PROJECT REPORT

Hybrid P2P-Cloud Anonymous Transfer Engine

A production-grade, privacy-first file transfer platform that races WebRTC peer-to-peer connections against cloud relay — and self-destructs the evidence.

Prepared by	Vineeth Sagar H L
Programme	B.E. Information Science and Engineering, BMSIT&M, Bengaluru
GitHub	github.com/Vineeth-Sagar
Portfolio	vineethsagar.co.in

Core Stack	FastAPI · React · WebRTC · Socket.IO · AWS S3/Lambda · Redis · Kubernetes · Docker · Prometheus

Table of Contents

1. Introduction & Motivation
2. System Architecture
3. Architecture Highlights
4. Key Features
5. Technology Stack
6. Data Pipeline & Storage Schema
7. API & Signaling Reference
8. Testing Strategy
9. Key Design Decisions
10. Security Architecture
11. Challenges Faced & Solutions
12. Lessons Learned
13. Setup & Execution Guide
14. Kubernetes Deployment Architecture
15. Future Enhancements
16. Project File Structure
17. Conclusion

1. Introduction & Motivation

Every day, millions of people transfer files using platforms such as WeTransfer, Google Drive, or Dropbox. The uncomfortable truth behind this convenience is that every one of those files passes through a corporate server, gets stored on infrastructure the sender does not control, and persists there for longer than most users would like.

This project set out to challenge that model with a fundamental question: *what if file transfers could happen directly between browsers — with no middleman, no storage, and no trace left behind?*

Answering that question required going deep into **WebRTC DataChannels**, **NAT traversal**, **STUN/TURN relay servers**, and the practical reality that peer-to-peer connections fail more often than they succeed in the real world. Corporate firewalls, symmetric NATs, and carrier-grade NATs are all designed, in effect, to prevent direct connections between devices.

The system therefore treats network connectivity as a **probability, not a guarantee**:

- **Best case (P2P succeeds):** files stream directly between browsers at LAN-like speeds. Zero servers touch the data and zero copies exist anywhere.
- **Worst case (P2P fails):** the system silently pivots to streaming the file through AWS S3 with a 1-hour presigned URL, then destroys the file via a serverless Lambda “Nuke Protocol” after download — backed by a 24-hour lifecycle failsafe.

This is not a toy demo. It is a production-grade distributed system complete with Kubernetes manifests, Prometheus monitoring, non-root containers, multi-stage Docker builds, and a real deployment pipeline spanning Render, Railway, and self-hosted Kubernetes.

2. System Architecture

The platform follows a **microservices architecture** with three independently deployable services orchestrated behind a unified Nginx Ingress layer: a React single-page application, a FastAPI REST backend, and a Node.js Socket.IO signaling service — backed by Redis, AWS S3/Lambda, a self-hosted Coturn STUN/TURN relay, and Prometheus.

High-Level Component Map

```
Browser (React SPA)
  Dashboard UI  --  Lobby + Dropzone + Progress
  useSignaling Hook  --  Socket.IO Client
  useWebRTC Hook  --  RTCPeerConnection + DataChannel
  useHybridTransfer Hook  --  5s Race Controller
  IndexedDB  --  Zero-RAM Chunk Buffer
        |
        |  WebSocket (signaling)         |  ICE candidates
        v                                v
Kubernetes Cluster                 Coturn STUN/TURN
  FastAPI Backend  (Auth, Heartbeat,     NAT Traversal Relay
  Upload)  -- Port 8000                  Ports 3478 / 5349
  Socket.IO Signaling (Node.js)
  -- Port 3001
  Nginx SPA Server -- Port 80
  Nginx Ingress (SSL + WS upgrade)
        |
        +--> Redis (Ephemeral Presence Store, TTL 60s)
        +--> AWS S3 (Multipart Streaming, 24h Lifecycle)
        +--> AWS Lambda (Nuke Protocol, Burn-After-Reading)
        +--> Prometheus (Scrapes /metrics every 15s)
```

Architectural overview: client hooks race a direct WebRTC path against a server-mediated fallback, while a signaling layer only ever relays connection metadata — never file content.

