ChatGPT

Reliability Deep Research for AI Agents, Discord Bots, and LLM Context Management

Scope and failure model

Long-running “agent” systems (AutoGPT-style loops, LangChain/LangGraph agents, tool-using assistants, autonomous coding/email agents) behave less like a single model call and more like a distributed system: a control loop coordinating external APIs, tool plugins, and state (memory) across many steps and many minutes/hours. [1]

A “crash” in practice spans a spectrum:

- **Hard crash**: process exits, watchdog restarts, runaway memory, or unrecovered exceptions.
- **Soft crash**: the process lives, but stops making progress (deadlocks / infinite loops / “recursion limit” reached / stuck retries).
- **Safety crash**: loss of critical constraints (e.g., “confirm before acting”) during context trimming/compaction, producing destructive actions.
- **Economic crash**: runaway retries and tool loops consume token budget or quota; you get throttled (429) or hit spend limits. [2]

This matters because the dominant failures in production are often *not* a single bug; they are feedback loops (retry storms, cascading failures, state blowups) where “small” partial failures amplify into system-wide outage behavior. [3]

AI agent crash prevention and recovery patterns

Memory leaks and context overflow patterns

In real systems, “memory leak” is often **unbounded state growth** (chat history, tool outputs, traces, caches) rather than a classic allocator leak. This is a reliability problem because state growth interacts with context windows (prompt size) and with process memory (RAM). LangGraph/LangChain documentation explicitly frames memory as state that persists and grows across steps and can be resumed, which is powerful but has predictable failure modes if unbounded. [4]

Common overflow patterns seen in agent frameworks:

- **Full transcript accumulation**: “store everything” designs (ConversationBuffer-style) rapidly balloon token usage; Pinecone’s LangChain memory walkthrough explicitly notes that storing every interaction can quickly exceed context window limits, motivating progressive summarization. [5]
- **Context-length crash**: multiple real AutoGPT issue reports show the same operational failure: an extended run accumulates messages until the model call throws “maximum context length” errors, sometimes producing loops rather than clean shutdown. [6]

- **Constraint loss during compaction**: the OpenClaw incident described a large "real inbox" triggering "context compaction," after which the agent lost the "don't act until I confirm" instruction and executed bulk deletions. This is a concrete example of a safety-critical instruction being dropped by context management under pressure. [7]

Crash prevention implications:

- Treat context as a **bounded resource**; enforce explicit caps and structured policies (trim, summarize, or externalize to retrieval).
- Treat "instruction anchors" as **non-compactable** (pinned constraints / policy blocks) in whatever memory system exists, because naive "compress oldest" can delete guardrails first. The OpenClaw failure mode demonstrates why this is not theoretical. [8]

API rate limiting cascades and retry storms

Agent systems often have *many* outbound calls per user request: LLM calls, tool APIs, databases, search, and sometimes stacked internal services. Rate limits and transient errors are normal; what causes outages is **retry amplification**.

Key, well-documented cascade mechanics:

- **Unsuccessful retries can worsen rate limiting**: OpenAI's guidance for 429 explicitly warns that unsuccessful requests still count toward per-minute limits; naïvely "resend until it works" self-amplifies throttling. [9]
- **Retry storms are a known distributed-systems antipattern**: Azure's "retry storm" antipattern notes services can struggle to recover because many clients hammer them during recovery. [10]
- **Multi-layer retry multiplication**: AWS' Builders' Library describes a five-deep call stack with three retries per layer amplifying load by 243× when the downstream starts failing—classic positive feedback leading to metastable failure. [11]
- **Cascading failures are overload-driven positive feedback**: Google SRE defines cascading failure precisely as failure that grows over time via positive feedback, commonly triggered by overload; mitigations include serving degraded results and deliberate load shedding. [12]
- **Recommended mitigation baseline**: bounded retries + exponential backoff + jitter + a max retry budget are explicitly recommended by AWS Well-Architected reliability guidance. [13]

Agent-specific cascade pattern that repeatedly appears in practice:

1. Tool API begins timing out or returning 429/5xx intermittently.
2. Agent loop interprets failure as "try again," often re-running the *entire* reasoning prompt + tool call.
3. Token spend increases (economic crash), tool traffic increases (rate-limit crash), and latency rises (queueing).
4. System hits quota/spend limits (OpenAI 429 quota) or global throttles; then *everything* fails. [14]

Tool execution deadlocks and "zombie" hangs

Deadlock in agent systems is rarely a mutex deadlock; it is more often:

- A tool call that never returns (waiting on network / subprocess / browser automation).

