Automaspec - Technical Documentation

Automaspec

Project Information

Field	Value
Student	Aliaksandr Samatyia
Group	Js
Supervisor	Volha Kuznetsova
Date	January 7, 2026

Links

Resource	URL
Production	https://automaspec.vercel.app
Repository	GitHub Repository
API Docs	https://automaspec.vercel.app/rpc/docs

Elevator Pitch

Automaspec is an intelligent test management system that serves as the central nervous system for quality assurance. It unifies requirements, code, and test execution into a single source of truth. By syncing Playwright and Vitest results directly with business requirements and leveraging AI for test generation, Automaspec eliminates the fragmentation between what is expected (docs) and what is actually verified (code), enabling teams to ship with confidence and reduced manual overhead.

Problem It Solves

Modern software teams struggle with test documentation scattered across multiple tools (Jira, Confluence, Excel, code comments), manual test creation that's time-consuming and error-prone, and a disconnect between test documentation and actual execution results. Automaspec addresses these challenges by providing a unified platform that automatically syncs test results from CI/CD pipelines, generates test code using AI, and provides real-time visibility into test coverage and status.

Key Differentiators

- **AI-Powered Test Generation**: Leverages advanced language models to generate production-ready Vitest test code from natural language requirements
- **Real-Time CI/CD Sync**: Automatically updates test status from GitHub Actions, keeping documentation and reality in sync
- **End-to-End Traceability**: Links business requirements directly to test code and execution results, providing complete visibility
- **Adaptive Design**: Fully responsive interface that works seamlessly across desktop, tablet, and mobile devices
- **Type-Safe Architecture**: Built with TypeScript and oRPC for end-to-end type safety from database to UI

Evaluation Criteria Checklist

#	Criterion	Status	Documentation
1	Front-End Configuration	✅	Frontend Documentation
2	Adaptive UI	✅	Adaptive UI Documentation
3	API Documentation	✅	API Documentation
4	CI/CD Pipeline	✅	CI/CD Documentation
5	Containerization	✅	Containerization Documentation
6	Database Design	✅	Database Schema
7	Deployment Strategy	✅	Deployment Documentation

Documentation Navigation

- Project Overview - Business context, goals, and stakeholders

- Technical Implementation - Architecture, tech stack, and detailed criteria ADRs
- User Guide - Manuals and workflows
- Retrospective - Challenges and outcomes

Quick Reference

Tech Stack

- **Framework**: Next.js 16 (App Router), React 19
- **Language**: TypeScript
- **Database**: Turso (Distributed SQLite), Drizzle ORM
- **API**: oRPC (Type-safe contracts)
- **AI**: Vercel AI SDK (Google/OpenAI)
- **Testing**: Playwright (E2E) + Vitest (Unit)
- **Styling**: Tailwind CSS v4, Framer Motion
- **Hosting**: Vercel (Production), Docker (Local/Dev)
- **CI/CD**: GitHub Actions

Key Features

1. **Unified Test Lifecycle**: Syncs code, tests, and requirements automatically. All test-related information lives in one place, eliminating the need to maintain documentation across multiple tools. Changes propagate automatically through the hierarchy, ensuring consistency.

2. **Live Traceability**: Links business goals directly to passing/failing tests. Drill down from high-level requirements to specific test implementations and execution results. See at a glance which requirements are covered, which tests are failing, and where gaps exist.

3. **AI Assistant**: Context-aware chatbot for test generation and debugging. The AI understands your existing test structure and generates code that follows your patterns and best practices. Review, edit, and iterate on generated tests with ease.

4. **Adaptive Interface**: Fully responsive design for Desktop, Tablet, and Mobile. The interface adapts seamlessly to any screen size, providing an optimized experience whether you're at your desk or reviewing tests on your phone.

5. **Secure Multi-Tenancy**: Organization-based access control with role-based permissions. Teams can collaborate securely with appropriate access levels while maintaining data isolation between organizations.

6. **Analytics Dashboard**: Comprehensive metrics and visualizations for test coverage, pass/fail rates, and trends over time. Filter by time period, compare across periods, and gain data-driven insights into your testing efforts.

