CodeSync - Real-Time Collaborative Code Editor

A production-ready, real-time collaborative code editor built with modern web technologies and deployed on AWS with a resilient, scalable infrastructure.

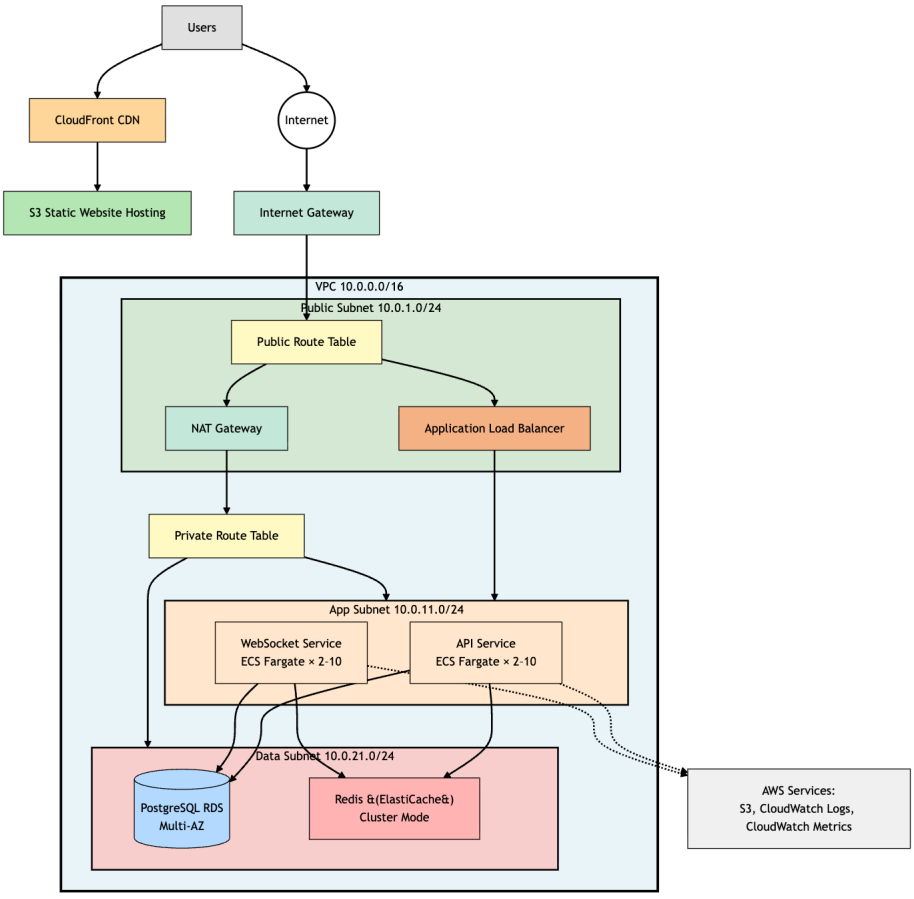
🏗️ Architecture Overview

Tech Stack:

- **Frontend**: React + TypeScript + Monaco Editor
- **Backend API**: Node.js + Express + PostgreSQL
- **WebSocket Service**: Node.js + Socket.io + Redis
- **Infrastructure**: AWS (ECS Fargate, RDS, ElastiCache, S3, CloudFront, ALB)
- **IaC**: Terraform

Key Features:

- ✅ Auto-scaling ECS services based on CPU and memory
- ✅ Health checks, automatic failover, and self-healing
- ✅ Multi-AZ deployment with eliminated SPOFs
- ✅ CloudFront CDN, Redis caching, connection pooling



🔧 Application Architecture

CodeSync follows a distributed three-tier architecture with containerized microservices:

Frontend Service

- **Technology**: React 18 + TypeScript + Vite
- **Editor**: Monaco Editor (VS Code's editor engine)
- **State Management**: React Context API for authentication
- **Communication**: REST API calls + WebSocket for real-time updates
- **Deployment**: Static files on S3, served via CloudFront CDN