Three-Tier Transfer Hierarchy

The system implements a cascading fallback strategy that always prioritizes privacy first:

Priority	Route	Latency	Privacy	When It's Used
1	WebRTC Direct P2P	Lowest	Maximum (zero server contact)	Both peers on open/cone NAT
2	TURN Relay via Coturn	Medium	High (encrypted relay, no storage)	One or both peers behind restricted NAT
3	AWS S3 Cloud Upload	Highest	Moderate (temporary storage, auto-destruct)	WebRTC completely fails (symmetric NAT, firewall)

3. Architecture Highlights

3.1 The 5-Second Race Algorithm

The heart of the system is the hybrid transfer controller, implemented in the `useHybridTransfer` hook. Rather than waiting indefinitely for WebRTC to negotiate, it runs an aggressive race: a 5-second countdown starts the moment a transfer begins. If the peer-to-peer connection completes first, the transfer proceeds over the DataChannel in 16KB chunks. If the timer expires first, the client falls back to a multipart HTTP upload streamed to S3 in 5MB chunks. Either path converges on the same `completed` state.

Why 5 seconds? WebRTC ICE negotiation typically completes in 1–3 seconds on open networks. A 5-second window gives STUN enough time to probe, without leaving the user staring at a spinner when they are behind a corporate firewall that will never allow a direct connection.

3.2 Zero-RAM Chunk Buffering with IndexedDB

Traditional file-transfer implementations accumulate received chunks in a JavaScript array — which means a 500MB file consumes 500MB of browser RAM, crashing tabs at scale. This engine writes every incoming 16KB DataChannel chunk directly to IndexedDB (the `TransferCache` object store), bypassing RAM entirely. Only at the final step does it read all chunks back, assemble a Blob, and trigger the download.

3.3 Burn-After-Reading Data Destruction

Cloud-relayed files are protected by a three-layer destruction cascade:

Layer	Mechanism	Trigger	Latency
1	abort_multipart_upload	Upload failure mid-stream	Immediate
2	AWS Lambda “Nuke Protocol”	Post-download deletion request	Seconds
3	S3 Lifecycle Policy	24-hour automatic expiration	≤ 24 hours

Even if a user never triggers the Nuke Protocol, the file cannot exist beyond 24 hours — AWS enforces this at the infrastructure level.

3.4 Graceful Degradation at Every Layer

The system is designed to never hard-fail:

- **Redis unavailable?** Falls back to an async InMemoryStore with an asyncio.Lock and manual TTL eviction.
- **WebRTC blocked?** The 5-second timer triggers S3 cloud relay.
- **S3 upload fails mid-stream?** abort_multipart_upload cleans up orphaned parts.
- **Container crashes?** Kubernetes restarts the pod, with resource limits preventing OOM cascading.

4. Key Features

WebRTC Peer-to-Peer Transfer

- RTCDataChannel with binary chunking at 16KB (the SCTP maximum-transmission-unit sweet spot).
- Backpressure flow control using bufferedAmountLowThreshold (64KB) to prevent buffer exhaustion on slow connections.
- ICE Trickle with Google STUN (stun:stun.l.google.com:19302) and a self-hosted Coturn TURN relay for NAT traversal.

Hybrid 5-Second Race Fallback

- Races the P2P connection against a 5-second timer.
- Seamless pivot to cloud relay — the user only sees a badge change from “P2P Direct” to “Cloud Server Relay”.
- The progress bar's gradient color changes dynamically based on the active route.

Anonymous Identity & Live Lobby

- Auto-generated hacker-style handles (e.g. Hacker_4821).
- JWT-based stateless sessions (HS256, 2-hour expiry).
- Redis ephemeral presence with 60-second TTL keys and 30-second heartbeat pulses.
- Real-time lobby showing online peers, refreshed every 10 seconds.

Zero-Disk S3 Streaming Upload

- aioboto3 async multipart upload with 5MB chunk streaming — files never touch the API server's disk or accumulate in memory.
- Returns 1-hour presigned download URLs.
- Three-layer file destruction: abort, Lambda nuke, and 24-hour lifecycle.

Prometheus Observability

- Auto-instrumented metrics via prometheus-fastapi-instrumentator.
- Kubernetes ServiceMonitor CRD scraping /metrics every 15 seconds.
- Request latency histograms, error-rate counters, and throughput gauges out of the box.

