ChatGPT

Five Foundational Principles for Autonomous Infrastructure and AI Agents

1. No Manual Core Infrastructure

Theory: Human intervention in core operations is the classic source of toil and error. Site Reliability Engineering (SRE) explicitly seeks to **eliminate "toil"** – repetitive, manual work – so that systems are self-driving and reliable. As the SRE handbook puts it, *"if a human operator needs to touch your system during normal operations, you have a bug."* [1]. In practice this means **automating all routine tasks** (deployments, scaling, configuration) and managing them via code. Automation not only frees engineers for higher-level work, but also **increases consistency and reliability**. Google SRE has demonstrated that *removing humans from the release process* ("Push On Green") *"can paradoxically reduce toil while increasing system reliability"* [2]. In other words, fully automated pipelines often yield fewer errors than ad-hoc manual fixes.

Cases: The industry has increasingly embraced GitOps and Infrastructure-as-Code to achieve this. For example, GitOps treats **Git as the single source of truth for all configuration** [3], and employs continuous deployment so that any desired state in Git is automatically realized. One guide notes that GitOps *"overwrites any configuration drift, such as manual changes or errors, ensuring the environment maintains the desired state defined in Git."* [3]. In practice, tools like ArgoCD or Flux continuously reconcile cluster state from version-controlled manifests. A Terrateam example describes an end-to-end GitOps workflow: "*everything flows from Git… There are no manual steps, no kubectl commands, and no drift between Git and reality*" [4]. Likewise, major cloud providers now publish GitOps patterns (e.g. Weave GitOps on AWS) that **ban manual console changes** and automate all provisioning. In SRE practice, sites like Google even aim for "**push-on-green**" deployment with zero human touch [2]. Case studies (e.g. at Google and Meta) consistently report that *"taking humans out of the release process"* leads to faster, safer rollouts [2].

Alignment: This principle aligns with classic DevOps goals (repeatability, traceability) and with Zero Trust ideals: by removing manual work, human error and ad-hoc shortcuts are eliminated. It also dovetails with policy-as-code – when all changes go through version control, every action is auditable. In the CNCF's vision of autonomous infrastructure, *"self-governing"* systems *"enforce policies instantly and consistently without human intervention"* [5]. No-manual infrastructure is thus consistent with self-governance: policies and configurations live in code, not in people's heads. It also supports security: if infrastructure is only changed via pipeline, it can be vetted, tested and rolled back systematically.

Refined v1: A concrete formulation of this principle might be: **"All production infrastructure and agent pipelines must be defined declaratively and managed via automated CI/CD; no direct manual changes."** For version 1, that means: store every cluster/VM/container definition in Git, require pull requests for any change, and enforce review/testing. Build a release pipeline that *only* takes code from trusted repositories. Avoid any "wizard/console mode" access to the core control plane. In short, move the human off the button: if an engineer thinks "I should SSH in and fix this", instead we refine the process so that the fix is a code change triggered through the normal CI/CD pipeline.

Control Plane Hooks: In the control plane (e.g. Kubernetes API server, Terraform Cloud, etc.), enforce this by intercepting changes. For example, use admission controllers or policy engines (like OPA/Gatekeeper) to **block any mutation** not originated from the GitOps pipeline. Implement webhooks or controllers that watch for manual actions and immediately revert or alert. For instance, if someone uses `kubectl apply` directly, a hook could immediately replace the resource with the last Git-committed version. Likewise, CI pipelines should automatically apply code changes and run sanity checks; humans should only affect the system via merging commits. Build integration tests into the control plane so that any change outside of Git fails validation.

Risks: Over-automation introduces its own risks. A mis-configured pipeline can wreak havoc quickly (e.g. a bad config could auto-deploy and break everything). Because humans no longer intervene manually, failures must be caught by monitoring and automated rollbacks. Relying entirely on code also means mistakes get versioned and replayed unless there are proper tests. Furthermore, if pipelines or Git repositories are compromised, an attacker can push malicious changes at machine speed. To mitigate these, it's crucial to have **high confidence in pipelines** (through thorough testing) and to maintain a robust audit trail of all commits/deploys. There's also an organizational risk: teams may feel the loss of control or experience slower turnaround if they must go through PR processes for every change. Balancing agility and safety requires good testing, canary deployments, and well-defined rollback plans.

