Oxford Tech Consulting

Senior software, AI & product · Oxford, UK

SAMPLE REPORT — PREPARED FROM PUBLIC SOURCE CODE

Technical Due Diligence Report

Subject: Umami — open-source web analytics platform

Repository: github.com/umami-software/umami (v3.2.0, MIT)

PREPARED BY	George Brooks, Oxford Tech Consulting
DATE	2 August 2026
SCOPE	Public repository review — codebase, architecture, security, delivery, team signals
CLASSIFICATION	Public specimen. Client reports are confidential and NDA-covered.

EXECUTIVE SUMMARY

A codebase I'd be comfortable inheriting — with three caveats

I spent a weekend inside this codebase the way I'd want someone to examine mine: reading the code, not the pitch deck. Umami is an open-source, privacy-focused web analytics platform — a Google Analytics alternative — built by Umami Software, Inc. What I reviewed is the public repository at version 3.2.0: roughly 58,000 lines of TypeScript across a Next.js application, a standalone tracking script, and dual database support, with an optional ClickHouse path for high-volume ingestion.

The short version: the engineering is good. The stack is modern without being fashionable, the code is organised the way disciplined teams organise it, the raw SQL that analytics products live on is parameterised properly, and the project ships continuously with tests running on every push. Plenty of venture-backed codebases I've seen don't clear that bar.

If I were advising on this deal, the code would not be my main worry — the concentration of knowledge would be. Roughly seventy percent of every change ever made came from one person, and the written documentation is too thin to catch anyone who has to pick it up cold. Alongside that sit a handful of security-hygiene defaults that are forgivable in a self-hosted open-source tool but need tightening before I'd trust them in a commercial cloud product. All of it is fixable; none of it is priced in until you ask.

AREA	RATING	HEADLINE
Architecture	LOW RISK	Clean monolith with a credible scaling path (ClickHouse, read replicas).
Code quality & testing	LOW RISK	TypeScript throughout, unit + end-to-end suites, automated lint/format.
Security	MODERATE	Solid fundamentals; weak secret default and legacy-token acceptance need fixing.
Scalability & infrastructure	LOW RISK	Purpose-built for volume; multi-stage Docker, pinned toolchain.
Delivery & CI/CD	MODERATE	CI runs tests and builds, but e2e tests and dependency scanning are not gated.
Team & key-person risk	ELEVATED	~70% of all commits from one founder; top three contributors dominate.
Licensing & IP	LOW RISK	MIT-licensed core; open-core boundary should be verified in full diligence.
Documentation	MODERATE	Minimal written docs and no decision records; knowledge lives in heads.

Overall: MODERATE RISK — investable, with conditions. Nothing here kills a deal. The security and delivery items are one to two sprints of remediation and belong in a 90-day plan. The key-person concentration is the one thing I would insist goes into the deal terms — retention, knowledge transfer and a senior hiring commitment — because no amount of engineering fixes it after the fact. Before reading further, please read the assumptions and limits of this review on the next page; a diligence report is only as honest as its stated boundaries.

SCOPE, ASSUMPTIONS & LIMITATIONS

What this review is — and what it is not

Every diligence report rests on assumptions, and I would rather state mine plainly than let you discover them later. This section is the contract between us: what I looked at, what I took on trust, and where the edges of my confidence are.

Assumptions

Point in time. Every finding refers to the repository at commit af1b6c6 (version 3.2.0), cloned on 2 August 2026. Code changes daily; conclusions have a shelf life. Public code only. This specimen was prepared exclusively from the public GitHub repository, its commit and contributor history, CI configuration, and published project metadata. I assume the public repository is representative of the team's engineering practice — a reasonable assumption here, since it is the product — but the commercial cloud offering may contain code I have not seen. Authentic history. I treat the git history as genuine and un-rewritten; contributor statistics are only as honest as the history behind them. Declared dependencies. Dependency analysis is based on the manifests as committed; no runtime verification or penetration testing was performed.