- A tool call that returns but the agent cannot parse it (schema mismatch), leading to repeated re-attempts.
- Concurrency collisions (two tools manipulating the same external state concurrently).
- Event-loop starvation where "progress" stops but process stays alive.

Hardening mechanisms supported by major agent tooling:

- The OpenAI [15] Agents SDK explicitly supports **per-tool timeouts** and explicitly documents two policies: return a model-visible timeout error (recoverable) vs hard-fail the run by raising an exception. It also supports tool error functions to return controlled error observations to the model rather than crashing the whole run. [16]
- LangGraph documents "GRAPH_RECURSION_LIMIT" as an operational symptom: graphs reach a max step limit due to cycles/infinite loops, and recommends debugging cycles or raising limits only when expected. [17]

Operationally, "tool deadlock prevention" is mostly: **timeouts + cancellation + idempotency + isolating side effects**. When tools are slow, you either (a) cap them, (b) split into async job dispatch + polling workflows, or (c) move them into a durable workflow engine so progress survives restarts. The existence of durable-agent patterns (e.g., workflow/orchestration integrations) is itself a response to tool unreliability. [18]

Token budget exhaustion and error propagation

There are two distinct "token budget" failure classes:

- **Context window exhaustion**: request fails because aggregated messages exceed model limits (explicitly shown in AutoGPT issues). [6]
- **Quota/spend exhaustion**: OpenAI documents 429 variants for both rate limit and "exceeded your current quota / maximum monthly spend," which becomes a systemic outage for the agent if not handled as a first-class state. [19]

Error propagation ("one failure causes system-wide crash") typically enters through one of these paths:

- **State poisoning**: incorrect intermediate results (tool outputs, earlier hallucinations) are retained and repeatedly reused; OpenAI's context-engineering guidance explicitly frames summaries as "clean rooms" to correct/omit prior mistakes and prevent compounding errors. [20]
- **Tight coupling**: if "LLM call succeeds" is required at every step to make progress, then LLM throttling becomes a total outage. This is why graceful degradation patterns (soft dependencies) matter. [21]

Recovery strategies that actually survive production

A realistic recovery stack draws from standard resilience patterns, adapted for agent loops:

- **Circuit breakers**: Microsoft's circuit breaker pattern is explicit: prevent repeated calls likely to fail; keep the system responsive; optionally return defaults during open state; combine with retry cautiously because the breaker's exceptions should stop retries. [22]
- **Graceful degradation**: AWS Well-Architected calls out converting hard dependencies to soft dependencies and prioritizing "most important functions" during failures. For agents, this maps to:

skipping optional tools, returning partial results, deferring work into a queue, or switching to cached/previous outputs. [21]
- **Controlled retries**: exponential backoff with jitter, bounded retry budgets, and eliminating "retries at every layer" (explicit AWS guidance). [23]
- **Bounded execution**: enforce max steps / recursion limits / wall-clock limits (LangGraph recursion limit exists precisely because infinite loops are routine in complex graphs). [17]
- **Checkpointing and resumable progress**: Anthropic [24] 's long-running agent harness work emphasizes external progress artifacts (feature lists, progress notes, commits) so each session can restart, verify state, and continue rather than pretending to "hold everything in context" forever. [25]
- **Safety via reversible side effects**: Replit [26] describes snapshottable filesystems and databases, explicit revertability, dev/prod split, and restricting agent access to development databases—classic "blast radius reduction" for agent actions. [27]

Discord bot reliability at scale

Token revocation and suspension drivers beyond "duplicate instances"

At scale, "token revoked" events are frequently self-inflicted by exceeding platform safety limits or mishandling credentials.