2. Declarative, Versioned Infrastructure (GitOps/IaC)

Theory: Closely related to eliminating manual work is the idea of **"everything as code"**. Infrastructure-as-Code (IaC) means all network, server, and cluster configurations are expressed in declarative code, stored in version control, and treated like software. GitOps takes this further by using Git *as the single source of truth* for system state [3]. In a GitOps model, changes to infrastructure happen by committing code: pull requests are reviewed and then merged, and automated pipelines apply those changes. This enforces consistency, auditability, and repeatability. The Splunk guide describes GitOps as *"IaC using Git as the version control system for all infrastructure configurations"*, where *"everything is defined as code and stored in Git"* [3]. Declarative definitions prevent configuration drift because the system continuously reconciles to the declared state; the GitOps workflow *"overwrites any configuration drift, such as manual changes or errors"* [3].

Cases: Real-world usage of GitOps/IaC abounds. Teams using Flux or ArgoCD report that any divergence between Git and the live cluster is automatically detected and fixed, so stray manual edits never persist [3] [4]. For example, one deployment team noted that with Terraform + ArgoCD, *"there are no manual steps… and no drift between Git and reality"* [4]. Platforms like AWS, Azure, and Google Cloud all encourage IaC (CloudFormation, ARM templates, Deployment Manager) and integrate with CI/CD. Weaveworks' GitOps best practices recommend that *"Infrastructure evolves only through pull requests; nothing is manually applied"* (Weaveworks GitOps Principles). In practice, teams that switch to GitOps/IaC see faster provisioning and far fewer unexpected outages. In one case, Cloudera reported *95% automated provisioning* using AI-driven GitOps in 2025 [6].

Alignment: A declarative, versioned model aligns well with reliability and security. It creates an immutable history of changes (good for auditing and rollback) and tightly couples code review to infra changes. It also underpins other principles: with IaC in Git, it becomes trivial to eliminate manual steps (#1), verify external changes (#3) (since all changes come via pull requests), and apply policy as code. From an SRE perspective, IaC is often cited as the foundation for reproducibility: an SRE book notes that infrastructure should grow without linear toil – ideally a service can scale *"by at least one order of*

magnitude with zero additional [manual] work" [7] . Declarative IaC is how that goal is met: once written, configurations scale automatically.

Refined v1: Concretely, the system should adopt a policy like: **"All infrastructure (networks, compute, clusters, databases, etc.) must be defined in code and managed via Git. No changes are allowed outside of the version-controlled pipeline."** Version 1 of this might mean splitting configuration into well-organized repositories (e.g. a Terraform repo, a Helm charts repo). Use branching and pull-request workflows for all updates. Ensure that all config files are tested by pipeline (e.g. syntax/lint checks, static analysis). Handle secret management via sealed secrets or Vault, also driven by code commits. In short: if it runs in production or hosts AI agents, its configuration has a Git history.

Control Plane Hooks: The control plane (e.g. the Kubernetes API server, Terraform Cloud, cloud provider APIs) must be hooked into this workflow. For example, use GitOps controllers (ArgoCD, Flux) that watch Git and continually apply the desired state. Set up webhooks so that when a pull request is merged, CI jobs automatically deploy the new configuration. Conversely, use admission webhooks to **reject any out-of-band change**. Cloud orchestration services often provide mechanisms (e.g. AWS Config rules or Kubernetes OPA Gatekeeper) to enforce that resources match the declared spec. Essentially, the control plane should be **read-only** unless changes originate from the Git pipeline.

Risks: Heavy reliance on GitOps has pitfalls. A single bad merge can propagate widely, so strong testing (unit, integration, and canary deployments) is crucial before going to production. There is also a risk of "GitOps fatigue": long pipelines and PR cycles can slow emergency fixes. Large monorepos of YAML/TF can become unwieldy. Secret leakage is another concern – embedding secrets in code is dangerous, so secure secret management must be integrated. Moreover, rollbacks may not be trivial if stateful data is involved (e.g. DB schema changes need special care). Finally, demanding that everything be code may impede ad-hoc debugging or exploration (teams need tools to inspect running state without bypassing Git policies). Mitigations include having robust change validation in CI, staging environments, and disaster recovery playbooks.

3. Always Verify External Actions

Theory: Autonomous agents must not be blindly trusted. An agent's external actions (API calls, commands, data fetches, etc.) can have far-reaching effects. Classic Zero Trust wisdom applies: *"never trust, always verify."* Every action by an AI agent should be checked against rules and validated. This addresses issues like hallucinations, injections, or malicious outputs. For example, generative LLMs are known to "hallucinate" plausible-sounding but false information. If an agent takes such hallucinated output as truth and acts on it, the consequences can be severe. Insecure output handling is even recognized by OWASP's GenAI Top 10: *"Neglecting to validate LLM outputs may lead to downstream security exploits"* [8] . In practice, this principle means implementing rigorous *post-decision safety checks*: every plan or external action generated by the agent should be verified by a higher-level system or policy engine.