Production Container Security

- All containers run as non-root users (appuser, nginx).
- Multi-stage Docker builds minimize attack surface (no build tools in production images).
- Kubernetes resource limits prevent memory leaks and CPU starvation.

Event-Driven Signaling

- Node.js Socket.IO server acting as a pure relay — it never sees file content, only SDP offers/answers and ICE candidates.
- In-memory Map for UUID-to-socket mapping with automatic cleanup on disconnect.

5. Technology Stack

Backend

Technology	Purpose
FastAPI	Async Python REST API framework
Uvicorn	High-performance ASGI server
Redis (async)	Ephemeral presence store with TTL-based session tracking
aioboto3 / botocore	Async AWS SDK for zero-disk S3 multipart streaming
PyJWT	HS256 JWT token generation and validation
python-multipart	Streaming multipart form parser for file uploads
Pydantic	Request/response data validation and serialization
prometheus-fastapi-instrumentator	Auto-instrumented Prometheus metrics

Frontend

Technology	Purpose
React 18	Component-based SPA framework
Vite 5	Lightning-fast HMR development server and bundler
Tailwind CSS 3	Utility-first dark theme styling
Socket.IO Client	WebSocket connection to the signaling server
Axios	HTTP client for REST API and multipart uploads
WebRTC APIs	RTCPeerConnection, RTCDataChannel, ICE/SDP
IndexedDB	Zero-RAM binary chunk storage during file reception

Infrastructure

Technology	Purpose
Docker	Multi-stage, non-root containerization
Kubernetes	Container orchestration with resource limits
Nginx Ingress	SSL termination, WebSocket upgrade, SPA routing
Coturn	Self-hosted STUN/TURN relay for NAT traversal
AWS S3	Temporary cloud file storage with lifecycle policies
AWS Lambda	Serverless burn-after-reading file deletion
Prometheus	Metrics collection via ServiceMonitor CRD
Render / Railway	PaaS deployment targets
Let's Encrypt	Automated SSL via cert-manager

6. Data Pipeline & Storage Schema

End-to-End Transfer Flow

1. The sender calls `POST /api/auth/join` and receives a JWT bound to a generated UUID and display name, valid for two hours.
2. Both peers send a heartbeat every 30 seconds, which refreshes a 60-second Redis presence key so the live lobby reflects who is currently online.
3. Once the receiver selects the sender from the lobby, both clients register their UUID with the Socket.IO signaling server, then exchange SDP offer/answer and trickled ICE candidates through it.
4. **If the P2P connection succeeds within 5 seconds**, the sender pushes a JSON metadata frame (filename, size, type) followed by binary chunks over the DataChannel; the receiver writes each chunk to IndexedDB and assembles a Blob for download once complete.
5. **If the 5-second window elapses**, the sender instead streams the file to S3 via a 5MB multipart upload and receives a 1-hour presigned URL, which is shared with the receiver. After download, the receiver (or a scheduled process) triggers the Lambda Nuke Protocol, which issues a head_object check followed by delete_object.

Data Storage Schema

Redis Presence Store

```
Key Pattern:   user_online:{uuid}
Value:         "{anonymous_name}"    (e.g., "Hacker_4821")
TTL:           60 seconds
Heartbeat:     Every 30 seconds (POST /api/heartbeat)
Scan Method:   SCAN (non-blocking) + batched MGET
```

S3 Object Layout

```
Path:          uploads/{user_id}/{uuid4}.{extension}
Metadata:      {"sender_uuid": "{user_id}"}
Presigned URL: GET, expires in 3600 seconds (1 hour)
Lifecycle:     Expiration after 1 day
Multipart:     AbortIncompleteMultipartUpload after 1 day
```

IndexedDB (Browser-Side)

```
Database:      TransferCache
Object Store:  chunks
Data:          Raw binary ArrayBuffer (16KB per entry)
Lifecycle:     Cleared after Blob assembly and download trigger
```

Core Pydantic Models

```
class JoinRequest(BaseModel):
    anonymous_name: str          # e.g., "Hacker_4821"

class TokenResponse(BaseModel):
    access_token: str            # HS256 JWT
    token_type: str              # "bearer"
```

7. API & Signaling Reference

REST API (FastAPI — Port 8000)

