

Article

Trigger, Not Root Cause: Re-Examining GitHub's 2024 to 2026 Availability Decline Under Artificial Intelligence-Driven Load

Vlad-Ștefan Dieaconu ¹, Răzvan Rughiniș ^{1,2,*}, Ebru Resul ¹ and Dinu Țurcanu ^{3,*}

¹ Faculty of Automatic Control and Computers, National University of Science and Technology POLITEHNICA Bucharest, 060042 Bucharest, Romania; vlad.dieaconu@stud.acs.upb.ro (V.-Ș.D.); ebru.resul@upb.ro (E.R.)

² Academy of Romanian Scientists, 3 Ilfov, 050044 Bucharest, Romania

³ Faculty of Electronics and Telecommunications and National Institute of Innovations in Cybersecurity "CYBERCOR", Technical University of Moldova, 168, Stefan cel Mare Blvd, MD-2004 Chisinau, Moldova

* Correspondence: razvan.rughinis@upb.ro (R.R.); dinu.turcanu@adm.utm.md (D.Ț.)

Abstract

Over two years, the code-hosting platform GitHub suffered a sustained decline in availability that became acute by mid-2026. A widely repeated explanation blames artificial intelligence, in particular AI coding assistants and autonomous agents acting on code at machine speed. We argue that this account confuses a trigger with a root cause. Treating the episode as an explanatory single-case study built only from public evidence, namely GitHub's reports and post mortems, independent incident tracking, disclosures from other operators, and the reliability literature, we separate proximate triggers from structural causes. Three pre-existing weaknesses recur: services coupled tightly enough for a localized fault to cascade, weak protection against misbehaving client traffic, and capacity that could not expand quickly enough to absorb non-diurnal load. Agentic traffic exposed and amplified these weaknesses but did not create them; GitHub's own statements and the theory of metastable failure support this reading, and incidents with the same structural signature predate the surge by more than a year. We set out architectural remedies, among them cell-based isolation with shuffle sharding, decoupling of critical paths, admission control and load shedding, quality-of-service tiering, and predictive elasticity, and close with a resilience agenda for an internet where automated requests now exceed human ones.

Keywords: distributed systems; availability; resilience; cascading failure; metastable failure; cell-based architecture; load shedding; autoscaling; agentic traffic; case study; GitHub



Academic Editors: Mara Nikolaidou, Teta Stamati, Cleopatra Bardaki, Georgia Dede, Alexandra Gasparinatu, Evangelia Filiopoulou and Elena Politi

Received: 15 June 2026

Revised: 14 July 2026

Accepted: 15 July 2026

Published: 20 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

GitHub is the world's most widely used platform for hosting and collaborating on source code, and over the past two years it has also become core infrastructure for AI-assisted software development. Its own annual report describes a platform operating at very large scale. More than 180 million developers now work on it, and over 36 million of them joined during 2025, an average of more than one new developer every second [1]. In the same year, developers created more than 230 new repositories every minute, merged an average of 43.2 million pull requests each month, and pushed close to one billion commits [1]. Hundreds of millions of projects depend on the platform not only for storage but for the automated pipelines that build, test, and deploy software. When GitHub is unavailable, the consequences reach well beyond any single company: continuous-integration pipelines stall, software releases are blocked, and large numbers of developers

lose access to the systems through which modern software is written and shipped. GitHub's availability has, in short, become a concern for a large part of the software economy.

That reliability has visibly deteriorated. GitHub publishes a monthly availability report that summarizes every incident degrading its services [2]. Over the twelve months from May 2025 to April 2026, an independent monitoring service that tracked the platform continuously recorded 257 incidents, of which 48 were major outages; February 2026 was the worst single month, with 37 incidents [3]. The deterioration was serious enough to prompt visible departures, including maintainers of well-known projects who moved their work off GitHub and cited reliability as one of their reasons [3]. It also led GitHub's Chief Technology Officer to publish two detailed and unusually frank posts acknowledging the problem and describing the engineering response [4,5]. By the company's own account, it had begun an effort to raise its capacity roughly tenfold, then concluded within months that it would need to design for considerably greater scale [5].

The most common public explanation for this decline is AI, and the reasoning behind it is intuitive. The same two-year period brought a large rise in machine-generated activity. AI coding assistants now write and revise code continuously; autonomous agents open pull requests and run searches without waiting for a human; and large crawlers harvest public repositories to train and serve models. GitHub's leadership reports that agentic development workflows accelerated sharply from the second half of December 2025 [5], which is also when the incidents intensified. Drawing the obvious conclusion, many observers have held AI traffic responsible for GitHub's availability problems.

Our argument is that this account is mistaken on a key point: it confuses a trigger with a root cause. A trigger is the event that sets an outage in motion. A root cause is the property of the system that lets that event turn into a large and lasting failure. Between the two stands the amplifier, a force that raises the frequency and severity of triggers without changing what the system does once a trigger arrives. AI and agentic traffic, on our reading, were the trigger and the amplifier of GitHub's decline rather than its root cause. The root causes lay in how the platform had been built for resilience. Its services were coupled tightly enough that trouble in one could spread to many; it had no effective way to detect and shed abnormal traffic from a small number of badly behaved clients; and its capacity could not grow quickly enough to absorb sudden surges that arrived outside the usual daily pattern of human use. The higher load did not produce these weaknesses, but it brought them to the surface. As the rest of the article shows, GitHub's own reports come close to saying as much.

This article makes four contributions. First, it reconstructs GitHub's availability decline from public sources and characterizes the incidents by frequency, severity, most-affected service, and stated cause. Second, it separates proximate triggers from structural causes, labelling each claim as GitHub-official, third-party, or our own inference, and grounds the distinction in the theory of metastable failure. Third, it sets out specific architectural changes that would have contained the documented incidents, with their costs and trade-offs made explicit. Fourth, it offers a forward-looking agenda for platform resilience in a setting where automated traffic has begun to exceed human traffic, pairing concrete threats with named mitigations.

The argument can be stated as four questions. First, what does the public record show about the frequency, severity, distribution, and stated causes of GitHub's availability decline? Second, were the structural weaknesses behind the largest incidents present before the rise of agentic traffic? Third, which established architectural mechanisms would likely have contained the documented incidents, and at what cost? Fourth, what does the case imply for the design of openly accessible platforms now that automated requests rival or

exceed human requests on the web? The four contributions below, and Section 4, Section 5, Sections 6 and 8, answer these questions in order.

GitHub is our central case, but it is not an isolated one. Other large, openly accessible services have reported the same basic problem of automated traffic overwhelming infrastructure that was provisioned for human use, as Section 2.4 documents. The stakes also extend past any one platform. The composition of internet traffic itself is changing, with automated requests now rivaling or exceeding human requests by at least one network-wide measurement [6], and how large services remain available under that traffic is a question about the architecture and governance of the future internet, not about the operations of a single company. We therefore treat GitHub as a well-documented instance of a wider industry pattern. The rest of the article proceeds as follows. Section 2 gives background on GitHub's scale and architecture, on the dependability concepts the analysis relies on, and on the rise of agentic traffic. Section 3 sets out the scope and method. Section 4 characterizes the incident record. Section 5 develops the cause-versus-trigger argument. Section 6 proposes architectural changes. Section 7 reviews related academic work. Section 8 turns to the agentic future and pairs threats with mitigations. Section 9 concludes.

2. Background

2.1. GitHub's Scale and Architecture

To see why GitHub's failures spread, it helps to picture how the platform is built. GitHub began as a large Ruby on Rails application, a so-called monolith in which a single codebase runs as one unit and shares core infrastructure, backed by MySQL relational databases [5]. Around this core sit many services: Git storage for repository data, GitHub Actions for continuous integration and deployment, a search subsystem built on Elasticsearch, and services for pull requests, issues, notifications, webhooks, packages, the Copilot family of AI features, and more. These services are not fully independent. Many share databases, caches, authentication paths, and network infrastructure. GitHub has said that it is now breaking the monolith into more isolated services and data domains, moving performance-sensitive code from Ruby to Go, shifting some workloads off MySQL, and relocating infrastructure to Microsoft Azure to gain both larger machines and more regions [4,5]. As of March 2026, the company reported that 12.5% of its traffic was served from an Azure region, with a target of 50% by mid-year [4]. The point for this article is that, during the period studied, GitHub ran a partially coupled architecture in which several critical services depended on shared components. That is the kind of arrangement in which a localized fault can grow into a platform-wide outage.

Two features of GitHub's operating context bear on how far this case generalizes. GitHub is owned by Microsoft, and the migration just described moves a growing share of its traffic onto Microsoft Azure, from about 12.5% toward a stated 50% target [4]. That move buys larger machines and more regions, and is part of GitHub's own remedy, but it also ties the platform to a proprietary operational substrate whose internals are not public and concentrates dependence on a single cloud provider, itself a shared-fate consideration of the kind this article examines. The architectural lessons drawn later are therefore portable across platforms, whereas GitHub's particular migration path is entangled with Microsoft and Azure economics and should not be assumed for operators without that backing.

The volume this architecture must absorb has grown quickly. Between the 2024 and 2025 editions of GitHub's annual Octoverse report, activity rose sharply: the 2024 report recorded a 59% year-on-year increase in contributions to generative-AI projects [7], while the 2025 report recorded close to 1 billion commits (a 25% rise), an average of 43.2 million pull requests merged each month (up 23%), and a steady pace of more than 230 new repositories per minute [1]. A single user action, opening one pull request, can touch Git

storage, mergeability checks, branch protection, Actions, search, notifications, permissions, webhooks, application programming interfaces (APIs), background job queues, caches, and databases [5]. That fan-out has consequences. When each unit of work runs through many subsystems, a small per-request inefficiency, multiplied across a surge in requests, can become a large aggregate load on whichever shared component is weakest.

2.2. Availability, Dependability, and How Systems Fail Under Load

The analysis rests on a small, standard vocabulary. Dependability is the umbrella property of a system that lets reliance be justifiably placed on the service it delivers, and its attributes include reliability, availability, safety, and maintainability [8,9]. Availability is the readiness of a system to deliver correct service, which is informally the fraction of time it is usable, whereas reliability is the continuity of correct service over an interval [8]. Operators often state an availability target in “nines”. Five nines, or 99.999% availability, permits only about 5.26 min of downtime per year; four nines (99.99%) permits roughly 52.6 min, and three nines (99.9%) roughly 8.8 h. These targets show why even brief, frequent degradations matter. A platform that many businesses treat as always-on has a very small budget for unavailability, and that budget is spent by the sort of repeated, partial outages described later.

Two further ideas describe how failures spread. The blast radius of a fault is the share of users, requests, or subsystems it affects, and resilient designs try to keep it small so that a problem in one place does not become a problem everywhere. A cascading failure is a chain reaction in which the failure of one component raises the load or latency on others and causes them to fail in turn. Tight coupling, by which we mean shared databases, caches, or code paths, widens the blast radius and makes cascades more likely, because a single overloaded dependency can degrade every service that relies on it at once.

A particular danger under load is the retry storm, a case of the broader “thundering herd” problem. When a service slows down or returns errors, its clients tend to retry, and if many of them retry at the same time, the extra requests add to the very load that caused the errors, which produces more errors and more retries. This self-reinforcing behavior is the heart of what the reliability literature calls a metastable failure: a system sitting in a vulnerable but still working state is pushed by a temporary trigger into a degraded state that then sustains itself through a feedback loop, often an efficiency optimization running in reverse, such as retries or cache misses turning into database load, and stays there even after the original trigger is gone [10,11]. The point that matters most for our argument is that this literature treats the root cause as the sustaining loop and the vulnerable state, not the trigger. Because many different triggers can push a system into the same failure, outages are often blamed, misleadingly, on whatever event set them off [10]. That is the mistake we believe the “AI broke GitHub” story makes.

