Below is a **detailed task implementation guide** for the **Backend Developer** working on the **VerseProjection SaaS web app** MVP, focused on real-time Bible verse auto-detection and projection using public-domain KJV/WEB texts. This guide aligns with the updated PRD (artifact_id: a7c7a8e1-9f68-4929-bc62-8d5d19d66186, artifact_version_id: ca07a5bd-a168-4b01-9a91-8cc217441994), emphasizing continuous speech analysis, top 10 verse matching, user selection, and model learning via user feedback. The Backend Developer's role is to build and maintain the server-side infrastructure, including APIs, database management, WebSocket communication, and cloud integration, to support the AI pipeline (Whisper, BERT) and frontend (React). The system must achieve <2-second end-to-end latency, 99.9% uptime, scalability for 1,000–10,000 users, and security (HTTPS, JWT, AES-256), within a 6–9-month timeline and $150K–$400K budget.

Task Implementation Guide for Backend Developer: VerseProjection MVP

1. Overview
Role: Backend Developer
Objective: Develop, deploy, and maintain the backend infrastructure for VerseProjection, including RESTful APIs, WebSocket communication, PostgreSQL database, and AWS cloud integration. Ensure seamless interaction between the AI pipeline (Whisper for transcription, BERT for verse matching), frontend (React for dashboard and projection), and database (KJV/WEB texts, user feedback). Support offline mode via Electron app synchronization and implement a feedback loop for AI model retraining.
Scope:
- **APIs**: User authentication, audio streaming, verse matching, feedback storage.
- **WebSocket**: Real-time audio streaming and match delivery.
- **Database**: Store KJV/WEB texts, user settings, and feedback data.
- **Cloud**: AWS (EC2, RDS, API Gateway, S3, CloudWatch, ElastiCache) for hosting, scaling, and monitoring.
- **Offline Sync**: Handle feedback synchronization for Electron app.

- **Deliverables**: Backend codebase (Node.js/Express), database schema (PostgreSQL), API documentation, cloud deployment scripts, test suites, monitoring dashboards.
Timeline: Months 3–8 (development, testing, deployment, per PRD).
Budget Allocation: ~$50K–$100K (part of salaries, included in $150K–$400K).
Tools: Node.js, Express, PostgreSQL (pgvector), AWS (EC2, RDS, API Gateway, S3, CloudWatch, ElastiCache), Socket.IO, JWT, GitHub, Jest.

2. Backend Requirements
- **Performance**:
  - API response time: <200ms (excluding AI pipeline).
  - WebSocket latency: <100ms for audio/match delivery.
  - End-to-end latency: <2 seconds (audio to projection, including AI).
- **Scalability**: Support 1,000–10,000 concurrent users (Sunday peak).
- **Reliability**: 99.9% uptime (AWS auto-scaling).
- **Security**: HTTPS, JWT authentication, AES-256 encryption for database.
- **Compatibility**: Integrate with React frontend, FastAPI AI pipeline, Electron app.
- **Cost**: ~$460–$900/month for AWS (EC2, RDS, API Gateway, S3, CloudWatch, ElastiCache).

3. Backend Architecture
- **Framework**: Node.js/Express (RESTful APIs, WebSocket).
- **Database**: PostgreSQL (KJV/WEB texts, user settings, feedback) with pgvector for embeddings.
- **WebSocket**: Socket.IO for real-time audio streaming and match delivery.
- **Cloud**:
  - AWS EC2: Host Node.js/Express (t3.medium, auto-scaling).
  - AWS RDS: PostgreSQL (db.t3.micro, 10GB).
  - AWS API Gateway: WebSocket management.
  - AWS S3: Logs, feedback backups.
  - AWS CloudWatch: Error logging, performance monitoring.
  - AWS ElastiCache: Redis for verse embedding caching (8MB).
- **Data Flow**:
  - Frontend → WebSocket (audio) → Node.js → FastAPI (AI pipeline) → PostgreSQL (verse query) → WebSocket (matches) → Frontend.

- User selection → API → PostgreSQL (feedback storage).
- Offline sync: Electron (SQLite) → API → PostgreSQL.

4. Task Breakdown
The Backend Developer's tasks are organized into six phases: Database Setup, API Development, WebSocket Implementation, Cloud Deployment, Offline Sync, and Testing/Optimization.

