Budget App GCP Deployment - Ultra-Detailed Step-by-Step Guide

Table of Contents

Prerequisites Setup

Step 1: Install Google Cloud SDK

macOS

```bash
# Install via Homebrew
brew install google-cloud-sdk
```

Expected Output:

```
==> Downloading https://dl.google.com/dl/cloudsdk/channels/rapid/downloads/google-cloud-sdk-...
...
🍺  google-cloud-sdk was successfully installed!
```

Verify Installation:

```bash
gcloud version
```

Expected Output:

```
Google Cloud SDK 460.0.0
bq 2.0.101
core 2024.01.19
gcloud-crc32c 1.0.0
gsutil 5.27
```

Linux/Cloud Shell

```bash
# Download and install
curl https://sdk.cloud.google.com | bash

# Restart shell or source
exec -l $SHELL

# Or source directly
source ~/.bashrc
```

Verify Installation:

```bash
which gcloud
```

Expected Output:

```
/home/youruser/google-cloud-sdk/bin/gcloud
```

Step 2: Initialize and Authenticate

```bash
# Initialize gcloud (interactive setup)
gcloud init
```

Interactive Prompts You'll See:

```
Welcome to the Google Cloud CLI!

Pick configuration to use:
 [1] Re-initialize this configuration [default] with new settings
 [2] Create a new configuration
Please enter your numeric choice: 1

Choose the account you would like to use to perform operations:
 [1] yourname@gmail.com
 [2] Log in with a new account
Please enter your numeric choice: 1
```

Then authenticate:

```bash
gcloud auth login
```

What Happens:

1. Opens browser window
2. Sign in with Google account
3. Grant permissions
4. Returns to terminal with success message

Expected Terminal Output:

```
Your browser has been opened to visit:
    https://accounts.google.com/o/oauth2/auth?...

You are now logged in as [yourname@gmail.com].
Your current project is [None].
```

Set Application Default Credentials:

```bash
gcloud auth application-default login
```

Why: Allows local development tools to authenticate with GCP

Step 3: Create GCP Project

```bash
# Set project name
export PROJECT_ID="budget-app-prod"

# Create project
gcloud projects create $PROJECT_ID --name="Budget App"
```

Expected Output:

```
Create in progress for [https://cloudresourcemanager.googleapis.com/v1/projects/budget-app-prod].
Waiting for [operations/cp.12345] to finish...done.
```

Verify Project Created:

```bash
gcloud projects describe $PROJECT_ID
```

Expected Output:

```
createTime: '2026-01-20T10:30:00.000Z'
lifecycleState: ACTIVE
name: Budget App
projectId: budget-app-prod
projectNumber: '123456789012'
```

Set as Active Project:

```bash
gcloud config set project $PROJECT_ID
```

Expected Output:

```
Updated property [core/project].
```

Verify Active Project:

```bash
gcloud config get-value project
```

Expected Output:

```
budget-app-prod
```

Step 4: Enable Billing

Via Console:

1. Go to: https://console.cloud.google.com/billing
2. Click "Link a billing account"
3. Select or create billing account
4. Click "Set account"

Verify Billing is Enabled:

```bash
gcloud beta billing projects describe $PROJECT_ID
```

Expected Output (billing enabled):

```
billingAccountName: billingAccounts/ABCDEF-123456-GHIJKL
billingEnabled: true
name: projects/budget-app-prod/billingInfo
projectId: budget-app-prod
```

⚠️ **IMPORTANT:** Without billing enabled, you cannot deploy to Cloud Run or create Cloud SQL instances.

Step 5: Enable Required APIs

```bash
# Enable all APIs at once
gcloud services enable \
  run.googleapis.com \
  sql-component.googleapis.com \
  sqladmin.googleapis.com \
  cloudbuild.googleapis.com \
  secretmanager.googleapis.com \
  compute.googleapis.com \
  vpcaccess.googleapis.com \
  artifactregistry.googleapis.com
```

Expected Output:

```
Operation "operations/acat.p2-123456789012-..." finished successfully.
```

Verify APIs are Enabled:

```bash
gcloud services list --enabled | grep -E 'run|sql|build|secret|compute|vpc'
```

Expected Output:

```
cloudbuild.googleapis.com            Cloud Build API
compute.googleapis.com               Compute Engine API
run.googleapis.com                   Cloud Run Admin API
secretmanager.googleapis.com         Secret Manager API
sql-component.googleapis.com         Cloud SQL
sqladmin.googleapis.com              Cloud SQL Admin API
vpcaccess.googleapis.com             Serverless VPC Access API
```