The distinction can be made operational with two questions that an incident record can answer. First, would a different trigger have produced the same failure? When faults of unrelated kinds, a migration, an expired certificate, a configuration change, a traffic surge, all spread in the same way, the pattern of spread belongs to the system rather than to any one trigger. Second, would the same trigger have produced a contained degradation in a system without the structural property at issue? When an operator’s own post mortem names the shared dependency or the missing control that turned a local fault into a platform-wide outage, the second question is answered by the primary source. Sections 4 and 5 apply both questions to GitHub’s record.

2.3. The Rise of AI and Agentic Traffic

The traffic reaching large internet platforms has changed in character, and three kinds of automated activity are worth keeping apart because they differ in volume and behavior. The first is AI coding assistants, such as GitHub’s Copilot, which generate and revise code and make frequent API calls on a user’s behalf. The second is AI crawlers, which harvest public content such as source code, documentation, and images to train models or answer user queries; their requests are dominated by training-data collection [12]. The third is AI agents, autonomous programs that navigate, decide, and act, whether by browsing, filling forms, opening pull requests, or completing transactions, at a pace and breadth no human matches. Crawlers read at very high volume. Agents interact rather than merely read, and for now they generate far less traffic than crawlers, though they are growing very fast [13]. A single agent working on one task may touch many more endpoints than a person would; an agent shopping for a product, for instance, may visit thousands of pages where a human would visit a handful.

Table 1 summarizes how the three classes differ along the dimensions that drive load, so that the rest of the article can keep them apart rather than aggregating them.

Table 1. Three classes of automated traffic and the dimensions that matter for availability. The classes differ in volume, timing, read/write mix, and cache behavior, and therefore call for different mitigations; the analysis keeps them distinct in Sections 5 and 8 rather than treating them as a single “AI load”. Sources: [12–16].

Dimension	AI Coding Assistants	AI Crawlers	Autonomous Agents
Primary purpose	Generate and revise code on a developer’s behalf	Harvest public content to train or serve models	Navigate, decide, and act to complete a task
Request volume	Moderate, per active developer	Very high; bulk reads	Lower today but growing very fast [13]
Temporal pattern	Tracks developer activity; partly diurnal	Bursty around training runs; non-diurnal [17]	Task-driven bursts; non-diurnal
Read/write mix	Mixed reads and writes (commits, pull requests)	Read-dominated	Mixed; many endpoints per task
Cache behavior	Mostly hot, popular content	Long-tail; defeats caches tuned for popular content [14]	Broad, unpredictable footprint
Matching mitigation	QoS priority for interactive traffic; per-client rate limits	Separate cache tiers; price or gate bulk access [18,19]	Cell isolation with shuffle sharding; verified identity [20–22]

The aggregate shift is large. Cloudflare, which observes a substantial share of web traffic, reported that automated systems came to generate the majority of requests to web content during 2026: its chief executive stated that, as of June 2026, roughly 57.5% of such requests were automated and 42.5% came from humans, the first such crossover in Cloudflare’s own measurement of requests to web content (a different operator, Imperva, dates an earlier crossover on a broader bot definition; see Section 8.1), one it had not expected before 2027 [6]. The figure concerns requests to web pages and is in part a leadership statement citing the company’s measurement dashboard, so we read the precise share as indicative rather than exact, though independent reports point the same way. The same operator’s year-in-review found that non-AI bots already exceeded human traffic to web pages at the start of 2025 by several percentage points, and that one category of AI “user-action” crawling grew more than fifteenfold over the year [12]. A separate security vendor measured autonomous-agent traffic growing by roughly 7851% year over year, and

reported that AI-driven traffic as a whole nearly tripled during the year [13]. The exact figures vary by source, but the broad picture is consistent: a large and fast-growing share of requests now comes from software rather than people, and that software does not keep human hours.

This shift matters for availability, and for a structural reason. Capacity planning and automatic scaling have long been tuned to human demand, which is diurnal and geographic, rising and falling with waking hours in each region and giving operators predictable daily peaks and troughs. Automated traffic keeps no such rhythm. A fleet of agents or crawlers can start at any time, from anywhere, and can aim itself at a platform during that platform's quietest local hours, at the very times when capacity has been scaled down and is slowest to recover. Automated load therefore tends to arrive at the least convenient moment and in the least predictable shape, as the incidents below repeatedly show.

2.4. GitHub as One Case Within an Industry-Wide Pattern

GitHub's experience is part of a broader pattern in which openly accessible services find that automated traffic overwhelms infrastructure built for human use. The Wikimedia Foundation reported that the bandwidth needed to serve multimedia grew by 50% from January 2024, driven not by human readers but by programs scraping its image catalogue to feed AI models, and that at least 65% of its most resource-intensive traffic came from bots even though bots accounted for only about 35% of page views, a disproportion that ate into the operational headroom it relies on to absorb genuine human surges [14]. The same wave of crawler traffic has strained other community-run hosts: the Fedora package repository blocked all traffic from Brazil, and one operator measured roughly 70% of the Diaspora network's traffic coming from LLM bots [23,24]. In open-source code hosting specifically, the service SourceHut reported blocking entire cloud providers because of the volume of crawler traffic coming from them [23], and an independent maintainer built a proof-of-work challenge system, later adopted by the KDE and GNOME projects, after aggressive AI crawlers repeatedly destabilized a Git service and slipped past conventional defenses by disguising their identity and rotating network addresses [24]. These cases share the shape of GitHub's: infrastructure provisioned for human-scale, human-paced demand, then put under sustained pressure by automated clients that behave nothing like human users. These cases establish that the pressure was industry-wide. Whether other platforms share GitHub's structural weaknesses is a separate question that their thinner public records do not answer: the comparison supports the premise about load, not the diagnosis about architecture.

3. Scope and Method of Analysis

The study is designed as an explanatory single-case study [15]: it asks how and why a contemporary episode unfolded, in a setting we do not control, using several kinds of evidence read against one another. GitHub serves as a revelatory case, because an acute decline coincided with unusually detailed public post mortems, giving access to material that is normally proprietary. It also serves as a critical case [25] for the claim that AI traffic destabilizes platforms: the claim should hold, if anywhere, where the AI surge was largest and best documented, so its failure here would require its qualification everywhere. The unit of analysis is the availability incident, and the case is bounded by the two years to mid-2026.

This article is built entirely from public, verifiable sources, with no access to proprietary GitHub data. Four kinds of evidence are used. The first is GitHub's own material: its monthly availability reports, which summarize each incident that degraded service [2]; the detailed root-cause posts published by its Chief Technology Officer in March and

April 2026 [4,5]; its real-time status page, which records individual incidents [2]; and its annual Octoverse activity reports [1,7]. The second is independent incident tracking, in the form of a third-party monitoring service that recorded GitHub's incidents continuously over a twelve-month window and published aggregate counts, severities, durations, and a comparison with competing platforms [3]. The third is operator disclosures and reputable engineering journalism describing comparable incidents at other services [14,15,17,18]. The fourth is the peer-reviewed and industry-standard reliability literature used to interpret the evidence, principally the dependability and metastable-failure literatures [8,10,11] together with the practitioner work on overload, isolation, autoscaling, and chaos engineering introduced in Sections 5–8.

The window of analysis is roughly the two years to mid-2026, with the closest attention on the period of acute decline from late 2025 to April 2026, where the public record is richest. We adopt each source's own definitions rather than imposing a new taxonomy, and we are explicit about which source a figure comes from, because the definitions differ. GitHub's availability reports count incidents that GitHub judges to have degraded performance; the independent monitor counts incidents it detected and labels an outage major when it caused significant disruption to a large number of users [3]. The two series are not directly comparable and are never combined, and where we report a count we name its source. Quantitative figures here are limited to those attributable to a dated source, and we have not estimated or interpolated missing values.

We also label every causal claim by where it comes from: GitHub-official where GitHub states it, third-party where an independent source reports it, and our inference where we draw a conclusion that the sources support but do not state outright. The labels function as a simple coding scheme, applied uniformly to every causal statement in Sections 4 and 5. The labelling is deliberate, because the central argument turns on the difference between what is observed, namely the incidents and their triggers, and what is interpreted, namely their structural causes.

Because the two series come from different observers, we treat their overlap as a cross-check on each. Where GitHub's own reports and the independent monitor both cover an incident, such as the October 2024 name-resolution outage and the multi-day Git-over-SSH degradation, we compare date, duration, and affected services and find them consistent, and we take GitHub's post mortems as the authority for causes while using the monitor only for independent frequency and severity. The direction of the residual bias is worth stating as well: a self-reported post mortem that concedes architectural coupling and missing safeguards works against the company's interest, so it is unlikely to overstate structural fault, while the monitor's detection thresholds bias toward under-counting rather than inflating the record. This is the calibration a public-evidence study can honestly perform without access to the operator's internal logs.

Four limitations follow from this design. First, GitHub's post mortems are self-reports written for a public audience, and a company describing its own failures chooses what to disclose and how to frame it. The risk is reduced, though not removed, by the direction of the admissions: conceding architectural coupling and missing safeguards is costlier for the company than blaming external load, so these are statements against interest. Second, the independent incident counts come from one monitoring service with its own detection thresholds, so they should be read as a single consistent series rather than as ground truth. Third, the counterfactual claims of Section 6, that named mechanisms would likely have contained particular incidents, cannot be tested directly; they rest on the mechanism descriptions in the engineering literature and on the match between each mechanism and the failure mode documented in the post mortems. Fourth, one case supports analytic

generalization to a class of systems, not statistical generalization, and Section 2.4 shows that other platforms faced similar load, not that they share GitHub's architecture.

4. GitHub's Availability Issues: Characterizing the Incident Record

4.1. Frequency, Severity, and Trend

The most direct evidence of decline is the incident record itself. Over the twelve months from May 2025 to April 2026, an independent monitor recorded 257 incidents at GitHub, of which 48 were classified as major outages, accounting for a combined 112 h and 18 min of major-outage downtime; across all tracked incidents the monitor reported an average mean-time-to-recovery of about 6 h [3]. The trend was not flat. Both minor and major incidents rose from December 2025 onward, February 2026 was the worst single month with 37 incidents, and February and April 2026 tied for the most major outages, at seven each [3]. Several individual outages were long by any standard. A degradation of Git-over-SSH in late October 2025 lasted roughly 47 h, and a domain-name-resolution failure a year earlier ran for more than 19 h, during which code search failed almost completely for about four hours [2,3]. GitHub's own monthly availability reports, which count incidents on a different basis, record for instance ten incidents in April 2025 and five in December 2025 [2]; the rising trend through the acute phase is clearest in the independent monitor's series [3]. Figure 1 places the principal incidents on a timeline, and Figure 2 sets GitHub's incident count against two comparable platforms over the same period.