4.1 Database Setup
Objective: Design and deploy the PostgreSQL database to store KJV/WEB texts, user settings, and feedback data, supporting fast verse queries and AI retraining.
Tasks:
1. **Schema Design**:
  - **Bible Table**: Stores KJV/WEB verses.
    - Columns:
    - `id` (UUID, primary key).
    - `version` (VARCHAR(10), e.g., "KJV", "WEB").
    - `book` (VARCHAR(50), e.g., "John").
    - `chapter` (INTEGER, e.g., 3).
    - `verse` (INTEGER, e.g., 16).
    - `text` (TEXT, e.g., "For God so loved the world…").
    - `embedding` (VECTOR(768), BERT embedding for similarity).
    - Size: ~4MB (texts), ~8MB (embeddings), ~12MB total.
    - Indexes:
    - Full-text search on `text` (GIN index, for override search).
    - Vector index on `embedding` (pgvector, cosine distance).
    - Composite index on `book, chapter, verse` (for explicit queries).
  - **Users Table**: Stores user data.
    - Columns:
    - `id` (UUID, primary key).
    - `email` (VARCHAR(255), unique).
    - `password_hash` (VARCHAR(255), bcrypt).
    - `subscription_tier` (VARCHAR(20), e.g., "free", "premium").
    - `settings` (JSONB, e.g., {"bible_version": "KJV", "font_size": 24}).
    - Size: ~1MB for 1,000 users.
    - Indexes: Unique index on `email`.

- **Feedback Table**: Stores user selections for AI retraining.
    - Columns:
    - `id` (UUID, primary key).
    - `user_id` (UUID, foreign key to Users).
    - `timestamp` (TIMESTAMP).
    - `transcription` (TEXT, e.g., “God so loved the world”).
    - `selected_verse_id` (UUID, foreign key to Bible).
    - `top_matches` (JSONB, e.g., [{“verse_id”: “uuid”, “confidence”: 0.92}, …]).
    - Size: ~10MB/month for 1,000 users (100 selections/user/month).
    - Indexes: Index on `timestamp` (for retraining queries).
- **Logs Table** (optional, debugging):
    - Columns:
    - `id` (UUID, primary key).
    - `timestamp` (TIMESTAMP).
    - `user_id` (UUID, foreign key).
    - `event` (VARCHAR(50), e.g., “transcription_error”).
    - `details` (JSONB, e.g., {“transcription”: “text”, “confidence”: 0.6}).
    - Size: ~1MB/month for 1,000 users.
    - Indexes: Index on `timestamp`.

2. **Data Ingestion**:
- **KJV/WEB Texts**:
    - Source: Public-domain JSON from CrossWire Bible Society (e.g., `kjv.json`, `web.json`).
    - Script: Python (psycopg2) to parse JSON, insert into `Bible` table.
    - Size: ~31,000 verses (KJV: ~23,000, WEB: ~8,000).
- **Embeddings**:
    - Generate via BERT (AI Developer provides script, `generate_embeddings.py`).
    - Insert into `Bible.embedding` (pgvector, 768-dimensional vectors).
    - Size: ~8MB for 31,000 verses.
- **Validation**:
    - Query sample verses (e.g., `SELECT * FROM Bible WHERE book='John' AND chapter=3 AND verse=16`).
    - Verify embedding queries (e.g., `SELECT id, 1 - cosine_distance(embedding, $1) AS confidence ORDER BY confidence DESC LIMIT 10`).

3. **Security**:

   - Enable AES-256 encryption for data at rest (RDS configuration).
   - Restrict database access: Allow only Node.js (EC2) and AI pipeline (SageMaker).
   - Use environment variables for credentials (AWS Secrets Manager).
4. **Setup**:
   - Deploy PostgreSQL on AWS RDS (db.t3.micro, 1 vCPU, 1GB RAM, 10GB storage).
   - Install pgvector extension (`CREATE EXTENSION vector`).
   - Configure backups (daily, 7-day retention).
   - Script: SQL (`db/schema.sql`, `db/seed.sql`).

Deliverables:
- Database schema (PostgreSQL, `db/schema.sql`).
- Data ingestion scripts (Python, `db/ingest.py`).
- RDS configuration (Terraform, `infra/rds.tf`).
- Documentation (schema, indexes, GitHub, `docs/db.md`).

Timeline: Month 3 (4 weeks).
Effort: ~80–120 hours.