Cases: There are many cautionary examples. One Microsoft reference architecture for LLM agents (hotel-booking scenario) emphasizes identity and audit: *"Each action is authenticated, authorized, and audited, enforcing 'never trust, always verify' at every step."* [9] . This shows each agent action is checked like a human's request. OWASP also highlights overreliance on LLMs as a vulnerability: *"Failing to critically assess LLM outputs can lead to compromised decision making, security vulnerabilities, and legal liabilities."* [10] . Concrete failures have occurred: in an AI red-team test, an agent hallucinogenically fabricated a bogus rule ("*All orders over $1,000 get automatic refunds*"), saved it to its knowledge base, and then automatically approved thousands of illegitimate refunds [11] – a clear disaster of unverified

output. In cybersecurity research, hidden malicious instructions embedded in web content have fooled agents into unsafe behaviors [12]. These examples underline that without verification, agents can propagate mistakes or attacker-crafted data.

Alignment: This principle ensures trustworthiness. It complements least-privilege (Principle 4) and tiered trust (Principle 5) by adding an active safety layer. It's aligned with regulatory and compliance needs as well: logged and validated actions can be audited, reducing liability. From an AI-safety standpoint, always verifying ties into feedback loops and redundancy: treat the agent's decisions as proposals, not commands. It also aligns with SRE practices: in critical systems we always validate actions (e.g. health checks, canaries, approvals). Here, the "external action" is any change or side-effect of the agent, and we treat it as an untrusted input to the real system.

Refined v1: Practically, implement this by **interposing a validation layer**. For example, require that any command or API call proposed by the agent be sent to an external "verifier" (itself a programmatic check, another model, or human) before execution. Use content filters, pattern matchers, or semantic analysis (as in LlamaGuard or Azure Content Safety) to catch bad outputs. Define clear guardrails for each tool or action: e.g. if the agent attempts to delete resources, the control system should flag it. In software terms, one might wrap every tool call in a policy check or "dry-run" mode that simulates the result and checks for anomalies. In a CI pipeline, any changes a tool makes (e.g. a pull request it opens) must pass automated tests and/or manual review. The refined rule might be: **"No external action by an agent is carried out without confirmation against a trusted policy or oracle."** This could start as automated rules (e.g. regex-based filters) and evolve to include human-in-the-loop for high-risk actions.

Control Plane Hooks: Implementing this in the control plane means adding **input/output validation and logging hooks** at every interface. For instance, if an agent calls an external API, route that call through an API gateway with schema validation and rate limiting. If it requests to run code, first run the code in a sandbox or a linter. One can also use *"interceptors"* or *"middlewares"* in the control plane: whenever the agent's command hits a service, it passes through a security policy engine (e.g. OPA, CASBIN) that either permits or denies based on rules. Additionally, log all such actions centrally and analyze logs for anomalies. Essentially, treat agent outputs as untrusted requests to your control plane: enforce authentication, validate the request body, and require approvals if it violates a policy.

Risks: Adding verification adds latency and potential bottlenecks. Overly strict filters may block legitimate actions (false positives), slowing down the agent. Agents might learn to game the checks if not careful (adversarial prompts). There's also a coverage risk: it's hard to anticipate every malicious or nonsensical output, so the validation logic itself can have blind spots. Under-scrutiny is dangerous (missed threats), over-scrutiny is annoying. To mitigate, validation should be multilayered (e.g. use both rule-based and ML-based detectors) and continually updated as new risks appear. Finally, misuse of this principle can lead to excessive alarm fatigue: too many alerts or checks could desensitize the team. It is important to focus verification on *high-impact actions* (e.g. financial transactions, code deployment) and monitor outputs with thresholds or confidence scores, rather than hard-block every single statement.

4. Principle of Least Privilege

Theory: Least privilege means **give each agent only the permissions it absolutely needs**, no more. In a multi-agent system, agents can execute powerful actions (e.g. modifying infrastructure, accessing sensitive data), so over-permissive rights are dangerous. The Zero Trust principle explicitly warns that agents should have *"narrowly defined permissions."* In practice, this might mean using strict IAM roles, short-lived tokens, or sandboxing so that an agent can't e.g. delete databases or read unrelated data. As one expert put it: if an agent is designed to read sales data, *"it shouldn't be able to write to billing records*

or access HR systems." [13] . Likewise, another source notes *"if an agent's purpose is to book travel, it should have no access to financial reporting systems"* [14] . This principle sharply limits the "blast radius" of a compromised or malfunctioning agent.