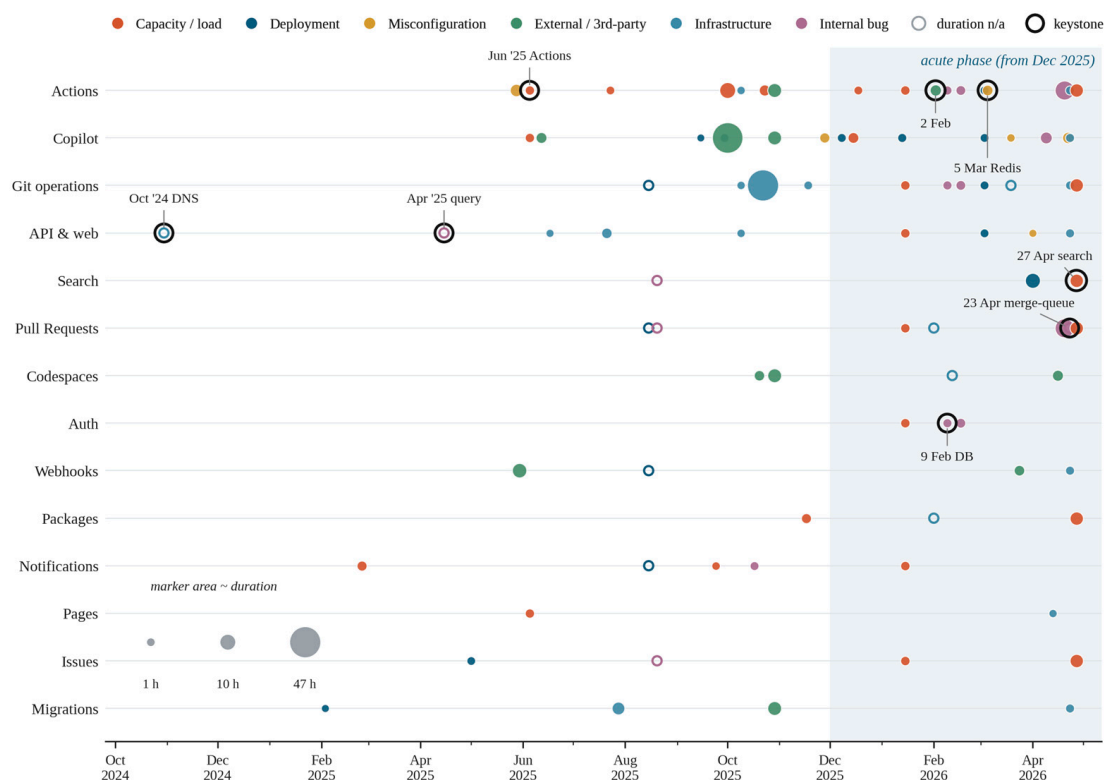


Figure 1. Timeline of GitHub's principal availability incidents, October 2024 to April 2026. Each marker is an incident, positioned by date (horizontal axis) and most-affected service (vertical axis); marker area is proportional to approximate duration, open markers denote incidents whose duration is not recorded, and color encodes the primary cause category. The shaded band marks the acute phase from December 2025 onward, and the ringed markers are the keystone events discussed in the text: the October 2024 DNS outage, the April 2025 expensive-query incident, the June 2025 Actions throttling incident, the 2 and 9 February 2026 incidents, the 5 March 2026 Redis-failover incident, the 23 April 2026 merge-queue corruption, and the 27 April 2026 search outage. Source: GitHub status page and availability reports [2]; independent tracking [3].

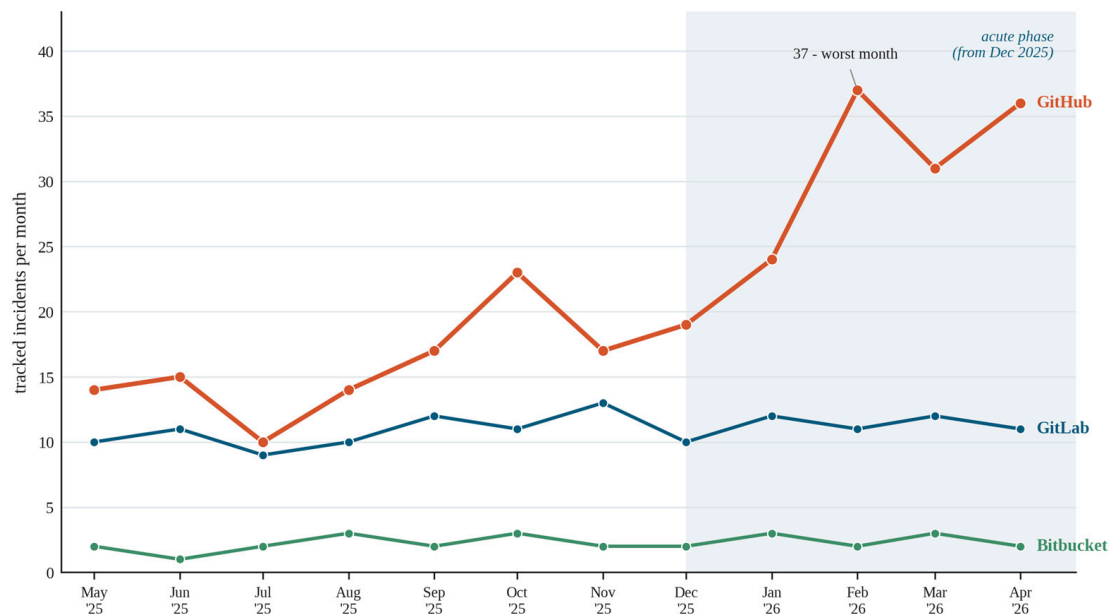


Figure 2. Monthly tracked incidents at GitHub, GitLab, and Bitbucket, May 2025 to April 2026, from one independent monitoring service. Counts are not normalized for platform size or traffic, so the figure supports only the modest claim that GitHub’s incident frequency was high in absolute terms during a period of acute strain. The shaded band marks the acute phase from December 2025 onward, over which GitHub’s monthly count climbed and peaked at 37 incidents in February 2026. Source: [3].

The decline shows up in relative terms as well. Over the same twelve months, the independent monitor recorded 257 incidents at GitHub against 132 at GitLab and 27 at Bitbucket [3] (Figure 2). This comparison has to be read with care, as the source itself notes. It does not adjust for the number of projects hosted or the traffic each platform carries, and GitHub is by far the largest of the three, so a higher absolute count is to be expected. The figure therefore supports only a modest claim, that GitHub’s incident frequency was high in absolute terms during a period of acute strain, and not a normalized reliability ranking.

4.2. Where GitHub Failed Most

The incidents were not spread evenly across the platform. The same tracking found GitHub Actions, the continuous-integration and deployment service, to be the most frequently affected, with 57 incidents over the year, followed by the Copilot AI features with 44, Issues with 28, Pull Requests with 27, core Git operations with 19, Webhooks with 17, Codespaces with 16, the API with 15, Packages with 10, and Pages with 9 [3] (Figure 3). Two patterns stand out. First, the services most central to automated software delivery, namely Actions and Copilot, were affected most often, which fits their heavy and growing use by both human and machine clients. Second, core Git operations, the one function a code host cannot do without, were affected on nineteen occasions, including a multi-day degradation of Git-over-SSH attributed to resource exhaustion on backend servers [3]. For a platform whose central promise is reliable access to code, that matters.

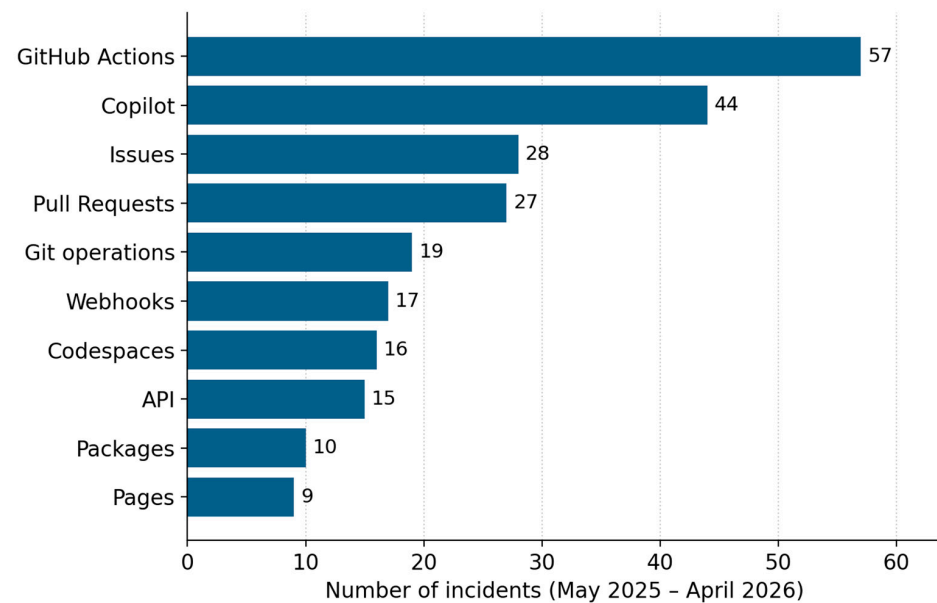


Figure 3. GitHub incidents by affected service, May 2025 to April 2026, as recorded by one independent monitoring service. A single incident can affect more than one service, so the counts are not mutually exclusive. Source: [3].

4.3. The Anatomy of Representative Incidents

Aggregate counts convey the scale of the decline. The individual post mortems show how it happened. We summarize several representative incidents, drawn from GitHub's own reports, because together they reveal a consistent structure: a trigger meets a shared or insufficiently protected dependency, and the failure spreads.

In October 2024, a database migration at one site caused the local domain-name-resolution infrastructure to start failing. Attempts to recover the database then set off cascading failures that widened the impact: for about four hours every code-search request failed, and a quarter of Actions workflows were delayed [2]. The migration was routine. What turned it into a long and wide outage was that the recovery actions interacted badly with a shared dependency.

A compute-intensive database query offers a clearer example. In April 2025, such a query prevented a critical database cluster from serving anything else, and because many services depended on that cluster's data, the damage was broad: Actions, the API, Pages deployments, and Git operations were all degraded [2]. The trigger was one expensive query. The breadth of the impact came from concentrating many services' fate in a single shared cluster.

June 2025 produced a small but pointed illustration of the argument made here. A spike in load between internal Actions services exposed a misconfiguration that throttled requests on the critical path for starting workflow runs. Some 47.2% of runs were delayed and 21% failed, and the trouble spread past Actions itself, cancelling 60% of Copilot coding-agent sessions and blocking every branch-based Pages deployment [2]. The load spike was the trigger. The failure and its spread came from a latent misconfiguration and from the coupling between Actions and the features that depend on it.

Two August 2025 incidents followed the same template. In one, a production database migration dropped a column that the application's data layer still referenced for some pull-request queries, which raised error rates across pushes, webhooks, notifications, and pull requests [2]. In the other, the search subsystem entered a degraded state, so that pages relying on search, such as issues, pull requests, and projects, failed to load correctly [2]. The proximate events were ordinary changes; their effects travelled through shared paths.

November 2025 brought two more. An expired internal certificate used for service-to-service communication caused all Git operations, over both SSH and HTTPS, to fail until the certificate was replaced [3]. In the same month the Dependabot service hit a rate limit in the container registry and could not finish more than half of its jobs on time [2]. The certificate incident shows how one shared security dependency can disable the platform's core function. The Dependabot incident shows an automated GitHub service overwhelming another GitHub service, a preview, internal to the company, of the misbehaving-client problem.

The acute phase began in early 2026. On 2 February, an Actions outage occurred when a gap in telemetry caused existing security policies to be applied, all at once and across every region, to key internal storage accounts. This blocked access to the metadata needed to create the virtual machines that run hosted jobs, and halted runner operations [4]. As GitHub noted, such infrastructure problems are normally confined to a few regions and routed around automatically; this time the failure defeated the regional isolation that would usually have contained it [4].

