1. File System Storage Issues 2

Detailed Explanation In development environments (e.g., local machines), your backend likely relies on the local file system for storing and accessing files, as:

Watermark logos (e.g., stored at src/assets/img/Screenshot...png).
Generated visual diff PDFs (e.g., in uploads/visual-diffs/).
Temporary files created during processing (e.g., intermediate PDFs or caches).

This works fine locally because the file system is persistent—files stay on disk across restarts. However, in production, especially on containerized or serverless platforms (e.g., Docker on Kubernetes, AWS ECes, Heroku, or Vercel): Containers are ephemeral: Each deployment or restart spins up new containers. The file system is reset or mounted as a temporary volume, so any files written during runtime (e.g., uploaded diffs) are lost. This leads to "file not found" errors when trying to access them later. No shared storage across instances: If your app scales to multiple containers (e.g., for load balancing), files written to one instance aren't available on others, causing inconsistencies. Temp file accumulation: Without automated cleanup, temporary files (e.g., from PDF processing) pile up, consuming disk space. In limited environments like serverless functions (e.g., AWS Lambda with 512MB-10GB ephemeral storage), this can lead to "out of space" errors, crashes, or throttled performance. Specific to your case: The watermark logo path is hardcoded to a local asset, which won't exist in a deployed container unless bundled (but even then, it's not scalable). Visual diffs in uploads/ vanish on redeploy, breaking features like downloading historical

In serverless setups, the file system is often read-only or temporary (/tmp in Lambda), exacerbating these issues under load or frequent deploys. Detailed Solution Migrate all file storage to a cloud-based, persistent object storage service like AWS S3, Google Cloud Storage, or Supabase Storage (since you mentioned Supabase, which integrates well with PostgreSQL via Prisma). This make it files are durable, scalable, and accessible across instances. Step-by-Step Implementation:

Choose a Storage Provider to
Use AWS S3 (cost-effective, integrates with most platforms) or Supabase Storage (if you're already using Supabase for DB, as it handles auth and buckets seamlessly). Set up a bucket (e.g., your-app-uploads) with folders like logos/, visual-diffs/, temp/.

Integrate with Your Backend:
Install SDK: npm install @aws-sdk/client-s3 (for S3) or use Supabase's JS client if applicable.
Replace local file operations with cloud uploads/downloads:

For uploads (e.g., visual diffs): Use multer or direct stream to S3. Example:JavaScriptconst { S3Client, PutObjectCommand } = require('@aws-sdk/client-s3');
const s3 = new S3Client({ region: 'us-east-1' });

async function uploadFile(filePath, key) {
  const fileContent = fs.readFileSync(filePath);
  const params = {
    Bucket: 'your-bucket',
    Key: key, // e.g., 'visual-diffs/file.pdf'
    Body: fileContent,
  };
  await s3.send(new PutObjectCommand(params));
  return `https://${params.Bucket}.s3.amazonaws.com/${key}`;
}
For downloads/reads (e.g., watermark logo): Fetch from URL instead of local path.JavaScriptconst { GetObjectCommand } = require('@aws-sdk/client-s3');
async function getFileStream(key) {
  const params = { Bucket: 'your-bucket', Key: key };
  const { Body } = await s3.send(new GetObjectCommand(params));
  return Body; // Stream for processing
}

Handle Temp Files:
Use cloud storage for temps too, but set lifecycle policies: In S3, configure rules to auto-delete objects in temp/ after 1-7 days.
Implement a cleanup job: Use a cron job (e.g., via node-cron) or a serverless scheduler (AWS EventBridge) to scan and delete old temps.JavaScriptconst cron = require('node-cron');
cron.schedule('0 0 * * *', async () => { // Daily
  // List and delete old objects in temp/ via S3 API
});

Migration and Testing:
Migrate existing files: Script to upload local files to S3.
Update code: Replace all fs.readFileSync, fs.writeFile with cloud equivalents.
Test: Deploy to staging; simulate restarts and check file persistence.
Security: Use IAM roles for access; signed URLs for public reads.

Benefits: Scalable (handles TBs of data), durable (99.999% availability), and cost ~$0.023/GB/month. Avoids all ephemeral storage pitfalls.

2.PDFProcessing Libraries Issues

Detailed Explanation

Your backend uses libraries like pdfjs-dist (for parsing), pdf-lib (for manipulation), and pdfkit (for generation), which often depend on native modules like canvas (for rendering) or Cairo (graphics backend).

