Family OS

Technical Blockers & Mitigation Report

■■ PRODUCTION RISK ANALYSIS ■■

Version 1.0 | February 13, 2026

Architecture: React Native (Expo) + Bun + Hono + tRPC + PostgreSQL RLS
AI Stack: Google Gemini (MCP, A2UI, A2A)
Target Scale: 100 → 5,000+ families

1. Executive Summary

1.1 Purpose & Criticality

This report identifies technical blockers, architectural risks, and mitigation strategies for Family OS, a production-grade AI-powered household coordination platform. Early blocker identification is critical because:

- **Cross-module automation creates cascading failure risks:** A single OCR error can create incorrect calendar events, tasks, and expenses across multiple families.
- **Multi-tenant architecture with RLS:** Misconfiguration can leak sensitive household data (financial records, documents, location).
- **AI-powered automation requires guardrails:** Gemini hallucinations in financial or document parsing can cause incorrect actions (wrong expense amounts, missed deadlines).
- **Mobile OS constraints limit real-time capabilities:** iOS background execution limits and Android lifecycle issues affect WebSocket reliability.
- **Scale amplifies risks:** What works for 10 families breaks at 1,000 (PostgreSQL connections, Redis memory, WebSocket concurrency).

1.2 Evaluation Methodology

This analysis employs a multi-layered risk assessment framework:

- **Architectural decomposition:** Each feature module analyzed for failure modes, data dependencies, and integration points.
- **Platform constraint analysis:** Expo limitations, iOS/Android OS restrictions, mobile network instability patterns.
- **AI system risk analysis:** Gemini API behavior patterns, MCP tool routing errors, A2UI/A2A edge cases.
- **Multi-tenancy security review:** PostgreSQL RLS policy validation, JWT token refresh vulnerabilities, file storage permissions.
- **Scale modeling:** Projected resource consumption at 100, 1,000, and 5,000 active families with concurrent usage patterns.
- **Failure scenario simulation:** Network partitions, background mode transitions, concurrent edit conflicts, API rate limits.

1.3 Production-Readiness Perspective

This report operates under the assumption that Family OS will handle real household data including financial records, legal documents, and family schedules. The evaluation prioritizes **data integrity, security, and reliability** over feature velocity. Risks are not minimized—they are presented with realistic failure scenarios and concrete mitigation paths.

2. Identified Technical Blockers

The following table identifies 18 critical technical blockers discovered during architecture review and library evaluation. Each blocker includes severity rating, impact assessment, root cause analysis, and mitigation strategy.

2.1 External Calendar & Document Processing Blockers

Area	Blocker Description	Severity	Mitigation Strategy	Phase
Calendar Sync	No React Native library for CalDAV/iCloud sy[illegible]	HIGH	Phase 1: Use [illegible]	Phase 2 [illegible]
OAuth Token Refresh	Access tokens expire (1 hour for Google). Ba[illegible]	HIGH	Implement [illegible]	Phase 2 [illegible]
Calendar Conflict Resolution	Two-way sync creates conflicts: event edited [illegible]	MEDIUM	[illegible]	Phase 2 [illegible]
PDF Encryption Performance	Decrypting 50MB PDF in memory causes cra[illegible]	HIGH	Implement [illegible]	[illegible]

3. Architectural Risk Zones (Deep Analysis)

This section provides in-depth analysis of five critical architectural risk zones. Each zone represents a system layer where multiple blockers converge, amplifying failure probability and impact.

3.1 Data Consistency & Cross-Module Automation

Risk Overview: Family OS's intelligent automation chains create cascading dependencies across modules. A document upload triggers OCR → AI extraction → calendar event creation → reminder task generation → expense logging. Each step introduces failure points.

Failure Scenario:
1. User uploads insurance policy PDF (20MB, 50 pages)
2. ML Kit OCR extracts text successfully (200k tokens)
3. Text sent to Gemini for structured extraction
4. Gemini API times out (30s limit, document too large)
5. Partial data returned: due date extracted, amount missing
6. Calendar event created with due date, NO amount
7. Reminder task created 30 days before (amount: $0)
8. Expense record NOT created (validation failed: amount required)
9. User sees calendar event, assumes automation complete
10. User misses insurance payment deadline

Root Cause: Lack of transactional semantics across module boundaries. Each module (Calendar, Tasks, Expenses) operates independently. No rollback mechanism for partial automation failures.