4.2 API Development
Objective: Build RESTful APIs to handle user authentication, settings, feedback, and manual verse overrides, integrating with the AI pipeline and database.
Tasks:
1. **Project Setup**:
   - Framework: Node.js (18.x), Express (4.x).
   - Structure:
      - `src/api`: Route handlers (`auth.js`, `settings.js`, `feedback.js`, `verses.js`).
      - `src/services`: Business logic (`authService.js`, `dbService.js`, `aiService.js`).
      - `src/middleware`: Authentication, error handling (`auth.js`, `error.js`).
      - `src/models`: Database queries (`bible.js`, `users.js`, `feedback.js`).
   - Dependencies: express, pg (PostgreSQL driver), bcrypt, jsonwebtoken, dotenv.
   - Linting: ESLint (Airbnb style guide).
2. **Authentication API**:

- **Endpoint**: `/auth/login` (POST).
    - Input: `{email, password}`.
    - Action: Verify credentials (bcrypt), generate JWT (jsonwebtoken, 24h expiry).
    - Output: `{token, user: {id, email, subscription_tier, settings}}`.
- **Endpoint**: `/auth/register` (POST).
    - Input: `{email, password, subscription_tier}`.
    - Action: Hash password (bcrypt), insert into `Users`, generate JWT.
    - Output: Same as login.
- **Endpoint**: `/auth/refresh` (POST).
    - Input: `{token}`.
    - Action: Validate JWT, issue new token.
    - Output: `{token}`.
- **SSO**:
    - Integrate AWS Cognito (OAuth2, Google/Microsoft providers).
    - Endpoint: `/auth/sso` (GET, redirects to Cognito).
    - Callback: `/auth/sso/callback` (GET, exchanges code for JWT).
- **Security**:
    - JWT middleware for protected routes (`Authorization: Bearer <token>`).
    - Rate limit (express-rate-limit, 100 requests/min).
    - Input validation (express-validator, e.g., email format, password strength).

3. **Settings API**:
- **Endpoint**: `/settings` (GET, protected).
    - Action: Fetch user settings from `Users.settings`.
    - Output: `{bible_version, font_size, text_color, bg_color, confidence_threshold}`.
- **Endpoint**: `/settings` (PUT, protected).
    - Input: `{bible_version, font_size, text_color, bg_color, confidence_threshold}`.
    - Action: Update `Users.settings` (JSONB).
    - Output: Updated settings.
- **Validation**:
    - Ensure `bible_version` in ["KJV", "WEB"].
    - Font size: 12–48 (integer).
    - Colors: Hex format (e.g., "#FFFFFF").
    - Confidence threshold: 0.7–0.9 (float).

4. **Feedback API**:
   - **Endpoint**: `/feedback` (POST, protected).
     - Input: `{transcription, selected_verse_id, top_matches}`.
     - Action: Insert into `Feedback` table (validate `selected_verse_id` exists).
     - Output: `{status: "success"}`.
   - **Validation**:
     - Transcription: Non-empty string (<1,000 chars).
     - Top_matches: Array of {`verse_id`, `confidence`} (max 10).
     - Filter low-confidence selections (confidence <0.5).
   - **Size**: ~10MB/month for 1,000 users (100 selections/user/month).
5. **Verse Override API**:
   - **Endpoint**: `/verses/search` (POST, protected).
     - Input: `{query}` (e.g., "John 3:16" or "God so loved").
     - Action:
     - If query matches regex `[A-Za-z]+ \d+:\d+`: Query `Bible` by `book, chapter, verse`.
     - Else: Full-text search on `Bible.text` (tsvector, ranked by relevance).
     - Output: [{`verse_id`, `reference`, `text`, `version`}, ...] (max 10).
   - **Validation**: Query non-empty (<100 chars).
6. **AI Pipeline Integration**:
   - **Endpoint**: `/infer` (POST, internal, called by WebSocket handler).
     - Input: Base64-encoded Opus audio (5-second chunk).
     - Action: Forward to FastAPI AI pipeline (HTTP POST, `http://ai-service/infer`).
     - Output: `{transcription, matches: [{verse_id, reference, text, version, confidence}, ...]}`.
   - **Security**: Restrict to internal traffic (VPC, SageMaker IP).
7. **Error Handling**:
   - Standardize responses: `{error: {code, message}}` (e.g., `{error: {code: 401, message: "Unauthorized"}}`).
   - Log errors to CloudWatch (e.g., `{"event": "api_error", "endpoint": "/infer", "details": {...}}`).
   - Handle AI pipeline failures (e.g., timeout after 1s, return `{matches: []}`).

