CRÍTICO	ALTO	MEDIO
Bloquea el deploy. Sin excepción.	El deploy queda en revisión hasta resolver.	Recomendado. No bloquea pero debe planificarse.

1. Dockerfile — Construcción de la Imagen

x	Criterio	¿Por qué importa?	¿Cómo verificarlo?	Nivel
	Imagen base con versión fija	**¿Por qué?** *Usar :latest significa que mañana puede descargarse una versión diferente con bugs o cambios que rompan tu app. En producción siempre debes saber exactamente qué versión estás corriendo.*	**¿Cómo verificar?** Revisar el Dockerfile: la línea FROM debe tener versión exacta. FROM node:18.19.0-alpine3.19 FROM node:latest	CRÍTICO
	Imagen base mínima (Alpine o slim)	**¿Por qué?** *Una imagen de Ubuntu completa trae miles de paquetes que no necesitas, cada uno es una posible vulnerabilidad. Alpine tiene solo lo esencial, ocupa ~5MB vs ~200MB de Ubuntu.*	**¿Cómo verificar?** El FROM debe terminar en -alpine o -slim. Verificar con: docker image ls mi-app — el tamaño no debería superar 200MB.	ALTO
	Usuario sin privilegios (NO root)	**¿Por qué?** *Por defecto Docker corre como root dentro del contenedor. Si un atacante explota tu app, tendrá acceso root. Un usuario sin privilegios limita enormemente el daño posible.*	**¿Cómo verificar?** El Dockerfile debe tener estas líneas antes del CMD: RUN addgroup -S appgroup && adduser -S appuser -G appgroup USER appuserVerificar ejecutando: docker exec mi-contenedor whoami → debe decir appuser, NUNCA root.	CRÍTICO
	No hay secretos dentro de la imagen	**¿Por qué?** *Todo lo que está en el Dockerfile queda grabado en las capas de la imagen para siempre, incluso si luego lo borras con RUN rm. Cualquiera con acceso a la imagen puede ver esos secretos.*	**¿Cómo verificar?** Revisar que en el Dockerfile no aparezcan: contraseñas, tokens, API keys ni conexiones a DB. Ejecutar: docker history mi-app:version para ver todas las capas.	CRÍTICO

x	Criterio	¿Por qué importa?	¿Cómo verificarlo?	Nivel
	Imagen escaneada sin vulnerabilidades críticas	**¿Por qué?** *Las vulnerabilidades conocidas (CVEs) son publicadas constantemente. Una imagen sin escanear puede tener decenas de fallas conocidas que un atacante puede explotar.*	**¿Cómo verificar?** Ejecutar: trivy image mi-app:1.0.0No debe aparecer ningún resultado con severidad CRITICAL. Las HIGH deben revisarse caso por caso.	CRÍTICO
	Build multi-stage (separar build de producción)	**¿Por qué?** *Si compilas tu código dentro del mismo contenedor que corres en producción, terminas con compiladores, herramientas de build y código fuente innecesario. Más peso y más superficie de ataque.*	**¿Cómo verificar?** El Dockerfile debe tener al menos dos bloques FROM. El segundo (producción) solo copia los archivos compilados del primero. FROM node:18-alpine AS builder ... FROM node:18-alpine AS production COPY --from=builder /app/dist ./dist	ALTO
	Archivo .dockerignore presente	**¿Por qué?** *Sin .dockerignore, Docker copia todo al contexto de build: .git, .env, node_modules, archivos de test. Eso hace las imágenes más grandes y puede filtrar información sensible.*	**¿Cómo verificar?** Debe existir un archivo .dockerignore en la raíz del proyecto con al menos: .git, .env, node_modules, *.test.js, /tests, README.md	ALTO
	Imagen etiquetada con versión semántica	**¿Por qué?** *Si siempre usas :latest no puedes saber qué versión está en producción ni hacer rollback fácilmente. Con versiones sabes exactamente qué desplegaste.*	**¿Cómo verificar?** El build debe usar un tag con versión: docker build -t mi-app:1.4.2 .Nunca hacer deploy con :latest en producción.	ALTO

2. Seguridad del Contenedor en Ejecución

✓	Criterio	¿Por qué importa?	¿Cómo verificarlo?	Nivel
	no-new-privileges activado	**¿Por qué?** *Sin esta opción, un proceso dentro del contenedor podría usar setuid/setgid para escalar sus propios privilegios dentro del sistema, incluso siendo usuario no-root.*	**¿Cómo verificar?** En docker-compose.yml bajo el servicio: security_opt: - no-new-privileges:true	CRÍTICO