Method	Endpoint	Auth	Purpose
GET	/ping	No	Health check
GET	/metrics	No	Auto-instrumented Prometheus metrics
POST	/api/auth/join	No	Anonymous session creation, returns JWT
POST	/api/heartbeat	Yes	Refresh Redis presence TTL to 60s
GET	/api/users/online	No	List active peers
POST	/api/upload	Yes	S3 cloud fallback upload (multipart/form-data)

WebSocket Signaling Events (Socket.IO — Port 3001)

Event	Direction	Purpose
register	Client to Server	Map UUID to socket ID
offer	Bidirectional	Relay WebRTC SDP offer
answer	Bidirectional	Relay WebRTC SDP answer
ice-candidate	Bidirectional	Trickle ICE candidates
disconnect	Automatic	Clean up UUID mapping

WebRTC Signaling Protocol — Phase Breakdown

The signaling server is intentionally minimal: it is a pure relay that never inspects or stores file data.

- **Phase 1 — Registration:** both peers emit register with their UUID; the server maps each UUID to a socket ID in memory.
- **Phase 2 — SDP Exchange:** the sender emits an offer addressed to the receiver's UUID; the server looks up the socket and relays it, and the answer follows the reverse path.
- **Phase 3 — ICE Trickle:** both sides exchange ICE candidates through the same relay pattern as they are discovered.
- **Phase 4 — Direct P2P Established:** once negotiation completes, the RTCDataChannel carries all further traffic in 16KB binary chunks, and the signaling server is no longer involved.

Why Socket.IO over raw WebSockets? Socket.IO provides automatic reconnection, event namespacing, and transport fallback (WebSocket to HTTP long-polling). In production, behind corporate proxies that block WebSocket upgrades, this fallback keeps signaling functional.

8. Testing Strategy

Current Testing Approach

The project currently relies on integration-level verification rather than a formal automated unit-test suite:

Method	What It Validates
/ping health check	Backend is alive and responsive
start.py orchestrator	All three services boot successfully and stay running
Process polling loop	Detects unexpected child-process crashes during development
Manual browser testing	End-to-end P2P and cloud-fallback transfer flows
Kubernetes liveness probes	Container health in production
Prometheus metrics	Request success/failure rates and latency distributions

Recommended Testing Roadmap

```
Unit Tests (pytest)
  test_auth.py          -> JWT generation, expiry, invalid token handling
  test_redis_client.py  -> InMemoryStore fallback, TTL eviction, scan
  test_uploads.py       -> Multipart abort on failure, presigned URL format
  test_security.py      -> Bearer token extraction, 401 scenarios

Integration Tests
  test_signaling.py     -> Socket.IO event relay (offer/answer/ice-candidate)
  test_presence.py      -> Heartbeat -> online listing -> TTL expiry
  test_s3_lifecycle.py  -> Upload -> presigned URL -> Lambda deletion

E2E Tests (Playwright / Cypress)
  test_lobby.cy.js      -> Auto-registration, peer discovery, selection
  test_p2p_transfer.js  -> WebRTC negotiation, chunk transfer, download
  test_fallback.js      -> Force P2P failure, verify cloud relay activation
```

9. Key Design Decisions

9.1 Why 16KB DataChannel chunks?

WebRTC DataChannels use SCTP under the hood, which has a maximum message size; while browsers can fragment larger messages, crossing roughly the 64KB boundary triggers fragmentation that destroys throughput. 16KB sits well below this threshold while remaining large enough to minimize per-chunk overhead — the sweet spot between reliability and speed.

9.2 Why Redis ephemeral keys instead of a database?

User presence is inherently temporal: a user is either online right now or not. A traditional database would require writes on connect, on every heartbeat, and on disconnect, plus scheduled cleanup for clients that crash without disconnecting cleanly. A Redis SETEX with a 60-second TTL eliminates all of this — if a client crashes, its key simply evaporates, with no cleanup jobs, stale data, or phantom users. The 30-second heartbeat interval gives a 2x safety margin.

9.3 Why SCAN instead of KEYS *?

Redis is single-threaded, and KEYS * scans the entire keyspace in one blocking operation, which can freeze Redis for seconds at scale. SCAN returns results incrementally in cursor-based pages, keeping Redis responsive even with thousands of presence keys.