Hard, explicit revocation/reset mechanisms:

- **IDENTIFY limit resets bot token**: Discord Gateway docs state a global limit of 1000 IDENTIFY calls per 24 hours across shards; hitting it terminates active sessions and **resets the bot token**, requiring an owner update. [28]
- **Gateway abuse can lead to revoked access**: Discord Gateway rate limiting documentation states apps can send 120 gateway events per connection per 60 seconds and that repeat offenders can have API access revoked. [29]

Policy/terms enforcement vectors (often experienced operationally as "suspended access"):

- Discord's Developer Policy explicitly references enforcement actions for violations. [30]
- Discord's Developer Terms prohibit abusive usage (including exceeding usage limits) and require protecting developer credentials (tokens/keys), including not embedding them in open source and keeping them encrypted in materials accessible to third parties. [31]

Operational "token compromise" mechanism:

- Even when not platform-enforced, a leaked bot token must be rotated because it functions as the bot's password; common developer documentation (including major library docs) treats regeneration as revoking the old token. [32]

Rate limiting patterns and how to avoid them in production

Discord rate limiting is multi-layered: per-route buckets, global limits, and an "invalid request" limiter that is operationally catastrophic if reached.

From official Discord docs:

- 429 responses include `retry_after` and distinguish global vs non-global rate limits; rate-limit headers define remaining capacity and reset timing. [33]
- Global rate limit baseline: "All bots can make up to 50 requests per second to our API." [33]
- Invalid Request Limit ("Cloudflare bans"): too many invalid requests (401/403/429) triggers temporary restriction; Discord documents the current threshold as 10,000 invalid requests per 10 minutes and recommends tracking invalid request rates at scale. [34]

Production-grade avoidance patterns implied by these mechanics:

- Centralize HTTP dispatch behind a single rate-limit aware queue per token (and, at scale, per shard/cluster), because the unit of enforcement is shared across your deployment. [35]
- Treat 401/403/429 as **faults that must reduce traffic**, not as recoverable "spam retry" signals, because they contribute directly to the invalid request ban limiter. [34]

WebSocket connection management at 24/7 scale

Discord Gateway reliability hinges on implementing the protocol exactly, including timing jitter, heartbeat acks, and correct resume behavior.

From official Discord Gateway docs:

- Clients receive a Hello with `heartbeat_interval` and must heartbeat every interval; first heartbeat should be delayed by `heartbeat_interval * jitter` where jitter is random 0–1 to avoid synchronized reconnect spikes. [36]
- Missing Heartbeat ACK indicates a failed ("zombied") connection; clients should terminate and reconnect, attempting Resume. [36]
- Correct resume requires caching `resume_gateway_url`, `session_id`, and last sequence number; failing to use `resume_gateway_url` increases disconnect rate. [37]

From Discord opcodes/status codes documentation:

- Specific close codes should stop reconnect loops (e.g., authentication failed, invalid intents, disallowed intents), while others indicate "reconnect and resume." [38]

At scale, connection management is inseparable from avoiding token resets:

- Every unnecessary IDENTIFY (especially in crash loops) consumes limited "session starts" and risks hitting the 1000/day IDENTIFY reset limit. [28]
- Discord documents max concurrency for identifying shards (`max_concurrency`) and how shards map to concurrency buckets, which becomes a hard deployment constraint for large sharded bots. [39]

Intent configuration and state management impact on stability

Gateway Intents are directly framed as a mechanism to reduce computational burden:

- Discord Gateway docs describe intents as a system to lower the burden of processing stateful event data; missing/invalid/disallowed intents can cause the connection to be closed (4013/4014 close codes). [40]
- For verified bots, message content is a privileged intent with a documented approval/revocation path; after transition periods, verified apps' access can be revoked if unapproved. [41]

Memory/state guidance from Discord is explicit:

- Discord Gateway documentation recommends storing only needed state because client state can grow large; some events include necessary member objects, reducing the need for full member caching. [42]

In practice, "bot memory leaks" at scale frequently come from "cache everything" patterns (members/messages) and from event listener accumulation. The correct baseline is: **minimal cache + external persistence + selective fetch**. Discord's own docs push toward this by warning about state growth and encouraging minimal in-memory storage. [43]

Multi-region and failover constraints

Discord's scaling unit is the shard, and failover must respect shard exclusivity and IDENTIFY limits.