✓	Criterio	¿Por qué importa?	¿Cómo verificarlo?	Nivel
	Modo privilegiado desactivado	**¿Por qué?** *privileged: true le da al contenedor acceso casi completo al sistema host. Es como apagar toda la seguridad de Docker. Nunca debe usarse en producción.*	**¿Cómo verificar?** Buscar en docker-compose.yml que NO exista: privileged: trueSi no aparece esa línea, está desactivado por defecto.	CRÍTICO
	Capacidades del kernel eliminadas	**¿Por qué?** *Los contenedores heredan capacidades del kernel que generalmente no necesitan (modificar la red, montar sistemas de archivos, etc.). Eliminarlas reduce lo que un atacante puede hacer si compromete el contenedor.*	**¿Cómo verificar?** En docker-compose.yml: cap_drop: - ALL cap_add: - NET_BIND_SERVICE # Solo si tu app necesita puerto <1024	ALTO
	Sistema de archivos de solo lectura	**¿Por qué?** *Si un atacante entra al contenedor, no podrá modificar archivos del sistema ni instalar herramientas maliciosas. Tu app solo debe poder escribir en carpetas específicas.*	**¿Cómo verificar?** En docker-compose.yml: read_only: true tmpfs: - /tmp # Para archivos temporales - /var/cache # Si la app lo necesita	ALTO
	Límites de CPU y memoria definidos	**¿Por qué?** *Sin límites, un contenedor comprometido o con un bug puede consumir todos los recursos del servidor y tumbar el resto de servicios. Los límites aíslan el impacto.*	**¿Cómo verificar?** En docker-compose.yml: deploy: resources: limits: cpus: '0.5' memory: 512MVerificar con: docker stats	ALTO

3. Red y Puertos

✓	Criterio	¿Por qué importa?	¿Cómo verificarlo?	Nivel
	Solo Nginx/proxy expuesto al exterior	**¿Por qué?** *Tu aplicación no debe recibir tráfico directamente de internet. Nginx actúa como guardián: filtra, redirige y protege. La app solo habla con Nginx, no con el mundo.*	**¿Cómo verificar?** En docker-compose.yml, solo el servicio nginx debe tener ports: con 80 y 443. El servicio de la app debe usar expose: (visible internamente) pero NO ports:.	CRÍTICO

✓	Criterio	¿Por qué importa?	¿Cómo verificarlo?	Nivel
	Base de datos sin puerto expuesto al host	**¿Por qué?** *Si la base de datos tiene un puerto mapeado al host (ej: '5432:5432'), cualquiera que acceda al servidor puede intentar conectarse directamente. No tiene por qué salir del contenedor.*	**¿Cómo verificar?** En el servicio de DB en docker-compose.yml NO debe existir la sección ports:. Solo debe existir expose: si otros contenedores la necesitan.	CRÍTICO
	Red interna nombrada y separada	**¿Por qué?** *La red default de Docker conecta todos los contenedores entre sí. Crear redes internas te permite aislar grupos de servicios: la app habla con la DB, pero el servicio de email no tiene por qué ver la DB.*	**¿Cómo verificar?** En docker-compose.yml definir: networks: red_interna: driver: bridgeY asignarla a cada servicio con: networks: [red_interna]	ALTO
	Socket de Docker no montado en contenedores	**¿Por qué?** *Montar /var/run/docker.sock dentro de un contenedor le da control total sobre Docker y el host. Si ese contenedor es comprometido, el atacante puede crear contenedores privilegiados y escapar al host.*	**¿Cómo verificar?** Buscar en docker-compose.yml que NINGÚN servicio tenga: volumes: - /var/run/docker.sock:/var/run/docker.sockSi no aparece, está bien.	CRÍTICO

4. Secretos y Variables de Entorno

✓	Criterio	¿Por qué importa?	¿Cómo verificarlo?	Nivel
	Sin contraseñas en texto plano en docker-compose.yml	**¿Por qué?** *El docker-compose.yml generalmente se sube al repositorio de código. Si tiene contraseñas en texto plano, cualquiera con acceso al repo (o si el repo es público) puede verlas.*	**¿Cómo verificar?** Revisar que en environment: no aparezcan valores reales de contraseñas, tokens o claves. DB_PASSWORD: ${DB_PASSWORD} (lee de .env) DB_PASSWORD: mipassword123	CRÍTICO
	Archivo .env excluido del repositorio	**¿Por qué?** *El archivo .env contiene los secretos reales. Si se sube a Git accidentalmente, queda en el historial para siempre incluso si luego lo borras.*	**¿Cómo verificar?** Verificar que en .gitignore existan estas líneas: .env .env.* .env.productionVerificar que no esté en el repo: git ls-files | grep .env	CRÍTICO

✓	Criterio	¿Por qué importa?	¿Cómo verificarlo?	Nivel
	Variables separadas por ambiente	**¿Por qué?** *Usar las mismas credenciales en desarrollo y producción es un riesgo. Alguien con acceso al ambiente de desarrollo puede acceder a producción.*	**¿Cómo verificar?** Deben existir archivos separados: .env.development → credenciales de dev .env.production → credenciales realesEl deploy usa: docker compose --env-file .env.production up -d	ALTO
	No hay secretos en el código fuente	**¿Por qué?** *A veces los secretos se 'hardcodean' directamente en el código por comodidad. Esto es igual de peligroso que tenerlos en el compose.*	**¿Cómo verificar?** Revisar el código con: grep -r 'password|secret|api_key|token' ./srcUsar herramienta como git-secrets o truffleHog para detectar secretos en el historial de Git.	CRÍTICO