Cases: Many LLM-agent guidelines emphasize least privilege. The Zero-Trust prompt article lists least privilege as a core bullet: e.g. *"Agents should only receive narrowly defined permissions needed to carry out their assigned tasks"* [14] . In real deployments, organizations are starting to use techniques like token scoping and open authorization. For example, the Microsoft/WSO2 hotel agent example issues each agent its own OAuth token with minimal scope [9] . In Kubernetes setups, one might use Pod Security Policies or Kubernetes RBAC so that even if a container escapes, it cannot escalate privileges. Case studies of breaches often highlight over-permissive service accounts as the root cause, underscoring why least privilege is vital.

Alignment: Least privilege works hand-in-hand with trust and verification. If an agent can only do what it needs to do, then even if an external action slips past verification, the damage is contained. It also aligns with micro-segmentation and defense-in-depth: agents are confined to "their lane." This helps meet compliance requirements (data-minimization, separation of duties). From an SRE angle, least privilege reduces risk of user error or conflicting changes. It means that failures in one agent won't cascade to unrelated services.

Refined v1: A refined rule might be: **"Define strict role permissions for each agent based on its job; do not grant blanket or root credentials to any agent."** Version 1 could involve inventorying all agent types and their actions, then writing IAM policies or ACLs that grant only the needed actions. Use ephemeral credentials: e.g. agents get temporary tokens just-in-time for a task, which expire immediately after. For example, if an agent needs to query a database, give it a SQL user that can only read certain tables and nothing else, with credentials that rotate frequently. Implement a robust secrets management system (like HashiCorp Vault) so agents obtain credentials on-demand with fine-grained scopes. Never bake credentials into the agent; instead, dynamically provision them when validating a task.

Control Plane Hooks: Enforce least privilege through the control plane by integrating with identity and access systems. For instance, hook into the orchestrator's RBAC: if using Kubernetes, define a minimal ServiceAccount and Roles for each agent deployment. Use admission controllers that verify an agent pod isn't requesting extra privileges. In cloud APIs, rely on IAM policies – the control plane should deny any API call from an agent identity that it isn't authorized for. Some platforms offer continuous policy engines (e.g. AWS IAM Access Analyzer) which can be configured to alert on any unexpected permission use. In general, any attempt by an agent to exceed its designated scope should trigger an automatic rollback or alert at the control plane.

Risks: Too-tight permissions can hamper functionality. If a policy is too restrictive, the agent may fail to accomplish its task (and there needs to be a process to adjust it). This can lead to frustrated developers or cascading failures if multiple agents depend on each other. Also, the complexity of maintaining many fine-grained roles can be high; mistakes in policy definitions can create security gaps or operational breakage. There's a trade-off between convenience and security: highly granular policies require more governance. To manage this, one can start with coarse roles and iteratively tighten them while monitoring failures, or use tools like principle-of-least-privilege automation engines. Finally, simply having many low-privilege accounts might create overhead in tracking them – it's important to have a central inventory and regular audits of agent permissions.

5. Tiered Trust and Human Oversight for High-Risk Actions

Theory: Not all actions by an agent carry the same risk. The final principle is to **tier trust and include human oversight for critical operations**. Routine, low-impact tasks (e.g. fetching weather data, generating draft text) can be fully automated. But *high-stakes actions* (e.g. deleting or reconfiguring resources, moving money, releasing to production) should require extra scrutiny. This is akin to a "dual-control" model: as one blog puts it, *"routine, low-risk tasks can be automated freely. High-risk operations… should require human-in-the-loop approval or multi-factor triggers."* [15] . In other words, establish gates: an agent's proposal must be confirmed by a human (or a separate high-trust system) before proceeding when it crosses a risk threshold.

Cases: The concept of graduated autonomy is gaining traction. The CNCF blog on autonomous infrastructure defines levels L1–L5, where "L3 copilot" agents always require human approval for significant changes, and even "L4 autopilot" systems operate within strict predefined boundaries [16] . Practically, many orgs are adopting a 2-phase commit: for example, an agent might automatically spin up a test environment, but only create live resources after a human clicks "Approve." The Token Security guide (BleepingComputer) explicitly recommends *"tiered trust models"* and *"scoped tokens and short-lived credentials,"* and notes that *"high-risk operations… should require human-in-the-loop approval or multi-factor triggers."* [15] . Enterprises often implement this by having separate "dev" vs "prod" roles: an agent might operate unsupervised in dev, but prod changes are gated through ServiceNow/approvals.