What Discord explicitly supports:

- Sharding: each shard supports up to 2500 guilds; apps in 2500+ guilds must shard. Discord provides the sharding formula and notes sharding requires no state-sharing between connections. [42]
- Discord also notes you can establish multiple sessions with the same shard parameters and even orchestrate "zero-downtime" scaling/updating by handing off traffic to a new deployment of sessions prepared in parallel. [42]

Implication for multi-region designs:

- Multi-region is viable, but "active-active" must be planned at the shard/orchestration layer so only one session is authoritative for a given guild's event stream at a time, while others remain standby or handle disjoint shard sets. This follows directly from the sharding model and resume/session semantics. [44]

Evidence of production uptime targets in real bots

Some major bot services publish uptime explicitly via status pages:

- MEE6 [45] reports 100.0% uptime over the past 90 days for its bot and related components (as of Feb 26, 2026). [46]
- Square [47] similarly reports 100.0% uptime over the past 90 days for its bot. [48]

Those pages do not disclose architecture, but any long-lived bot meeting that uptime still must obey the hard protocol constraints documented above (heartbeat correctness, resume correctness, sharding thresholds, identify ceilings, and rate-limit compliance). [49]

LLM context management and token optimization

When to compact versus keep full history

The most operationally grounded pattern is **bounded session context + explicit compaction policy**:

- OpenAI's Agent SDK "session memory" guidance centers on trimming and compression as two "proven techniques," explicitly linking them to coherence, tool-call accuracy, cost/latency reduction, and error containment ("clean rooms" to avoid compounding bad facts). [20]
- LangChain/LangGraph documentation frames short-term memory as thread-scoped state persisted via checkpoints and long-term memory as cross-thread storage; this implies (and operationally requires) a strategy for what belongs in which tier. [4]

Compact (summarize/trim) when:

- The interaction is long enough that full history limits tool accuracy or pushes context toward model limits. [50]
- Earlier turns contain exploratory noise that is no longer relevant (risk of "yesterday's plan" overriding current goals). [20]

Keep full history when:

- You need legal/medical-style traceability of exact phrasing, commitments, or user-provided data, and errors from summarization would be high cost. This is the same reason OpenAI frames summaries as clean-room transformations—useful, but a semantic change. [51]

Semantic versus chronological context selection

Chronological context (last N messages) is simple but wasteful. Semantic selection is the usual route for maximizing "intelligence per token":

- Retrieval-Augmented Generation (RAG) is a canonical approach: combine parametric knowledge with non-parametric memory accessed via retrieval. [52]
- LangChain's memory concepts separate "semantic memory" (stored facts) from "semantic search" (embedding-based retrieval), reinforcing that production systems typically need both: stable stored facts and meaning-based retrieval. [4]

Practical implication: treat the context window as a scarce working set and use retrieval to inject only the *minimum sufficient evidence* for the current turn, rather than dragging entire dialogue history forward. [53]

Cache optimization strategies

Two distinct caches matter:

- **Prompt caching (provider-side)**: OpenAI documents automatic prompt caching that reuses recently processed prompt prefixes; it is enabled for recent models and requires structuring prompts so static prefix content is identical and placed first. [54]
- **KV caching (inference-side)**: for self-hosted transformer inference, KV cache avoids recomputing attention for previously processed tokens by storing key/value pairs, improving generation efficiency at inference time. [55]

Economics:

- OpenAI's pricing page shows cached input tokens priced materially below standard input pricing (example entries explicitly list "cached input" pricing). This makes prompt-prefix stability a direct cost lever. [56]