Not assessable from a repository — covered in a full engagement

Backups and disaster recovery (whether restores are actually tested); production security posture (secrets management, access control, monitoring, incident history); infrastructure cost and cost-per-customer; regulatory and compliance evidence (GDPR processing records, SOC 2 status — material for an analytics business); contractual and SLA exposure; and the team itself, beyond what the commit history reveals. A full engagement covers all of these through private access and interviews, and I will not speculate on them here.

How I worked

I read the code the way an incoming technical leader would: entry points first, then the authentication path, the database layer, the public ingestion endpoint, the build and deploy pipeline, and the test suites — verifying every claim against a specific file and line rather than sampling by folder. The areas assessed: architecture (structure, boundaries, fitness for the product's claims), code quality (readability, typing, test depth), security (authentication, injection surfaces, secret handling), scalability (what breaks first under 10× load), delivery (how safely and how often the team can ship), team (authorship concentration as visible in history), and licensing (open-source obligations and IP hygiene).

FINDINGS — 1 OF 4

Architecture & scalability

The application is a Next.js 16 (App Router) monolith with a clearly separated source tree: UI components, API route handlers, a query layer, a permissions module with its own tests, and an internationalisation layer. The analytics tracker and session recorder are built as independent lightweight bundles via Rollup — the right call, since the tracker runs on customers' websites and must stay small and dependency-free.

Persistence is dual-path: PostgreSQL through Prisma 7 for standard deployments, with an optional ClickHouse backend for high-volume event ingestion and Redis for session state. Read-replica support is wired into the Prisma layer. This is exactly the architecture you would want to see in an analytics product whose economics depend on ingesting large event volumes cheaply — the scaling path is designed in, not bolted on.

Deployment is a multi-stage Dockerfile with pinned Node, pnpm and Prisma versions, plus maintained Docker Compose, Podman and Netlify configurations. The build pipeline (database client generation, tracker build, geo-data build, app build) is scripted and reproducible.

Assessment: LOW RISK. The architecture supports both the self-hosted open-source product and a commercial cloud offering without a rewrite.

FINDINGS — 2 OF 4

Code quality, testing & delivery

The codebase is TypeScript end to end (~58,000 lines), formatted and linted with Biome, and organised consistently enough that a new senior engineer could navigate it within days. Naming is disciplined and the query layer keeps database access out of UI code.

Testing

Unit tests (Vitest) cover the core library code, the permissions module, and a sample of API routes — including the areas that most deserve tests, such as authentication and data formatting. A Playwright end-to-end suite exercises login, user, website and team API flows. This is genuinely better test discipline than most venture-stage codebases I review.

The gap: the CI pipeline runs unit tests and a production build on every push, but the end-to-end suite is not part of CI, and there is no dependency-vulnerability scanning or static security analysis gating merges. The safety net exists; it just isn't fully attached to the release process.

Documentation

Written documentation inside the repository is thin — a short README and contributing guide, no architecture decision records. For an open-source project with external docs this is survivable; for an acquirer it means the operational knowledge lives with the people, which compounds the key-person finding below.

Assessment: LOW RISK on quality, MODERATE on delivery. Wiring the e2e suite and a dependency audit into CI is roughly a week of work and should be a post-investment condition.

FINDINGS — 3 OF 4

Security

The fundamentals are done properly. Passwords are hashed with bcrypt. Authentication tokens carry a fingerprint of the user's current password so that changing a password invalidates outstanding sessions — a detail many teams miss. The extensive raw SQL needed for analytics queries is routed through a templating layer that converts named placeholders into bound positional parameters, keeping the SQL-injection surface controlled despite heavy use of dynamic queries.

