Project Proposal Narrative- Hybrid Cloud-Bursting for coffee-project

- Smeet Nagda

- Jinish Shah

- Nisarg Jasani

Problem Statement & Description

The problem. Many enterprises still run the majority of their workloads on on-premises servers. During traffic spikes (e.g., seasonal sales, class registration, product launches), these fixed-capacity servers come under peak workload and cannot handle the requests. Scaling on-prem capacity quickly is expensive and slow. More importantly buying more hardware for short windows leads to poor utilization and wasted capital. Also during this time, manual interventions which are required to restart or scale services causes burnout to engineers and introduces risk of human error.

Why it matters. when systems can't handle sudden demand, everything slows down. Users get frustrated, requests start failing, and that frustration eventually turns into lost revenue or SLA penalties. Meanwhile, ops teams spend their time putting out fires instead of actually improving reliability or scaling smarter. A solution that can automatically absorb those traffic spikes without throwing away existing on-prem investments changes that equation. It keeps things running smoothly when demand surges, protects user experience, and gives engineers their time back to focus on long-term reliability instead of constant crisis management.

Our solution. We're planning to build an automated hybrid cloud-bursting pipeline that keeps performance steady without wasting resources. Most of the time, the app will run on our enterprise servers in Docker. But when the system detects sustained pressure, say CPU stays above 70% for five minutes or the queue depth crosses a limit it'll automatically add temporary capacity in the cloud. The same container image will launch on ECS Fargate or Kubernetes (we haven't decided yet), the load balancer will update to include those new instances, and once demand drops, the cloud side will scale back down to zero. Everything will run through GitHub Actions and Ansible, with policy-as-code ensuring tests and linting pass before any deployment kicks off. The idea is to make scaling invisible, safe, and hands-off for the ops team.

Why a pipeline? We're planning to build a continuous pipeline that makes every decision and automation step repeatable, visible, and easy to audit. The idea is to tie metrics-driven triggers like Alertmanager or webhooks to infrastructure-as-code tools such as Ansible and cloud CLIs, so there's no need for ad-hoc manual fixes.

The pipeline will react automatically to key events, like a code push to a release branch or a runtime alert, while still keeping human checks where they matter such as protected branches, required reviews, and clear approval gates.

Tagline. *"Scale when it matters, save when it doesn't."*

Primary Use Case

Use Case: *PR to release builds, tests, and deploys and runtime alerts trigger automatic cloud-bursting.*

1. **Preconditions**
    - On-prem Docker host(s) provisioned and reachable via Ansible.
    - Cloud account (e.g., AWS) with ECS Fargate service (or managed K8s) created, initially scaled to 0.
    - GitHub repository mirrors coffee-project at /<team-repo>/coffee-project with branch protections.
    - GitHub Actions secrets configured: AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, AWS_REGION, ONPREM_SSH_KEY, REGISTRY_TOKEN, etc.
    - Prometheus + Alertmanager (or CloudWatch Alarms) posting webhooks to a GitHub Actions endpoint.
2. **Main Flow**
    1. Developer opens a PR targeting release/x.y.
    2. **CI** runs lint + unit tests; image is built and pushed to GHCR with the PR SHA tag.
    3. Release Engineer approves PR branch protection requires checks to pass.
    4. Merge to release/x.y triggers **Continuous Deployment** which deploys the tagged image to on-prem Docker via Ansible; updates LB to point to on-prem service.
    5. When traffic spikes, Alertmanager sends a webhook to **GitHub Actions**. That triggers a "Burst" job, which scales up the cloud service to N tasks and adds them to the load balancer's target group.

6. When load subsides for a cooldown window, a **Unburst** job scales cloud tasks back to 0 and removes them from the LB.

3. **Subflows**
    - [S1] PR creation automatically requests Release Engineer review.
    - [S2] ci.yml executes: lint → test → build → push.
    - [S3] cd-release.yml runs Ansible playbook to update on-prem service and health-check.
    - [S4] burst.yml handles scale up/down on cloud based on alert payload.
4. **Alternative Flows**
    - [E1] Lint fails → PR marked with failing status; merge blocked.
    - [E2] Tests fail → PR failing; merge blocked.
    - [E3] On-prem deploy fails health checks → automatic rollback to previous tag; issue opened.
    - [E4] Cloud burst scale-out fails → on-call notified via Actions/Slack; on-prem continues to serve.