Backend API Service

- **Technology**: Node.js + Express + TypeScript
- **Responsibilities**:
  - User authentication (JWT token generation/validation)
  - Document CRUD operations (create, read, update, delete)
  - Access control and permissions management
  - Rate limiting to prevent abuse
- **Data Storage**:
  - PostgreSQL for structured data (users, documents metadata, permissions)
  - S3 for document content with versioning
  - Redis for session caching and rate limiting counters
- **Routes**: `/api/auth/*`, `/api/documents/*`, `/api/users/*`

WebSocket Service

- **Technology**: Node.js + Socket.io + TypeScript
- **Responsibilities**:
  - Maintain persistent connections with active users
  - Real-time document synchronization across collaborators
  - Broadcast code changes to all clients in a document room
  - Track active users and cursor positions
- **Coordination**: Redis Pub/Sub for inter-instance message distribution
- **Route**: `/socket/*`

Data Layer

- **PostgreSQL (RDS)**: Primary database for users, documents, permissions
- **Redis (ElastiCache)**:
  - Session cache (JWT validation)
  - Rate limiting counters
  - Pub/Sub messaging (WebSocket coordination)
- **S3**: Document content storage with versioning

Communication Flow

1. **Client → Frontend**: User interactions in browser
2. **Frontend → API**: REST calls for CRUD operations (HTTP/HTTPS)
3. **Frontend → WebSocket**: Real-time collaboration (Socket.io over HTTP)
4. **API → PostgreSQL**: Structured data queries
5. **API → S3**: Document content read/write

6. **API/WebSocket → Redis**: Caching and pub/sub messaging
7. **WebSocket Instances → Redis Pub/Sub**: Cross-instance message broadcasting

Key Design Patterns

- **Microservices**: Separate API and WebSocket services for independent scaling
- **Stateless Services**: No local state in containers (enables horizontal scaling)
- **Event-Driven**: WebSocket uses pub/sub pattern for real-time updates
- **Cache-Aside**: Redis caches frequently accessed data (sessions, rate limits)
- **Operational Transformation**: Conflict resolution for concurrent edits

☁️ AWS Cloud Architecture

This project leverages multiple AWS services to create a highly available, scalable, and resilient cloud infrastructure:

Compute Services

Amazon ECS (Elastic Container Service) with Fargate

- Serverless container orchestration service
- Runs API and WebSocket services as Docker containers
- Fargate eliminates need to manage EC2 instances
- Auto-scaling based on CPU/Memory metrics (2-10 tasks per service)
- Deployed across 2 Availability Zones for high availability

Application Load Balancer (ALB)

- Layer 7 load balancer distributing traffic to ECS tasks
- Health checks every 30 seconds to ensure task availability
- Sticky sessions for WebSocket connections (maintain user-to-instance mapping)
- Routes `/api/*` to API service and `/socket/*` to WebSocket service
- Automatically removes unhealthy targets from routing pool

Storage Services

Amazon S3 (Simple Storage Service)

- Object storage for document content and frontend static files
- Document versioning enabled for content recovery
- Separate buckets: frontend assets and document storage
- Lifecycle policies for cost optimization (archival to Glacier)
- 99.999999999% (11 9's) durability

Amazon RDS (Relational Database Service) - PostgreSQL

- Managed PostgreSQL database for structured data (users, documents, permissions)
- Multi-AZ deployment with automatic failover (60 second RTO)
- Automated backups with 7-day retention
- Point-in-time recovery for disaster recovery

- Automatic minor version patches and maintenance

Amazon ElastiCache - Redis

- In-memory data store for three purposes:
  - Session caching (JWT token validation)
  - Rate limiting (prevent API abuse)
  - Pub/Sub messaging (WebSocket message broadcasting)
- Cluster mode with 2+ nodes for redundancy
- Automatic failover in ~15 seconds
- Sub-millisecond latency for cache operations