Cost–quality tradeoffs and context window utilization

Two empirically grounded constraints shape long-context behavior:

- **"Lost in the middle"**: long-context models often use information near the beginning and end of a prompt more effectively than information buried in the middle; this is a documented failure mode for long contexts. [57]
- **Compaction risk**: aggressive compaction can delete critical constraints; the OpenClaw incident demonstrates that compaction can drop a safety rule and cause destructive actions. [7]

Operational heuristic derived from these constraints:

- Running at ~95% context utilization is an outage-multiplier: it maximizes truncation risk, increases the chance that essential constraints fall into the "middle," and drives up latency/cost per step (more tokens processed per call). The "lost in the middle" evidence supports maintaining structural headroom so essential instructions and current task state can remain in the most reliable prompt positions (front/back). [58]

Production patterns that persist across resets

The strongest cross-source convergence is: **externalize durable state**.

- Anthropic's agent harness patterns rely on feature lists, progress notes, and git commits so each run begins by reloading "truth" from durable artifacts and ends by committing verifiable progress. This makes the system resilient across context windows and restarts. [25]
- OpenAI's session memory approach similarly treats summaries/trimming as explicit state transformations to keep sessions bounded and debug-friendly. [20]

Early warning system design that predicts crashes

Instrumentation baseline

A production early-warning design needs three telemetry layers:

- **Metrics**: health and saturation numbers.
- **Traces**: end-to-end step-level visibility across LLM calls and tool calls.
- **Logs/events**: high-cardinality details for incident reconstruction.

This aligns with standard monitoring definitions (collect, process, and display quantitative data) and with modern agent observability framing. [59]

OpenTelemetry [60] is explicitly positioned as a single set of APIs/libraries/collectors for capturing metrics and distributed traces, making it a natural substrate for agent "run traces" that span tools and model calls. [61]

Predictive signals for AI agent crashes

Predictive signals are those that trend *before* failure:

- **Context pressure**: tokens per step, context utilization %, and compaction events; these precede "context length" crashes and safety-rule loss. [62]
- **Retry pressure**: rising 429 rate, falling "successful call ratio," rising backoff time; retry storms precede total outage. [63]
- **Tool latency drift**: increasing p95/p99 tool-call latency, timeout rates; this precedes deadlocks and step explosions. [64]
- **Step explosion**: recursion/iteration counter growth without progress markers; LangGraph's recursion limit error exists because infinite loops are common and detectable via step counting. [17]
- **Resource saturation**: CPU, memory RSS, queue depths; saturation is a canonical "pre-crash" signal in SRE monitoring methods. [65]

Predictive signals for Discord bot incidents

From Discord's explicit protocol and limits, the highest-signal pre-incident indicators are:

- **Heartbeat health**: missing Heartbeat ACKs ("zombied" connections) and increasing resume frequency. [66]
- **IDENTIFY burn rate**: IDENTIFY calls/day, session start limit remaining, and max concurrency bucket behavior; this directly predicts token reset events. [67]
- **Invalid request rate**: 401/403/429 per 10-minute window; Discord documents this as the trigger for temporary API restriction. [34]
- **Global and bucket rate-limit pressure**: repeated 429s with low remaining capacity and rising retry_after. [33]

Alerting model that avoids noise and catches real failure

SLO-based alerting is the scalable pattern: alert on **error budget burn**, not on every transient spike.