9.4 Why an in-memory fallback for Redis?

Local development should not require running Redis. The InMemoryStore class mirrors the Redis async API using an asyncio.Lock for thread safety and manual TTL tracking, so python start.py works out of the box with zero external dependencies.

9.5 Why aioboto3 instead of boto3?

Standard boto3 is synchronous and blocks the event loop during S3 uploads; for a FastAPI async backend this would serialize all uploads through a single thread. aioboto3 wraps boto3 operations in async context managers, letting the server handle multiple concurrent uploads without blocking.

9.6 Why separate Ingress resources for API and frontend?

Nginx Ingress regex rewrite annotations apply globally to all paths within an Ingress resource. If the frontend SPA route and the API route shared the same Ingress, the regex rewrite would break static asset paths (CSS, JS, images). Splitting into two Ingress resources — one with regex rewrites for /api, one without for / and /socket.io — prevents this collision.

9.7 Why network_mode: “host” for Coturn?

Docker's bridge networking translates ports through iptables NAT rules, but WebRTC TURN relay requires dynamic UDP port allocation across the range 49152-65535. Bridge NAT cannot efficiently handle this scale of dynamic port mapping and introduces latency, so host networking gives Coturn direct access to the host's network stack instead.

9.8 Why IndexedDB instead of in-memory arrays?

A JavaScript array holding 500MB of ArrayBuffer chunks pushes the browser tab to 500MB+ of RAM. IndexedDB writes to the browser's internal on-disk storage, keeping RAM usage flat — the difference between a system that transfers files up to 50MB and one that transfers files of arbitrary size.

10. Security Architecture

Authentication Flow



Container Security Posture

- Non-root execution across all containers (appuser:appgroup for the backend, nginx:nginx for the frontend).
- Multi-stage builds strip build dependencies (Node.js, npm, pip) from production images.
- Kubernetes resource limits prevent any single pod from consuming cluster resources — API: 500m CPU / 512Mi RAM max, Frontend: 200m CPU / 128Mi RAM max, Signaling: 250m CPU / 256Mi RAM max.
- AWS IAM least-privilege — the Lambda function has only s3:DeleteObject and s3:GetObject permissions.
- No persistent credentials in containers — secrets are injected via Kubernetes Secret objects.

11. Challenges Faced & Solutions

11.1 The Symmetric NAT Problem

Problem: WebRTC connections worked perfectly on home networks but failed 100% of the time on mobile data and corporate networks, because symmetric NATs assign a different external port for every destination, making STUN-based hole punching impossible.

Solution: A self-hosted Coturn TURN server with network_mode: “host” relays encrypted media when direct connections fail, and the 5-second race timer prevents users from waiting indefinitely for connections that will never establish.

11.2 Railway Dynamic Port Injection

Problem: Railway assigns a random port via the $PORT environment variable at container startup; hardcoding --port 8000 in the Dockerfile caused the container to listen on the wrong port and become unreachable behind Railway's reverse proxy.

Solution: An entrypoint.sh script dynamically reads $PORT at runtime: exec uvicorn app.main:app --host 0.0.0.0 --port "${PORT:-8000}".

11.3 Redis Connection Blocking the Event Loop

Problem: The early implementation used eager Redis pool initialization at import time; if Redis was slow or unreachable, the entire FastAPI startup would hang, blocking all endpoints, not just presence-related ones.

Solution: Lazy Redis initialization wrapped in the InMemoryStore fallback: the Redis connection is attempted on first use, and if it fails, the in-memory store transparently takes over without crashing the server.

11.4 Browser RAM Exhaustion on Large Files

Problem: The initial WebRTC receiver implementation stored incoming chunks in a JavaScript Array; transferring a 200MB file consumed 200MB+ of tab RAM, causing Chrome to kill the tab on mobile devices.

Solution: Pivoted to IndexedDB chunk buffering, writing each 16KB chunk directly to the TransferCache object store on disk. RAM usage stays constant regardless of file size, at the cost of slightly slower assembly at the end.

11.5 DataChannel Buffer Overflow

Problem: Sending file chunks as fast as FileReader could read them caused the DataChannel's internal buffer to overflow, silently dropping chunks and corrupting transfers.