Networking & Content Delivery

Amazon VPC (Virtual Private Cloud)

- Isolated network with CIDR block 10.0.0.0/16
- Public subnets (for ALB, NAT Gateway) in 2 AZs
- Private subnets (for ECS tasks, RDS, ElastiCache) in 2 AZs
- Internet Gateway for outbound internet access
- NAT Gateway for private subnet internet access

Amazon CloudFront

- Global Content Delivery Network (CDN) with 400+ edge locations
- Caches frontend static assets close to users
- Reduces latency from 100-300ms to 10-50ms globally
- Automatic DDoS protection with AWS Shield Standard
- Custom error pages and cache behaviors

Security Services

AWS IAM (Identity and Access Management)

- ECS Task Roles for service-to-service authentication
- Principle of least privilege (only required permissions)
- No hardcoded credentials in application code
- Cross-service access control (ECS → S3, RDS, Secrets Manager)

AWS Secrets Manager

- Encrypted storage for sensitive data:
  - JWT secret for token generation
  - Database master password
- Automatic rotation capability
- Integrated with ECS for runtime secret injection
- Encryption at rest with KMS

Security Groups

- Virtual firewalls controlling traffic between services:

    - ALB → ECS tasks (ports 3000, 3001)
    - ECS tasks → RDS (port 5432)
    - ECS tasks → ElastiCache (port 6379)
- Stateful inspection (automatic return traffic)
- Deny-by-default with explicit allow rules

AWS Certificate Manager (ACM) (Optional)

- Free SSL/TLS certificates for HTTPS
- Automatic renewal (no manual intervention)
- Integration with ALB and CloudFront

Monitoring & Operations

Amazon CloudWatch

- **Metrics**: CPU, Memory, Request Count, Response Time, Database connections
- **Logs**: Centralized logging from ECS tasks (/ecs/codesync-*)
- **Dashboards**: Visual representation of 10+ key metrics
- **Alarms**: 9 configured alarms with SNS notifications
    - High CPU/Memory (ECS, RDS, Redis)
    - High response time or 5XX errors (ALB)
    - Low disk space (RDS)
    - Unhealthy host count (ALB)

Amazon SNS (Simple Notification Service)

- Email notifications when CloudWatch alarms trigger
- Configured for alert escalation
- Multi-subscriber support (email, SMS, Lambda)

Amazon ECR (Elastic Container Registry)

- Private Docker image registry
- Stores codesync-api and codesync-websocket images
- Integration with ECS for automatic image pulls
- Image scanning for security vulnerabilities

Infrastructure as Code

Terraform

- Declarative infrastructure provisioning
- Modular design (10 separate modules: VPC, security, database, cache, storage, ALB, ECS cluster, ECS service, CloudFront, monitoring)
- State management for tracking infrastructure changes
- Repeatable deployments across environments (dev/staging/prod)
- Complete infrastructure deployed in ~20 minutes

Cloud Architecture Benefits

High Availability:

- Multi-AZ deployment eliminates single points of failure
- Automatic failover for RDS, ElastiCache, and ECS tasks
- Target: 99.99% uptime (52 minutes downtime/year)

Scalability:

- Horizontal scaling: ECS tasks scale from 2 to 10+ based on demand
- Vertical scaling: Easy instance type upgrades (db.t3.micro → db.t3.large)
- Auto-scaling policies respond within 1-2 minutes

Cost Efficiency:

- Pay-per-use pricing (no upfront costs)
- Auto-scaling reduces costs during low traffic (60% savings)
- Fargate eliminates EC2 management overhead
- Development environment: ~$50-100/month

Security:

- Multiple security layers (network, application, data)
- Encryption at rest (RDS, S3) and in transit (TLS/SSL)
- Private subnets isolate application and data tiers
- No public internet access to databases

Resilience:

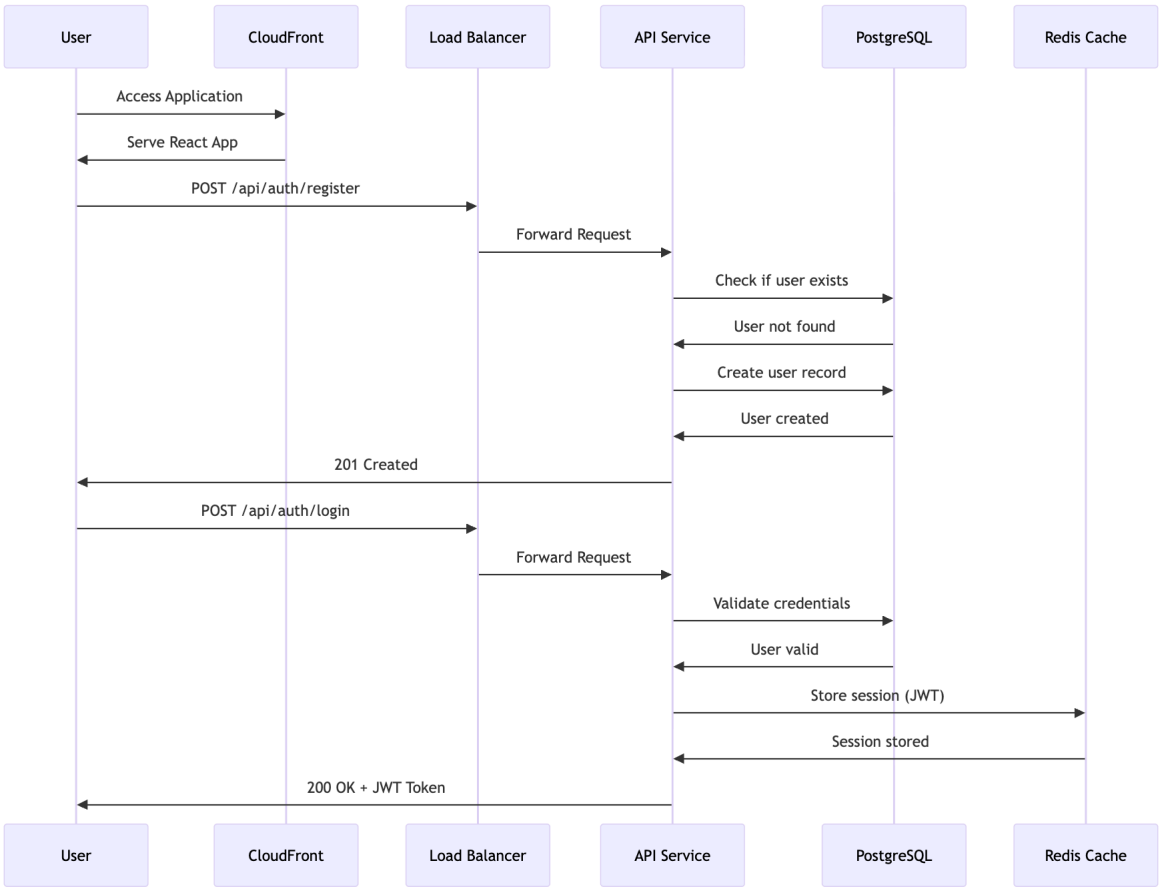
- Automated backups and point-in-time recovery
- Health checks with automatic recovery
- Data replication across availability zones
- Disaster recovery capability