Deliverables:
- Backend codebase (Node.js/Express, `src/api/`).
- API documentation (Swagger, `docs/api.yaml`).

- Database queries (PostgreSQL, `src/models/`).
- Test suite (Jest, `tests/api/`).

Timeline: Months 4–5 (8 weeks).
Effort: ~160–240 hours.

4.3 WebSocket Implementation
Objective: Enable real-time audio streaming from frontend to AI pipeline and match delivery to frontend, supporting continuous speech analysis.
Tasks:
1. **Setup**:
   - Library: Socket.IO (server: `socket.io`, client: `socket.io-client`).
   - Server: Integrate with Express (`src/websocket/index.js`).
   - Client: React frontend connects to `wss://app.verseprojection.com`.
2. **Audio Streaming**:
   - **Event**: `audio_chunk` (client → server).
      - Payload: Base64-encoded Opus audio (5-second chunk, <500 kbps).
      - Frequency: Every 2 seconds.
      - Action:
      - Validate payload (non-empty, <1MB).
      - Forward to FastAPI `/infer` (HTTP POST).
      - Cache session data (user_id, settings) in Redis (TTL: 1 hour).
   - **Error Handling**:
      - Invalid audio: Emit `error` event (`{code: "invalid_audio", message: "Invalid audio format"}`).
      - Log to CloudWatch (`{"event": "websocket_error", "details": {...}}`).
3. **Match Delivery**:
   - **Event**: `matches` (server → client).
      - Payload: `{transcription, matches: [{verse_id, reference, text, version, confidence}, ...]}`.
      - Action: Emit to client on AI pipeline response.
      - Frequency: Every 2 seconds (aligned with audio chunks).
   - **Filtering**: Apply user's `confidence_threshold` (e.g., 0.7) from `Users.settings`.
4. **Connection Management**:
   - **Authentication**: Validate JWT on connect (Socket.IO middleware).
   - **Reconnect**: Handle client disconnects (auto-reconnect with 5s retry).

   - **Scalability**: Use API Gateway for WebSocket routing (1,000–10,000 connections).
   - **Timeout**: Disconnect idle clients after 30 minutes.
5. **Monitoring**:
   - Log connection events to CloudWatch (e.g., `{"event": "websocket_connect", "user_id": "uuid"}`).
   - Track latency (audio to matches, target: <1s excluding AI).
   - Alert on high error rates (CloudWatch alarm, >1% errors).

Deliverables:
- WebSocket server (Socket.IO, `src/websocket/`).
- Client integration guide (React, `docs/websocket.md`).
- Monitoring dashboard (CloudWatch, `monitoring/websocket.json`).
- Test suite (Jest, `tests/websocket/`).

Timeline: Month 5–6 (6 weeks).
Effort: ~120–180 hours.

4.4 Cloud Deployment
Objective: Deploy backend to AWS, ensuring scalability, reliability, and cost efficiency for 1,000–10,000 users.
Tasks:
1. **EC2 Setup**:
   - Instance: t3.medium (2 vCPUs, 4GB RAM).
   - Auto-scaling: 1–10 instances (CPU utilization >70%).
   - Image: Amazon Linux 2, Node.js 18.x, PM2 for process management.
   - Script: Dockerfile (`docker/Dockerfile`), build in GitHub Actions.
   - Cost: ~$50–$200/month for 1,000 users.
2. **RDS Setup**:
   - Instance: db.t3.micro (1 vCPU, 1GB RAM, 10GB storage).
   - Configuration: PostgreSQL 15.x, pgvector, AES-256 encryption.
   - Backups: Daily, 7-day retention.
   - Access: Restrict to EC2, SageMaker (VPC security group).
   - Cost: ~$50–$100/month.
3. **API Gateway**:
   - Setup: WebSocket API for Socket.IO (`wss://app.verseprojection.com`).
   - Routes: `connect`, `audio_chunk`, `matches`, `disconnect`.