The keystone incident occurred on 9 February, 2026, and GitHub's own account of it is the most important single piece of evidence in this article. A core database cluster supporting authentication and user management became overloaded, and the mistakes behind it, in GitHub's telling, had been made days and weeks earlier [4]. Early that month, two very popular client applications that make heavy API calls were updated in ways that unintentionally raised the read traffic they generated more than tenfold; because such applications are adopted gradually as users upgrade, the rising load did not show up at once [4]. On 7 February, in order to ship a new model to a limited set of customers, GitHub shortened the refresh interval of a cache holding user settings from twelve hours to two, which increased write traffic; weekend load was low, and the monitoring in place was too coarse to reveal the danger [4]. On 9 February, the regular Monday peak, a wave of client-application upgrades, and another model release arrived together, and the combination of higher writes and elevated reads overwhelmed the cluster [4]. The decisive detail is architectural. That cluster also held user settings, including model-policy data that had grown over time from a few bytes to kilobytes per user, for reasons of historical simplicity, and because the same cluster held authentication and user-management data, every dependent service suffered [4]. GitHub added that, once the cluster was overwhelmed, it had no sufficiently granular control to block the offending traffic further up the stack [4].

Read closely, GitHub's account of 9 February draws the same line between trigger and cause that this article does. The triggers were external and AI-related: client applications generating far more requests, and new model rollouts. The causes were structural and already present: model-policy data placed in the authentication database, a cache change whose risk was invisible without better monitoring, and the lack of fine-grained controls to shed abnormal load. The trigger had grown only because of AI. The failure was possible only because of the architecture.

On 5 March 2026, an automated failover of a Redis cluster used to orchestrate Actions jobs ran as designed, but a latent configuration error left the cluster with no writable primary, so the mechanism meant to provide resilience could not be used to recover, and operators had to step in by hand [4]. GitHub described this not as an aggressive change or a missing safeguard but as latent configuration exposed by a routine production event, and concluded that it needed to rehearse its failover procedures in production more thoroughly [4].

Two April 2026 incidents close the record. On 23 April, a regression in the merge-queue feature caused squash-merges of groups containing more than one pull request to revert changes that had already been merged, leaving the affected default branches in an incorrect state. By GitHub's count 658 repositories and 2092 pull requests were affected;

no data was lost, since all commits remained in Git, but not every repository could be repaired automatically [5]. GitHub attributed the regression to an incomplete feature flag and inadequate test coverage, a process failure rather than a load problem [5]. On 27 April, the Elasticsearch subsystem that powers search across pull requests, issues, and projects became overloaded, which GitHub’s preliminary assessment put down to a likely botnet attack, and stopped returning results, so search-dependent parts of the interface showed nothing while Git operations and the APIs kept working [5]. GitHub said plainly that this subsystem was one it had not yet isolated as a single point of failure, because other work had ranked higher [5].

One shape recurs across these incidents. The triggers differ from case to case: migrations, an expired certificate, configuration changes, load spikes, a feature-flag bug, a probable attack. The reason each trigger produced a large or lasting outage does not differ. In every case there was a shared or insufficiently protected dependency, weak isolation between services, and too little ability to shed or contain abnormal load. The dates matter as much as the shape, and Section 5 returns to them: the earliest of these incidents preceded the acceleration of agentic traffic by more than a year.

4.4. Stated Causes in Aggregate

The pattern in the individual post mortems holds in aggregate, too. Classifying GitHub’s incidents over the year by their primary stated cause, the independent monitor attributed 83 to capacity, 71 to deployment, 31 to external dependencies, 30 to configuration, and 24 to internal causes such as software bugs, with the rest uncategorized or attributed to infrastructure and network problems [3] (Figure 4). Capacity, meaning the inability to serve the load presented, was the single largest category, and the source noted that in many cases capacity did not scale up on its own and needed manual intervention [3]. That is direct evidence for two of the three pre-existing weaknesses identified here: insufficient elasticity, and a high rate of change that interacted badly with the system, since deployment and configuration together account for more than a hundred incidents. These categories classify incidents by their stated cause, not by the origin of the traffic that exposed them, so the 83 capacity incidents are not, and are not offered as, a count of AI-caused outages; they are direct evidence that the platform repeatedly could not serve the load presented, whatever its source, which is the structural reading Section 5.2 develops rather than a claim about who generated the requests.

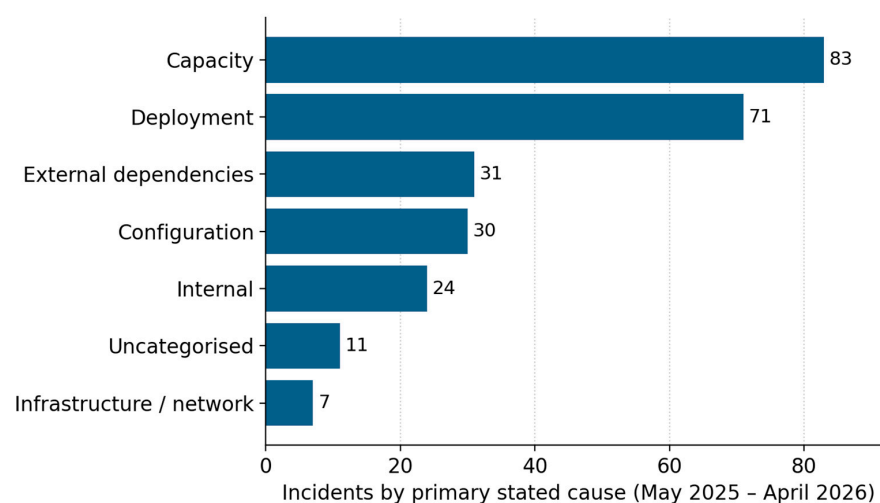


Figure 4. GitHub incidents by primary stated root-cause category, May 2025 to April 2026, as classified by one independent monitoring service from GitHub’s incident reports. Assigning a single primary cause to a distributed-systems failure is necessarily approximate. Source: [3].

4.5. What GitHub Tried

GitHub's response, as set out by its Chief Technology Officer, speaks directly to these weaknesses. In the near term the company reported redesigning the user-settings cache to hold far more data in a segmented database cluster, auditing the health of critical data and compute infrastructure, isolating key dependencies so that Actions and Git would not be harmed by shared-infrastructure problems, and adding protection for downstream components during spikes while giving priority to critical traffic [4]. Over a longer horizon it described moving off the MySQL-backed monolith by pulling out isolated services and data domains, shifting performance-sensitive code from Ruby to Go, moving webhooks to a different backend, and migrating to Azure for both larger machines and more regions, with a path toward several clouds [5]. In its own words the work comes down to "reducing hidden coupling, limiting blast radius" and making the platform degrade gracefully when one subsystem is under pressure [5]. That the remedies are decoupling, isolation, load shedding, and faster scaling is corroboration, though not proof, that the matching weaknesses were the causes, which is the subject of the next section.

5. Cause Versus Trigger

5.1. The Strongest Case That AI Caused the Decline

The view that AI caused GitHub's decline deserves to be put at its strongest before it is questioned. The timing fits: GitHub's leadership reports that agentic development workflows accelerated sharply from the second half of December 2025 [5], and the incident data show both minor and major outages climbing from that same month [3]. The mechanism is plausible: AI assistants, agents, and crawlers do increase request volume, and several of GitHub's own incidents were set off by such increases, among them the more-than-tenfold rise in read traffic from client applications in February 2026 [4], the probable botnet that overwhelmed search in April 2026 [5], and the broad growth in repository, pull-request, and API activity that GitHub names as the driver of its capacity crisis [5]. The scale is real as well: GitHub says it began planning for a tenfold increase in capacity and then found it needed to design for thirty times its then-current scale, which it attributes to the change in how software is being built [5]. On this view, a platform was simply overwhelmed by a historically large, AI-driven surge in demand, and no architecture could have been expected to absorb it. The argument is a serious one, and any honest analysis has to grant that the AI-driven surge was real, large, and the immediate occasion of many incidents.

5.2. Why the Surge Was the Trigger and Amplifier, Not the Root Cause

The argument still mistakes the trigger for the cause. The literature on metastable failure draws the distinction carefully. A system runs for a long time in a vulnerable but working state; a temporary trigger pushes it into a degraded state; and a self-sustaining feedback loop then holds it there even after the trigger has passed. Because many different triggers can push the same system into the same failure, this literature treats the sustaining loop and the vulnerable state, not the trigger, as the root cause [10,11]. By that measure, blaming GitHub's decline on AI is like blaming a building's collapse on the particular gust of wind that preceded it rather than on the structural fault that made collapse possible. The useful question is not what increased the load, but why increased load produced large, lasting, cascading failures instead of graceful, contained degradation.

GitHub's own statements answer that question. Its Chief Technology Officer wrote that the recent instability was driven by rapid load growth together with "architectural coupling that allowed localized issues to cascade across critical services" and an inability to shed load from clients that were behaving abnormally [4]. Those are not descriptions of too much demand; they are descriptions of design weaknesses that turned demand into

outages. GitHub's list of contributing factors goes further, naming insufficient isolation between critical-path components, inadequate safeguards for load shedding and throttling, and gaps in validation and early-warning monitoring [4]. Each of these was present before the surge and would have stayed a latent risk without it.

Chronology supports the claim of pre-existence directly. GitHub dates the sharp acceleration of agentic workflows to the second half of December 2025 [5]. The October 2024 incident, in which recovery actions cascaded through shared name-resolution infrastructure and code search failed platform-wide, occurred more than a year earlier [2]. The April 2025 incident, in which one expensive query on a shared cluster degraded Actions, the API, Pages, and Git operations at once, occurred eight months earlier [2]. The June 2025 incident, in which a latent misconfiguration spread from Actions to Copilot and Pages, occurred six months earlier [2]. These incidents carry the same structural signature as the failures of 2026, a localized fault crossing a shared or unprotected dependency, and they occurred when agentic load was a fraction of what it later became. The weaknesses cannot, therefore, be products of the surge. What changed from December 2025 onward was the frequency and force of the triggers, not the structure that received them.

The evidence points to three specific pre-existing weaknesses. The first is tight coupling that let localized faults cascade. The keystone February 2026 incident happened because user-settings and model-policy data sat, for historical convenience, in the same database cluster that serves authentication and user management, so overloading it degraded every dependent service [4]. The April 2025 expensive-query incident spread because many services shared one critical cluster [2]. The 2nd of February Actions outage spread because a change defeated the regional isolation that would normally have contained it [4]. GitHub's remedy, isolating Git and Actions from shared infrastructure and breaking the monolith into separate data domains [4,5], amounts to an admission that the coupling was the problem.

The second weakness is weak protection against anomalous or misbehaving client traffic. GitHub states that, once its database was overwhelmed in February 2026, it could not identify and block the offending traffic higher up the stack because it lacked sufficiently granular controls [4]. The April 2026 search outage came from an external client, a probable botnet, overwhelming a subsystem that had not been isolated [5]. The November 2025 Dependabot incident showed an internal automated client using up another service's rate limit [2]. Running through all three is the absence of effective admission control and load shedding, that is, the ability to recognize abnormal demand and refuse or deprioritize it before it does harm. This is the capability the standard literature prescribes for avoiding overload [25,26], and a retry-amplified version of the same dynamic is the textbook route into a metastable failure [10,27].

The third weakness is insufficient and slow elasticity. Capacity was the largest single category of stated cause, and in many incidents it did not scale on its own and needed manual intervention [3]. The February 2026 capacity crisis is the clearest instance: GitHub set out to raise capacity roughly tenfold, then concluded within months that it had to design for some thirty times its scale, the mark of a system that could not grow quickly enough to meet the load arriving [5]. GitHub's framing also captures the compounding behavior that turns a capacity shortfall into a cascade: at high scale, it notes, queues lengthen, cache misses turn into database load, and retries amplify traffic [5]. That is the same self-reinforcing loop that defines metastable failure [10] and the same latency amplification the "tail at scale" work describes when one slow dependency degrades a whole request path [28].