Pipeline Design

High-Level Architecture

- **Developer**
  Owns feature work, creates PRs, writes tests, and responds to review feedback. Uses branching strategy (feature/*, release/*). Ensures commits are small and atomic, with meaningful messages and updated docs. May run pre-commit hooks (lint/tests) locally for faster feedback.

- **GitHub Repo (branch protection, events)**
  Single source of truth for code and IaC. Branch protection enforces required reviews, status checks, CODEOWNERS on `release/*`, and disallows force-push. Events (PR opened/synchronized, push to release) are tracked and used to trigger workflows. Secrets/Environments (with approvals) store credentials and per-env config.

- **GitHub Actions CI/CD (lint, test, build, Ansible deploy)**
  CI: runs language-aware linting and tests (matrix as needed), gathers artifacts (coverage, test reports). Build: produces immutable Docker image, tags (`:sha`, `:x.y.z`, `:release`) and pushes to GHCR. CD: uses Ansible to deploy to on-prem, with health checks and rollback on failure. Logs and annotations make failures traceable; caching speeds builds.

- **On-Prem Server (Docker app)**
  Primary runtime for normal load; pulls signed images from GHCR. Runs container(s) with env/secret files and resource limits. Exposes health/readiness endpoints. Exports metrics (e.g., `/metrics`, node exporter). Stores minimal secrets locally (prefer short-lived tokens). Connectivity (VPN/peering) allows talking to shared DB/cache if needed.

- **Monitoring (Prometheus/CloudWatch)**
  Continuously scrapes system/app metrics (CPU, memory, RPS, latency, queue depth, error rates). Alert rules include thresholds and durations (e.g., sustained CPU > 70% for 5m) to avoid flapping. Dashboards (Grafana/CloudWatch) visualize SLOs. Alerts include rich context (labels, last values) to inform automated decisions.

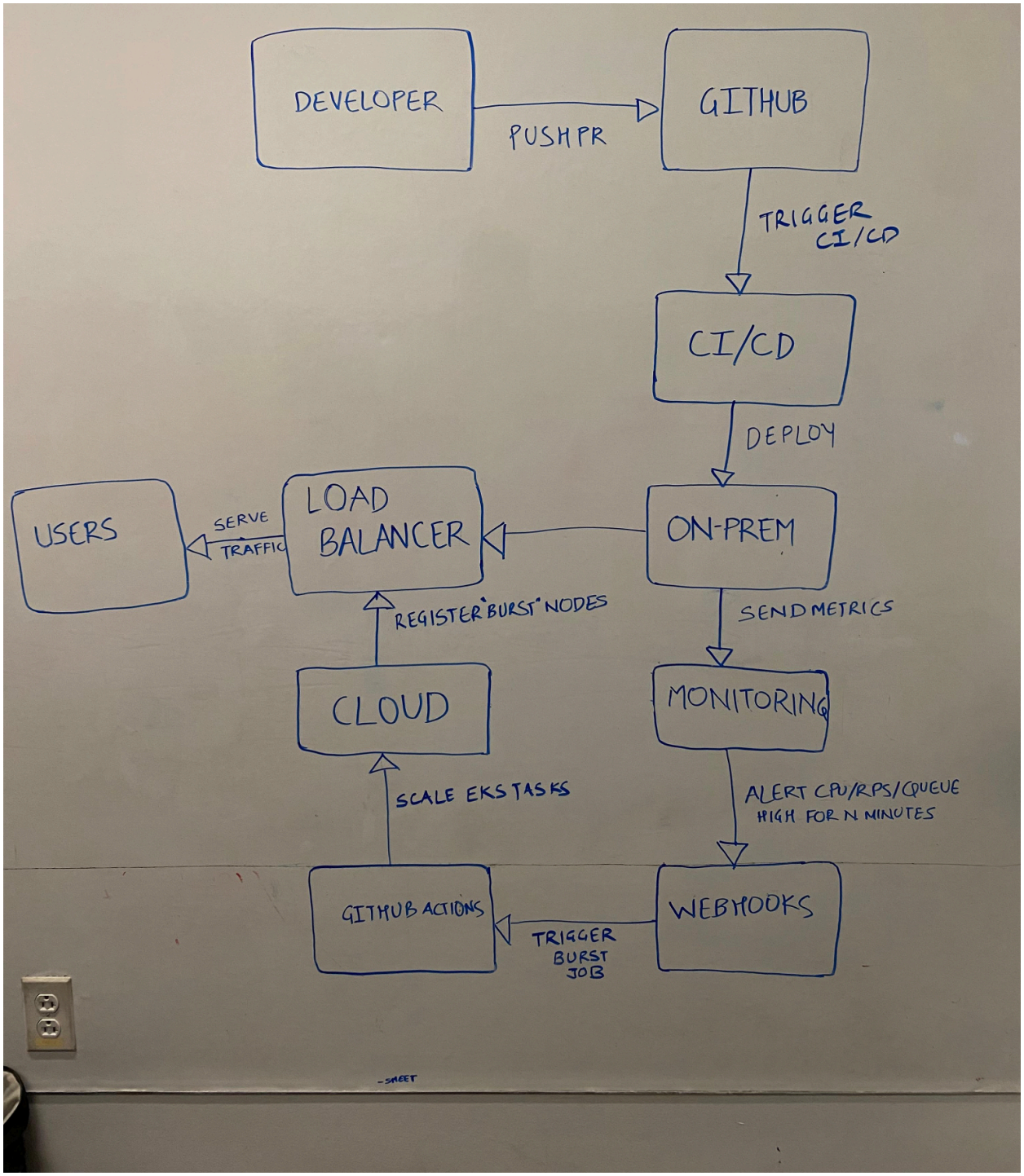
- **Webhook**
  Hardened endpoint that receives alert payloads (signed/HMAC). Validates source and schema, rate-limits to prevent storm effects, and forwards structured parameters (desired scale, reason, correlation IDs) to downstream automation. Logs every invocation for auditability.