✅ **Checkpoint:** All prerequisites complete! Time: ~15 minutes

---

Phase 1: Application Preparation

Step 1.1: Set Up Project Structure

Create Next.js Project:

```bash
# Navigate to your projects directory
cd ~/projects

# Create Next.js app
npx create-next-app@latest budget-app

# Select options:
# ✔ Would you like to use TypeScript? Yes
# ✔ Would you like to use ESLint? Yes
# ✔ Would you like to use Tailwind CSS? Yes
# ✔ Would you like to use `src/` directory? Yes
# ✔ Would you like to use App Router? Yes
# ✔ Would you like to customize the default import alias? No

cd budget-app
```

Expected Directory Structure:

```
budget-app/
├── src/
│   └── app/
│       ├── favicon.ico
│       ├── globals.css
│       ├── layout.tsx
│       └── page.tsx
├── public/
├── next.config.js
├── package.json
├── tsconfig.json
└── tailwind.config.ts
```

Install Additional Dependencies:

```bash
# Database client
npm install @prisma/client
npm install -D prisma

# Authentication
npm install next-auth
npm install @auth/prisma-adapter

# Environment variables
npm install dotenv

# HTTP client
npm install axios
```

Expected Output:

```
added 127 packages, and audited 367 packages in 8s

123 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Step 1.2: Initialize Prisma

```bash
# Initialize Prisma
npx prisma init
```

Expected Output:

```
✔ Your Prisma schema was created at prisma/schema.prisma
  You can now open it in your favorite editor.

Next steps:
1. Set the DATABASE_URL in the .env file to point to your existing database
2. Run prisma db pull to turn your database schema into a Prisma schema
3. Run prisma generate to generate the Prisma Client
```

Files Created:

```
prisma/
└── schema.prisma
.env
```

Edit prisma/schema.prisma:

prisma

```prisma
// This is your Prisma schema file

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

// User model
model User {
  id           String    @id @default(cuid())
  email        String    @unique
  name         String?
  createdAt    DateTime  @default(now())
  updatedAt    DateTime  @updatedAt

  incomeSources IncomeSource[]
  categories    ExpenseCategory[]
  transactions  Transaction[]
  goals         SavingsGoal[]
}

// Income source
model IncomeSource {
  id        String   @id @default(cuid())
  userId    String
  user      User     @relation(fields: [userId], references: [id])
  name      String
  amount    Decimal  @db.Decimal(10, 2)
  frequency String   // monthly, biweekly, weekly
  isActive  Boolean  @default(true)
  createdAt DateTime @default(now())
}

// Expense category
model ExpenseCategory {
  id             String       @id @default(cuid())
  userId         String
  user           User         @relation(fields: [userId], references: [id])
  name           String
  parentCategory  String
  budgetedAmount  Decimal      @db.Decimal(10, 2)
  isActive       Boolean      @default(true)
  transactions    Transaction[]
}

// Transaction
model Transaction {
  id          String          @id @default(cuid())
  userId      String
  user        User            @relation(fields: [userId], references: [id])
  date        DateTime
  amount      Decimal         @db.Decimal(10, 2)
  description String?
  type        String          // income, expense
  categoryId  String?
  category    ExpenseCategory? @relation(fields: [categoryId], references: [id])
  createdAt   DateTime        @default(now())
}

// Savings goal
model SavingsGoal {
  id           String   @id @default(cuid())
  userId       String
  user         User     @relation(fields: [userId], references: [id])
  name         String
  targetAmount  Decimal  @db.Decimal(10, 2)
  currentAmount Decimal  @db.Decimal(10, 2) @default(0)
  deadline      DateTime?
  priority      Int      @default(1)
  createdAt     DateTime @default(now())
}
```
Save the file.

Step 1.3: Create Dockerfile

Create Dockerfile in project root:

dockerfile

```dockerfile
# Dockerfile
FROM node:18-alpine AS base

# Install dependencies only when needed
FROM base AS deps
RUN apk add --no-cache libc6-compat
WORKDIR /app

# Copy package files
COPY package.json package-lock.json* ./
RUN npm ci

# Rebuild source code only when needed
FROM base AS builder
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY . .

# Generate Prisma Client
RUN npx prisma generate

# Disable telemetry during build
ENV NEXT_TELEMETRY_DISABLED 1

# Build Next.js
RUN npm run build

# Production image, copy all files and run next
FROM base AS runner
WORKDIR /app

ENV NODE_ENV production
ENV NEXT_TELEMETRY_DISABLED 1

RUN addgroup --system --gid 1001 nodejs
RUN adduser --system --uid 1001 nextjs

# Copy necessary files
COPY --from=builder /app/public ./public
COPY --from=builder --chown=nextjs:nodejs /app/.next/standalone ./
COPY --from=builder --chown=nextjs:nodejs /app/.next/static ./.next/static
COPY --from=builder /app/prisma ./prisma
COPY --from=builder /app/node_modules/.prisma ./node_modules/.prisma

