IMPLEMENTATION GUIDE

ChatOps Bot
— Step by Step

9 implementation steps to fix bugs, add features, and ship production-ready code

9 Steps | 7 Bug Fixes | 2 Features

Node.js | Discord.js | MongoDB | GitHub Actions | Docker | Jest | Prometheus

github.com/Sandilya69/chatops-bot

01 Startup Env Validation

Quick Win

■ 10 min

Before the Discord bot connects, validate that every required environment variable exists.
If any are missing, print exactly which ones are absent and exit immediately with a clear error.
This prevents silent crashes later when a missing token causes a confusing runtime failure.

CODE SNIPPET

```
// config/validate.js
const REQUIRED = [
  'DISCORD_TOKEN', 'CLIENT_ID', 'MONGODB_URI',
  'GITHUB_TOKEN', 'GITHUB_OWNER', 'GITHUB_REPO'
];

function validateEnv() {
  const missing = REQUIRED.filter(k => !process.env[k]);
  if (missing.length > 0) {
    console.error('MISSING ENV VARS:', missing);
    process.exit(1);
  }
}

module.exports = { validateEnv };
```

✓ WHAT TO DO

1. Create config/validate.js with the code above.
2. Import and call validateEnv() at the very top of bot.js, before anything else runs.
3. Test by removing one env var — confirm it exits with the correct var name listed.

■ TIP

This takes 10 minutes but saves hours of debugging 'undefined is not a function' errors later.

02 Clean Up the Repository

Quick Win

15 min

Your repo shows 69.8% HTML — that's auto-generated coverage reports inside the coverage/ folder.
These should never be version-controlled. Remove them from git tracking and add to .gitignore.
This flips your language breakdown to mostly JavaScript, which is what it actually is.

CODE SNIPPET

```
# Remove coverage from git tracking (keep files locally)
git rm -r --cached coverage/

# Add to .gitignore
echo 'coverage/' >> .gitignore
echo 'logs/' >> .gitignore

# Also add a .env.example so people know what's needed
# .env.example:
# DISCORD_TOKEN=your_token_here
# CLIENT_ID=your_client_id_here
# MONGODB_URI=mongodb+srv://...
# GITHUB_TOKEN=your_pat_here
# GITHUB_OWNER=your_github_user
# GITHUB_REPO=your_repo_name
```

✓ WHAT TO DO

1. Run 'git rm -r --cached coverage/' to stop tracking the folder.
2. Add coverage/ and logs/ to .gitignore.
3. Create .env.example with placeholder values for all required vars.
4. Commit with message: 'chore: remove generated files, add env example'.

03 Production Approval Flow

Critical

■ 2 hours

The README says prod requires Admin approval, but no confirmation step exists. An Admin can deploy to production in one command with zero verification. Add Discord buttons: Approve and Cancel. Only when an Admin clicks Approve does the actual GitHub workflow_dispatch fire.

CODE SNIPPET

```
const { ButtonBuilder, ActionRowBuilder } = require('discord.js');

// Store pending deploys in memory
const pendingDeploys = new Map();

// In /deploy handler — if env === 'prod':
const id = crypto.randomUUID();
pendingDeploys.set(id, { service, env, userId });

const approve = new ButtonBuilder()
  .setCustomId('approve_' + id)
  .setLabel('Approve').setStyle(1);  // Primary
const cancel = new ButtonBuilder()
  .setCustomId('cancel_' + id)
  .setLabel('Cancel').setStyle(4);   // Danger

const row = new ActionRowBuilder().addComponents(approve, cancel);
await interaction.reply({ content: 'Awaiting approval...', components: [row] });
```

✓ WHAT TO DO

1. When env is 'prod', do NOT trigger GitHub immediately. Reply with Approve/Cancel buttons.
2. Listen for button clicks in client.on('interactionCreate') — check interaction.isButton().
3. On Approve: verify the clicker is Admin, then trigger the workflow. On Cancel: delete the message.
4. Add a 5-minute timeout — if no one clicks, auto-expire and edit the message.

■ TIP

Only gate production. Staging and dev can still fire instantly for fast iteration.

04 Workflow Status Polling

Critical

1.5 hours

After triggering a workflow, the bot never checks if it succeeded or failed. The audit log only ever says 'triggered'. Poll GitHub's API every 10 seconds until the workflow reaches a terminal state, then update both the Discord message and the MongoDB audit record with the real result.