5. Health Checks y Disponibilidad

✓	Criterio	¿Por qué importa?	¿Cómo verificarlo?	Nivel
	Health check definido en cada servicio	**¿Por qué?** *Sin health check, Docker considera el contenedor 'saludable' si el proceso no ha muerto, aunque tu app esté colgada, en loop infinito o sin poder conectarse a la DB. El health check verifica que realmente responda.*	**¿Cómo verificar?** En docker-compose.yml bajo cada servicio: healthcheck: test: ["CMD", "curl", "-f", "http://localhost:3000/health"] interval: 30s timeout: 10s retries: 3 start_period: 40s	CRÍTICO
	Endpoint /health implementado en la app	**¿Por qué?** *El health check necesita un punto de la app que confirme que todo está bien: que la app responde, que puede llegar a la base de datos, etc.*	**¿Cómo verificar?** La app debe tener una ruta GET /health que devuelva HTTP 200 cuando todo está bien.Verificar desde el host: curl http://localhost:PUERTO/health	CRÍTICO
	Política de reinicio: unless-stopped	**¿Por qué?** *restart: always reinicia el contenedor incluso si lo paras manualmente (molesto en mantenimientos).*	**¿Cómo verificar?** En cada servicio del docker-compose.yml: restart: unless-stopped	ALTO

✓	Criterio	¿Por qué importa?	¿Cómo verificarlo?	Nivel
		unless-stopped se reinicia si cae solo, pero respeta las paradas manuales.		
	Orden de arranque con espera real	**¿Por qué?** *depends_on por defecto solo espera que el contenedor arranque, no que esté listo. Tu app puede intentar conectarse a la DB antes de que esta esté lista y fallar.*	**¿Cómo verificar?** En docker-compose.yml: depends_on: db: condition: service_healthyRequiere que el servicio db tenga su propio healthcheck definido.	ALTO

6. Configuración General del docker-compose.yml

✓	Criterio	¿Por qué importa?	¿Cómo verificarlo?	Nivel
☐	**Sin campo 'version' en el compose**	**¿Por qué?** *Desde Docker Compose v2 (el actual), el campo version: está obsoleto y genera advertencias. La documentación oficial de Docker indica que ya no debe usarse.*	**¿Cómo verificar?** El archivo docker-compose.yml NO debe tener la línea version: '3.8' ni ninguna versión al inicio. El archivo debe comenzar directamente con services:	MEDIO
☐	**Nombres de contenedores definidos explícitamente**	**¿Por qué?** *Sin container_name, Docker genera nombres aleatorios como proyecto-servicio-1. Con nombres fijos es mucho más fácil hacer docker logs, docker exec o monitoreo.*	**¿Cómo verificar?** En cada servicio del compose: container_name: mi-app-produccionVerificar: docker ps → debe mostrar el nombre definido.	MEDIO
☐	**Imagen de producción con tag fijo, no latest**	**¿Por qué?** *Si el compose dice image: mi-app:latest y alguien hace docker compose pull, puede bajar una versión diferente a la que estaba corriendo sin darse cuenta.*	**¿Cómo verificar?** En el servicio del compose: image: mi-app:1.4.2Nunca: image: mi-app:latest	ALTO
☐	**Archivo separado para producción**	**¿Por qué?** *El docker-compose.yml de desarrollo tiene herramientas de debug, volúmenes de código*	**¿Cómo verificar?** Debe existir un archivo docker-compose.prod.yml solo con la configuración de producción. Desplegar	ALTO

✓	Criterio	¿Por qué importa?	¿Cómo verificarlo?	Nivel
		fuente y configuraciones que no deben estar en producción.	con: docker compose -f docker-compose.prod.yml up -d	

7. Proceso de Deploy

✓	Criterio	¿Por qué importa?	¿Cómo verificarlo?	Nivel
☐	**Deploy sin bajar todos los servicios**	**¿Por qué?** *Hacer docker compose down antes del deploy borra los contenedores y genera downtime. Con --no-deps puedes actualizar solo el servicio que cambió sin tocar los demás.*	**¿Cómo verificar?** Para actualizar solo la app sin tocar la DB ni otros servicios: docker compose up -d --no-deps --build appVerificar que otros servicios siguieron corriendo durante el deploy.	ALTO
☐	**Verificación automática post-deploy**	**¿Por qué?** *Después de un deploy puede que el contenedor arranque pero la app falle internamente. Hay que verificar que realmente esté saludable antes de dar el deploy por exitoso.*	**¿Cómo verificar?** Después del deploy ejecutar: docker ps → estado debe ser healthy, no starting docker logs mi-app --tail 50 → sin errores críticos curl http://localhost/health → debe responder 200	CRÍTICO
☐	**Procedimiento de rollback documentado**	**¿Por qué?** *Cuando algo sale mal en producción, el estrés hace que se cometan más errores. Si el procedimiento de vuelta atrás está escrito, cualquier persona del equipo puede ejecutarlo en minutos.*	**¿Cómo verificar?** Debe existir un documento o script con los pasos exactos para volver a la versión anterior: docker compose up -d --no-deps mi-app mi-app:VERSION_ANTERIORProbar el rollback antes de necesitarlo de verdad.	ALTO