   - Security: JWT validation (Cognito integration).
   - Cost: ~$20–$50/month.
4. **S3**:
   - Buckets:
     - `verseprojection-logs`: Store API/WebSocket logs (~1MB/month).
     - `verseprojection-backups`: Feedback data backups (~10MB/month).
   - Lifecycle: Archive logs after 30 days (Glacier).
   - Cost: ~$10–$50/month.
5. **CloudWatch**:
   - Metrics: API latency, WebSocket errors, database queries, EC2 CPU.
   - Logs: API errors, WebSocket events, feedback insertions.
   - Alarms:
     - API errors >1% (5-minute window).
     - Latency >200ms (API), >1s (WebSocket).
     - CPU >80% (EC2).
   - Cost: ~$10–$20/month.
6. **ElastiCache**:
   - Instance: cache.t3.micro (1 vCPU, 0.5GB RAM).
   - Use: Cache verse embeddings (8MB, LRU, TTL: 1 week).
   - Access: Restrict to EC2, SageMaker.
   - Cost: ~$20–$30/month.
7. **Deployment Pipeline**:
   - Tool: GitHub Actions (`workflows/deploy.yml`).
   - Steps:
     - Build Docker image (EC2).
     - Deploy to EC2 (AWS CLI).
     - Run migrations (PostgreSQL, `db/migrate.sql`).
     - Test endpoints (`/auth/login`, `/settings`).
   - Rollback: Revert to previous image on failure.
8. **Security**:
   - HTTPS: Enable via AWS Application Load Balancer (ACM certificate).
   - Secrets: Store credentials in AWS Secrets Manager (RDS, Cognito, JWT).
   - Firewall: VPC security group (allow ports 80, 443 from internet, 5432 from EC2).
9. **Cost Optimization**:
   - Use Reserved Instances for EC2, RDS (save ~20%).
   - Monitor usage (CloudWatch Budgets, target: $460–$900/month).

Deliverables:
- Cloud infrastructure (Terraform, `infra/`).
- Deployment pipeline (GitHub Actions, `workflows/`).
- Monitoring dashboards (CloudWatch, `monitoring/`).
- Cost report (GitHub, `docs/cost.md`).

Timeline: Months 6–7 (6 weeks).
Effort: ~120–180 hours.

4.5 Offline Sync
Objective: Enable the Electron app to sync feedback data (SQLite) with the cloud (PostgreSQL) when reconnected.
Tasks:
1. **Sync API**:
   - **Endpoint**: `/sync/feedback` (POST, protected).
      - Input: `[{transcription, selected_verse_id, top_matches, timestamp}, ...]`.
      - Action: Insert into `Feedback` table, deduplicate by `timestamp`.
      - Output: `{status: "success", synced: n}`.
   - **Validation**: Same as `/feedback`, batch size <1,000.
2. **Electron Integration**:
   - **Schema**: SQLite `Feedback` table (mirrors PostgreSQL).
      - Columns: `id`, `timestamp`, `transcription`, `selected_verse_id`, `top_matches`.
   - **Logic**:
      - On reconnect: Query SQLite (`SELECT * FROM Feedback WHERE synced=0`).
      - Send to `/sync/feedback` (Node.js `axios`).
      - Update SQLite (`UPDATE Feedback SET synced=1 WHERE id=$1`).
   - **Error Handling**:
      - Retry on failure (3 attempts, 5s interval).
      - Log to SQLite (`Logs` table), sync to CloudWatch when online.
3. **Security**:
   - Authenticate sync requests (JWT).
   - Encrypt SQLite data (AES-256, Electron `crypto` module).
4. **Testing**:
   - Simulate offline mode: Store 100 selections, sync on reconnect.

  - Verify deduplication (no duplicate `Feedback` entries).
  - Metrics: 100% sync success, <1s for 100 selections.

Deliverables:
- Sync API (Node.js, `src/api/sync.js`).
- Electron sync script (Node.js, `offline/sync.js`).
- Test suite (Jest, `tests/sync/`).
- Documentation (GitHub, `docs/offline.md`).

Timeline: Month 7–8 (4 weeks).
Effort: ~80–120 hours.