- Google's SRE workbook provides concrete burn-rate math (e.g., for 99.9% SLO, 0.1% error rate consumes the budget at burn rate 1; higher burn rates exhaust budgets rapidly). [68]

To connect SLO alerting to "predict crashes":

- Use short-window + long-window multi-burn alerts (catch fast meltdowns and slow degradation).
- Couple SLO burn alerts with resource saturation (USE method) and request health (RED method): saturation + rising error rate is a crash precursor, not a mere blip. [69]

Checklists and reference designs

AI agent crash prevention checklist

Bounded execution and loop control
- Enforce max-step / recursion limits and treat "limit reached" as a first-class controlled shutdown, not an exception cascade. [17]
- Enforce wall-clock limits per run and per tool call; timed-out tools must return model-visible errors or terminate runs deterministically. [16]

Context safety and memory hygiene
- Separate pinned constraints ("must confirm before acting," safety policies) from compactable history; compaction systems that drop constraints are demonstrably dangerous. [7]
- Replace "store everything" memory with progressive summaries + retrieval-backed long-term memory; full-buffer strategies are documented to exceed context window limits quickly. [70]
- Maintain external progress artifacts (feature lists, progress notes, commits) so restarts are normal and safe. [25]

Rate limiting and dependency hardening
- Implement bounded retries with exponential backoff + jitter and a retry budget; never allow retries at every layer. [71]
- Add circuit breakers on flaky tools and LLM providers; open-state should fail fast or return safe defaults. [72]
- Implement graceful degradation paths ("soft dependencies") so the agent can still deliver partial value when one dependency fails. [21]

Blast radius reduction
- Default all external side effects to reversible modes: sandboxed environments, dev/prod separation, and revertable snapshots for stateful resources. [73]

Early warning system reference design for agent crashes

Telemetry (minimum viable)
- Per-run traces: (1) model call spans with tokens, latency, 429/5xx; (2) tool call spans with timeout/exception

types; (3) step counters and termination reasons. [74]
- Metrics: context utilization %, tokens per minute, cost per task, retry counts, queue depth, memory RSS, saturation. [75]

Predictors and automated mitigation
- Trigger circuit breaker when tool error rate or timeout rate crosses a threshold; switch to degraded mode (skip optional tools, return partial, enqueue for later). [76]
- Trigger "context pressure" mitigations when utilization crosses threshold: summarize, trim, retrieve, or restart with external state reloaded. [77]

Discord bot reliability checklist for 99.9%+ uptime targets

Gateway correctness
- Heartbeats: respect `heartbeat_interval`, initial jitter, and Heartbeat ACK handling; treat missing ACK as zombied connection and force reconnect+resume. [66]
- Resume: always use `resume_gateway_url`; cache `session_id` and last sequence; handle opcode 7 reconnect and opcode 9 invalid session correctly. [40]
- Close codes: terminate reconnect loops on non-recoverable close codes (auth failed, invalid/disallowed intents). [78]

IDENTIFY hygiene (prevents forced token reset)
- Track IDENTIFY calls/day and session start limit remaining; never let deployment crash loops burn IDENTIFY budget toward the documented 1000/day token reset threshold. [79]
- Implement shard startup concurrency exactly per `max_concurrency` buckets. [39]

HTTP rate-limit hygiene
- Respect per-route headers and `retry_after`; treat 429 as a backpressure signal. [33]
- Track invalid request rate (401/403/429) to avoid hitting the documented 10,000/10-minute restriction. [34]

Intents and state footprint
- Enable only necessary intents to reduce event volume/compute; ensure privileged intents are enabled/approved or connections will be closed. [80]
- Minimize in-memory cached state; Discord explicitly warns state can grow large and recommends storing only what is required. [42]

Evidence targets
- External status pages show some bots can sustain 100% uptime over recent 90-day windows; achieving similar requires strict adherence to the protocol limits and defensive deployment practices above. [81]

---

[1] [25] Effective harnesses for long-running agents \ Anthropic
https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents

[2] [17] [26] GRAPH_RECURSION_LIMIT - Docs by LangChain
https://docs.langchain.com/oss/python/langgraph/errors/GRAPH_RECURSION_LIMIT

[3] [12] Google SRE - Cascading Failures: Reducing System Outage
https://sre.google/sre-book/addressing-cascading-failures/

4 24 Memory overview - Docs by LangChain
https://docs.langchain.com/oss/javascript/concepts/memory

5 70 Conversational Memory for LLMs with Langchain | Pinecone
https://www.pinecone.io/learn/series/langchain/langchain-conversational-memory/