USER nextjs
USER nextjs

EXPOSE 8080

ENV PORT 8080
ENV HOSTNAME "0.0.0.0"

CMD ["node", "server.js"]
```

Why Each Section:

- `deps stage`: Install dependencies in isolated layer for caching
- `builder stage`: Build the app
- `runner stage`: Minimal production image
- `nodejs user`: Security best practice (don't run as root)
- `PORT 8080`: Cloud Run requirement

Step 1.4: Create .dockerignore

```bash
# Create .dockerignore
cat > .dockerignore << 'EOF'
# Dependencies
node_modules
npm-debug.log*
yarn-debug.log*
yarn-error.log*

# Build output
.next
out
dist
build

# Git
.git
.gitignore

# Environment
.env
.env.local
.env.*.local

# IDE
.vscode
.idea
*.swp
*.swo

# OS
.DS_Store
Thumbs.db

# Misc
README.md
.prettierrc
.eslintrc.json
EOF
```

Verify File Created:

```bash
cat .dockerignore
```

Step 1.5: Update next.config.js

```javascript
// next.config.js
/** @type {import('next').NextConfig} */
const nextConfig = {
  // Required for Cloud Run deployment
  output: 'standalone',

  // External packages for Prisma
  experimental: {
    serverComponentsExternalPackages: ['@prisma/client', 'prisma']
  },

  // Optional: Enable compression
  compress: true,

  // Optional: Optimize images
  images: {
    domains: ['storage.googleapis.com'], // If using Cloud Storage
  }
}

module.exports = nextConfig
```

Why:

- `output: 'standalone'`: Creates minimal production build for containers
- `serverComponentsExternalPackages`: Prevents Prisma bundling issues
- `compress`: Built-in gzip compression

Verify Configuration:

```bash
npm run build
```

Expected Output:

```
  ✓ Creating an optimized production build
  ✓ Compiled successfully
  ✓ Linting and checking validity of types
  ✓ Collecting page data
  ✓ Generating static pages (5/5)
  ✓ Collecting build traces
  ✓ Finalizing page optimization

Route (app)                              Size     First Load JS
┌ ○ /                                    137 B          87.2 kB
└ ○ /_not-found                          871 B          87.9 kB
+ First Load JS shared by all            87.1 kB
  ├ chunks/864-...                       26.8 kB
  ├ chunks/fd9d1056-...                  53.4 kB
  ├ chunks/main-app-...                  227 B
  └ chunks/webpack-...                   6.66 kB

○  (Static)  prerendered as static content
```

✅ **Checkpoint:** Application structure ready for containerization! Time: ~20 minutes

---

Phase 2: Database Setup

Step 2.1: Set Environment Variables

```bash
# Set variables for easy reuse
export PROJECT_ID="budget-app-prod"
export REGION="us-central1"
export INSTANCE_NAME="budget-db"
export DB_NAME="budgetapp"
export DB_USER="postgres"

# Generate secure random password
export DB_PASSWORD="$(openssl rand -base64 32)"

# Display password (save this!)
echo "Database Password: $DB_PASSWORD"
```

Expected Output:

```
Database Password: xK7mP2nQ9vL4wE8sR3jF6tY1hU5zN0bA8cD==
```

⚠️ **IMPORTANT:** Save this password! You'll need it later.

Save to file (optional):

```bash
echo "DB_PASSWORD=$DB_PASSWORD" >> ~/budget-app-secrets.txt
chmod 600 ~/budget-app-secrets.txt
```

Step 2.2: Create Cloud SQL Instance

```bash
# Create PostgreSQL instance
gcloud sql instances create $INSTANCE_NAME \
  --database-version=POSTGRES_15 \
  --tier=db-f1-micro \
  --region=$REGION \
  --network=default \
  --no-assign-ip \
  --root-password="$DB_PASSWORD" \
  --backup-start-time=03:00
```

What This Does:

- `--database-version`: Uses PostgreSQL 15
- `--tier=db-f1-micro`: Smallest (cheapest) instance (~$7/month)
- `--region`: Where instance runs (choose closest to users)
- `--network=default`: Uses default VPC network
- `--no-assign-ip`: No public IP (more secure, private only)
- `--backup-start-time`: Automated daily backups at 3 AM

Expected Output:

```
Create request issued for: [budget-db]
Waiting for operation to complete...done.
Created [https://sqladmin.googleapis.com/sql/v1beta4/projects/budget-app-prod/instances/budget-db].
NAME      DATABASE_VERSION  LOCATION       TIER         PRIMARY_ADDRESS  PRIVATE_ADDRESS  STATUS
budget-db  POSTGRES_15      us-central1-b  db-f1-micro  -                10.123.45.67     RUNNABLE
```

⏱️ **Time:** This takes 5-10 minutes

Verify Instance is Running:

```bash
gcloud sql instances describe $INSTANCE_NAME
```

Expected Output (key fields):

```yaml
state: RUNNABLE
databaseVersion: POSTGRES_15
settings:
  tier: db-f1-micro
  ipConfiguration:
    ipv4Enabled: false
    privateNetwork: projects/budget-app-prod/global/networks/default
```

Step 2.3: Create Database

```bash
# Create database within instance
gcloud sql databases create $DB_NAME \
  --instance=$INSTANCE_NAME
```

Expected Output:

```
Creating Cloud SQL database...done.
Created database [budgetapp].
instance: budget-db
name: budgetapp
project: budget-app-prod
```

Verify Database Created:

```bash
gcloud sql databases list --instance=$INSTANCE_NAME
```

Expected Output:

```
NAME       CHARSET  COLLATION
budgetapp  UTF8     en_US.UTF8
postgres   UTF8     en_US.UTF8
template0  UTF8     en_US.UTF8
template1  UTF8     en_US.UTF8
```

Step 2.4: Store Password in Secret Manager

```bash
# Create secret
echo -n "$DB_PASSWORD" | gcloud secrets create db-password \
  --data-file=- \
  --replication-policy="automatic"
```

Expected Output:

```
Created secret [db-password].
```

Verify Secret Created:

```bash
gcloud secrets describe db-password
```

Expected Output:

```
createTime: '2026-01-20T12:00:00.000Z'
name: projects/123456789012/secrets/db-password
replication:
  automatic: {}
```

Test Secret Retrieval:

```bash
gcloud secrets versions access latest --secret="db-password"
```

Expected Output:

```
xK7mP2nQ9vL4wE8sR3jF6tY1hU5zN0bA8cD==
```

Step 2.5: Set Up Cloud SQL Proxy (for Local Development)

```bash
# Download Cloud SQL Proxy
curl -o cloud-sql-proxy \
  https://storage.googleapis.com/cloud-sql-connectors/cloud-sql-proxy/v2.8.2/cloud-sql-proxy.linux.amd64

# Make executable
chmod +x cloud-sql-proxy

# Move to PATH
sudo mv cloud-sql-proxy /usr/local/bin/
```

Verify Installation:

```bash
cloud-sql-proxy --version
```

Expected Output:

```
Cloud SQL Proxy v2.8.2
```

Get Connection Name:

```bash
export CONNECTION_NAME=$(gcloud sql instances describe $INSTANCE_NAME \
  --format='value(connectionName)')

echo "Connection Name: $CONNECTION_NAME"
```

Expected Output:

```
Connection Name: budget-app-prod:us-central1:budget-db
```

Start Proxy (in separate terminal):

```bash
cloud-sql-proxy $CONNECTION_NAME
```

Expected Output:

```
2026/01/20 12:00:00 Listening on 127.0.0.1:5432
2026/01/20 12:00:00 The proxy has started successfully and is ready for new connections!
```

✅ **Keep this terminal open while developing locally**

Step 2.6: Configure Local Environment

Create .env file in your project:

```bash
cd ~/projects/budget-app

cat > .env << EOF
# Database
DATABASE_URL="postgresql://$DB_USER:$DB_PASSWORD@127.0.0.1:5432/$DB_NAME"

# NextAuth
NEXTAUTH_SECRET="$(openssl rand -base64 32)"
NEXTAUTH_URL="http://localhost:3000"

# App
NODE_ENV="development"
EOF
```

Verify .env file:

```bash
cat .env
```

Expected Output:

```
# Database
DATABASE_URL="postgresql://postgres:xK7mP2nQ9vL4wE8sR3jF6tY1hU5zN0bA8cD==@127.0.0.1:5432/
budgetapp"

# NextAuth
NEXTAUTH_SECRET="aB3cD4eF5gH6iJ7kL8mN9oP0qR1sT2uV3wX==
NEXTAUTH_URL="http://localhost:3000"

# App
NODE_ENV="development"
```

Step 2.7: Run Database Migrations

```bash
# Generate Prisma client
npx prisma generate
```

Expected Output:

```
✔ Generated Prisma Client (v5.8.0) to ./node_modules/@prisma/client in 89ms

Start using Prisma Client in Node.js (See: https://pris.ly/d/client)
```

Create and run migrations:

```bash
# Create migration
npx prisma migrate dev --name init
```

Expected Output:

```
Environment variables loaded from .env
Prisma schema loaded from prisma/schema.prisma
Datasource "db": PostgreSQL database "budgetapp", schema "public" at "127.0.0.1:5432"

PostgreSQL database budgetapp created at 127.0.0.1:5432

Applying migration `20260120120000_init`

The following migration(s) have been created and applied from new schema changes:

migrations/
  └─ 20260120120000_init/
    └─ migration.sql

Your database is now in sync with your schema.