It helps to state the separation outright for the most serious incidents. For 9 February 2026, the proximate triggers were a tenfold, client-driven rise in reads and a cache-refresh change during a model rollout, while the structural cause was the colocation of model-policy

data in the authentication database together with missing fine-grained load controls [4]. For 5 March 2026, the proximate trigger was a routine Redis failover, while the structural cause was a latent configuration error that left no writable primary and failover procedures that had never been exercised in production [4]. For 27 April 2026, the proximate trigger was a probable botnet, while the structural cause was an un-isolated search subsystem that remained a single point of failure [5]. In each case the proximate trigger varied, while the structural weakness behind the impact stayed the same (see Table 2).

Table 2. Proximate trigger and structural cause for the principal documented incidents, the mitigation that would likely have contained each, and the source of the causal claim. The 23 April 2026 merge-queue incident is included for completeness; GitHub describes it as a process failure rather than a load failure [5]. Sources: [2–5].

Incident	Proximate Trigger	Structural Cause	Mitigation (Section)	Source of Causal Claim
Oct 2024 DNS outage	Database migration and the recovery actions that followed	Shared name-resolution and database dependency serving many services	Sections 6.1 and 6.2	GitHub-official (trigger); our inference (cause)
Apr 2025 expensive query	One compute-intensive query	Many services dependent on a single shared cluster	Sections 6.1 and 6.2	GitHub-official
Jun 2025 Actions throttling	Internal load spike	Latent misconfiguration on the critical path; coupling of dependent features	Sections 6.1 and 6.3	GitHub-official
2 February 2026 Actions outage	Telemetry gap applying security policies in all regions at once	Defeat of the regional isolation that normally contains such faults	Section 6.2	GitHub-official
9 February 2026 database overload	Tenfold client read growth; cache-refresh change; model rollout	Settings and model-policy data colocated with authentication; no granular load controls	Sections 6.1 and 6.3	GitHub-official
5 March 2026 Redis failover	Routine automated failover	Latent configuration error; failover never rehearsed in production	Section 8.3 (chaos engineering)	GitHub-official
23 April 2026 merge-queue corruption	Regression behind an incomplete feature flag	Inadequate test coverage and rollout safeguards (process, not load)	Section 8.3	GitHub-official
27 April 2026 search outage	Probable botnet	Un-isolated search subsystem, a known single point of failure	Sections 6.2 and 6.3	GitHub-official (preliminary)

One objection deserves a direct answer. GitHub reports that it planned for ten times its scale and then concluded it had to design for thirty times [5], and it can be objected that no architecture absorbs growth of that order, so that naming structural weaknesses is hindsight presented as analysis. The objection misreads the claim. The argument is not that GitHub should have served every request during the surge. It is that architecture decides the form overload takes. A platform with isolated cells, working admission control, and disciplined retries meets a surge by slowing down, shedding its lowest-priority work, and confining damage to a fraction of users. The documented incidents show the opposite form: platform-wide impact from localized faults, degradation that sustained itself after the trigger had passed, and recovery that needed manual intervention. The overload was unavoidable. Its form was not.

5.3. A Minimal Model of the Tipping Point

The distinction between trigger and cause can be given a compact quantitative form that matches the metastable-failure literature [10,11]. Consider a single shared resource, of the kind at the centre of the keystone incidents, with a maximum service rate C (requests

per second) and a base arrival rate λ from clients. Retries add to this: when a fraction f of requests fails or times out and each failed request is in turn retried, the load actually offered to the resource is not λ but the retry-amplified quantity

$$\lambda_{\text{eff}}(f) = \lambda / (1 - rf), \text{ valid while } rf < 1.$$

Here r is the mean number of retries a failed request generates and f is the fraction of offered requests that fail. The form is the closed sum of the whole retry cascade, in which the retries of failed requests may themselves fail and be retried again: the offered load is the geometric series $\lambda(1 + rf + (rf)^2 + \dots)$, which converges to $\lambda / (1 - rf)$ while $rf < 1$. The first-order estimate $\lambda(1 + rf)$ keeps only a single round of retries, whereas the closed form captures the compounding that makes a retry storm self-sustaining, and its amplification factor $1 / (1 - rf)$ grows without bound as the product rf approaches one.

The failing fraction f itself rises with utilization, because a resource driven past C sheds or delays work, so f and λ_{eff} reinforce one another. This coupling admits two stable equilibria. In the healthy equilibrium f is near zero, the amplification factor $1 / (1 - rf)$ is close to one, and the offered load λ_{eff} sits comfortably below C ; in the collapsed equilibrium f is high, the amplification factor is large, retries dominate, and λ_{eff} exceeds C so the queue never drains. The boundary between them is the failing fraction at which the amplified load just reaches capacity,

$$f_c = (1 - \lambda/C)/r.$$

found by setting $\lambda_{\text{eff}} = C$. A transient trigger, a load spike, an expensive query, a failover, is what carries the system across this boundary; while f stays below f_c the system relaxes back to health, but once the trigger drives f past f_c the amplified load exceeds C , which pushes f higher still, and the retry loop sustains the collapse after the trigger has gone. The tipping condition can be read straight off f_c : collapse becomes self-sustaining once the failing fraction is forced past the point where the retry budget r overwhelms the headroom ratio C/λ , and because a smaller headroom or a larger retry budget lowers f_c , either one brings the collapse within reach of an ordinary disturbance.

Two of the article's recommendations act directly on this condition. Disciplined retries, exponential backoff with jitter and a bounded retry budget, lower the effective r , which raises the critical fraction f_c and shrinks the amplification factor, so that overload yields slowdown rather than collapse [29]; admission control and load shedding cap λ_{eff} by refusing or deprioritizing work before the queue runs away [26,27]. Elasticity acts on the other term: reactive autoscaling raises C only after utilization crosses a threshold, and because a non-diurnal burst arrives when provisioned C is at its lowest, the lag drives C down toward λ exactly when λ is spiking, so f_c collapses toward zero and the smallest disturbance tips the system, whereas predictive, pre-warmed capacity holds C well above the amplified load through the surge [30]. The 9 February 2026 incident instantiates the model: the more-than-tenfold growth in client reads and the cache-refresh change from twelve hours to two raised λ against a fixed C , and because no sufficiently granular control could reduce λ_{eff} once f had risen [4], the system settled into the collapsed equilibrium and stayed there until operators intervened by hand. The model claims no fitted constants, which GitHub does not publish, but it makes the sustaining loop explicit and shows why one architecture yields the same collapse under many different triggers.

5.4. What Is Observed, What Is Reported, and What We Infer

We are explicit about where each claim comes from. That the incidents happened, with the triggers and impacts described, is GitHub-official and, where noted, corroborated

by third-party tracking [2–5]. That the recent instability involved architectural coupling, inadequate load shedding, and insufficient isolation is also GitHub-official, stated by its Chief Technology Officer [4,5]. Our own inference, which the sources support but do not put in these words, is the unifying claim: that these weaknesses predated the AI surge and are analytically prior to it, so that AI is properly classed as the trigger and amplifier of the decline rather than its root cause, and that the metastable-failure framework is the right lens for that classification [10,11]. We regard the inference as well founded for two reasons: GitHub’s own diagnosis and its own remedies point to the same structural causes, and incidents with the same structural signature predate the surge, as shown above.

6. What GitHub Could Have Done

If the causes were structural, the remedies are structural, too. This section sets out architectural changes that, on the evidence of GitHub’s own post mortems, would likely have contained the incidents described above. The aim is to draw out the design lessons, not to second-guess a large engineering team working under pressure. The claims are claims of containment, not prevention: none of these mechanisms would have stopped the triggers from occurring, and each is matched in Table 2 to the incidents whose blast radius or duration it would likely have reduced. It is also worth saying that each proposal matches work GitHub has since reported doing, an indication that the proposals are apt rather than speculative.

6.1. Decouple Critical Paths and Isolate Data Domains

The most direct lesson is that data and services with different purposes, and different blast-radius needs, should not share fate. The February 2026 keystone incident would have been far smaller had user-settings and model-policy data not lived in the authentication cluster: an overload of a settings store ought to degrade settings, not logins [4]. The broader principle is to separate the critical path, the small set of operations a code host must always perform, such as authenticating users, serving Git data, and orchestrating builds, from everything else, and to give each its own data and infrastructure so that a failure in one cannot starve the others. This is what GitHub now describes as breaking the monolith into isolated services and data domains and isolating Git and Actions from shared infrastructure [4,5]. A related discipline is to insulate the high-volume path that serves requests from the lower-volume path that manages configuration, so that a problem in management cannot bring down serving; the site-reliability literature has recommended this separation for years [26,28].

6.2. Contain the Blast Radius with Cell-Based Isolation and Shuffle Sharding

Decoupling reduces which services share fate. Cell-based architecture reduces how many users share fate within a service. In a cell-based design a service is deployed as many independent copies, called cells, each a complete instance with a fixed maximum size; incoming requests are routed to a cell by a partition key, and because cells share no state, a failure stays inside the cell where it happens while the others carry on [20]. For GitHub, partitioning repositories or organizations across cells would mean that an expensive query, a poisoned cache, or an overloaded database harmed only the fraction of customers in the affected cell rather than the whole platform, which would directly limit the kind of platform-wide impact seen in April 2025 and February 2026 [2,4].

Shuffle sharding extends this at very little cost. Rather than putting each customer in a single physical cell, the system assigns each customer to a small virtual shard, a randomly chosen combination of workers or cells, so that any two customers are very unlikely to share the same full combination. A problem caused by one customer, or aimed at one

customer, then reaches only the few who happen to overlap rather than everyone in a shard [21]. The canonical account observes that ordinary sharding already cuts a fault's reach from the whole fleet to a fraction, and that shuffle sharding does "exponentially better again," usually at no extra hardware cost, since it is a smarter arrangement of resources the operator already has [21]. The trade-off is real and worth stating. Cell-based isolation adds operational complexity, because there are many cells to deploy, watch, and migrate between, and it can raise fixed costs, because some resources have to be replicated per cell. The usual guidance is to bundle smaller cells, or to share fixed resources with care, so that the cost rises modestly while the blast radius still shrinks [20]. Figure 5 sets the failure behavior of a tightly coupled design beside that of a cell-based design using shuffle sharding.

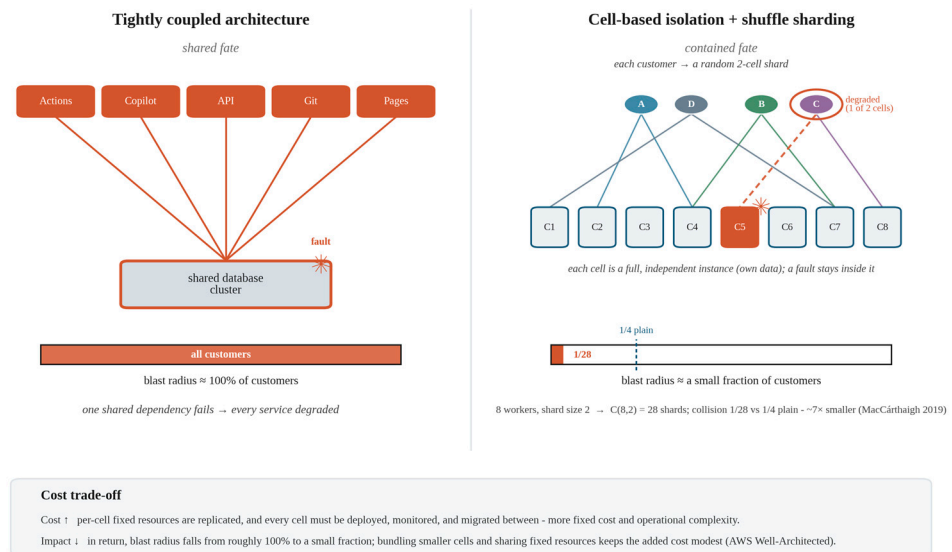


Figure 5. Blast-radius comparison: tightly coupled architecture versus cell-based isolation with shuffle sharding. On the left, a single shared dependency (one database cluster) feeds every service, so a fault propagates to all of them and the blast radius approaches 100% of customers. On the right, the same workload is partitioned into independent cells and each customer is mapped to a randomly chosen virtual shard of two cells, so a fault stays inside the affected cell and reaches only a small fraction of customers; with eight cells and shard size two there are $C(8,2) = 28$ possible shards, a collision probability of about 1/28 against 1/4 for plain partitioning, roughly seven times smaller. The lower panel summarizes the cost trade-off: per-cell fixed resources are replicated and every cell must be deployed, monitored, and migrated between, but in return the blast radius falls from roughly 100% to a small fraction. Source: synthesized from [20,21].