CODE SNIPPET

```
const GITHUB_API = 'https://api.github.com';

async function pollWorkflow(runId, interaction, auditDoc) {
  const maxWait = 600000; // 10 min
  const start = Date.now();

  while (Date.now() - start < maxWait) {
    await sleep(10000); // wait 10s
    const res = await fetch(
      `${GITHUB_API}/repos/.../actions/runs/${runId}`,
      { headers: { Authorization: `token ${TOKEN}` } }
    );
    const run = await res.json();
    if (run.status === 'completed') {
      auditDoc.status = run.conclusion; // success/failure
      await auditDoc.save();
      await interaction.editReply(`Deploy: ${run.conclusion}`);
      return run.conclusion;
    }
  }
  // Timeout after 10 min
  auditDoc.status = 'timeout';
  await auditDoc.save();
}
```

✓ WHAT TO DO

1. After workflow_dispatch succeeds, get the run ID from GitHub's response or list runs API.
2. Start pollWorkflow() — it loops every 10s checking the run status.
3. On completion: update MongoDB status field and editReply() the Discord message.
4. Decrement your Prometheus active_deployments gauge when done.

■ TIP

Use interaction.editReply() to update the original message in-place instead of sending new messages.

05 GitHub Webhook (Replaces Polling)

Medium

■ 1.5 hours

Polling works but wastes API calls. A webhook is the production-grade solution: GitHub POSTs to your server the moment a workflow finishes. Set up an Express endpoint that receives and verifies the event, then processes it the same way polling did.

CODE SNIPPET

```
const express = require('express');
const crypto = require('crypto');
const app = express();
app.use(express.json());

app.post('/webhook/github', async (req, res) => {
 // Verify GitHub signature
  const sig = req.headers['x-hub-signature-256'];
  const hmac = crypto.createHmac('sha256', WEBHOOK_SECRET)
    .update(JSON.stringify(req.body)).digest('hex');
  if (sig !== `sha256=${hmac}`) return res.status(401).end();

  const { action, workflow_run } = req.body;
  if (action === 'completed') {
  // Update MongoDB + Discord message
    await handleWorkflowComplete(workflow_run);
  }
  res.status(200).end();
});

app.listen(3001, () => console.log('Webhook server on 3001'));
```

✓ WHAT TO DO

1. Create a small Express server (can live in the same process) listening on port 3001.
2. In GitHub repo Settings → Webhooks, add your URL with event type 'workflow_run'.
3. Verify the signature on every incoming request using HMAC SHA-256.
4. Once this works reliably, you can remove the polling loop from Step 4.

■ TIP

Use a separate WEBHOOK_SECRET env var (not your GitHub token). Generate one with: openssl rand -hex 20

06 Rate Limiting on /deploy

Medium

45 min

A Developer can currently spam /deploy endlessly — burning CI minutes and risking deployment collisions. Add a per-user cooldown (60 seconds) and a global max concurrent deployments cap. Track cooldowns in-memory with a simple Map. No database needed for this.

CODE SNIPPET

```
// Rate limiter module
const cooldowns = new Map();   // userId -> lastDeployTime
let activeDeploys = 0;
const COOLDOWN_MS = 60000;     // 60 seconds
const MAX_CONCURRENT = 3;

function checkRateLimit(userId) {
  if (activeDeploys >= MAX_CONCURRENT) {
    return 'Max concurrent deployments reached. Wait.';
  }
  const last = cooldowns.get(userId) || 0;
  const remaining = COOLDOWN_MS - (Date.now() - last);
  if (remaining > 0) {
    return `Please wait ${Math.ceil(remaining/1000)}s before deploying again.`;
  }
  return null; // OK to proceed
}

function recordDeploy(userId) {
  cooldowns.set(userId, Date.now());
  activeDeploys++;
}

function deployComplete() { activeDeploys--; }
```

✓ WHAT TO DO

1. Create a rateLimiter.js module with checkRateLimit(), recordDeploy(), deployComplete().
2. In your /deploy handler, call checkRateLimit() first. If it returns a message, reply and stop.
3. Call recordDeploy() when a deploy starts. Call deployComplete() when polling/webhook confirms done.

TIP

Wire deployComplete() into your webhook handler from Step 5 so the counter stays accurate.

07 MongoDB Reconnect Logic

Medium

30 min

If MongoDB Atlas goes down briefly (common), the bot crashes or silently fails on role checks.
Add automatic retry on initial connection and a reconnect listener for runtime disconnects.
This keeps the bot alive through short database outages without manual restarts.