Solution: Implemented backpressure control using bufferedAmountLowThreshold (64KB): the sender pauses when the buffer exceeds this threshold and resumes via the onbufferedamountlow callback, creating a natural flow-control mechanism that adapts to network speed.

11.6 Nginx Ingress Regex vs. SPA Routing Collision

Problem: Using a single Kubernetes Ingress resource with a regex rewrite (/api(/|$)(.*) -> /$2) caused the frontend's static assets (JS, CSS) to be rewritten and return 404s.

Solution: Split into two separate Ingress resources — one handling /api with regex rewrites, and another handling / (SPA) and /socket.io (WebSocket) without rewrites.

11.7 S3 Orphaned Multipart Uploads

Problem: If a cloud upload failed midway (network timeout, client disconnect), the partially uploaded S3 multipart chunks would persist indefinitely, silently accumulating storage costs.

Solution: Three-layer cleanup: an immediate abort_multipart_upload in the except block of the upload handler, a lifecycle rule (AbortIncompleteMultipartUpload set to 1 day) via setup_s3_lifecycle.py, and a Lambda function for

manual on-demand cleanup.

12. Lessons Learned

Distributed Systems Are About Failure Modes

The most important code in this project is not the happy path — it is the fallback logic. Every component has a failure mode, and the quality of the system is defined by how gracefully it handles each one. Every feature is now considered in terms of “what happens when this breaks?”

WebRTC Is Powerful but Hostile

WebRTC gives browser-to-browser communication, but the internet actively fights direct connections. NAT traversal is a probabilistic game, not a deterministic protocol, so building a reliable product on WebRTC means accepting that P2P will fail and having a seamless fallback ready.

Async Python Changes Everything

Moving from synchronous boto3 to aioboto3, and from blocking Redis to redis.asyncio, are not just performance optimizations — they are architectural decisions that determine whether the server can handle 10 concurrent uploads or 1,000.

Kubernetes Forces Good Architecture

Writing K8s manifests forced explicit thinking about resource limits (what happens if code leaks memory?), health checks (how does the cluster know a service is alive?), secret management (no more hardcoded credentials), and scaling (can two replicas run without breaking state?).

Browser APIs Are More Capable Than You Think

IndexedDB, WebRTC DataChannels, the File API, and Blob construction show that modern browsers are legitimate application platforms — real distributed systems can be built entirely in the browser with careful architecture.

Security Is a Spectrum, Not a Checkbox

From non-root containers to least-privilege IAM policies to JWT expiration to S3 lifecycle rules, security is not one thing added at the end — it is dozens of small decisions made throughout the entire stack.

13. Setup & Execution Guide

Option 1 — One-Command Local Development

```
# Clone the repository
git clone https://github.com/your-repo/hybrid-p2p-engine.git
cd hybrid-p2p-engine

# Set up backend virtual environment
cd backend
python -m venv venv
venv\Scripts\activate        # Windows
# source venv/bin/activate    # Linux/macOS
pip install -r requirements.txt
cd ..

# Install frontend dependencies
cd frontend
npm install
cd ..

# Install signaling dependencies
cd signaling
npm install
cd ..

# Launch all three services simultaneously
python start.py
```

The start.py orchestrator boots the FastAPI backend on port 8000 (with hot-reload), the Socket.IO signaling server on port 3001, and the Vite dev server on port 5173; waits 4 seconds for services to initialize; opens http://localhost:5173 automatically; and monitors all processes, reporting crashes in real time. If Redis is not running locally, the backend automatically falls back to its InMemoryStore, so no additional setup is required.

Option 2 — Docker Containers

```
# Build all images
docker build -t hybrid-p2p/api:latest ./backend
docker build -t hybrid-p2p/frontend:latest ./frontend
docker build -t hybrid-p2p/socket:latest ./signaling

# Run with environment variables
docker run -d -p 8000:8000 \
  -e REDIS_URL=redis://your-redis:6379 \
  -e AWS_ACCESS_KEY_ID=your-key \
  -e AWS_SECRET_ACCESS_KEY=your-secret \
  -e S3_BUCKET_NAME=your-bucket \
  hybrid-p2p/api:latest

docker run -d -p 3001:3001 hybrid-p2p/socket:latest
docker run -d -p 80:80 hybrid-p2p/frontend:latest
```