The trade-off can be made concrete. Table 3 shows, for a service partitioned into N cells with each customer assigned to a shuffle shard of k cells, how the number of distinct shards $C(N,k)$ grows and how the chance that two customers share an identical shard, an upper bound on how far one customer's fault can reach, falls. Even modest values are effective: eight cells with shard size two give twenty-eight shards and a collision probability near 3.6%, while sixteen cells with shard size three give 560 shards and a collision below 0.2%. The cost is per-cell fixed overhead, a control plane, monitoring, and a minimum replica set for each cell, so the practical design keeps cells few enough to amortize that overhead and many enough to bound the blast radius. At a scale of order 10^8 developers a few tens of cells still leave millions of users per cell, large enough to amortize fixed cost, while holding any single customer-scoped fault to a few percent of customers. Exact figures need capacity data GitHub does not publish, so these illustrate the shape of the trade-off rather than a budget.

Table 3. Blast-radius bound versus cell count for shuffle sharding. For N cells and shard size k there are $C(N,k)$ distinct shards; the collision probability, the chance that two customers are assigned an identical shard, upper-bounds the fraction of customers a single customer-scoped fault can reach. Larger N and k shrink the blast radius at the cost of more per-cell fixed overhead. Source: computed from the shuffle-sharding model of [21].

Cells N	Shard k	Shards $C(N,k)$	Collision Probability	Blast-Radius Bound
4	2	6	16.7%	~1 in 6
8	2	28	3.6%	~1 in 28
8	3	56	1.8%	~1 in 56
16	2	120	0.83%	~1 in 120
16	3	560	0.18%	~1 in 560

6.3. Recognize and Refuse Abnormal Load: Admission Control, Load Shedding, and Disciplined Retries

Isolation limits how far a failure spreads. Admission control and load shedding limit whether overload sets in at all. The idea is that a service under stress should protect its own health by refusing or deprioritizing work it cannot finish, instead of accepting everything and collapsing [25,26]. In practice that means rejecting low-priority requests first while protecting essential ones, among them the health-check traffic that keeps a service marked alive, since shedding those checks can shrink the very fleet needed to recover [25]. It also means spotting abnormal traffic from a small number of clients and throttling it at the edge, well before it reaches a fragile database, which is the capability GitHub said it lacked in February 2026 when it could not block the offending load higher up the stack [4]. Standard mechanisms for this already exist, from per-client rate limits and quotas to web-application firewalls and gateway throttling that can shed load along several dimensions [20,25]. Concretely, such a control can be stated as a short rule set: classify each request by client identity and priority; admit interactive and health-check traffic first; meter each client with a token bucket at its quota and, when a bucket empties, return HTTP 429 with a Retry-After header rather than queueing the request; and, once utilization crosses a high-water mark, shed the lowest-priority class first while never shedding the health checks that keep the fleet marked live [25].

A second discipline addresses the retry storms that turn a brief slowdown into a sustained collapse. When clients retry failed requests at once and in step, they add to the load that caused the failure. The established answer is exponential backoff with jitter, which lengthens the gap between retries and randomizes it so that clients do not retry in synchronized waves, paired with idempotent APIs so that safe retries do not cause duplicate side effects [29]. The same applies to GitHub's own automated clients and scheduled jobs, which should spread their work rather than bunch it [29]. Because retry amplification is one of the most common engines of metastable failure [10], disciplined retry behavior is one of the main defenses rather than a fine-tuning detail; combined with effective load shedding, it would have stopped several of the cascades described above before they spread.

6.4. Make Elasticity Predictive Rather than Reactive

Finally, capacity should anticipate load rather than chase it. Most automatic scaling is reactive: it adds servers after a monitored metric such as processor utilization crosses a threshold. Reactive scaling carries two built-in limitations that hurt under sudden load. Newly started servers cannot serve the requests already queued and failing, and provisioning is not instantaneous, so there is a window in which the system is underprovisioned [30]. Predictive, or proactive, elasticity instead forecasts demand and provisions capacity ahead of time, and the production literature recommends pairing a predictive component with a reactive safety net [30]. For a platform facing surges that do not follow the daily cycle, the case for prediction is stronger again: because agentic load does not follow the human

rhythm that reactive scaling quietly assumes, a system tuned only to react will keep being caught short at the moment a fleet bursts into a quiet region. Pre-warming capacity ahead of known model or client releases, several of which coincided with GitHub's incidents [4], is a concrete and low-risk way to apply the principle.

7. Related Work

This article draws on several settled strands of the reliability literature and connects them to a phenomenon, agentic traffic, that the peer-reviewed literature has only started to address. The conceptual base is the dependability framework of Avizienis and colleagues, which fixes the meanings of availability, reliability, and the faults, errors, and failures that threaten them [8]. The most directly relevant strand is the work on metastable failures by Bronson and colleagues, who gathered a range of long-known but superficially different outages, such as persistent congestion, retry storms, and death spirals, under one framework in which a trigger pushes a vulnerable system into a self-sustaining failure, and who argued that the sustaining feedback loop, not the trigger, is the real root cause [10]; later work catalogued such failures in production systems and studied the conditions under which they arise [11]. This framework is the backbone of the cause-versus-trigger argument.

Several practitioner literatures supply the mitigations. The handling of overload and cascading failure is covered in the site-reliability literature [26] and in detailed engineering accounts of load shedding [25] and of timeouts, retries, and backoff with jitter [27]. Fault isolation through bulkheads, cells, and shuffle sharding is documented in the cloud-architecture literature [28,30]. The way a single slow dependency amplifies latency across a large fan-out request is the subject of the well-known “tail at scale” analysis [29], which explains why GitHub's remark that one slow dependency can degrade several products is to be expected rather than surprising [5]. Autoscaling, including the limits of reactive scaling and the case for prediction, is studied in large production systems such as Google's Autopilot [21] and in a sizeable follow-on literature. Methods for verifying resilience before an incident, by deliberately injecting faults into production under controlled conditions, belong to chaos engineering, which Basiri and colleagues introduced and formalized [31] and which has been built into automated platforms that experiment on a small fraction of traffic [32].

What is comparatively thin is peer-reviewed work on the reliability implications of AI and agentic traffic in particular. The strongest current evidence on the scale and behavior of this traffic comes not from the academic literature but from infrastructure operators and security vendors: network-wide measurements of bot and AI-crawler traffic [12], reports of the crossover between automated and human requests [6], and vendor analyses of autonomous-agent growth [13], alongside operator post mortems of the kind that prompted this article [4,5,14]. This is an evidence gap rather than a settled field. The engineering community has good data on how automated traffic behaves and a mature theory of how systems fail under load, but little peer-reviewed work joining the two. This article is meant as one step toward closing that gap, by reading a well-documented case through established reliability theory, and it identifies the resilience of large platforms under agentic load as a promising and under-studied research area. The analytic sequence used here, reconstructing an incident record from public sources, coding causal claims by origin, applying the trigger-versus-cause criteria of Section 2.2, and mapping structural weaknesses to named mitigations, is not specific to GitHub and can be repeated for any platform that publishes post mortems.

8. Preparing for the Agentic Future

8.1. A New Baseline: Automated Requests Now Rival or Exceed Human Requests

The backdrop to everything that follows is a change in who, or what, uses the web. By the measurement of one major infrastructure operator, automated systems came to generate the

majority of requests to web content during 2026, at roughly 57.5% against 42.5% from humans, a crossover that arrived earlier than the same operator had predicted [6] (Figure 6). Even if the precise share is read as indicative, the direction is clear. On a broad definition that counts all automated traffic, one long-running industry measurement put the human/automated crossover earlier still, in 2024 [16]. Non-AI bots rivalled human traffic to web pages throughout 2025, leading early in the year before humans edged back ahead by December [12]; autonomous-agent traffic grew by roughly 7851% over the year [13]; and weekly agent requests on one network more than doubled within January 2026 alone [6]. Platforms can no longer treat automated clients as a minor share of demand to be handled when convenient. They are becoming the majority case, and they have to be designed for on purpose.

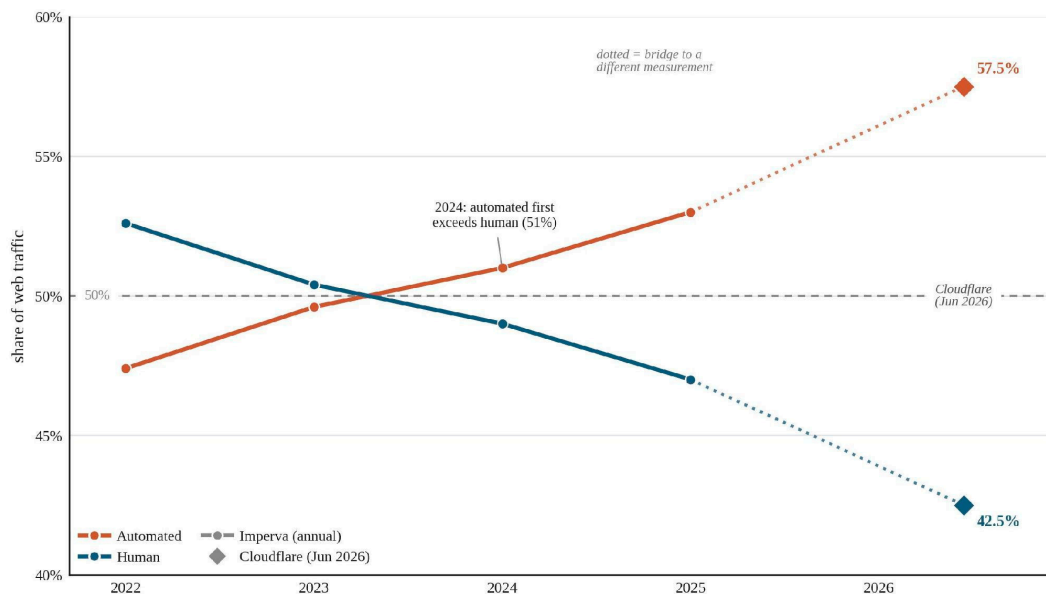


Figure 6. Share of web traffic from automated systems versus humans, 2022 to June 2026. The solid lines are one security vendor’s annual figures (Imperva), by which automated traffic first overtook human traffic in 2024 at about 51%; the dotted segments bridge to a single later data point from a second operator (Cloudflare), whose June 2026 measurement of requests to web content put the split at roughly 57.5% automated against 42.5% human. Because the two series measure related but not identical quantities, the dotted bridge marks a change of measurement and the precise values are best read as indicative of a crossover rather than as an exact, continuous trend. Source: [6,16].