🔍 How It Works

User Flow

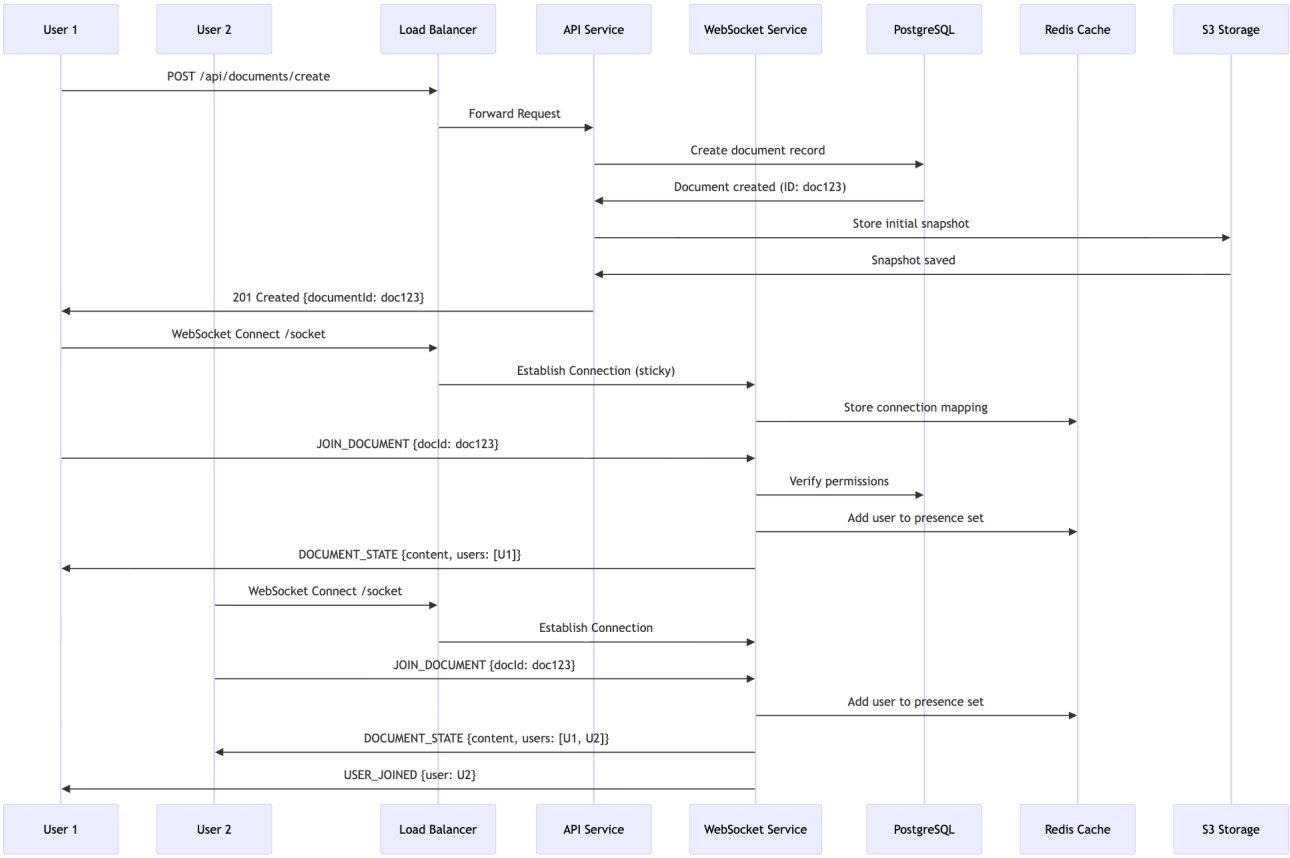
1. Registration & Login

- JWT-based authentication with tokens cached in Redis
- Credentials validated against PostgreSQL