Alignment: Tiered trust ensures safety without killing automation. It allows confidence-building: as you trust the agent more (through testing and monitoring), you can automate more. It also dovetails with compliance and accountability: having a human reviewer provides an audit point and legal responsibility. For SRE, this is similar to canary releases and gradual rollouts where major changes are gated by checks. It keeps the human in loop for *exceptional* cases while letting routine tasks run at machine speed. Importantly, it signals to stakeholders that the system isn't "letting the AI run wild": there is always a path to stop or correct.

Refined v1: Implement this by **classifying actions by risk level** and encoding clear escalation policies. For example, give each agent request a "sensitivity score" and require supervisor approval if above a threshold. Version 1 might be to define a list of critical operations (e.g. `destroy cluster`, `modify security policy`, `exfiltrate data`) that automatically open a ticket or Slack message for human approval. For configuration changes, integrate a manual review step in the CI pipeline for high-impact merges. Use multi-factor authentication for elevated commands – e.g. even if an agent has an API key, issuing a deletion command might require a signed 2FA code. The refined principle could be: **"No critical action is final without human sign-off; automate fully only after passing risk checks."**

Control Plane Hooks: In practice, hook your workflow manager or CI/CD to require approval flows. For instance, use Jenkins pipelines with "input" steps for risky deployments, or configure Terraform Cloud's run triggers to pause before apply. Attach labels or metadata to operations so the control plane knows which require manual intervention. In Kubernetes, you could use an admission webhook that blocks pods annotated "requires-approval" until a gatekeeper flips a switch. Some platforms (AWS, GCP) offer conditional approvals (e.g. a IAM permission that allows resource creation only if a Cloud Function returns "ok"). Essentially, the control plane should enforce "stop points" – operations above a risk bar are not executed automatically, and are routed to human approvers or safelisted processes.

Risks: The main downside is latency and human burden. Agents may stall waiting for approval, reducing agility. Teams might over-classify actions as "high-risk" and defeat automation. Human approvers can become a bottleneck or point of friction. There is also the potential issue of mistrust: if

humans override too many agent actions, the system may degrade back into manual mode. To balance this, it's important to **continuously refine the boundaries** – start with conservative gating and gradually automate more as the agent proves itself. Tools that allow easy delegation of approval (mobile apps, chatbots) can reduce friction. Finally, if not well logged, the approval process itself could become a vector (attacker could try to hijack an "approve" workflow), so that too should be monitored.

Sources: In compiling these guidelines we drew on SRE literature and modern agent research. For example, Google's SRE book on toil [1] and DevOps discussions on automation [2] emphasize eliminating manual ops. Industry best-practices on GitOps/IaC [3] [4] inform our infrastructure-as-code stance. Security experts from OWASP and token security outline the need for verification and least privilege [9] [10] [17]. The CNCF and practitioners detail autonomous infrastructure levels and trust pillars [5] [16]. We encourage readers to consult these sources for deeper examples and to validate any implementation.

---

[1] [7] Google SRE - What is Toil in SRE: Understanding Its Impact
https://sre.google/sre-book/eliminating-toil/

[2] Google SRE Principles: SRE Operations and How SRE Teams Work
https://sre.google/sre-book/part-II-principles/

[3] What is GitOps: The Beginner's Guide | Splunk
https://www.splunk.com/en_us/blog/learn/gitops.html

[4] End-to-End GitOps: Terraform for Infrastructure, ArgoCD for Application Delivery
https://terrateam.io/blog/argocd-terraform-integration-guide-for-end-to-end-gitops

[5] [6] [16] Why Autonomous Infrastructure is the future: From intent to self-operating systems | CNCF
https://www.cncf.io/blog/2025/10/17/why-autonomous-infrastructure-is-the-future-from-intent-to-self-operating-systems/

[8] [10] OWASP Top 10 for Large Language Model Applications | OWASP Foundation
https://owasp.org/www-project-top-10-for-large-language-model-applications/

[9] [14] The Zero-Trust Prompt: Re-thinking Identity in the Age of LLM Agents | by YUSUFF ADENIYI GIWA | Medium
https://medium.com/@adeniyi221/the-zero-trust-prompt-re-thinking-identity-in-the-age-of-llm-agents-88583f144c1b

[11] [12] 15 Security Threats to LLM Agents (with Real-World Examples)
https://research.aimultiple.com/security-of-ai-agents/

[13] [15] [17] Extending Zero Trust to AI Agents: "Never Trust, Always Verify" Goes Autonomous
https://www.bleepingcomputer.com/news/security/extending-zero-trust-to-ai-agents-never-trust-always-verify-goes-autonomous/