8.2. Why Agentic Traffic Breaks Human-Era Assumptions

Agentic traffic breaks three assumptions on which platforms were built. The first is diurnality. Human demand rises and falls with local waking hours, which gives the predictable peaks and troughs that capacity planning and reactive scaling rely on. Automated fleets keep no such hours. They can start at any time and aim themselves at a region during its quietest local hours, when capacity has scaled down and is slowest to recover, hitting the system at its least-provisioned moment (Figure 7). The second assumption is a modest request volume per task. A person performs one action at human speed, whereas an agent may issue many requests to finish a single task, and a crawler may request a site’s entire long tail of rarely visited pages, which defeats caches tuned for popular content and forces expensive fetches from origin. This is the mechanism Wikimedia reported, in which bots produced a disproportionate share of its most resource-intensive traffic by requesting unpopular pages [14]. Crawler traffic is, moreover, dominated by training-data collection, which one operator puts at roughly 80% of AI-crawler requests and which arrives in irregular bursts rather than predictable cycles [17]. The third assumption is predictability of origin. A small number of organizations can decide, at any moment, to

train or deploy a new model or agent, and so create an unscheduled global spike that no single platform controls. That is the dynamic behind GitHub’s February 2026 incident, where client-application updates quietly multiplied read traffic [4].

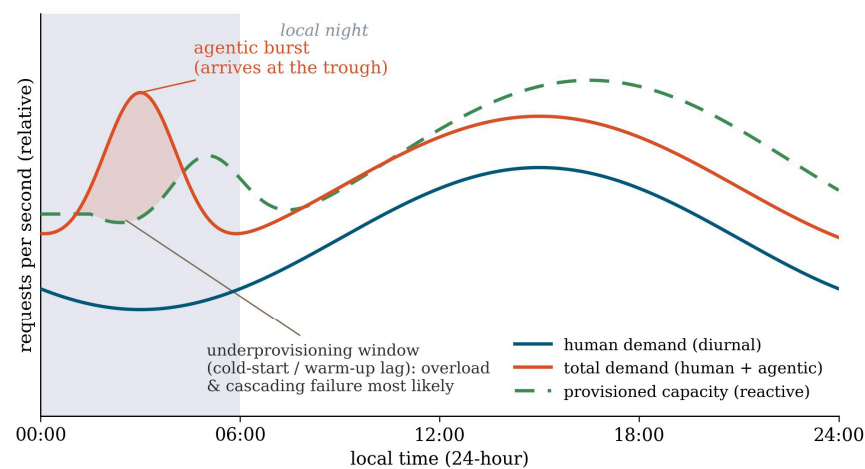


Figure 7. Human (diurnal) load versus total load once non-diurnal agentic traffic is added, and the autoscaling-lag problem. On a 24-h axis, a smooth diurnal human-demand curve (blue) is met by capacity that scales reactively with it (dashed green). Because agentic clients keep no human hours, total demand (red) stays above human demand at every hour, including the local night, and swells into a sharp burst during the night-time trough, when provisioned capacity is at its lowest and slowest to recover; the shaded gap between the burst and the lagging capacity curve is the window of underprovisioning (cold-start or warm-up lag) in which overload and cascading failure are most likely. Source: synthesized from [5,30].

8.3. Concrete Threats and Grounded Mitigations

We close the analysis by pairing particular ways that agentic traffic threatens availability with particular, named mitigations. Some extend the architectural measures of Section 6 into the forward-looking setting; others are specific to the economics and identity of automated traffic.

Consider first the threat of unscheduled, fleet-driven demand from a small number of actors. Because a handful of organizations can launch large crawls or agent fleets without warning, platforms need levers that price or gate automated access rather than only absorbing it. The most developed of these is the revival of the long-dormant HTTP “402 Payment Required” status as an access-and-payment signal. An operator can allow a crawler’s request, charge for it, or block it, returning a 402 with a price when payment is required, with the infrastructure provider acting as intermediary [19,33]. The same provider has moved to block AI crawlers by default, which makes access opt-in rather than opt-out [19]. For a code host, similar levers, such as charging or rate-limiting bulk programmatic access to public data while keeping ordinary developer use free, turn an uncontrolled cost into a managed one and create a signal that discourages the most abusive patterns. These are economic and policy instruments more than purely technical ones, and they work best alongside the technical controls below.

A second threat is agents and assistants reading and writing at machine rates. When automated clients push, pull, and search far faster than people, the isolation measures of Section 6 become preventive infrastructure rather than after-the-fact repair. Per-repository or per-organization cell-based isolation with shuffle sharding keeps one repository targeted by an aggressive agent, or one organization whose automation misbehaves, from degrading the platform for everyone [20,21]. The adaptation that agentic traffic specifically calls for is to design the partitioning so that the busiest automated workloads are spread across cells rather than concentrated within them.

A third threat is the mismatch between autoscaling and the daily cycle. As argued in Section 6, reactive scaling is structurally mismatched to load that does not follow human hours. The forward-looking answer is predictive elasticity that forecasts demand and provisions capacity in advance, backed by a reactive safety net [21], together with deliberate pre-warming of capacity ahead of expected model or client releases [4]. Because the triggering events, new model and agent launches, are often known to the platform or its ecosystem in advance, even rough forecasting can remove the cold-start window that turns a surge into an outage.

A fourth threat is retry storms and the amplification of search and indexing load. High-volume automated reads, especially to search and to long-tail content, can swamp indexes and caches, and the resulting errors can set off retry storms. The mitigations are admission control and load shedding that recognize and deprioritize abnormal automated demand before it reaches fragile subsystems [26], disciplined backoff with jitter on both client and server side [29], and, for crawler load in particular, routing automated reads to separate cache tiers sized for breadth and tolerant of somewhat higher latency, so that bulk crawling does not evict the hot content human users rely on. A major operator has proposed exactly this split between caches for automated and for human traffic [18], and GitHub's April 2026 search outage shows why an un-isolated, unprotected search subsystem is dangerous under such load [5].

A fifth threat is machine demand crowding out human users when capacity is scarce. Under contention, not all traffic deserves equal treatment. A quality-of-service scheme that sorts requests into classes, with interactive human traffic ranked highest, then essential automation, then bulk or speculative agentic and crawler traffic, lets a platform degrade gracefully by slowing or shedding the lowest class first while protecting human users and critical automation (Figure 8). This is the constructive side of load shedding [26,27], shedding the least important work first rather than simply shedding work. It depends on being able to tell the traffic classes apart reliably, which is the subject of the identity problem below.

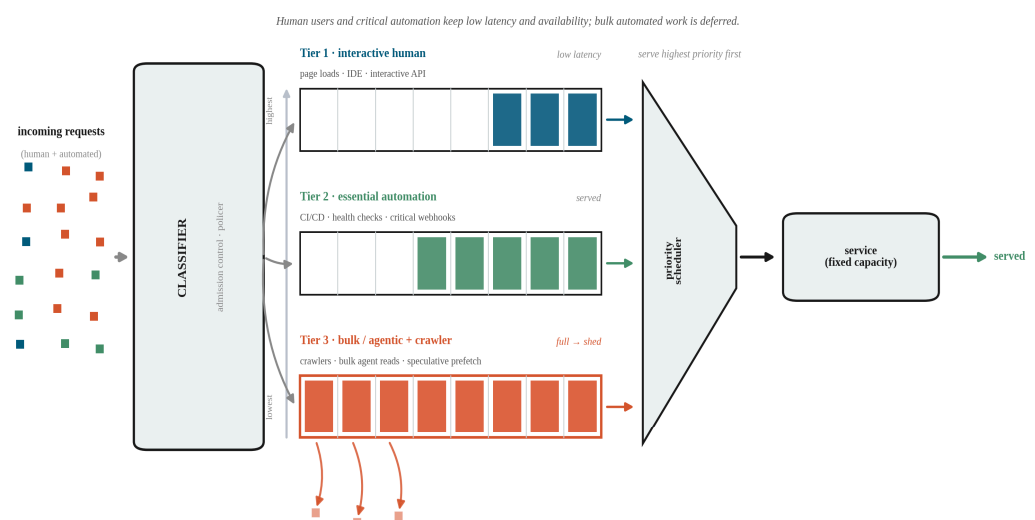


Figure 8. Quality-of-service tiering of human and automated traffic under contention. A classifier and admission-control policer sort incoming requests into tiers: interactive human traffic (Tier 1, highest priority), then essential automation such as CI/CD and health checks (Tier 2), then bulk or speculative agentic and crawler traffic (Tier 3, lowest). A priority scheduler drains higher-priority queues first, so as load approaches the fixed service capacity the lowest tier fills and is shed first (tail-drop or HTTP 429), preserving latency and availability for human users and critical automation. Source: synthesized from [26,27].

A final threat is latent faults that surface only under real failure. Several of GitHub's most damaging incidents were latent problems exposed by routine events, such as a failover that left no writable primary, security policies misapplied across regions, and a feature flag with too little test coverage [4,5]. The discipline that brings such faults to light on purpose, before users do, is chaos engineering: stating a hypothesis about steady-state behavior and then injecting controlled faults, including the loss of a dependency, a regional failure, or a simulated traffic surge, on a small fraction of production traffic, to confirm that the system degrades gracefully [32,34]. GitHub's own conclusion that it needed to rehearse failover procedures in production more thoroughly [4] is in effect a call for this practice. Extending such drills to cover agentic-style load surges and misbehaving-client scenarios would test the very conditions this article flags as dangerous.

8.4. Identity and Trust: Telling Agents Apart

Every mitigation that treats automated traffic differently from human traffic depends on being able to tell them apart, and that is getting harder. Operators note that the old binary of bot or human is breaking down as agents navigate, fill forms, and transact in ways that look human [13], and that ill-behaved crawlers already slip past defenses by disguising their identity and rotating network addresses [24]. The emerging response is verified agent identity, meaning cryptographic signatures that let well-behaved automated clients prove who they are, so that platforms can apply pricing, prioritization, and rate limits accurately instead of guessing [19]. Standardization of cryptographic agent identification is under way: dedicated work at the Internet Engineering Task Force on web bot authentication [22] builds on the HTTP message-signatures standard [35] to let automated clients sign their requests, and the pace of that work will set how soon such controls can operate at internet scale. The requirement cuts both ways. It lets the quality-of-service, economic, and load-shedding controls above work without harming legitimate human users, but it also puts a great deal of weight on standards and infrastructure for agent identification that are still immature. Building platforms for the agentic web [36] therefore means investing not only in isolation, elasticity, and load shedding, but in the means to tell apart the traffic those mechanisms are supposed to govern.

9. Conclusions

GitHub's availability fell over roughly two years, and sharply in early 2026, and the urge to blame artificial intelligence is understandable, since AI-driven demand was the immediate occasion of many incidents. The evidence, much of it from GitHub itself, points elsewhere: AI was the trigger and the amplifier of the decline, not its root cause. The instability was made possible by design weaknesses that were already in place, namely tight coupling that let localized faults cascade, weak protection against anomalous and misbehaving client traffic, and elasticity that could not keep up with sudden load arriving outside the daily human cycle. Incidents bearing the same structural signature occurred well before agentic traffic accelerated, which places the weaknesses before the surge in time as well as in logic. GitHub's own Chief Technology Officer described the instability in those terms, and the company's remedies, namely decoupling, isolation, load shedding, and faster scaling, amount to an implicit confirmation that the matching weaknesses were the causes [4,5]. The literature on metastable failure supplies the principled basis for the distinction: many triggers can drive a vulnerable system into the same self-sustaining failure, so the structural vulnerability, not the trigger, is the cause [10,11].