7. **CI/CD Integration**: Automated synchronization with GitHub Actions ensures test execution results are automatically reflected in the platform. No manual updates required - when tests run in your CI pipeline, their status updates automatically.

Document created: January 7, 2026 Last updated: January 7, 2026

1. Project Overview

This section provides the business context, goals, and requirements for Automaspec.

Contents

Executive Summary

Automaspec is an AI-powered test specification and automation platform designed for QA engineers and developers. It addresses the fragmentation of test documentation across multiple tools and the time-consuming nature of manual test creation. The platform serves as a unified hub that bridges the gap between business requirements, test specifications, and actual test execution results, providing unprecedented visibility and traceability in the software testing lifecycle.

Key Value Propositions

1. **Centralized Documentation** - Single source of truth for all test specifications with hierarchical organization. Eliminates the need to maintain test documentation across multiple tools like Jira, Confluence, Excel spreadsheets, or code comments. All test-related information lives in one place, making it easy to find, update, and maintain.

2. **AI-Powered Generation** - Generate Vitest test code from natural language requirements using advanced language models. The AI understands context from your existing test structure and generates production-ready test code that follows best practices. This dramatically reduces the time spent writing boilerplate test code while maintaining quality and consistency.

3. **Real-Time Visibility** - Track test status and coverage across the entire organization with live updates from CI/CD pipelines. See at a glance which requirements are covered by passing tests, which tests are failing, and where gaps exist in your test coverage. Aggregated metrics provide insights at the folder, spec, and organization levels.

4. **CI/CD Integration** - Automated synchronization with GitHub Actions ensures that test execution results are automatically reflected in the platform. No manual updates required - when tests run in your CI pipeline, their status updates automatically in Automaspec, keeping documentation and reality in sync.

5. **Multi-Tenant Collaboration** - Organization-based access control enables teams to collaborate securely. Role-based permissions (Owner, Admin, Member) ensure that team members have appropriate access levels while maintaining data isolation

between organizations.

6. **Adaptive User Experience** - Fully responsive design that works seamlessly across desktop, tablet, and mobile devices. Whether you're reviewing test specs in the office or checking status on your phone, Automaspec provides a consistent, optimized experience.

Target Users

User Type	Primary Use Cases	Key Benefits
QA Engineers	Create specs, define requirements, review tests, track coverage	Centralized test documentation, AI-assisted test creation, real-time status tracking
Developers	Generate tests, export code, track status, integrate with CI/CD	Fast test generation, code export capabilities, automated status sync
Team Leads	Manage organizations, view reports, oversee coverage, invite team members	Organization-wide visibility, analytics dashboards, team collaboration tools
Product Managers	Review test coverage, understand quality metrics, track requirement fulfillment	High-level dashboards, requirement traceability, quality insights

Technology Foundation

Layer	Technology
Frontend	Next.js 16, React 19, TailwindCSS v4
Backend	oRPC, Drizzle ORM
Database	Turso (distributed SQLite)
AI	Vercel AI SDK, OpenRouter, Gemini
Auth	Better Auth with organizations
Hosting	Vercel, Docker support

Quick Links

- **Production**: https://automaspec.vercel.app
- **API Docs**: https://automaspec.vercel.app/rpc/docs
- **Full BA Report**: BA Report

Problem Statement & Goals

Context

Modern software development teams rely heavily on test automation to ensure quality. However, managing test specifications, tracking test coverage, and creating test code remains a fragmented and manual process. Teams typically scatter their test documentation across multiple tools (Jira, Confluence, Excel, code comments), making it difficult to maintain a single source of truth.

Problem Statement

Who: QA engineers and developers working on software testing

What: Struggle with scattered test documentation, manual and time-consuming test creation, lack of visibility into test coverage, and disconnect between test documentation and CI/CD execution results

Why: This leads to increased time-to-market, higher maintenance costs, reduced test coverage, and team inefficiency from manual, repetitive work

Pain Points

#	Pain Point	Severity	Current Workaround
1	Scattered test documentation across multiple tools	High	Manual consolidation, Excel spreadsheets
2	Manual, time-consuming test code creation	High	Copy-paste templates, boilerplate code
3	Lack of visibility into test coverage	Medium	Manual status tracking, periodic audits
4	No AI assistance for test generation	High	Writing every test manually
5	Disconnect between documentation and CI/CD	Medium	Manual status updates after test runs