✔ Generated Prisma Client (v5.8.0) to ./node_modules/@prisma/client in 76ms
```

Verify Tables Created:

```bash
npx prisma studio
```

What Happens:

- Opens Prisma Studio in browser at http://localhost:5555
- You should see all your tables: User, IncomeSource, ExpenseCategory, Transaction, SavingsGoal

✅ **Checkpoint:** Database fully configured and migrations applied! Time: ~25 minutes

---

Phase 3: Networking Configuration

Step 3.1: Create VPC Connector

Why: Cloud Run needs a VPC connector to access Cloud SQL via private IP

```bash
# Create VPC Access connector
gcloud compute networks vpc-access connectors create budget-connector \
  --network=default \
  --region=$REGION \
  --range=10.8.0.0/28
```

What This Does:

- `--network=default`: Uses default VPC network
- `--region`: Must match Cloud Run region
- `--range`: IP range for connector (small /28 = 16 IPs)

Expected Output:

```
Create request issued for: [budget-connector]
Waiting for operation to complete.......done.
Created connector [budget-connector].
```

⏱️ **Time:** 2-3 minutes

Verify Connector Created:

```bash
gcloud compute networks vpc-access connectors describe budget-connector \
  --region=$REGION
```

Expected Output:

```
ipCidrRange: 10.8.0.0/28
name: projects/budget-app-prod/locations/us-central1/connectors/budget-connector
network: default
state: READY
```

⚠️ **IMPORTANT:** Wait for `state: READY` before proceeding

Step 3.2: Configure Firewall Rules (if needed)

Check existing firewall rules:

```bash
gcloud compute firewall-rules list --filter="network:default"
```

For private Cloud SQL, you typically don't need additional rules, but verify:

```bash
# Check if Cloud SQL can be reached
gcloud compute firewall-rules describe default-allow-internal
```

Expected Output:

```
allowed:
- IPProtocol: tcp
  ports:
  - '0-65535'
- IPProtocol: udp
  ports:
  - '0-65535'
- IPProtocol: icmp
```

✅ **Checkpoint:** Network configured for Cloud Run ↔ Cloud SQL communication! Time: ~5 minutes

---

Phase 4: Build & Deploy

Step 4.1: Build Container Image

Method 1: Using Cloud Build (Recommended)

```bash
cd ~/projects/budget-app

# Submit build to Cloud Build
gcloud builds submit --tag gcr.io/$PROJECT_ID/budget-app
```

What Happens:

1. Uploads source code to Cloud Storage
2. Builds Docker image using your Dockerfile
3. Pushes image to Google Container Registry
4. Returns image URL

Expected Output:

```
Creating temporary archive of 234 file(s) totalling 45.2 MiB before compression.
Uploading tarball of [.] to [gs://budget-app-prod_cloudbuild/source/...]
Created [https://cloudbuild.googleapis.com/v1/projects/budget-app-prod/builds/...]
Logs are available at [https://console.cloud.google.com/cloud-build/builds/...].
-------------------------------- REMOTE BUILD OUTPUT --------------------------------
starting build "abc123..."

FETCHSOURCE
Fetching storage object: gs://budget-app-prod_cloudbuild/source/...
Copying gs://...

BUILD
Already have image (with digest): gcr.io/cloud-builders/docker
Sending build context to Docker daemon  47.32MB
Step 1/20 : FROM node:18-alpine AS base
...
Step 20/20 : CMD ["node", "server.js"]
Successfully built 1234567890ab
Successfully tagged gcr.io/budget-app-prod/budget-app:latest

PUSH
Pushing gcr.io/budget-app-prod/budget-app
...
latest: digest: sha256:abc123... size: 4321

DONE
```

⏱️ **Time:** 5-8 minutes for first build

Method 2: Build Locally (Alternative)

```bash
# Build locally
docker build -t gcr.io/$PROJECT_ID/budget-app .