Impact: Data inconsistency, user confusion, missed deadlines, financial loss. User trust in automation degrades.

Mitigation Strategy:
• Implement 'automation_chains' table tracking multi-step automations
• Store state machine: pending → processing → complete → failed
• On failure: rollback all completed steps (delete created events/tasks)
• Show user: "Automation incomplete: X succeeded, Y failed. Retry?"
• Implement idempotency: retry safe, no duplicates
• Add manual override: "Create anyway (without amount)"
• Monitor automation success rate per document type

3.2 AI System Risks (MCP + A2UI + A2A)

Risk Overview: Family OS relies on Gemini AI for voice commands (MCP routing), UI generation (A2UI pattern), and backend actions (A2A pattern). AI non-determinism creates unpredictable failures.

Tool Misuse Scenario (MCP):
User: "Remind me to pay rent on the 1st"
Expected: MCP routes to create_task tool with title="Pay rent", due_date="1st of next month"
Actual: MCP misinterprets "pay" → routes to create_expense tool
Result: Expense record created: "Rent - $0" on today's date
No reminder task created
User misses rent payment

Agent Routing Errors:
Gemini confidence scores for tool selection:
• create_task: 0.62
• create_expense: 0.58
• create_calendar_event: 0.51
Threshold: 0.60
Result: Task created (correct), but low confidence indicates ambiguous intent
Should have asked clarification: "Do you want a task reminder or calendar event?"

UI Manipulation Edge Cases (A2UI):
User uploads receipt. Gemini extracts:
{
"merchant": "Costco",
"items": ["Milk", "Bread", "Eggs"],
"total": "$45.23"
}
A2UI generates expense form pre-filled with data
Edge case: Total doesn't match sum of items (items missing prices)
User clicks "Save" without verifying
Expense record created with incorrect data

Response Caching Inconsistencies:
Gemini response cached for "Add milk to list"
Cache hit rate: 80% (performance optimization)
Problem: User has two lists: "Groceries" and "Weekly Shopping"
Cached response always adds to "Groceries"
User intended "Weekly Shopping" but cache returns wrong list
Result: Item added to wrong list, user doesn't notice until shopping

Mitigation Strategies:
• Intent Classification Layer: Pre-classify intent BEFORE MCP routing. Use structured output (Zod schema validation).
• Confidence Thresholds: Require >0.75 confidence for auto-execution. Below threshold → ask clarification.
• A2UI Validation: ALWAYS show preview with "Verify and Save" button. Never auto-save AI-generated data.
• Cache Invalidation: Include context in cache key (list_id, user_id, timestamp). TTL: 5 minutes max.

• Hallucination Detection: Cross-check AI outputs against source data. Financial amounts must match OCR text exactly.
• Feedback Loop: "Was this correct?" after AI action. Train on user corrections (future enhancement).

3.3 Mobile Platform Constraints

Risk Overview: iOS and Android impose strict constraints on background execution, permissions, and network access. Family OS's real-time features (WebSocket sync, notifications) are severely limited.

iOS Background Execution Limits:
• WebSocket connections terminated after 30 seconds in background
• Background fetch: Limited to 15-minute intervals (unreliable)
• Push notifications required for critical updates (APNs complexity)
Impact: User misses task assignments, expense splits, calendar changes when app backgrounded

Android Lifecycle Issues:
• Doze mode: Network access restricted after screen off for 30+ minutes
• App standby buckets: Frequent apps → priority. Infrequent → severe restrictions
• Battery optimization: Users enable aggressive battery saver → app killed frequently
Impact: Offline queue not synced. Changes lost if device rebooted before sync.

