Low‑Level Technical Specification: Backend

### **1. Project Structure & Modules**

* **Entry Point**
  + src/index.ts / app.js
  + Bootstraps HTTP server, WebSocket server, database connections, middleware
* **API Layer**
  + src/routes/  
    - sessions.ts (session CRUD and vote endpoints)
    - stories.ts (story management)
    - ai.ts (AI suggestion endpoints)
    - auth.ts (token exchange, user info)
* **WebSocket Layer**
  + src/ws/  
    - socketServer.ts (initialization)
    - events.ts (join-session, vote-submitted, reveal-votes handlers)
* **Service Layer**
  + src/services/  
    - SessionService (create/read/update sessions)
    - VoteService (persist and retrieve votes)
    - AIService (call local Llama.cpp microservice)
    - IntegrationService (webhooks, external tool sync)
* **Data Access Layer**
  + src/db/  
    - models/ (ORM entities: User, Session, Story, Vote, AISuggestion)
    - repositories/ (CRUD operations using Knex or TypeORM)
* **Auth & Security**
  + src/middleware/  
    - authMiddleware.ts (JWT validation, role checks)
    - rateLimiter.ts (IP or user-based throttling)
* **Utilities**
  + src/utils/  
    - logger.ts (Winston or Bunyan setup)
    - errorHandler.ts (central Express error handler)
    - config.ts (env var validation via Joi/zod)

### **2. Dependencies & Versions**

* **Runtime & Frameworks**
  + node v18.x (OpenJS Foundation)
  + express v4.x (MIT)
  + socket.io v4.x (MIT)
* **Database & ORM**
  + postgresql v15+
  + knex v2.x (MIT) *or* TypeORM v0.3.x (MIT)
* **Caching & Pub/Sub**
  + redis v7.x (BSD‑3)
  + ioredis v5.x (MIT)
* **AI Service Client**
  + axios v1.x (MIT) *or* got v12.x (MIT) for HTTP calls to AI microservice
* **Auth & Security**
  + jsonwebtoken v9.x (MIT)
  + express-jwt v8.x (MIT)
  + joi v17.x (MIT) *or* zod v3.x (MIT) for config and payload validation
* **Logging & Monitoring**
  + winston v3.x (MIT)
  + prom-client v14.x (MIT) for metrics
* **Testing**
  + jest v29.x (MIT)
  + supertest v6.x (MIT)
  + socket.io-client for integration tests

### **3. REST API Endpoints**

* **Sessions**
  + GET /api/sessions → list sessions (query params: mode, active)
  + POST /api/sessions → create session (body: title, mode, deadline)
  + GET /api/sessions/:id → session details + stories + votes
  + POST /api/sessions/:id/terminate → close session & trigger summary
* **Stories & Votes**
  + POST /api/sessions/:id/stories → add user story (body: title, description)
  + POST /api/sessions/:id/vote → async vote (body: userId, storyId, value)
  + GET /api/sessions/:id/results → aggregated votes + AI suggestions
* **AI Suggestions**
  + GET /api/stories/:storyId/suggestion → fetch or generate AI estimate
* **Auth**
  + POST /api/auth/login → exchange credentials for JWT
  + GET /api/auth/me → retrieve user profile from JWT

### **4. WebSocket Server & Events**

* **Initialization**
  + Bind to same HTTP server at /socket.io/
  + Configure CORS, auth handshake via socket.use() validating JWT
* **Core Events**
  + join-session (payload: sessionId) → socket.join(sessionId)
  + vote-submitted (payload: storyId, userId, vote) → persist & broadcast vote-submitted to room
  + reveal-votes (payload: sessionId) → broadcast reveal-votes to room
  + disconnect → clean up any transient socket data

### **5. Data Persistence & Schema**

* **User** (users table)  
  + id, username, email, role, created\_at
* **Session** (sessions table)  
  + id, title, mode, deadline, created\_by, created\_at
* **Story** (stories table)  
  + id, session\_id, title, description, created\_at
* **Vote** (votes table)  
  + id, story\_id, user\_id, value, created\_at
* **AI Suggestion** (ai\_suggestions table)  
  + id, story\_id, session\_id, suggested\_value, confidence, generated\_at

### **6. AI Service Integration**

* **Endpoint**
  + Internal microservice at http://localhost:8000/estimate
* **Request Contract**
  + { storyId: string, description: string }
* **Response Contract**
  + { suggestedValue: number, confidence: number, modelVersion: string }
* **Timeout & Retries**
  + 500ms timeout, up to 2 retries on network errors
* **Fallback**
  + If AI fails, record null suggestion and proceed without blocking user

### **7. Caching & Messaging**

* **Redis Pub/Sub**
  + Channel: session:{sessionId} → publish new vote or reveal command
* **Cache**
  + Store live session state (votes array, user list) with 1h TTL
  + Invalidate cache on session update or termination
* **Locking**
  + Use Redis locks (SETNX with TTL) to prevent race conditions on session terminate

### **8. Authentication & Authorization**

* **JWT**
  + Access token expires in 1 hour; refresh token in 7 days
  + Signed with RSA256 or HS256 (env var JWT\_SECRET)
* **Middleware**
  + authMiddleware to verify token, attach req.user
  + roleMiddleware(['ScrumMaster']) for endpoints like session create/terminate
* **Keycloak (Optional)**
  + OIDC middleware to validate tokens issued by Keycloak

### **9. Configuration & Environment**

* **Env Vars** (validated on startup)  
  + PORT, DB\_HOST, DB\_PORT, DB\_USER, DB\_PASS, DB\_NAME
  + REDIS\_HOST, REDIS\_PORT, REDIS\_PASS
  + JWT\_SECRET, JWT\_ISSUER, JWT\_AUDIENCE
  + AI\_SERVICE\_URL, AI\_TIMEOUT\_MS
* **Config Module**
  + Central config.ts exporting typed values, throwing on missing/invalid

### **10. Logging & Monitoring**

* **Logger**
  + Winston with JSON format, separate transports for console & file
  + Log levels: error, warn, info, debug (configurable)
* **Metrics**
  + Expose /metrics for Prometheus
  + Track: HTTP request durations, WebSocket event rates, AI latency, DB query timings
* **Health Checks**
  + GET /healthz → checks DB and Redis connectivity

### **11. Error Handling**

* **Express Error Handler**
  + Catch all unhandled errors, log stack, return standardized JSON { code, message }
* **Validation**
  + Use Joi/Zod to validate request bodies and query params; return 400 Bad Request on failure
* **Global Promise Rejection**
  + Process-level handler to log and gracefully shutdown if necessary

### **12. Testing Strategy**

* **Unit Tests**
  + Mock repositories and services; test service logic and middleware
* **Integration Tests**
  + Spin up in-memory Postgres (e.g., pg-mem) and Redis (e.g., redis-mock)
  + Use Supertest for REST; socket.io-client for WebSocket flows
* **Contract Tests**
  + Validate AI service request/response contracts against OpenAPI spec
* **Coverage**
  + Aim for ≥ 80% coverage on critical modules

### **13. CI/CD & Deployment**

* **Pipeline**
  + Stages: lint → unit tests → build → integration tests → image build → deploy to staging
* **Docker**
  + Multi-stage Dockerfile (build + production)
* **Kubernetes**
  + Deployments for API, AI service, Postgres, Redis
  + ConfigMaps/Secrets for env vars
  + Horizontal Pod Autoscaler on CPU/memory