

Answer Key — Ticket-to-PR Agent

Sandbox Design

Reference design · self-check for the homework

The point is not matching this design word-for-word. The point is that every policy below is concrete — it names a mechanism, not an aspiration — and that the escape-budget answer assumes the model *is* compromised, not that it might be.

1. Network policy

Default deny. The container sits on an internal-only Docker bridge (`internal: true`) with no outbound route of any kind. The only network access the agent has is through a small allowlist proxy that also lives on that internal bridge, which permits exactly two destinations: the ticket-system API host and the git remote host. Both are pinned by hostname and port. No raw DNS, no general internet, no package mirrors, no model API from inside the agent container — if the agent needs to call the LLM, that call happens in a separate, smaller container that also talks only to the proxy. A prompt-injected agent on this network cannot exfiltrate data, cannot beacon, cannot pull a second-stage payload, because the kernel refuses every destination except the two in the allowlist.

2. Filesystem policy

The only read-write paths inside the container are `/workspace`, which holds the per-run repo checkout, and `/tmp`, which is a tmpfs for scratch space. Both are backed by ephemeral storage that is destroyed at the end of the run. Everything else in the container is read-only: the base image, the tool binaries, the config. Secrets are never written to disk at all — they are passed in as short-lived environment variables at container start and never serialised. No host bind mounts. No access to `/home` outside `/workspace`. No access to the host's SSH keys, docker socket, or kubeconfig. If the agent wants to persist something across runs, it goes through git, not through the

filesystem — which means it also goes through the PR review described in the escape-budget section.

3. Capability policy

`cap_drop: - ALL`, and nothing in `cap_add`. The agent runs as a non-root user with `no-new-privileges:true`. Git and the test runner are the only workloads inside the container, and neither of them legitimately needs elevated Linux capabilities — git talks over HTTPS (or SSH through the proxy), and the test runner runs ordinary user-space tests. If the test suite needs something unusual (docker-in-docker, a privileged syscall), that is a signal to redesign the workload to not need it, not a signal to add a capability. Every capability is a bypass of the other layers, and the default list for this agent is empty.

4. Secret scoping

Two tokens, both minimally scoped and both rotated per run. The git token is a fine-grained personal access token (or GitHub App installation token) scoped to a single repository, with permissions limited to "read contents, write contents, write pull requests" — no admin, no secrets, no workflows. The ticket API token is scoped to a single project in the ticket system, with read-issues and comment-on-issues permissions only. Both tokens are minted at the start of each run from a host-side secret broker, injected as environment variables at `docker run` time, and invalidated the moment the container exits — either by calling the issuing API's revoke endpoint, or by using native short-lived credentials (15 minute TTL) that expire on their own. At no point does a long-lived credential exist inside the agent's process space.

5. Escape budget

Assume the agent is fully prompt-injected and does exactly what an attacker tells it to do. The worst case is: the agent opens a hostile pull request against the single repository the git token has write access to, using the bot account. That is the entire blast. It cannot reach the internet, so it cannot exfiltrate source code; it cannot write outside `/workspace`, so it cannot persist on the host; its tokens expire in minutes and are scoped to one repo and one project, so a stolen token is worth little by the time it is stolen. The PR itself is contained by branch protection: the bot is not a code owner, PRs from the bot account require human review from a real engineer before merge, branch protection enforces that rule at the repo level, and the bot cannot approve its own PR or bypass the check. The worst-case outcome is therefore "a human reviewer sees a bad

PR and rejects it," which is an event the review pipeline already handles every day for non-compromised contributors. That is a budget you can live with.