.	Every call is time-bounded and retries back off with jitter.
Bulkheads.	Agents and resource pools are isolated; fan-out is capped.
Fallbacks.	Safe defaults and output validation keep degradation graceful.
Cascade monitoring.	Retry spikes, queue growth, and correlated errors are alerted early.

16 · Knowledge check and discussion

SLIDE 20

Q1. What turns one agent's failure into a system-wide outage?

Tight coupling plus naive retries. Synchronised retries with no backoff amplify a local fault into a storm.

Q2. Which control stops the fleet from hammering a dead dependency?

A circuit breaker that trips on repeated failure and stops the calls, combined with fast timeouts.

Q3. Do you need an attacker for a cascading failure?

No. Ordinary bugs plus coupling and bad retry logic are enough — it's an availability risk as much as a security one.

Take it back to your own systems

Close the session with these. They convert the module into action items:

1. If our most-used tool went down right now, how many agents would start retrying, and how hard?
2. Do we have a circuit breaker anywhere in the agent path, or only timeouts?
3. Have we ever deliberately failed a component to measure the blast radius?

---

17 · Glossary

Term	Meaning
Circuit breaker	A guard that stops calling a failing dependency until it recovers.
Bulkhead	Isolation that keeps one failure from consuming shared resources.
Backoff	Increasing the delay between retries to reduce load.
Blast radius	How far the impact of a single failure spreads.
Graceful degradation	Falling back to reduced function instead of total failure.

---

18 · Key takeaways

SLIDE 22

1. **Small failures don't stay small.** Coupling and retries spread them fast.
2. **Break the circuit.** Trip on failure; time out; back off with jitter.
3. **Isolate with bulkheads.** Keep one fault from draining the fleet.
4. **No attacker required.** Design for failure before it cascades.

---

19 · References and further reading

Every link below was checked when this guide was produced. Share them freely — the point of citing sources is that your audience can verify the claims themselves.

The framework

> **OWASP Top 10 for Agentic Applications (2026)**
>
> OWASP GENAI SECURITY PROJECT · 2025-12-09
>
> The official framework this training is based on. Peer-reviewed with input from 100+ researchers and an Expert Review Board including NIST, the European Commission, and the Alan Turing Institute.
>
> https://genai.owasp.org/resource/owasp-top-10-for-agentic-applications-for-2026/

> **OWASP Agentic Security Initiative**
>
> OWASP GENAI SECURITY PROJECT · 2025
>
> Programme behind the framework; publishes the companion guides.
>
> https://genai.owasp.org/initiatives/agentic-security-initiative/

> **deepteam — OWASP Top 10 for Agentic Applications reference**
>
> CONFIDENT AI · 2026
>
> Practitioner reference and red-teaming toolkit mapping to the framework.
>
> https://www.trydeepteam.com/docs/frameworks-owasp-top-10-for-agentic-applications

Incidents and research cited in this guide

Source	Link
Knight Capital — an automated router that could not tell it was done	https://www.sec.gov/newsroom/press-releases/2013-222
AWS DynamoDB — a textbook retry storm on a metadata service	https://aws.amazon.com/message/5467D2/
AWS US-EAST-1 — backoff that existed but did not engage	https://aws.amazon.com/message/12721/
Google Cloud — recovery that caused a second outage	https://status.cloud.google.com/incidents/ow5i3PPK96RduMcb1SsW
Cloudflare — an automated pipeline distributing a bad artifact every five minutes	https://blog.cloudflare.com/18-november-2025-outage/
Why do multi-agent LLM systems fail? (the MAST taxonomy)	https://arxiv.org/abs/2503.13657

Citation. OWASP GenAI Security Project, *OWASP Top 10 for Agentic Applications (2026)*, risk ASI08:2026 — Cascading Agent Failures.