# Push to GCR
docker push gcr.io/$PROJECT_ID/budget-app
```

Verify Image in Container Registry:

```bash
gcloud container images list
```

Expected Output:

```
NAME
gcr.io/budget-app-prod/budget-app
```

Check image details:

```bash
gcloud container images describe gcr.io/$PROJECT_ID/budget-app:latest
```

Step 4.2: Create Production Environment Secrets

Create NEXTAUTH_SECRET:

```bash
# Generate and store NextAuth secret
openssl rand -base64 32 | gcloud secrets create nextauth-secret \
  --data-file=- \
  --replication-policy="automatic"
```

Expected Output:

```
Created secret [nextauth-secret].
```

Verify Secrets:

```bash
gcloud secrets list
```

Expected Output:

```
NAME           CREATED           REPLICATION_POLICY  LOCATIONS
db-password      2026-01-20T12:00:00  automatic          -
nextauth-secret   2026-01-20T13:00:00  automatic          -
```

Step 4.3: Deploy to Cloud Run

```bash
# Get connection name if not already set
export CONNECTION_NAME=$(gcloud sql instances describe $INSTANCE_NAME \
  --format='value(connectionName)')

# Get database password from Secret Manager
export DB_PASSWORD=$(gcloud secrets versions access latest --secret="db-password")

# Deploy to Cloud Run
gcloud run deploy budget-app \
  --image gcr.io/$PROJECT_ID/budget-app:latest \
  --platform managed \
  --region $REGION \
  --allow-unauthenticated \
  --set-env-vars "DATABASE_URL=postgresql://postgres:$DB_PASSWORD@localhost:5432/$DB_NAME?host=/clou
  --set-env-vars "NEXTAUTH_URL=https://budget-app-PLACEHOLDER.run.app" \
  --set-secrets "NEXTAUTH_SECRET=nextauth-secret:latest" \
  --add-cloudsql-instances $CONNECTION_NAME \
  --vpc-connector budget-connector \
  --vpc-egress private-ranges-only \
  --min-instances 0 \
  --max-instances 10 \
  --memory 512Mi \
  --cpu 1 \
  --timeout 300 \
  --concurrency 80 \
  --port 8080
```

What Each Flag Does:

- `--image`: Container image to deploy
- `--platform managed`: Use fully managed Cloud Run
- `--region`: Where to deploy
- `--allow-unauthenticated`: Public access (change for production)
- `--set-env-vars`: Environment variables
- `--set-secrets`: Mount secrets as env vars
- `--add-cloudsql-instances`: Connect to Cloud SQL
- `--vpc-connector`: Use VPC connector for private IP
- `--vpc-egress private-ranges-only`: Only private traffic uses VPC
- `--min-instances 0`: Scale to zero when idle
- `--max-instances 10`: Max concurrent instances
- `--memory 512Mi`: RAM per instance
- `--cpu 1`: CPUs per instance
- `--timeout 300`: Request timeout (5 min)
- `--concurrency 80`: Requests per instance
- `--port 8080`: Container port

Expected Output:

```
Deploying container to Cloud Run service [budget-app] in project [budget-app-prod] region [us-central1]
✓ Deploying new service... Done.
  ✓ Creating Revision...
  ✓ Routing traffic...
Done.
Service [budget-app] revision [budget-app-00001-abc] has been deployed and is serving 100 percent of traffic.
Service URL: https://budget-app-abc123xyz-uc.a.run.app
```

⏱️ **Time:** 2-3 minutes

Save the Service URL:

```bash
export SERVICE_URL=$(gcloud run services describe budget-app \
  --region $REGION \
  --format 'value(status.url)')

echo "Service URL: $SERVICE_URL"
```

Step 4.4: Update NEXTAUTH_URL

```bash
# Update deployment with correct URL
gcloud run services update budget-app \
  --region $REGION \
  --set-env-vars "NEXTAUTH_URL=$SERVICE_URL"
```

Expected Output:

```
✓ Deploying... Done.
  ✓ Creating Revision...
  ✓ Routing traffic...
Service [budget-app] revision [budget-app-00002-def] has been deployed and is serving 100 percent of traffic.
```

Step 4.5: Grant Secret Access

```bash
# Get Cloud Run service account
export SERVICE_ACCOUNT=$(gcloud run services describe budget-app \
  --region=$REGION \
  --format='value(spec.template.spec.serviceAccountName)')

echo "Service Account: $SERVICE_ACCOUNT"

# Grant access to secrets
for secret in db-password nextauth-secret; do
  gcloud secrets add-iam-policy-binding $secret \
    --member="serviceAccount:$SERVICE_ACCOUNT" \
    --role="roles/secretmanager.secretAccessor"
done
```

Expected Output (for each secret):

```
Updated IAM policy for secret [db-password].
bindings:
- members:
  - serviceAccount:123456789012-compute@developer.gserviceaccount.com
  role: roles/secretmanager.secretAccessor
...
```

Step 4.6: Test Deployment

```bash
# Test the deployment
curl $SERVICE_URL
```

Expected Output: Should return HTML from your Next.js app

Test API endpoint:

```bash
curl $SERVICE_URL/api/health
```

Check logs:

```bash
gcloud run services logs tail budget-app --region $REGION
```

Expected Log Output:

```
2026-01-20 13:30:00.123 INFO Request received: GET /
2026-01-20 13:30:00.456 INFO Database connected successfully
2026-01-20 13:30:00.789 INFO Response sent: 200 OK
```

✅ **Checkpoint:** App deployed and running on Cloud Run! Time: ~15 minutes

---

Phase 5: Custom Domain (Optional)

Step 5.1: Verify Domain Ownership

Via Google Search Console:

1. Go to https://search.google.com/search-console
2. Add property: `budget.yourdomain.com`
3. Verify ownership (DNS TXT record or HTML file)

Step 5.2: Map Custom Domain

```bash
# Map domain to Cloud Run service
gcloud run domain-mappings create \
  --service budget-app \
  --domain budget.yourdomain.com \
  --region $REGION
```

Expected Output:

```
Mapping [budget.yourdomain.com] to service [budget-app]...done.

Please add the following DNS records:
  NAME                    TYPE  DATA
  budget.yourdomain.com.  A     216.239.32.21
  budget.yourdomain.com.  A     216.239.34.21
  budget.yourdomain.com.  A     216.239.36.21
  budget.yourdomain.com.  A     216.239.38.21
  budget.yourdomain.com.  AAAA  2001:4860:4802:32::15
  budget.yourdomain.com.  AAAA  2001:4860:4802:34::15
  budget.yourdomain.com.  AAAA  2001:4860:4802:36::15
  budget.yourdomain.com.  AAAA  2001:4860:4802:38::15
```

Step 5.3: Configure DNS

Add these records to your DNS provider:

For A records (IPv4):

```
Type: A
Name: budget
TTL: 3600
Data: 216.239.32.21
Data: 216.239.34.21
Data: 216.239.36.21
Data: 216.239.38.21
```

For AAAA records (IPv6):

```
Type: AAAA
Name: budget
TTL: 3600
Data: 2001:4860:4802:32::15
Data: 2001:4860:4802:34::15
Data: 2001:4860:4802:36::15
Data: 2001:4860:4802:38::15
```

Step 5.4: Verify Domain Mapping

```bash
# Check status
gcloud run domain-mappings describe budget.yourdomain.com \
  --region $REGION
```

Expected Output:

```
apiVersion: domains.cloudrun.com/v1
kind: DomainMapping
metadata:
  name: budget.yourdomain.com
spec:
  routeName: budget-app
status:
  conditions:
  - status: "True"
    type: Ready
  observedGeneration: 1
  url: https://budget.yourdomain.com
```

⏱️ **DNS propagation:** 10 minutes to 48 hours (usually < 1 hour)

Test custom domain:

```bash
curl https://budget.yourdomain.com
```

✅ **Checkpoint:** Custom domain configured! Time: ~5 minutes + DNS propagation

---

Phase 6: CI/CD Pipeline

Step 6.1: Create cloudbuild.yaml

yaml

```yaml
# cloudbuild.yaml
steps:
  # Build the container image
  - name: 'gcr.io/cloud-builders/docker'
    args:
      - 'build'
      - '-t'
      - 'gcr.io/$PROJECT_ID/budget-app:$COMMIT_SHA'
      - '-t'
      - 'gcr.io/$PROJECT_ID/budget-app:latest'
      - '.'

  # Push the container image to Container Registry
  - name: 'gcr.io/cloud-builders/docker'
    args:
      - 'push'
      - '--all-tags'
      - 'gcr.io/$PROJECT_ID/budget-app'

  # Run database migrations
  - name: 'gcr.io/cloud-builders/gcloud'
    entrypoint: bash
    args:
      - '-c'
      - |
        echo "Running database migrations..."
        # This would connect via Cloud SQL Proxy and run migrations
        # For now, migrations are run manually

  # Deploy container image to Cloud Run
  - name: 'gcr.io/google.com/cloudsdktool/cloud-sdk'
    entrypoint: gcloud
    args:
      - 'run'
      - 'deploy'
      - 'budget-app'
      - '--image=gcr.io/$PROJECT_ID/budget-app:$COMMIT_SHA'
      - '--region=us-central1'
      - '--platform=managed'

images:
  - 'gcr.io/$PROJECT_ID/budget-app:$COMMIT_SHA'
  - 'gcr.io/$PROJECT_ID/budget-app:latest'
```

```
options:
  logging: CLOUD_LOGGING_ONLY
  machineType: 'N1_HIGHCPU_8'

timeout: '1200s'  # 20 minutes
```

Save this file in your project root.

Step 6.2: Connect GitHub Repository

Install Cloud Build GitHub App:

1. Visit: https://github.com/apps/google-cloud-build
2. Click "Install"
3. Select repositories
4. Authorize

Create Build Trigger:

```bash
# Create trigger via gcloud
gcloud builds triggers create github \
  --name="budget-app-deploy" \
  --repo-name=budget-app \
  --repo-owner=YOUR_GITHUB_USERNAME \
  --branch-pattern="^main$" \
  --build-config=cloudbuild.yaml \
  --description="Deploy budget app on push to main"
```

Expected Output:

```
Created [https://cloudbuild.googleapis.com/v1/projects/budget-app-prod/locations/global/triggers/...].
NAME              TRIGGER_TEMPLATE.BRANCH_NAME  TRIGGER_TEMPLATE.REPO_NAME  CREATE_TIME
budget-app-deploy   main                          budget-app                  2026-01-20T14:00:00+00:00
```

Verify Trigger:

```bash
gcloud builds triggers list
```

Step 6.3: Test CI/CD

```bash
# Make a change and push
cd ~/projects/budget-app
echo "# CI/CD test" >> README.md
git add README.md
git commit -m "Test CI/CD pipeline"
git push origin main
```

Watch Build:

```bash
gcloud builds list --limit 1
```

Expected Output:

```
ID                    CREATE_TIME          DURATION  SOURCE  STATUS
abc123-def456-789              2026-01-20T14:05:00+00:00  5M30S    GitHub  SUCCESS
```

View Build Logs:

```bash
gcloud builds log $(gcloud builds list --limit 1 --format='value(id)')
```

✅ **Checkpoint:** CI/CD pipeline active! Every push to main auto-deploys. Time: ~15 minutes

---

Phase 7: Monitoring & Alerts

Step 7.1: View Logs

Real-time logs:

```bash
gcloud run services logs tail budget-app --region $REGION
```

Filter logs by severity:

```bash
gcloud logging read "resource.type=cloud_run_revision AND severity>=ERROR" \
  --limit 50 \
  --format json
```

Step 7.2: Set Up Uptime Checks

```bash
# Create uptime check
gcloud monitoring uptime-check-configs create budget-app-uptime \
  --display-name="Budget App Uptime" \
  --http-check-path="/" \
  --http-check-port=443 \
  --monitored-resource-type="uptime_url" \
  --monitored-resource="host=$( echo $SERVICE_URL | sed 's|https://||' )"
```

Expected Output:

```
Created uptime check [budget-app-uptime].
```

Verify:

```bash
gcloud monitoring uptime-check-configs list
```

Step 7.3: Create Alert Policies

Create alert for high error rate:

```bash
# First, create notification channel (email)
gcloud alpha monitoring channels create \
  --display-name="Email Alerts" \
  --type=email \
  --channel-labels=email_address=your@email.com
```

Get channel ID:

```bash
export CHANNEL_ID=$(gcloud alpha monitoring channels list \
  --filter="displayName:'Email Alerts'" \
  --format="value(name)")

echo "Channel ID: $CHANNEL_ID"
```

Create alert policy:

```bash
cat > alert-policy.yaml << EOF
displayName: "High Error Rate"
conditions:
  - displayName: "Error rate > 5%"
    conditionThreshold:
      filter: 'resource.type="cloud_run_revision" AND metric.type="run.googleapis.com/request_count" AND metric.la
      comparison: COMPARISON_GT
      thresholdValue: 0.05
      duration: 60s
notificationChannels:
  - $CHANNEL_ID
EOF

gcloud alpha monitoring policies create --policy-from-file=alert-policy.yaml
```

Verify Alerts:

```bash
gcloud alpha monitoring policies list
```

✅ **Checkpoint:** Monitoring and alerts configured! Time: ~10 minutes

---

Troubleshooting

Issue: Build Fails

Check build logs:

```bash
gcloud builds log $(gcloud builds list --limit 1 --format='value(id)')
```

Common fixes:

- Check Dockerfile syntax
- Verify package.json dependencies
- Increase Cloud Build timeout

Issue: Cloud Run Can't Connect to Database

Check VPC connector:

```bash
gcloud compute networks vpc-access connectors describe budget-connector --region=$REGION
```

Verify Cloud SQL instance:

```bash
gcloud sql instances describe $INSTANCE_NAME
```

Check service account permissions:

```bash
gcloud projects get-iam-policy $PROJECT_ID
```

Issue: 503 Errors

Check Cloud Run logs:

```bash
gcloud run services logs tail budget-app --region $REGION
```

Increase memory:

```bash
gcloud run services update budget-app \
  --memory 1Gi \
  --region $REGION
```

Total Deployment Time

- Prerequisites: ~15 minutes
- Phase 1: ~20 minutes
- Phase 2: ~25 minutes
- Phase 3: ~5 minutes
- Phase 4: ~15 minutes
- Phase 5: ~5 minutes + DNS
- Phase 6: ~15 minutes
- Phase 7: ~10 minutes

Total: ~2 hours (excluding DNS propagation)

Monthly Cost: $10-30 for low-traffic app