Option 3 — Kubernetes Deployment

```
# Apply secrets
kubectl apply -f infrastructure/k8s/app-secrets.yaml

# Deploy all services
kubectl apply -f infrastructure/k8s/api-deployment.yaml
kubectl apply -f infrastructure/k8s/frontend-deployment.yaml
kubectl apply -f infrastructure/k8s/socket-deployment.yaml

# Set up ingress with SSL
kubectl apply -f infrastructure/k8s/ingress.yaml

# Enable Prometheus monitoring
kubectl apply -f infrastructure/k8s/api-servicemonitor.yaml
```

Option 4 — PaaS Deployment (Render)

Pushing to the Git repository triggers render.yaml to auto-deploy the signaling server (p2p-signaling-server, Node.js) and the FastAPI backend (p2p-fastapi-backend, Python 3.11).

Environment Variables Reference

Variable	Required	Default	Description
SECRET_KEY	Yes	—	JWT HS256 signing secret
REDIS_URL	No	None (InMemoryStore)	Redis connection URI
AWS_ACCESS_KEY_ID	Yes	—	AWS IAM access key
AWS_SECRET_ACCESS_KEY	Yes	—	AWS IAM secret key
AWS_REGION	Yes	eu-north-1	AWS region for S3
S3_BUCKET_NAME	Yes	—	Target S3 bucket name
PORT	No	8000 / 3001	Dynamic port (Railway/Render)

14. Kubernetes Deployment Architecture

The Kubernetes deployment uses two Nginx Ingress resources — one routing /api traffic (with regex rewrite) to the backend, and another routing / (SPA) and /socket.io (WebSocket) traffic without rewrites, avoiding the routing collision described in Section 11.6. Each service runs behind its own ClusterIP Service and Deployment, sized independently by expected load.

Deployment	Replicas	Resource Limits
api-deployment	1	500m CPU / 512Mi RAM
frontend-deployment	2 (High Availability)	200m CPU / 128Mi RAM
socket-deployment	1	250m CPU / 256Mi RAM

A Kubernetes Secret object (app-secrets) supplies the Redis URL and AWS credentials to the API deployment. A ServiceMonitor custom resource instructs Prometheus to scrape the API's /metrics endpoint every 15 seconds. cert-manager, backed by Let's Encrypt, issues and renews TLS certificates for both Ingress resources automatically.



15. Future Enhancements

Short-Term

Enhancement	Impact
JWT validation in signaling server	Prevents UUID spoofing in WebSocket registration
Automated test suite (pytest + Cypress)	Catch regressions before deployment
CI/CD pipeline (GitHub Actions)	Automated build, test, and deploy on push
File encryption at rest	AES-256 encryption before S3 upload
Transfer history dashboard	Track past transfers with timestamps and routes used

Medium-Term

Enhancement	Impact
Redis Adapter for Socket.IO	Enable horizontal scaling of signaling to multiple replicas
Multi-file & folder transfers	Batch multiple files into a single transfer session
Resume interrupted transfers	Checkpoint-based recovery for large files
End-to-end encryption (E2EE)	Encrypt DataChannel payloads so even TURN relay is zero-knowledge
QR code pairing	Scan to connect — no lobby selection needed

Long-Term

Enhancement	Impact
Mesh networking (multi-peer)	Transfer files to multiple recipients simultaneously
Progressive Web App (PWA)	Install as a native app with offline queuing
WebTransport migration	Replace WebRTC DataChannel with HTTP/3 WebTransport for better performance
Grafana dashboards	Visualize transfer success rates, P2P vs. cloud ratio, latency percentiles
Rate limiting & abuse prevention	Token bucket rate limiting per JWT identity