Items requiring remediation

HIGH	Weak signing-secret default. If the APP_SECRET environment variable is unset, the application derives its token-signing secret from the database connection string. Connection strings routinely leak into logs, backups and CI output; anyone holding one could forge session tokens. Fix: make APP_SECRET mandatory at startup.
MEDIUM	Legacy token acceptance. Older stateless tokens minted without a password fingerprint are still honoured, so the password-change invalidation above does not apply to them. Fix: expire the legacy format on a published schedule.
MEDIUM	Session tokens never expire. Stateless login tokens are signed without an expiry, so a stolen token is valid indefinitely (unless the password changes). Fix: sign with a bounded lifetime and add refresh. Evidence in Appendix A.4.
MEDIUM	No login throttling. The login endpoint has no rate limiting or lockout; the only brute-force cost is bcrypt itself. Fix: per-IP and per-account throttling. Evidence in Appendix A.5.
MEDIUM	No dependency or static-analysis gate. With 66 runtime dependencies, the supply-chain surface is moderate and unmonitored in CI. Fix: add automated audit and SAST steps to the pipeline.
LOW	Interpolated schema statement. One database statement interpolates a schema name from configuration rather than binding it. The value is operator-controlled, so exploitability is low, but the pattern is worth eliminating.
LOW	MD5 utility present. An MD5 helper exists in the crypto module. Usage appears limited to non-security identifiers, but a full engagement would verify every call site.

Assessment: MODERATE RISK. No architectural security flaws; the findings are hygiene items, all remediable within one to two sprints. File-and-line evidence for every item above is given in Appendix A.

FINDINGS — 4 OF 4

Team, key-person risk & licensing

Contribution history is the clearest risk signal in this review. The founder accounts for roughly 70% of all commits in the project's history; the top three contributors — two of whom share a surname — account for over 95% of the top-five contribution volume. Community contributions exist but are a thin tail.

For an open-source project this is common. For an investment it is the item that most deserves attention in deal structure: if the founder became unavailable, the combination of concentrated authorship and thin written documentation means velocity would drop sharply and unpredictably. Mitigations belong in the term sheet — key-person retention, a documented knowledge-transfer plan, and a senior hiring commitment — and in engineering practice (architecture decision records, runbooks, broader code review).

Project health

The project is demonstrably alive and loved: ~38,000 GitHub stars, a modest open-issue count (~106) for its popularity, and commits landing the day of this review. Release cadence is steady and version 3.x shows sustained investment rather than maintenance mode.

Licensing & IP

The core repository is MIT-licensed, which is clean and permissive. The commercial questions sit outside this repository: where the open-core boundary lies (what is proprietary to the cloud product), whether contributor IP assignment is in place, and whether any dependency licences conflict with commercial plans. A full engagement audits all three and delivers a software bill of materials.

Assessment: ELEVATED RISK (team), LOW RISK (licence). Addressable through deal terms rather than engineering.

RECOMMENDATIONS

What I would ask of this company

Sequenced by where each item belongs: in the price, in the terms, or in the first ninety days. Items 1–5 total roughly two to three engineer-months of remediation — worth reflecting in valuation discussions, but not a reason to walk.

#	ACTION	WHEN
1	Make APP_SECRET mandatory; rotate and re-issue tokens.	Pre-close
2	Key-person retention and knowledge-transfer plan for the founder; senior hire commitment.	Deal terms
3	Expire legacy token format on a published schedule.	30 days
4	Add expiry + refresh to session tokens; throttle the login endpoint.	30 days
5	Gate CI on the Playwright suite; add dependency audit and static analysis.	60 days
6	Introduce architecture decision records and operational runbooks.	90 days
7	Full licence/SBOM audit across dependencies and the open-core boundary.	Full DD

FOR YOUR NEXT MEETING

Questions I'd put to management

Every finding in this report should become a question in the room, not just a line in a risk register. These are the six I would ask, and what a good answer sounds like.