Permission Volatility:
iOS 17: Users can grant "Allow Once" for camera, photos, location
Next app launch: Permission revoked. OCR scanning fails silently.
No clear UX for "permission was temporary, grant again"

Network Instability Patterns:
Mobile networks exhibit:
• Frequent DNS failures (10-15% of requests in subway, rural areas)
• Packet loss: 5-20% in weak signal
• High latency: 500ms - 5s during network transitions (WiFi ↔ cellular)
• Connection drops: Tunnel, elevator, airplane mode toggle
Result: WebSocket disconnections, failed API calls, corrupted sync state

Mitigation Strategies:
• Accept iOS background limitations. Design for foreground sync. Use push notifications for critical updates only.
• Implement robust permission handling: Check permission before EVERY use. Clear error messages: "Camera access required for receipt scanning".
• Network resilience: Exponential backoff (1s, 2s, 4s, 8s, 30s max). Request deduplication (idempotency keys). Queue mutations in op-sqlite WAL.
• Offline-first architecture: All features work offline. Sync when connection restored. Show sync status clearly.
• Test on real devices in real conditions: Subway commute, airplane mode toggle, low battery, weak signal.

3.4 Security & Compliance Risks

Risk Overview: Family OS handles highly sensitive data: financial records, legal documents, children's schedules, location data. Multi-tenant architecture with Row-Level Security creates attack surface.

Multi-Tenant RLS Leakage Risk:
Scenario: User crafts malicious tRPC query exploiting SQL injection in family_id filter
Normal query: SELECT * FROM expenses WHERE family_id = $1
Malicious: family_id = '1 OR 1=1' → bypasses RLS, returns ALL families' expenses
Impact: CRITICAL data breach. Competitor families, financial data exposed.

RLS Policy Examples (Vulnerable): ```sql -- VULNERABLE CREATE POLICY family_isolation ON expenses FOR ALL TO authenticated USING (family_id::text = current_setting('app.current_family')::text); -- Problem: String concatenation vulnerable to injection -- SECURE CREATE POLICY family_isolation ON expenses FOR ALL TO authenticated USING (family_id = (current_setting('app.current_family_id', true))::uuid); -- Uses UUID casting, parameterized, no injection possible ```
JWT Token Handling Vulnerabilities:
• Access token stored in AsyncStorage (unencrypted). Device forensics extracts token.
• Refresh token rotation not implemented. Single compromised refresh token valid for 90 days.
• No token revocation. User changes password but old tokens still valid.
• CSRF vulnerability: tRPC mutations don't validate origin header.

File Storage Permission Exposure:
Google Cloud Storage bucket configuration:
• Default ACL: public-read (WRONG) → All uploaded documents world-accessible
• Signed URL expiry: 7 days (TOO LONG) → Link leaked on social media remains valid
• No audit logs → Unauthorized access undetected

Sensitive Document Parsing Risks:
Insurance PDFs contain:
• Policy numbers, SSNs (masked but present)
• Bank account numbers for auto-pay
• Beneficiary information
Gemini AI processes full document. Logs stored by Google.
Compliance risk: GDPR, CCPA violations if personal data logged.

Mitigation Strategies:
• Mandatory RLS testing: Automated tests attempt cross-family access. CI/CD fails if test succeeds.
• JWT best practices: Store in expo-secure-store. Short-lived access tokens (15 min). Implement token revocation list (Redis).
• GCS security: Private buckets only. Signed URLs with 15-minute expiry. Generate on-demand per request. Uniform ACL (no public access).
• PII redaction: Strip SSNs, account numbers BEFORE sending to Gemini. Use regex patterns. Log redaction events.
• Penetration testing: Hire security firm for RLS bypass testing, token manipulation, file access vulnerabilities.
• Compliance audit: GDPR data processing agreement with Google. User data export capability. Right to deletion implementation.

3.5 Scaling Risks

Risk Overview: Architecture designed for 100 families may collapse at 1,000. Linear scaling assumptions fail. Resource contention creates cascading failures.

Resource Consumption Projections:

100 Families (400 users):
• PostgreSQL connections: ~50 concurrent
• WebSocket connections: ~200 (50% active)
• Redis memory: ~50MB (sessions + rate limits)
• Gemini API: ~500 requests/day (~$15/month)
• Google Cloud STT: ~1,000 requests/day (~$12/month)
Status: No issues. Single-instance handles load.

1,000 Families (4,000 users):
• PostgreSQL connections: ~500 concurrent → connection pool exhausted
• WebSocket connections: ~2,000 → Bun single-process limit (~5k)
• Redis memory: ~500MB → approaching 1GB instance limit
• Gemini API: ~5,000 requests/day → approaching 60 RPM limit
• Google Cloud STT: ~10,000 requests/day (~$120/month)
Status: Degraded performance. Requires infrastructure upgrades.