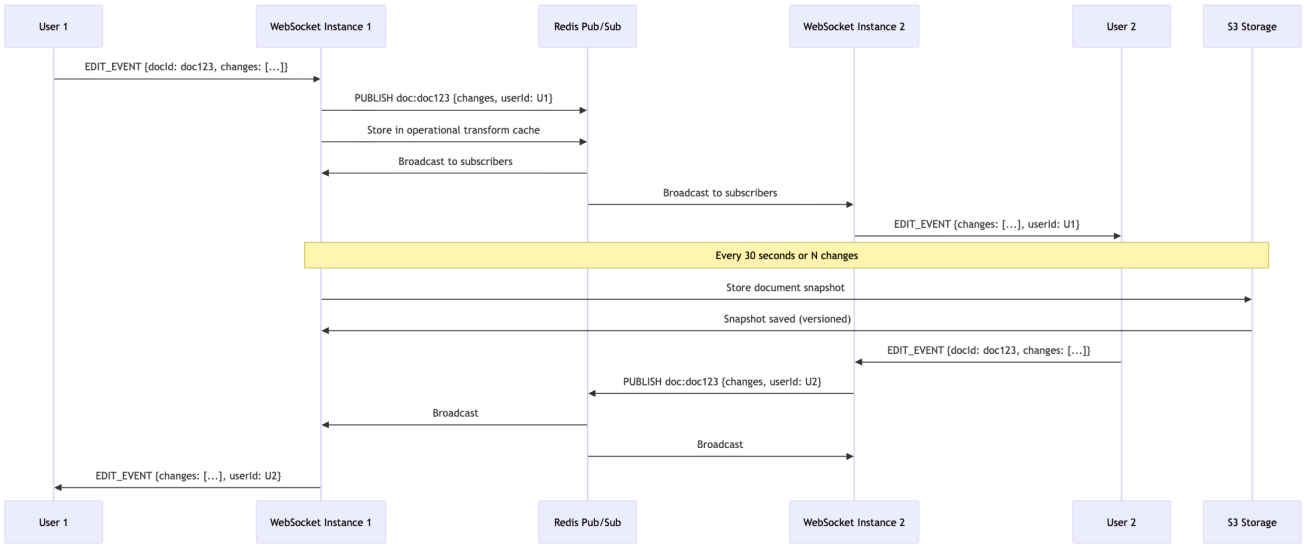
2. Document Management

- Metadata in PostgreSQL, content in S3
- Create, share, edit, and delete documents



3. Real-Time Collaboration

- Monaco Editor for code editing
- WebSocket connection with Redis Pub/Sub for message broadcasting
- Operational Transformation for conflict resolution
- Auto-save snapshots to S3 every few seconds
- Sticky sessions maintain user-to-instance mapping



Request Flow

1. **Authentication**: CloudFront → ALB → API → PostgreSQL validation → JWT cached in Redis
2. **Document Load**: API fetches metadata (PostgreSQL) and content (S3)
3. **WebSocket**: Authenticated connection joins document room
4. **Real-Time Edits**: Changes broadcast via Redis Pub/Sub with Operational Transformation
5. **Auto-Save**: Periodic snapshots to S3 every 5 seconds

Cloud Requirements Met

✅ **Elasticity**: ECS auto-scaling (2-10 tasks, responds in 60s)
✅ **Auto Recovery**: Health checks, automatic failover, 11 CloudWatch alarms
✅ **Failure Isolation**: 5 SPOFs eliminated (ALB, ECS, RDS, Redis, NAT across 2 AZs)
✅ **Performance**: CloudFront CDN, Redis caching, handles 5x traffic bursts

📋 Prerequisites

- AWS Account with appropriate permissions
- AWS CLI configured (`aws configure`)

- Terraform >= 1.0
- Docker
- Node.js >= 18

🚀 Quick Deployment

1. Create JWT Secret

```
JWT_SECRET=$(openssl rand -base64 32)
aws secretsmanager create-secret \
  --name codesync/jwt-secret \
  --secret-string "$JWT_SECRET" \
  --region us-east-1
```