1	“Who, other than the founder, has shipped a significant change to production unaided in the last quarter?” A good answer names people and features. A bad answer talks about plans to hire.
2	“What parts of the cloud product are not in the public repository, and who owns their IP?” The open-core boundary is where the commercial value sits; it should be crisp, documented and assigned.
3	“When did you last test restoring a backup, and how long did it take?” Not ‘do you have backups’ — everyone says yes. Restores that have never been rehearsed are a hope, not a plan.
4	“What does infrastructure cost per active website today, and how does that curve bend at ten times the event volume?” The architecture supports scale (§ Architecture); the question is whether the unit economics do.
5	“Where is your GDPR processing documentation, and has any customer's counsel reviewed it?” A privacy-positioned analytics product will be held to its own marketing. The code supports the claims (Appendix B); the paperwork must too.
6	“If APP_SECRET had to be rotated tomorrow, what happens to logged-in customers and how long does it take?” This tests whether the team has operational answers to the security findings in Appendix A, or just fixes-in-theory.

APPENDIX A

Code-level evidence for security findings

Every finding in this report traces to specific code. References are to repository paths at commit af1b6c6 (v3.2.0). Excerpts are reproduced under the project's MIT licence and abbreviated for clarity.

A.1 — Signing secret falls back to the database connection string (HIGH)

EVIDENCE — src/lib/crypto.ts, lines 56–58

```
export function secret() {
  return hash(process.env.APP_SECRET || process.env.DATABASE_URL);
}
```

This value signs and encrypts every session token (src/lib/jwt.ts) and seeds deterministic UUID generation (crypto.ts line 62). When APP_SECRET is unset, the secret is SHA-512 of the connection string — a value that appears in deploy scripts, CI logs, error traces and database backups. Anyone who obtains it can mint valid admin sessions offline. The startup validation script (scripts/check-env.js) verifies DATABASE_URL is set but does not require APP_SECRET, so the weak path is the silent default. Fix: fail startup when APP_SECRET is absent; one guard clause in check-env.js, plus a documented token-rotation step.

A.2 — Legacy tokens bypass password-change invalidation (MEDIUM)

EVIDENCE — src/lib/auth.ts, lines 34–37

```
// Reject tokens issued before the current password.
// Allow legacy stateless tokens that were minted without a password fingerprint.
if (user && payload.pwd && hash(user.password) !== payload.pwd) {
  user = null;
}
```

The design is good: tokens carry a hash of the password at issue time, so changing the password invalidates them. But the check only fires if payload.pwd exists — a token minted before this feature (or forged without the field, given A.1) skips invalidation entirely and remains valid forever. The same conditional pattern appears in the Redis path at line 46. Fix: reject tokens lacking the pwd claim after a published sunset date.

A.3 — Schema name interpolated into SQL (LOW)

EVIDENCE — src/lib/prisma.ts, line 423

```
await client.$executeRawUnsafe(`SET search_path TO "${schema}";`);
```

The only string interpolation found in the database layer. The value comes from operator configuration, not user input, so this is not exploitable today — but it sits inside rawQuery(), the hot path every analytics query flows through, where a future refactor could make it dangerous. Notably, the rest of this function is exemplary: named placeholders are converted to bound positional parameters (lines 426–439), which is why heavy raw-SQL use across the 87-file query layer is not an injection concern. Fix: validate the schema name against an identifier whitelist at config-load time.

A.4 — Stateless session tokens have no expiry (MEDIUM)

EVIDENCE — src/app/api/auth/login/route.ts, line 42 · src/lib/jwt.ts, lines 16–18

```
token = createSecureToken({ userId: user.id, role, pwd }, secret());

export function createSecureToken(payload: any, secret: any, options?: any) {
  return encrypt(createToken(payload, secret, options), secret);
}
```

createSecureToken accepts JWT options, but the login path never passes expiresIn, so non-Redis deployments issue tokens that are valid indefinitely. A token lifted from a device, log or backup works until the user changes their password (and, per A.2, sometimes even then). Redis-backed deployments fare better — sessions are stored with a TTL (src/lib/redis.ts, lines 49–51). Fix: sign with a bounded lifetime and implement refresh; roughly a day of work.