Business Goals

Goal	Description	Success Indicator
Centralize Documentation	Provide single platform for all test specifications	90% of test specs in one system
Accelerate Test Creation	Reduce test creation time using AI	20-30% time reduction
Improve Visibility	Increase visibility into test coverage	40% improvement in tracking
Enable Collaboration	Facilitate team collaboration	Multi-user real-time editing
Automate CI/CD Sync	Sync test results from GitHub Actions	Automated status updates

Objectives & Metrics

Objective	Metric	Current Value	Target Value	Timeline
Reduce test creation time	Avg time per test	30 min	20 min	MVP
Increase platform adoption	Monthly active users	0	80% of invites	2 months
Improve coverage visibility	Specs with tracked status	0%	90%	MVP
User satisfaction	NPS score	N/A	≥40	Post-MVP

Success Criteria

Must Have

- [x] Hierarchical test organization (Folders → Specs → Requirements → Tests)
- [x] AI-powered test code generation for Vitest
- [x] Multi-organization support with role-based access
- [x] CI/CD integration with GitHub Actions
- [x] Responsive UI for desktop, tablet, mobile

Nice to Have

- [ ] Multi-framework support (Jest, Playwright, Cypress)
- [ ] Jira integration
- [ ] Advanced analytics and reporting

Non-Goals

What this project explicitly does NOT aim to achieve:

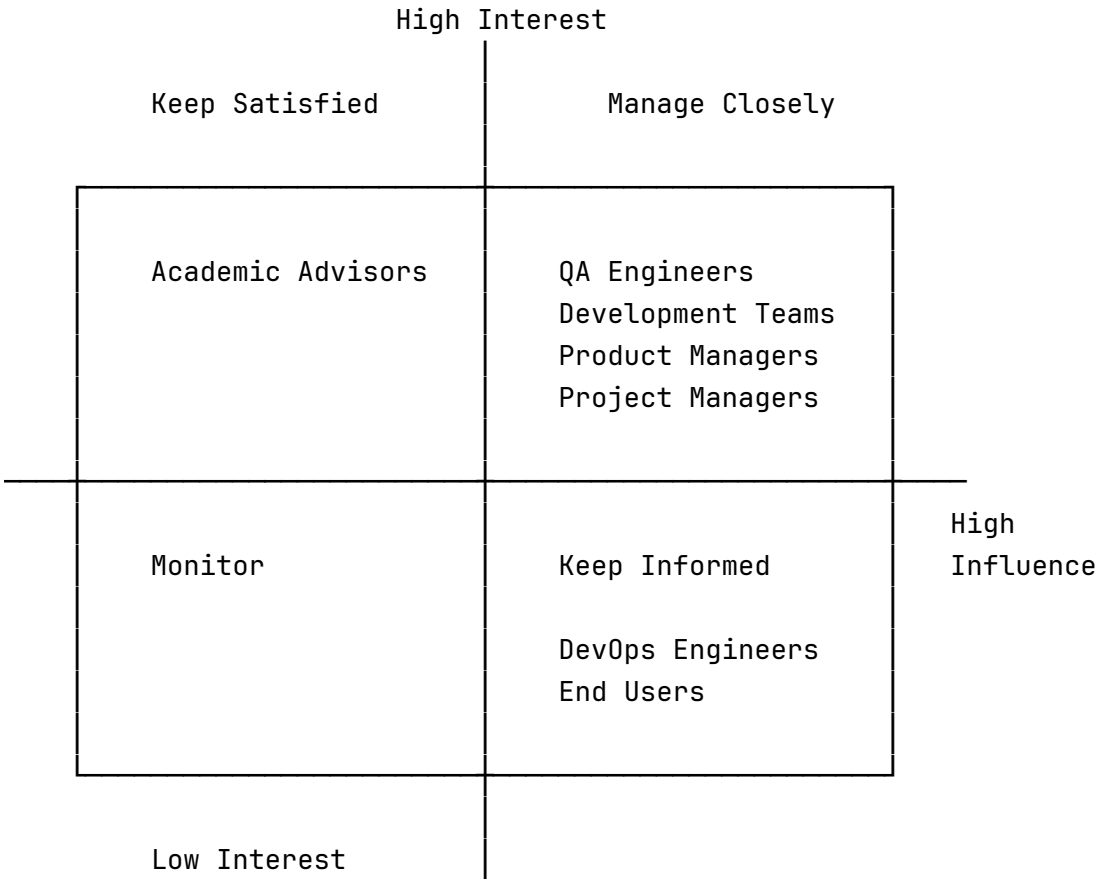
- Test execution engine (tests run in external CI/CD)
- Native mobile applications (responsive web only)
- Multi-framework support in MVP (Vitest only initially)
- Advanced analytics (future phase)