2. Build and Push Docker Images

```
AWS_REGION="us-east-1"
AWS_ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output
text)

# Create ECR repositories
aws ecr create-repository --repository-name codesync-api --region
$AWS_REGION
aws ecr create-repository --repository-name codesync-websocket --region
$AWS_REGION

# Login to ECR
aws ecr get-login-password --region $AWS_REGION | \
  docker login --username AWS --password-stdin
$AWS_ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com

# Build and push API
cd backend/api
docker build -t codesync-api .
docker tag codesync-api:latest
$AWS_ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/codesync-api:latest
docker push $AWS_ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/codesync-
api:latest

# Build and push WebSocket
cd ../websocket
docker build -t codesync-websocket .
docker tag codesync-websocket:latest
$AWS_ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/codesync-
websocket:latest
docker push $AWS_ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/codesync-
websocket:latest
cd ../..
```

3. Configure and Deploy Terraform

```
cd terraform
cp terraform.tfvars.example terraform.tfvars
# Edit terraform.tfvars with your values

terraform init
terraform plan
terraform apply
```

Key variables in `terraform.tfvars`:

```
aws_region   = "us-east-1"
project_name = "codesync"
environment  = "dev"

jwt_secret_arn = "arn:aws:secretsmanager:us-east-
1:ACCOUNT_ID:secret:codesync/jwt-secret-XXXXX"
alert_email    = "your-email@example.com"
```

4. Deploy Frontend

```
cd ../frontend
npm install

# Set API URLs from Terraform outputs
export VITE_API_URL="http://YOUR_ALB_DNS"
export VITE_WS_URL="http://YOUR_ALB_DNS"

npm run build

# Upload to S3
FRONTEND_BUCKET=$(cd ../terraform && terraform output -raw
frontend_bucket_name)
aws s3 sync dist/ s3://$FRONTEND_BUCKET/ --delete

# Invalidate CloudFront cache
CLOUDFRONT_ID=$(cd ../terraform && terraform output -raw cloudfront_id)
aws cloudfront create-invalidation --distribution-id $CLOUDFRONT_ID --
paths "/*"
```

5. Verify Deployment

```
# Get CloudFront URL
CLOUDFRONT_URL=$(cd terraform && terraform output -raw cloudfront_url)
```

```
echo "Application available at: $CLOUDFRONT_URL"

# Test health endpoints
ALB_DNS=$(cd terraform && terraform output -raw alb_dns_name)
curl http://$ALB_DNS/api/health
curl http://$ALB_DNS/socket/health
```

📊 Monitoring

CloudWatch Dashboard: AWS Console → CloudWatch → Dashboards → `codesync-dev-dashboard`

Configured Alarms:

- ALB high response time, 5XX errors, unhealthy hosts
- ECS CPU/Memory high
- RDS CPU high, storage low
- Redis CPU/Memory high

View Logs:

```
aws logs tail /ecs/codesync-dev-api --follow
aws logs tail /ecs/codesync-dev-websocket --follow
```

🔄 Updates

Backend Services:

- Build and push Docker image to ECR
- Use `aws ecs update-service --force-new-deployment` to trigger rolling update

Frontend:

- Run `npm run build` and sync to S3
- Invalidate CloudFront cache with `aws cloudfront create-invalidation`

🧹 Cleanup

```
cd terraform
terraform destroy
```

Warning: Create RDS snapshot before destroying if you need to preserve data.

📈 Scaling

Dev: 2-10 ECS tasks (512 CPU, 1GB), db.t3.micro, cache.t3.micro
Prod: 4-50 ECS tasks (1024 CPU, 2GB), db.t3.medium, cache.t3.medium

Adjust in `terraform.tfvars`: `api_max_capacity`, `db_instance_class`, `redis_node_type`

📝 Local Development

```
# Start PostgreSQL and Redis
docker run -d -p 5432:5432 -e POSTGRES_PASSWORD=postgres postgres:15
docker run -d -p 6379:6379 redis:7-alpine

# API Service
cd backend/api
npm install
npm run dev

# WebSocket Service
cd backend/websocket
npm install
npm run dev

# Frontend
cd frontend
npm install
npm run dev  # Available at http://localhost:3000
```