4.6 Testing and Optimization
Objective: Validate backend performance, optimize for latency and scalability, and ensure reliability during pilot testing.
Tasks:
1. **Unit Testing**:
  - APIs: Test `/auth`, `/settings`, `/feedback`, `/verses/search` (Jest, 100% coverage).
  - WebSocket: Test `audio_chunk`, `matches` events (Jest, Socket.IO client).
  - Database: Test queries (explicit, paraphrase, feedback, pg-mock).
  - Framework: Jest (`tests/`).
2. **Integration Testing**:
  - Simulate flow:
    - Login → Update settings → Stream audio → Receive matches → Submit feedback.
    - Override search → Sync offline feedback.
  - Tools: Supertest (API), Socket.IO client (WebSocket).
  - Metrics:
    - API latency: <200ms.
    - WebSocket latency: <100ms.
    - Database queries: <100ms.
3. **Load Testing**:
  - Simulate 1,000 concurrent users (Sunday peak).
    - Tool: Artillery (`tests/load.yml`).
    - Scenario: 1,000 clients streaming audio (5s chunks, 2s interval).
    - Metrics:

        - 99.9% uptime.
        - <200ms API response.
        - <1s end-to-end (excluding AI).
  - Verify auto-scaling (EC2: 1–10 instances, CPU >70%).
4. **Optimization**:
  - **Latency**:
        - Profile APIs (Node.js `clinic.js`, identify bottlenecks).
        - Cache frequent queries (Redis, e.g., verse embeddings).
        - Use connection pooling (pg-pool, max 20 connections).
  - **Memory**: Cap Node.js to 1GB (t3.medium).
  - **Database**:
        - Optimize indexes (e.g., GIN for full-text search).
        - Partition `Feedback` table by `timestamp` if size >1GB.
  - **Cost**: Use Reserved Instances, monitor CloudWatch Budgets.
5. **Pilot Testing**:
  - Deploy to 5–10 churches (Month 8).
  - Collect metrics:
        - API/WebSocket errors (<1%).
        - Feedback insertions (~1,000/church).
        - Uptime (99.9%).
  - Iterate: Fix bugs (e.g., slow queries), adjust Redis TTL (1 week to 2 weeks).
6. **Documentation**:
  - API specs (Swagger, `docs/api.yaml`).
  - Database queries (`docs/db.md`).
  - WebSocket events (`docs/websocket.md`).
  - Cloud setup (`docs/infra.md`).
  - Store in GitHub.

Deliverables:
- Test suites (Jest, Artillery, `tests/`).
- Performance benchmarks (latency, scalability, `tests/benchmarks.md`).
- Pilot test report (metrics, fixes, `tests/pilot.md`).
- Documentation (GitHub, `docs/`).

Timeline: Months 7–8 (6 weeks).
Effort: ~120–180 hours.

5. Developer Collaboration
Work closely with AI/ML, frontend, and DevOps teams:
- **AI/ML Developer**:
  - Integrate FastAPI `/infer` endpoint (HTTP POST, JSON response).
  - Share PostgreSQL queries for verse matching (explicit, paraphrase).
  - Coordinate feedback storage (`Feedback` table).
- **Frontend Developer**:
  - Define WebSocket events (`audio_chunk`, `matches`).
  - Share API specs (`/auth`, `/settings`, `/feedback`, `/verses/search`).
  - Support real-time rendering (<200ms for matches).
- **DevOps Engineer**:
  - Deploy EC2, RDS, API Gateway, S3, CloudWatch, ElastiCache.
  - Configure auto-scaling, monitoring, cost optimization.
  - Set up GitHub Actions for CI/CD.
- **Meetings**:
  - Weekly sync (30min, align on APIs, performance).
  - Sprint reviews (Month 4–8, verify integration).
- **GitHub**:
  - Create issues for bugs (e.g., "API /feedback fails on large payloads").
  - Store code (`src/`, `tests/`, `docs/`).

Deliverables: API specs, integration tests, GitHub issues, meeting notes.

6. Risk Mitigation
- **Performance**:
  - Risk: API/WebSocket latency exceeds 200ms/100ms.
  - Mitigation: Optimize queries (indexes, caching), profile with clinic.js, test with Artillery.
- **Scalability**:
  - Risk: System fails under 1,000+ users.
  - Mitigation: Configure EC2 auto-scaling, test with Artillery, monitor CloudWatch.
- **Reliability**:
  - Risk: Downtime during Sunday peak.
  - Mitigation: Ensure 99.9% uptime (RDS backups, EC2 redundancy), test failover.
- **Security**:

  - Risk: Unauthorized access to APIs/database.
  - Mitigation: Enforce JWT, HTTPS, VPC restrictions, audit Secrets Manager.
- **Offline Sync**:
  - Risk: Data loss during sync.
  - Mitigation: Implement retries, deduplication, test with 100 selections.
- **Cost**:
  - Risk: AWS costs exceed $900/month.
  - Mitigation: Use Reserved Instances, monitor CloudWatch Budgets, cap S3 storage.