5,000 Families (20,000 users):
• PostgreSQL connections: ~2,500 concurrent → SEVERE contention
• WebSocket connections: ~10,000 → multi-instance required
• Redis memory: ~2.5GB → cluster required
• Gemini API: ~25,000 requests/day → rate limits hit frequently
• Google Cloud STT: ~50,000 requests/day (~$600/month)
Status: System failure without horizontal scaling.

Specific Bottlenecks:
1. **PostgreSQL write contention:** Shared lists with 10+ concurrent editors → row-level locks. Deadlocks increase.
2. **Redis pub/sub fan-out:** 10,000 WebSocket subscribers. Single message → 10,000 Redis PUBLISH commands. CPU spike.
3. **Gemini rate limits:** Peak hours (8 AM breakfast time) → 500 simultaneous voice commands → 429 errors cascade.
4. **GCS bandwidth:** 5,000 families uploading receipts (5MB avg) daily → 25GB/day → $5/day egress costs.

Mitigation Roadmap:
• **Phase 1 (MVP, 0-100 families):** Single instance. No scaling required.
• **Phase 2 (100-1,000 families):** Implement PgBouncer. Redis Sentinel (HA). Bun cluster mode (multi-process).
• **Phase 3 (1,000-5,000 families):** PostgreSQL read replicas. Redis Cluster. Horizontal scaling with load balancer. Gemini API tier upgrade.
• **Infrastructure monitoring:** Set up alerts at 70% capacity. Auto-scaling for Bun instances. Database query optimization (indexes, query plans).

4. Risk Prioritization Matrix

This matrix prioritizes identified risks based on probability and business impact. Priority score = Probability × Impact. Risks scored ≥12 require immediate mitigation before MVP launch.

Risk	Probability (1-5)	Impact (1-5)	Priority Score	Mitigation Timeline
RLS Policy Bypass (data breach)	2	5	10	Before MVP (security testing)
JWT Token Leakage	3	5	15	Before MVP (secure storage)
File Storage Public Exposure	3	5	15	Before MVP (GCS configuration)
PDF Encryption Memory Crash	4	4	16	Before MVP (chunked decryption)
AI Hallucination (Financial)	4	4	16	Before MVP (validation layer)
Cross-Module Cascade Failures	3	4	12	Phase 2 (transaction rollback)
iOS Background WebSocket Kill	5	3	15	MVP (accept limitation + push notifs)
Concurrent Edit Conflicts	3	3	9	Phase 2 (OT/CRDTs)
PostgreSQL Connection Exhaustion	2	5	10	Before 1,000 families (PgBouncer)
Gemini API Rate Limits	4	4	16	Before 1,000 families (queue + tier upgrad
Google Cloud STT Cost Explosion	3	3	9	MVP (usage caps + monitoring)
OAuth Token Refresh Failures	3	3	9	Phase 2 (custom OAuth wrapper)
OCR Accuracy Drops	4	2	8	MVP (confidence thresholds + preview)
Network Partition Split-Brain	2	2	4	MVP (UUID primary keys)

Priority Scoring:

- **12-25 (HIGH):** Critical risk. Must resolve before proceeding to next phase.
- **6-11 (MEDIUM):** Significant risk. Mitigation plan required. Can defer to Phase 2.
- **1-5 (LOW):** Manageable risk. Monitor and address opportunistically.

5. Mitigation Roadmap

This roadmap categorizes mitigation efforts by implementation phase and assigns clear ownership and timelines.