6 62 autogpt crash by exceding token limit. · Issue #3979 · Significant-Gravitas/AutoGPT · GitHub
https://github.com/Significant-Gravitas/Auto-GPT/issues/3979

7 8 Meta's safety director loses emails to OpenClaw AI agent | Windows Central
https://www.windowscentral.com/artificial-intelligence/meta-summer-yue-director-openclaw-ai-email-deletion

9 63 How can I solve 429: 'Too Many Requests' errors? | OpenAI Help Center
https://help.openai.com/en/articles/5955604-how-can-i-solve-429-too-many-requests-errors

10 Retry Storm Antipattern - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/antipatterns/retry-storm/

11 Timeouts, retries and backoff with jitter
https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/

13 23 71 REL05-BP03 Control and limit retry calls - Reliability Pillar
https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/rel_mitigate_interaction_failure_limit_retries.html

14 19 Error codes | OpenAI API
https://developers.openai.com/api/docs/guides/error-codes/

15 16 47 64 Tools - OpenAI Agents SDK
https://openai.github.io/openai-agents-python/tools/

18 Tool calls that take a long time - LangGraph
https://forum.langchain.com/t/tool-calls-that-take-a-long-time/300?utm_source=chatgpt.com

20 50 51 77 Context Engineering - Short-Term Memory Management with Sessions from OpenAI Agents SDK
https://developers.openai.com/cookbook/examples/agents_sdk/session_memory/

21 REL05-BP01 Implement graceful degradation to transform applicable hard dependencies into soft dependencies - Reliability Pillar
https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/rel_mitigate_interaction_failure_graceful_degradation.html

22 72 76 Circuit Breaker Pattern - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker

27 73 Replit — Inside Replit’s Snapshot Engine: The Tech Making AI Agents Safe
https://blog.replit.com/inside-replits-snapshot-engine

28 29 36 37 39 40 42 43 44 49 60 66 67 79 80 Gateway - Documentation - Discord
https://docs.discord.com/developers/events/gateway

30 Discord Developer Policy – Developers
https://support-dev.discord.com/hc/en-us/articles/8563934450327-Discord-Developer-Policy

31 Discord Developer Terms of Service – Developers
https://support-dev.discord.com/hc/en-us/articles/8562894815383-Discord-Developer-Terms-of-Service

32 Creating a Bot Account - Discord.py
https://discordpy.readthedocs.io/en/stable/discord.html?utm_source=chatgpt.com

33 34 35 Rate Limits - Documentation - Discord
https://docs.discord.com/developers/topics/rate-limits

38 78 Opcodes and Status Codes - Documentation - Discord
https://docs.discord.com/developers/topics/opcodes-and-status-codes

41 Message Content Privileged Intent FAQ – Developers
https://support-dev.discord.com/hc/en-us/articles/4404772028055-Message-Content-Privileged-Intent-FAQ

45 65 69 75 The USE Method
https://www.brendangregg.com/usemethod.html?utm_source=chatgpt.com

46 81 MEE6 Status
https://mee6.statuspage.io/

48 Square Status
https://squarebot.statuspage.io/

52 53 Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks
https://arxiv.org/abs/2005.11401?utm_source=chatgpt.com

54 Prompt caching | OpenAI API
https://developers.openai.com/api/docs/guides/prompt-caching/

55 Caching
https://huggingface.co/docs/transformers/en/cache_explanation?utm_source=chatgpt.com

56 Pricing | OpenAI
https://openai.com/api/pricing/

57 58 Lost in the Middle: How Language Models Use Long ...
https://huggingface.co/papers/2307.03172?utm_source=chatgpt.com

59 Chapter 6 - Monitoring Distributed Systems
https://sre.google/sre-book/monitoring-distributed-systems/?utm_source=chatgpt.com

61 74 OpenTelemetry
https://opentelemetry.io/?utm_source=chatgpt.com

68 Prometheus Alerting: Turn SLOs into Alerts
https://sre.google/workbook/alerting-on-slos/?utm_source=chatgpt.com