Deliverables: Risk assessment, mitigation plan.

7. Success Criteria
- **Performance**:
  - API: <200ms response time.
  - WebSocket: <100ms latency.
  - Database: <100ms query time.
  - End-to-end: <2 seconds (with AI).
- **Scalability**: Supports 1,000–10,000 users (Artillery tests).
- **Reliability**: 99.9% uptime (CloudWatch metrics).
- **Security**: Zero vulnerabilities (JWT, HTTPS, AES-256).
- **Pilot**: 90%+ of pilot churches (5–10) successfully use backend (feedback, matches).
- **Cost**: AWS costs within $460–$900/month (CloudWatch billing).

8. Resources
- **PRD Reference**: artifact_id: a7c7a8e1-9f68-4929-bc62-8d5d19d66186, artifact_version_id: ca07a5bd-a168-4b01-9a91-8cc217441994.
- **Tools**:
  - Node.js (18.x), Express (4.x), Socket.IO (4.x).
  - PostgreSQL (15.x, pgvector), pg (Node.js driver).
  - AWS (EC2, RDS, API Gateway, S3, CloudWatch, ElastiCache).
  - JWT (jsonwebtoken), bcrypt, express-validator.
  - Jest, Supertest, Artillery, GitHub Actions.
- **Data Sources**:
  - CrossWire Bible Society (KJV/WEB, CC0).
- **Documentation**:

  - Express: https://expressjs.com/
  - Socket.IO: https://socket.io/docs/v4/
  - AWS RDS: https://docs.aws.amazon.com/AmazonRDS/
  - pgvector: https://github.com/pgvector/pgvector
- **Team Contacts**:
  - AI/ML Developer: FastAPI integration, database queries.
  - Frontend Developer: WebSocket, API consumption.
  - DevOps Engineer: Cloud deployment, monitoring.
  - Product Manager: Requirements, pilot testing.

9. Implementation Timeline
- **Month 3**:
  - Design database schema (PostgreSQL, Bible, Users, Feedback).
  - Ingest KJV/WEB texts, embeddings (Python, psycopg2).
  - Deploy RDS (Terraform, db.t3.micro).
- **Month 4**:
  - Setup Node.js/Express project (APIs: `/auth`, `/settings`).
  - Implement authentication (JWT, Cognito SSO).
  - Write database queries (Bible, Users).
- **Month 5**:
  - Develop APIs (`/feedback`, `/verses/search`, `/infer`).
  - Implement WebSocket (Socket.IO, audio, matches).
  - Integrate with FastAPI AI pipeline.
- **Month 6**:
  - Deploy EC2, API Gateway, S3, CloudWatch, ElastiCache (Terraform).
  - Setup GitHub Actions (CI/CD).
  - Test APIs, WebSocket (Jest, Artillery).
- **Month 7**:
  - Implement offline sync API (`/sync/feedback`).
  - Support Electron integration (SQLite sync).
  - Optimize performance (Redis caching, query indexes).
- **Month 8**:
  - Conduct pilot testing (5–10 churches).
  - Fix bugs, optimize scalability (auto-scaling).
  - Finalize documentation (GitHub).
  - Deliver test reports and benchmarks.

Total Effort: ~680–1,000 hours (~4–5 months full-time, within $50K–$100K budget).

Conclusion
This guide equips the Backend Developer to build a robust server-side infrastructure for the VerseProjection MVP, supporting real-time Bible verse detection with <2-second latency, 99.9% uptime, and scalability for 1,000–10,000 users. The backend leverages Node.js/Express for APIs, Socket.IO for WebSocket, PostgreSQL for data storage, and AWS for cloud hosting, integrating seamlessly with the AI pipeline and frontend. Offline sync ensures accessibility, and the feedback loop supports AI model improvement (5% accuracy increase in 3 months). By testing rigorously and collaborating with other teams, the developer will deliver a secure, efficient solution within the 6–9-month timeline and $150K–$400K budget. Please review this guide and provide feedback or proceed with implementation. Let me know if you need additional details, code snippets, or further refinements!

Note: As requested, this guide is structured to be opened in a side tab for easy reference alongside other documents.