The design lessons are not new; they are well-established resilience principles that the surge made urgent. Each of the three structural weaknesses has a direct remedy:

- Tight coupling that let localized faults cascade is met by decoupling critical paths and isolating data domains, so that a settings-store overload cannot take down authentication as it did on 9 February 2026, and by cell-based isolation with shuffle sharding, which confines an expensive query, a poisoned cache, or an aggressive client to a fraction of users rather than the whole platform.
- Weak protection against anomalous or misbehaving client traffic is met by admission control, load shedding, and disciplined retries, which recognize and refuse abnormal load and halt retry storms before a cascade spreads.
- Insufficient and slow elasticity is met by predictive, pre-warmed capacity that closes the cold-start window non-diurnal surges exploit.

None of these mechanisms would have prevented the triggers, but each would likely have reduced the blast radius or the duration of specific documented incidents.

The wider point is that GitHub is an early, well-documented case of a general condition. By at least one major measurement, automated requests have come to outnumber human requests on the web, and that traffic keeps no human hours, respects no single platform's capacity plan, and can be launched at will by a small number of actors [6]. Designing for this world takes more than extra capacity. It takes architectures that contain failure by default, controls that recognize and refuse abnormal load, elasticity that anticipates instead of reacting, economic and policy levers that price and gate bulk automated access, and the means to verify resilience and to tell automated traffic apart before it has to be governed. The platforms that do well in the agentic era will be the ones that treat resilience, rather than capacity, as the first problem to solve. The argument made here can be tested as other platforms publish their own records, and the mitigations of Sections 6 and 8 can be evaluated under controlled agentic-load experiments; both are tasks for the research community as much as for operators. That, in the end, is the lesson GitHub has drawn in public, and other openly accessible services would be wise to draw it before their own surge arrives rather than after.

Author Contributions: Conceptualization, V.-Ş.D., R.R., E.R. and D.T.; methodology, V.-Ş.D. and R.R.; investigation, V.-Ş.D.; formal analysis, V.-Ş.D. and R.R.; writing, original draft preparation, V.-Ş.D. and R.R.; writing, review and editing, V.-Ş.D., R.R., E.R. and D.T.; visualization, V.-Ş.D.; supervision, R.R. and D.T.; funding acquisition, R.R. and E.R. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by UEFISCDI and National University of Science and Technology POLITEHNICA Bucharest through project PN-IV-P7-7.1-PTE-2024-0505.

Data Availability Statement: This study uses publicly available sources. The incident reports, status pages, blog posts, industry reports, and technical documents analyzed in the article are cited in the reference list.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AI	Artificial intelligence
API	Application programming interface
CI/CD	Continuous integration and continuous deployment
DNS	Domain Name System
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
LLM	Large language model
SSH	Secure Shell

References

1. GitHub. Octoverse 2025: A New Developer Joins GitHub Every Second as AI Leads TypeScript to #1. *The GitHub Blog*, 28 October 2025. Available online: <https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/> (accessed on 10 June 2026).
2. GitHub. GitHub Availability Report (Monthly Series) and GitHub Status. *The GitHub Blog* and Githubstatus.com, 2024–2026. Available online: <https://github.blog/tag/github-availability-report/> (accessed on 6 June 2026).
3. Barua, H. GitHub Outages 2025–2026: Reliability Analysis and Outage History. *The IncidentHub Blog*, 30 April 2026. Available online: <https://blog.incidenthub.cloud/github-reliability-outage-history-2025-2026> (accessed on 6 June 2026).
4. Fedorov, V. Addressing GitHub’s Recent Availability Issues. *The GitHub Blog*, 11 March 2026. Available online: <https://github.blog/news-insights/company-news/addressing-githubs-recent-availability-issues-2/> (accessed on 6 June 2026).
5. Fedorov, V. An Update on GitHub Availability. *The GitHub Blog*, 28 April 2026. Available online: <https://github.blog/news-insights/company-news/an-update-on-github-availability/> (accessed on 3 June 2026).
6. Tom’s Hardware. ‘Bots Have Now Passed Human Traffic Online,’ Cloudflare Boss Laments. Available online: <https://www.tomshardware.com/tech-industry/artificial-intelligence/bots-have-now-passed-human-traffic-online-cloudflare-boss-laments-says-agentic-traffic-wasnt-expected-to-eclipse-real-people-until-next-year> (accessed on 14 July 2026).
7. GitHub. Octoverse 2024: The State of Open Source and Rise of AI. *The GitHub Blog*, 29 October 2024. Available online: <https://github.blog/news-insights/octoverse/octoverse-2024/> (accessed on 2 June 2026).
8. Avizienis, A.; Laprie, J.-C.; Randell, B.; Landwehr, C. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.* **2004**, *1*, 11–33. [CrossRef]
9. Vogel, E.; Dyka, Z.; Klann, D.; Langendörfer, P. Resilience in the Cyberworld: Definitions, Features and Models. *Future Internet* **2021**, *13*, 293. [CrossRef]
10. Bronson, N.; Aghayev, A.; Charapko, A.; Zhu, T. Metastable Failures in Distributed Systems. In Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS’21), Ann Arbor, MI, USA, 1–3 June 2021; pp. 221–227. [CrossRef]
11. Huang, L.; Magnusson, M.; Bangalore Muralikrishna, A.; Estyak, S.; Isaacs, R.; Aghayev, A.; Zhu, T.; Charapko, A. Metastable Failures in the Wild. In Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI’22), Carlsbad, CA, USA, 11–13 July 2022; pp. 73–90.
12. Cloudflare. The 2025 Cloudflare Radar Year in Review. *The Cloudflare Blog*, 15 December 2025. Available online: <https://blog.cloudflare.com/radar-2025-year-in-review/> (accessed on 10 June 2026).
13. HUMAN Security. The 2026 State of AI Traffic & Cyberthreat Benchmark Report. HUMAN Security, 2026. Available online: <https://www.humansecurity.com/learn/resources/2026-state-of-ai-traffic-cyberthreat-benchmarks/> (accessed on 1 June 2026).
14. Wikimedia Foundation. How Crawlers Impact the Operations of the Wikimedia Projects. *Diff (Wikimedia Foundation)*, 1 April 2025. Available online: <https://diff.wikimedia.org/2025/04/01/how-crawlers-impact-the-operations-of-the-wikimedia-projects/> (accessed on 1 June 2026).
15. Yin, R.K. *Case Study Research and Applications: Design and Methods*, 6th ed.; SAGE Publications: Thousand Oaks, CA, USA, 2018.
16. Imperva (Thales). 2025 Imperva Bad Bot Report; Imperva: San Mateo, CA, USA, 2025. Available online: <https://www.imperva.com/resources/resource-library/reports/2025-bad-bot-report/> (accessed on 2 June 2026).
17. Cloudflare. A Deeper Look at AI Crawlers: Breaking Down Traffic by Purpose and Industry. *The Cloudflare Blog*, 28 August 2025. Available online: <https://blog.cloudflare.com/ai-crawler-traffic-by-purpose-and-industry/> (accessed on 2 June 2026).
18. Cloudflare. Why We’re Rethinking Cache for the AI Era. *The Cloudflare Blog*, 2 April 2026. Available online: <https://blog.cloudflare.com/rethinking-cache-ai-humans/> (accessed on 2 June 2026).
19. Cloudflare. Introducing Pay per Crawl (Content Independence Day). *The Cloudflare Blog*, 1 July 2025. Available online: <https://blog.cloudflare.com/content-independence-day-no-ai-crawl-without-compensation/> (accessed on 2 June 2026).
20. Amazon Web Services. Use Bulkhead Architectures to Limit Scope of Impact (Cell-Based Architecture), REL10, AWS Well-Architected Framework, Reliability Pillar. Available online: <https://docs.aws.amazon.com/wellarchitected/> (accessed on 4 June 2026).
21. MacCárthaigh, C. Workload Isolation Using Shuffle-Sharding. Amazon Builders’ Library. Amazon Web Services, 2019. Available online: <https://aws.amazon.com/builders-library/workload-isolation-using-shuffle-sharding/> (accessed on 2 June 2026).
22. Meunier, T.; Major, S. HTTP Message Signatures for Automated Traffic: Architecture; Internet-Draft Draft-Meunier-Web-Bot-Auth-Architecture-05; IETF, March 2026; Work in Progress. Available online: <https://datatracker.ietf.org/doc/draft-meunier-web-bot-auth-architecture/> (accessed on 4 June 2026).
23. The Register. AI Crawlers Haven’t Learned to Play Nice with Websites (SourceHut). 18 March 2025. Available online: https://www.theregister.com/2025/03/18/ai_crawlers_sourcehut/ (accessed on 3 June 2026).
24. Edwards, B. Open Source Devs Say AI Crawlers Dominate Traffic, Forcing Blocks on Entire Countries (the Anubis Proof-of-Work System; KDE and GNOME). *Ars Technica*, 25 March 2025. Available online: <https://arstechnica.com/ai/2025/03/devs-say-ai-crawlers-dominate-traffic-forcing-blocks-on-entire-countries/> (accessed on 1 June 2026).

25. Flyvbjerg, B. Five Misunderstandings About Case-Study Research. *Qual. Inq.* **2006**, *12*, 219–245. [CrossRef]
26. Yanacek, D. Using Load Shedding to Avoid Overload. Amazon Builders' Library, Amazon Web Services. Available online: <https://aws.amazon.com/builders-library/using-load-shedding-to-avoid-overload/> (accessed on 9 June 2026).
27. Beyer, B.; Jones, C.; Petoff, J.; Murphy, N.R. (Eds.) Handling Overload; Addressing Cascading Failures; Managing Load. In *Site Reliability Engineering: How Google Runs Production Systems*; O'Reilly Media: Santa Rosa, CA, USA, 2016.
28. Dean, J.; Barroso, L.A. The Tail at Scale. *Commun. ACM* **2013**, *56*, 74–80. [CrossRef]
29. Brooker, M. Timeouts, Retries, and Backoff with Jitter. Amazon Builders' Library, Amazon Web Services. Available online: <https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/> (accessed on 9 June 2026).
30. Rzacca, K.; Findeisen, P.; Świdorski, J.; Zych, P.; Broniek, P.; Kusmieriek, J.; Nowak, P.; Strack, B.; Witusowski, P.; Hand, S.; et al. Autopilot: Workload Autoscaling at Google. In Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20), Heraklion, Greece, 27–30 April 2020. [CrossRef]
31. Orosz, P.; Nagy, B.; Varga, P. Real-Time Detection and Mitigation Strategies Newly Appearing for DDoS Profiles. *Future Internet* **2025**, *17*, 400. [CrossRef]
32. Basiri, A.; Behnam, N.; de Rooij, R.; Hochstein, L.; Kosewski, L.; Reynolds, J.; Rosenthal, C. Chaos Engineering. *IEEE Softw.* **2016**, *33*, 35–41. [CrossRef]
33. Cloudflare. AI Crawl Control: Custom HTTP 402 Responses for AI Crawlers (General Availability). *Cloudflare Developer Changelog and Blog*, 27 August 2025. Available online: <https://developers.cloudflare.com/changelog/2025-08-27-ai-crawl-control-launch/> (accessed on 4 June 2026).
34. Basiri, A.; Hochstein, L.; Jones, N.; Tucker, H. Automating Chaos Experiments in Production. In Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'19), Montreal, QC, Canada, 25–31 May 2019; pp. 31–40. [CrossRef]
35. Backman, A.; Richer, J.; Sporny, M. *HTTP Message Signatures*; RFC 9421; IETF: Reston, VA, USA, 2024. [CrossRef]
36. Duan, Q.; Lu, Z. AI Agent Communications in the Future Internet—Paving a Path Toward the Agentic Web. *Future Internet* **2026**, *18*, 171. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.