5.1 Critical Path: Must Solve Before MVP Launch

• **Security Hardening (Week 1-2):**
• Implement RLS policies with UUID casting (no string interpolation)
• Automated RLS bypass tests in CI/CD
• Store JWTs in expo-secure-store (iOS Keychain / Android KeyStore)
• Configure GCS buckets: private, signed URLs (15min expiry)
• Penetration testing: RLS, JWT manipulation, file access

• **Encryption Implementation (Week 2):**
• Integrate react-native-quick-crypto for AES-256-GCM
• Chunked decryption for files >10MB (stream-based)
• Test on iPhone 8 and Android API 23 devices
• Memory profiling: ensure no leaks after decryption
• Implement secure key storage with expo-secure-store

• **AI Validation Layer (Week 2-3):**
• Financial amount validation: must match OCR text exactly (regex)
• Intent classification before MCP routing (confidence >0.75)
• A2UI preview: NEVER auto-save AI-generated data
• Implement 'Verify and Save' workflow
• Hallucination detection: cross-check AI outputs against source

• **Mobile Platform Resilience (Week 3):**
• Offline queue with Write-Ahead Log (op-sqlite)
• Idempotency with client-side UUIDs
• Exponential backoff for network retries (1s → 30s max)
• Permission handling: check before every use, clear error messages
• Accept iOS background limitations: design for foreground sync

• **Cost Controls (Week 3):**
• Implement Google Cloud STT usage caps (50 commands/user/day)
• Cost monitoring dashboard (per-user tracking)
• Alerts on usage anomalies (>100 commands/user/day)
• Cache common voice patterns (deduplication)

5.2 Phase 2 Enhancements (Post-MVP, Month 2-4)

• **External Calendar Sync:**
• Custom OAuth implementation (Google Calendar API + MS Graph API)
• expo-auth-session for OAuth flows
• Token refresh with backend proxy
• Conflict resolution: version vectors, manual user resolution

• **Cross-Module Transaction Rollback:**
• Implement 'automation_chains' state machine table
• Rollback on partial failures (delete created events/tasks/expenses)
• Idempotency for safe retries
• User notification: 'Automation incomplete: Retry?'

• **Real-time Sync Improvements:**
• Operational Transformation (OT) for list items
• Vector clocks for conflict detection
• 'User X is editing' presence indicators
• Client-side conflict merging

• **Encryption Key Recovery:**
• Key escrow: encrypt master key with recovery key
• Derive recovery key from user password (PBKDF2, 100k+ iterations)
• Store encrypted master key on server
• Require strong password (12+ chars) + 2FA

5.3 Infrastructure Scaling (Before 1,000 Families)

• **Database Scaling:**
• Implement PgBouncer (transaction mode, pool_size=20)
• PostgreSQL read replicas for analytics queries
• Connection monitoring and alerts (70% threshold)
• Query optimization: indexes, EXPLAIN ANALYZE reviews

• **WebSocket Scaling:**
• Bun cluster mode (multi-process)
• Redis pub/sub for cross-instance messaging
• Sticky sessions (Redis-backed session store)
• Reconnection rate limiting (exponential backoff with jitter)

• **Redis High Availability:**
• Redis Sentinel for failover
• Separate instances: sessions, cache, rate limits, pub/sub
• maxmemory policy: allkeys-lru
• TTLs on all keys (no unbounded growth)

• **AI API Management:**
• Gemini API tier upgrade (60 RPM → 360 RPM)
• Request queueing with prioritization
• Response caching (deduplicate similar queries, 5min TTL)
• Exponential backoff on rate limits

• **Monitoring & Observability:**
• Set up DataDog / Sentry for error tracking
• PostgreSQL query performance monitoring
• WebSocket connection metrics (active, reconnects/sec)
• AI API usage dashboard (costs, rate limits, errors)
• Automated alerts: CPU >80%, memory >80%, error rate >1%

6. Technical Go / No-Go Assessment

6.1 Feasibility with Current Stack

Verdict: GO (with conditions)

Family OS is technically feasible with the current architecture (React Native Expo + Bun + Hono + tRPC + PostgreSQL RLS + Gemini AI). However, success depends on implementing critical mitigations before MVP launch.

Stack Strengths:
• Expo Development Builds support custom native modules (ML Kit, react-native-quick-crypto, react-native-pdf)
• tRPC provides end-to-end type safety, reducing runtime errors
• PostgreSQL RLS enforces multi-tenancy at database level (if configured correctly)
• Bun offers excellent WebSocket performance and fast startup times
• Gemini AI (MCP, A2UI, A2A) enables powerful automation

Stack Weaknesses:
• Expo background execution severely limited (iOS 30s WebSocket limit)
• No comprehensive library for CalDAV/iCloud calendar sync (custom implementation required)
• RLS misconfiguration risk HIGH (requires disciplined testing)
• AI non-determinism creates unpredictable failures (requires validation layers)

6.2 Red Flags Requiring Redesign

No Critical Red Flags Identified

All identified blockers have viable mitigation strategies within the current architecture. No fundamental redesign required. However, the following areas warrant close monitoring:

• ■■ Multi-Tenant RLS Security: Requires exceptional discipline. One misconfigured policy = data breach. Recommend mandatory security review for every RLS policy change.