CODE SNIPPET

```
const mongoose = require('mongoose');

const MAX_RETRIES = 5;
const RETRY_DELAY = 3000; // 3 seconds

async function connectDB(uri) {
  for (let i = 1; i <= MAX_RETRIES; i++) {
    try {
      await mongoose.connect(uri);
      console.log('MongoDB connected');
      return;
    } catch (err) {
      console.error(`Attempt ${i}/${MAX_RETRIES} failed:`, err.message);
      if (i === MAX_RETRIES) { process.exit(1); }
      await sleep(RETRY_DELAY * i); // exponential-ish
    }
  }
}

mongoose.on('disconnected', () => {
  console.warn('MongoDB disconnected. Reconnecting...');
  connectDB(process.env.MONGODB_URI);
});

module.exports = { connectDB };
```

✓ WHAT TO DO

1. Replace your current mongoose.connect() call with this connectDB() function.
2. The retry loop handles startup failures. The 'disconnected' event handles runtime drops.
3. Test by temporarily using an invalid URI — confirm it retries and reports clearly.

08 Rollback Command

Feature

■ 2 hours

Add a /rollback <service> command that reverts to the last known-good deployment. It reads your MongoDB audit log, finds the most recent successful deploy for that service, grabs its commit SHA, and re-triggers the GitHub workflow with that SHA.

CODE SNIPPET

```
// commands/rollback.js
const { Deployment } = require('../models/deployment');

async function handleRollback(interaction) {
  const service = interaction.options.getString('service');

  // Find last successful deploy for this service
  const lastGood = await Deployment.findOne({
    service: service,
    status: 'success'
  }).sort({ createdAt: -1 });

  if (!lastGood) {
    return interaction.reply('No successful deploy found for ' + service);
  }

  // Re-trigger workflow with the old commit SHA
  await triggerWorkflow(service, lastGood.environment, lastGood.commitSha);

  await interaction.reply(
    `Rolling back ${service} to commit ${lastGood.commitSha.slice(0,7)}`
  );
}
```

✓ WHAT TO DO

1. Register /rollback as a new slash command in deploy-commands.js with a 'service' string option.
2. Query MongoDB for the latest doc where service matches AND status is 'success'.
3. Re-trigger the workflow using the commitSha from that document.
4. Apply the same approval flow from Step 3 if the rollback targets production.

■ TIP

Make sure your audit docs store commitSha — if they don't, add that field in Step 4 when you update status.

09 Multi-Repo Support

Feature

■ 2 hours

Right now GITHUB_OWNER and GITHUB_REPO are hardcoded env vars — the bot can only deploy one repo. Store a service-to-repo mapping in MongoDB. When /deploy runs, look up which repo and workflow file to use based on the service name. One bot now manages your entire organization.

CODE SNIPPET

```
// models/service.js
const serviceSchema = new mongoose.Schema({
  name:     { type: String, unique: true, required: true },
  owner:    { type: String, required: true },
  repo:     { type: String, required: true },
  workflow: { type: String, default: 'deploy.yml' },
  envs:     [{ type: String }] // allowed environments
});

// In /deploy handler:
const svc = await Service.findOne({ name: serviceName });
if (!svc) {
  return interaction.reply(`Unknown service: ${serviceName}`);
}

// Use svc.owner, svc.repo, svc.workflow instead of env vars
await triggerWorkflow(svc.owner, svc.repo, svc.workflow, env);

// Add /addservice command for admins to register new services
```

✓ WHAT TO DO

1. Create a Service mongoose model with name, owner, repo, workflow, and allowed envs.
2. In /deploy, look up the service by name. If not found, reply with an error.
3. Use the looked-up values instead of env vars when calling the GitHub API.
4. Add an /addservice command so Admins can register new services without code changes.

■ TIP

Seed your first service doc manually in MongoDB to match your current repo, so existing deploys keep working.

SUMMARY

Recommended Order

Phase 1 — Foundation (1 hour)

Step 1	Env Validation	10 min
Step 2	Repo Cleanup	15 min

Phase 2 — Core Fixes (3–4 hours)

Step 3	Production Approval Flow	2 hrs
Step 4	Workflow Status Polling	1.5 hrs

Phase 3 — Hardening (2–3 hours)

Step 5	GitHub Webhook	1.5 hrs
Step 6	Rate Limiting	45 min
Step 7	MongoDB Reconnect	30 min

Phase 4 — New Features (3–4 hrs)

Step 8	Rollback Command	2 hrs
Step 9	Multi-Repo Support	2 hrs

✓ VIBE CODING TIP

Feed each step one at a time to your AI. After each step compiles and runs, move to the next.
Don't paste all 9 steps at once — let the code stabilize between each phase.