Native Dependencies Missing: Serverless platforms (Vercel, AWS Lambda) run in lightweight environments without system-level deps like libcairo, libjpeg. Building these in Docker is possible, but default setups fail with "module not found" or "native binding errors."
Memory Errors on Large PDFs: PDFs can be 10-100MB+; processing (e.g., watermarking pages) loads them into memory. Serverless limits (512MB default on Lambda, up to 3GB) cause OOM (Out of Memory) kills.
Timeouts: Serverless functions time out (e.g., 30s on Vercel, 15min max on Lambda). Complex ops like diffing multi-page PDFs exceed this, leading to incomplete processing or 504 errors.
Production-Specific: Locally, your machine has ample RAM (8-32GB) and no timeouts, so issues hide until deployment.

This breaks features like watermarking or visual diffs under real load.
Detailed Solution
Switch to a containerized deployment with pre-installed deps (e.g., Docker on a VM/EC2) or optimize for serverless with lighter libs/workarounds. Prefer Docker for reliability.
Step-by-Step Implementation:

Use Docker with Dependencies:
Create a Dockerfile installing natives:dockerfileFROM node:20-alpine
RUN apk add --no-cache cairo-dev pango-dev jpeg-dev giflib-dev pixman-dev libpng-dev
RUN npm install -g pkg-config
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
CMD ["node", "server.js"]
Deploy to: AWS ECS, Google Cloud Run, or a VPS (DigitalOcean/EC2) for persistent runtime.

Handle Memory and Timeouts:
Increase limits: On Lambda, set 2-8GB RAM and 900s timeout.
Offload heavy tasks: Use queues (see Issue 4) to process asynchronously.
Optimize libs: Switch to pure-JS alternatives if possible (e.g., pdf-lib is mostly JS, but pair with sharp for images instead of canvas).

Error Handling:
Wrap processing in try-catch; retry on OOM.
Monitor: Use Sentry/New Relic for memory usage alerts.

Alternative: Dedicated Server:
If serverless is mandatory, use AWS EC2 or similar for full control over env.

// already done with the direct link
Benefits: Ensures deps are available; scales vertically for memory needs.

3. Database Connection Exhaustion

Detailed Explanation
Prisma (ORM for PostgreSQL) creates a new DB connection per query in serverless mode, as each function invocation is isolated.

Connection Pool Exhaustion: PostgreSQL defaults to 100-200 max connections. Under load (e.g., 100 concurrent requests), connections spike, leading to "too many clients already" errors and app crashes.
No Pooling by Default: Prisma's built-in pooler isn't optimized for serverless; each cold start grabs a new connection without releasing properly.
Production-Specific: Locally, fewer requests mean no exhaustion. In prod (e.g., API traffic), it hits quickly.

This causes downtime during peaks.
Detailed Solution
Implement connection pooling with PgBouncer and Prisma's adapter.
Step-by-Step Implementation:

Set Up PgBouncer:
Deploy PgBouncer (lightweight proxy) in front of PostgreSQL (e.g., on Supabase, enable it; on AWS RDS, use as sidecar).
Config: Transaction mode, pool size 20-50.

Integrate with Prisma:
Install: npm install @prisma/adapter-pg pg
Update prisma/schema.prisma:prismadatasource db {
  provider = "postgresql"
  url   = env("DATABASE_URL")
}
In code, use adapter:JavaScriptconst { Pool } = require('pg');
const { PrismaPg } = require('@prisma/adapter-pg');
const pool = new Pool({ connectionString: process.env.DATABASE_URL });
const adapter = new PrismaPg(pool);
const prisma = new PrismaClient({ adapter });

Best Practices:
Use prisma.$disconnect() in finally blocks.
Monitor connections via PG admin tools.
Scale DB: Increase max_connections if needed.

Benefits: Reuses connections, handles 1000s of requests without exhaustion.

4. Memory-Intensive Operations

Detailed Explanation

Operations like PDF watermarking, diff calc, and visual diff gen load large buffers into memory.

Concurrent Crashes: Multiple requests (e.g., 5 users processing PDFs) exceed per-function memory, causing kills.
Serverless Limits: Isolated functions can't share resources; no horizontal scaling for single heavy tasks.
Production-Specific: Load tests reveal this; dev is single-threaded.

Detailed Solution

Implement a queue system with dedicated workers.
Step-by-Step Implementation:

Choose Queue Lib: Use BullMQ (Redis-based): npm install bullmq ioredis
Set Up:
Redis for queue (e.g., Upstash or AWS ElastiCache).
Worker file (worker.js):JavaScriptconst { Worker } = require('bullmq');
const worker = new Worker('pdf-queue', async job => {
  // Process PDF here (watermark, diff)
}, { connection: { host: 'redis-host' } });
In API: Enqueue instead of sync process:JavaScriptconst { Queue } = require('bullmq');
const queue = new Queue('pdf-queue', { connection: {...} });
await queue.add('process-pdf', { fileId });
// Return job ID for polling/status

Deploy Workers: Run on separate instances (e.g., EC2) with higher RAM.
Monitoring: Add retries, failed job alerts.

Benefits: Offloads heavy work, prevents crashes, scales independently.