- **GitHub Actions (Burst Workflow)**
  Dedicated automation for runtime scaling. Authenticates to AWS via OIDC (no long-lived keys). Executes idempotent steps: compute target count, update ECS desired-count or run `kubectl scale` for EKS, wait for health, then optionally notify. Includes guardrails (max scale, cooldowns) and symmetric scale-in logic when metrics normalize.

- **AWS Cloud (Burst Capacity)**
  Runs the same container image as on-prem for uniform behavior. ECS Fargate: serverless tasks, per-task IAM, target group registration. EKS: Deployments with HPA/cluster autoscaler; readinessProbes and PodDisruptionBudgets. Uses centralized logging/metrics, and minimal secrets via IAM roles. Designed to be stateless or backed by shared state.

- **Load Balancer**
  Fronts user traffic and distributes to healthy backends. On-prem NGINX/HAProxy handles ingress and can proxy to cloud targets; cloud ALB health-checks cloud tasks/pods. Supports weight adjustments during bursts, connection draining on scale-in, TLS termination, and per-route policies. Access logs provide observability.

- **Users**
  Consumers of the service (humans or clients). Experience consistent endpoints and SLOs regardless of scaling decisions. Benefit from reduced latency/errors during spikes due to transparent capacity bursts. Feedback (latency/error budgets) feeds into SLO monitoring and capacity planning.



Constraints & Guidelines

- All code changes must pass lint and tests before deployment (enforced via branch protection rules).
- Container images must be immutable and reproducible; only signed images are admitted to deploy.
- Rollback must be automatic if health checks fail post-deploy.

Required Elements (How We Meet Them)

- **GitHub Actions**: .github/workflows/ci.yml, .github/workflows/cd-release.yml, .github/workflows/burst.yml.
- **Linting step**: ESLint/Flake8/Checkstyle step chosen based on repo content.
- **Testing step**: Jest/Pytest/Maven Surefire (auto-selected).
- **Ansible playbook**: ansible/deploy_onprem.yml deploys the container and updates NGINX upstreams.
- **Branch Protection Rules**: Require PRs, 1+ code owner review, status checks (lint, test, build), linear history, and restrict who can push to release/* and main.
- **Release Branch Trigger**: on: push: branches: ["release/**"] in cd-release.yml.
- **Deployment in a container**: Dockerfile included; image published to GHCR and run on on-prem Docker and cloud ECS/EKS.

Team Division of Work

Our team of three plans to divide the work to cover all major aspects of the hybrid DevOps pipeline. **Smeet Nagda** will focus on setting up the **CI/CD workflows**, including GitHub Actions for linting, testing, building Docker images, and deploying to the on-prem server using Ansible. **Jinish Shah** will handle the **infrastructure and monitoring setup**, configuring the on-prem Docker environment, Prometheus or CloudWatch metrics collection, and the load balancer for hybrid traffic management. **Nisarg Jasani** will develop the **cloud-bursting automation**, implementing the webhook that triggers the GitHub Actions burst workflow to scale ECS or EKS instances during high workloads. All members will collaborate on integration testing, pipeline documentation, and ensuring a consistent configuration across local, on-prem, and cloud environments.