• ■■ AI-Powered Automation Complexity: Cross-module automation chains are fragile. Consider starting with simpler, single-module automations in MVP. Add cross-module chains in Phase 2 after validating single-module reliability.

• ■■ iOS Background Limitations: Cannot be solved with current stack. Must design UX around this constraint. Consider this a permanent limitation, not a temporary blocker.

6.3 Expo Long-Term Suitability

Verdict: Suitable for MVP, Re-evaluate at 5,000+ families

Expo Development Builds provide necessary flexibility for Family OS MVP and early growth (0-1,000 families). However, certain limitations may require brownfield migration to bare React Native at scale:

Expo Advantages:
• Fast development velocity (OTA updates, managed build pipeline)
• Good developer experience (clear documentation, active community)
• Config plugins support custom native modules
• Suitable for 95% of Family OS features

Expo Limitations:
• Background execution more restricted than bare RN (though iOS limits apply to both)
• Some advanced native modules incompatible (rare, but possible)
• OTA updates have size limits (not an issue for Family OS)
• Potential performance overhead (negligible for Family OS use cases)

Recommendation: Proceed with Expo for MVP. Monitor performance and feature requirements. If complex native features emerge (e.g., advanced HealthKit integration, custom camera processing), evaluate bare React Native migration at that time. Migration path exists but is non-trivial (2-4 weeks engineering effort).

6.4 Scalability to 5,000+ Families

Verdict: Scalable with Infrastructure Investment

Current architecture can scale to 5,000+ families (20,000+ users) with planned infrastructure upgrades. Critical scaling path:

0-100 Families: Single-instance deployment. No changes required.

100-1,000 Families:
• Implement PgBouncer for connection pooling
• Redis Sentinel for high availability
• Bun cluster mode (multi-process WebSocket handling)
• Cost: ~$200/month infrastructure

1,000-5,000 Families:
• PostgreSQL read replicas (analytics queries)
• Redis Cluster for horizontal scaling
• Load balancer with sticky sessions
• Gemini API tier upgrade (360 RPM)
• Cost: ~$800/month infrastructure + ~$600/month AI APIs = ~$1,400/month

5,000+ Families:
• Multi-region deployment for latency
• CDN for static assets (GCS → Cloudflare)
• Database sharding (by family_id ranges)
• Cost: ~$2,500/month infrastructure + ~$1,500/month AI APIs = ~$4,000/month

Economic Viability: At 5,000 families, $4,000/month = $0.80 per family. With subscription pricing of $10-15/month per family, infrastructure costs are 5-8% of revenue. Economically sustainable.

6.5 Final Architectural Confidence Rating

Confidence Rating: HIGH (8/10)

Family OS architecture is sound and production-ready with identified mitigations implemented. The stack is modern, type-safe, and performant. All critical blockers have clear mitigation paths.

Confidence deductions:
• -1: Multi-tenant RLS requires exceptional discipline (security risk)
• -1: AI-powered automation complexity (cascading failures possible)

Architectural Approval: GRANTED (Conditional)

Conditions for Production Launch:
1. Complete all "Critical Path: Must Solve Before MVP Launch" items (Section 5.1)
2. Pass penetration testing for RLS bypass, JWT manipulation, file access
3. Load testing: 500 concurrent users, validate no connection pool exhaustion
4. Encryption performance testing on iPhone 8 and Android API 23
5. AI validation layer tested with 100+ real receipts/invitations
6. Monitoring dashboard operational with automated alerts

Once conditions met, architecture is approved for production deployment with up to 1,000 families. Re-assessment required before scaling to 5,000+ families.

— End of Technical Risk Assessment —

Generated on February 13, 2026 at 05:26 PM

Approved for architectural review and production planning