16. Project File Structure

```
hybrid-p2p-engine/
|
+-- backend/                          # FastAPI REST API
|   +-- Dockerfile                    # Multi-stage, non-root Python container
|   +-- entrypoint.sh                 # Dynamic $PORT injection for PaaS
|   +-- requirements.txt              # Python dependencies
|   +-- nixpacks.toml                 # Nixpacks build config
|   +-- railway.toml                  # Railway deployment config
|   +-- app/
|       +-- main.py                   # FastAPI app, CORS, Prometheus
|       +-- api/
|       |   +-- auth.py               # POST /join -- JWT creation
|       |   +-- uploads.py            # POST /upload -- S3 multipart streaming
|       |   +-- users.py              # POST /heartbeat, GET /users/online
|       +-- core/
|       |   +-- config.py             # Environment variable loading
|       |   +-- security.py           # JWT encode/decode, Bearer dependency
|       +-- models/
|       |   +-- user.py               # Pydantic schemas
|       +-- services/
|           +-- redis_client.py       # Async Redis + InMemoryStore fallback
|
+-- frontend/                         # React SPA
|   +-- Dockerfile                    # Multi-stage Node -> Nginx container
|   +-- nginx.conf                    # SPA fallback + asset caching
|   +-- vite.config.js                # Vite + React plugin
|   +-- tailwind.config.js            # Tailwind CSS config
|   +-- package.json                  # Node dependencies
|   +-- src/
|       +-- main.jsx                  # React DOM mount
|       +-- App.jsx                   # Root component
|       +-- Dashboard.jsx             # Lobby + Dropzone + Progress UI
|       +-- index.css                 # Tailwind directives
|       +-- hooks/
|           +-- useSignaling.js       # Socket.IO + heartbeat management
|           +-- useWebRTC.js          # RTCPeerConnection + IndexedDB buffer
|           +-- useHybridTransfer.js  # 5-second P2P/Cloud race controller
|
+-- signaling/                        # WebRTC Signaling Microservice
|   +-- Dockerfile                    # Node 18 Alpine container
|   +-- index.js                      # Socket.IO event relay server
|   +-- package.json                  # Express + Socket.IO + CORS
|   +-- nixpacks.toml                 # Nixpacks config
|   +-- railway.toml                  # Railway config
|
+-- infrastructure/                   # Cloud & K8s Infrastructure
|   +-- coturn/
|   |   +-- docker-compose.yml        # STUN/TURN relay server
|   +-- k8s/
|   |   +-- api-deployment.yaml       # Backend K8s Deployment + Service
|   |   +-- frontend-deployment.yaml  # Frontend K8s Deployment + Service
|   |   +-- socket-deployment.yaml    # Signaling K8s Deployment + Service
|   |   +-- ingress.yaml              # Dual Nginx Ingress (API + Core)
|   |   +-- app-secrets.yaml          # Kubernetes Secrets
|   |   +-- api-servicemonitor.yaml   # Prometheus ServiceMonitor CRD
|   +-- lambda/
|   |   +-- nuke_protocol.py          # Burn-after-reading S3 deletion
|   |   +-- iam_policy.json           # Least-privilege Lambda IAM policy
|   +-- scripts/
|       +-- setup_s3_lifecycle.py     # S3 24h expiration + multipart abort
|
+-- start.py                          # Local dev orchestrator (all 3 services)
+-- render.yaml                       # Render PaaS multi-service deployment
+-- Steps.txt                         # Implementation checklist (18/18 complete)
+-- .gitignore                        # Dependency & env exclusions
```

17. Conclusion

The Hybrid P2P-Cloud Anonymous Transfer Engine is not just a file-transfer tool — it is a case study in building resilient distributed systems that respect user privacy. At its core, the project embodies a single architectural philosophy: *hope for the best, plan for the worst.* WebRTC offers the possibility of direct, private, zero-server file transfers, but the internet is messy — firewalls block, NATs restrict, connections drop. Rather than pretending these problems do not exist, the system embraces them as first-class design constraints.

The result is a platform where:

- Every transfer starts with maximum privacy (a direct P2P attempt).
- Every failure is handled gracefully (automatic cloud fallback).
- Every cloud-stored file has a death sentence (Lambda nuke plus a 24-hour lifecycle).
- Every component can fail independently (Redis fallback, multipart abort, Kubernetes restart policies).
- Every container runs with minimum privilege (non-root, resource-limited, secret-injected).

Building this project demonstrated that the gap between “it works on my machine” and “it works in production” is vast — filled with NAT traversal failures, dynamic port injection, regex rewrite collisions, buffer overflow bugs, and orphaned cloud storage. Each of those challenges strengthened both the system and the engineering practices behind it.

Built with FastAPI · React · WebRTC · Socket.IO · AWS S3 · Redis · Kubernetes · Docker · Prometheus