A.5 — No throttling on the login endpoint (MEDIUM)

EVIDENCE — src/app/api/auth/login/route.ts, lines 12–30

```
const user = await getUserByUsername(username, { includePassword: true });

if (!user || !checkPassword(password, user.password)) {
  return unauthorized({ code: 'incorrect-username-password' });
}
```

No rate limiting, attempt counting, lockout or CAPTCHA exists on this route or in middleware; searching the codebase finds no throttling mechanism anywhere in the auth path. The only brute-force cost is bcrypt verification itself (cost factor 10, src/lib/password.ts, line 3). For self-hosters behind a reverse proxy this is arguable; for a commercial cloud product it is not. Fix: per-IP and per-account throttling at the edge or in middleware. One credit: the error message is properly uniform, so the endpoint does not leak whether a username exists.

APPENDIX B

What holds up under scrutiny

Diligence that only lists faults is half a report. These are the places where close reading raised confidence rather than concerns — each verifiable at the reference given.

Encryption done properly	AES-256-GCM with random IV and salt, PBKDF2 key derivation, and authentication tags verified on decrypt — src/lib/crypto.ts, lines 5–46. Session tokens are JWTs additionally wrapped in this encryption (jwt.ts, lines 16–26), so payloads are not readable in transit logs.
Injection surface controlled	All 87 query-layer files route through a templating function that binds named parameters positionally (prisma.ts, lines 426–439; mirrored for ClickHouse in clickhouse.ts, line 408). A grep for direct interpolation into SQL across src/queries returns zero hits.
Ingest endpoint is hardened	The public /api/send collection endpoint validates every field with zod schemas, filters bots via isbot, checks blocked IPs, bounds numeric web-vitals, and — unusually thorough — rejects strings starting with spreadsheet formula triggers to prevent CSV-injection in exports (src/app/api/send/route.ts, lines 24–29).
Sensible auth details	Passwords hashed with bcrypt (password.ts); minimum length enforced at the API boundary (users/route.ts, line 15: z.string().min(8)); uniform login errors that don't disclose account existence; Redis sessions carry TTLs.
Headers configured	A Content-Security-Policy and per-route header sets are defined centrally in next.config.ts (line 61 onward) rather than left to defaults.
Visitor privacy by design	Visitor identifiers are derived with a salt that rotates on a schedule (crypto.ts, getSalt, lines 72–78), consistent with the product's cookieless privacy claims — the marketing and the implementation agree.

APPENDIX C

Measurements

METRIC	VALUE	READING
Application code	~58,300 lines TS/TSX	Mid-sized; one senior engineer can hold it in their head.
Runtime dependencies	66 (+39 dev)	Moderate supply-chain surface; unmonitored in CI (see body).
Test files	23 (17 unit, 6 e2e)	Well-aimed (auth, permissions, core libs) but thin for 58k lines.
TypeScript strictness	strict: true, strictNullChecks: false	A meaningful carve-out: null-safety is off, and 327 explicit 'any' annotations remain.
Raw-SQL query files	87	Large surface, uniformly parameterised (Appendix A.3).
Commit concentration	~70% single author	Top contributor 3,860 commits; next two 1,073 and 434.
Community signals	~38k stars · ~106 open issues	Active the day of review; healthy issue ratio for its popularity.

ABOUT THIS SPECIMEN

This sample was prepared by George Brooks of Oxford Tech Consulting from publicly available source code, to demonstrate the structure, depth and plain-English style of a real engagement. It reflects the repository at a single point in time (2 August 2026, v3.2.0), involves no non-public information, and is not investment advice or a recommendation regarding Umami Software, Inc. Client engagements are confidential, NDA-covered, and include private code, infrastructure, cost analysis and team interviews. Findings noted here may already have been addressed by the maintainers.

Oxford Tech Consulting · oxfordtechconsultants.com · george@oxfordtechconsultants.com · Oxford, UK