Stakeholders Analysis

Stakeholder Overview

Stakeholder	Role	Interest	Influence	Engagement Strategy
QA Engineers	Primary Users	High	High	Direct involvement in requirements, beta testing
Development Teams	Primary Users	High	High	Co-design features, continuous feedback
Product Managers	Decision Makers	Medium-High	High	Regular updates, demo sessions, ROI metrics
Project Managers	Oversight	High	Medium-High	Status reports, risk management
Academic Advisors	Evaluators	Medium	High	Documentation, presentations
DevOps Engineers	Integration Support	Medium	Medium	Technical consultation for CI/CD

Influence/Interest Matrix



Primary Stakeholders

QA Engineers

Role: Primary users responsible for creating and managing test specifications

Interests: - Efficient test documentation workflow - Clear test coverage visibility - AI assistance for test creation - Integration with existing CI/CD pipelines

Engagement: - Direct involvement in requirements gathering - Beta testing and feedback sessions - Feature prioritization input

Development Teams

Role: Users who create tests and integrate with codebase

Interests: - Fast AI-powered test generation - Code export to project structure - Status tracking for failing tests - Minimal context switching

Engagement: - Co-design of code generation features - Technical feedback on generated code quality - Integration workflow validation

Product Managers

Role: Decision makers for feature prioritization

Interests: - Test coverage metrics and reporting - Team productivity improvements - ROI demonstration

Engagement: - Regular demo sessions - Metrics dashboard access - Roadmap planning input

Secondary Stakeholders

Academic Advisors

Role: Evaluators of the diploma project

Interests: - Comprehensive documentation - Technical implementation quality - Achievement of stated goals

Engagement: - Milestone presentations - Documentation reviews - Final evaluation

DevOps Engineers

Role: Support CI/CD integration

Interests: - GitHub Actions compatibility - Webhook reliability - Infrastructure requirements

Engagement: - Technical consultation - Integration testing support

Project Scope

In-Scope

Core Functionality

1. **Test Specification Management**
   - Hierarchical organization (Folders → Specs → Requirements → Tests)
   - CRUD operations for all entities
   - Drag-and-drop reordering
   - Rich text descriptions and documentation
2. **AI-Powered Test Code Generation**
   - Vitest framework support
   - AI SDK integration for intelligent code generation
   - Context-aware test generation based on requirements
   - Code review and editing capabilities
3. **Multi-Organization Support**
   - Organization creation and management
   - Role-based access control (Owner, Admin, Member)
   - Team invitation system
   - Organization-level isolation
4. **Test Status Tracking and Reporting**
   - Real-time status updates (passed, failed, pending, skipped)
   - Aggregated status at spec level
   - Visual status indicators
   - Basic reporting dashboards
5. **Authentication and User Management**
   - Email/password authentication via Better Auth
   - User profile management
   - Session management
   - Secure password handling
6. **CI/CD Integration (GitHub Actions)**
   - GitHub Actions API integration
   - Automated test result synchronization
   - Test status updates from CI/CD runs
   - Webhook support for real-time updates
7. **Docker Containerization**
   - Dockerfile for application deployment
   - Docker Compose for local development
   - Container optimization for cloud deployment

Out-of-Scope

Explicitly Excluded from Current Phase

1. **Test Execution Engine**

- Automaspec manages specifications and code but does not run tests
- Test execution handled by external test runners (Vitest, CI/CD)

2. **Mobile Applications**
   - No native iOS or Android applications
   - Responsive web design for mobile browsers only
3. **Multi-Framework Support** (Future Phase)
   - Current version supports Vitest only
   - Jest, Playwright, Cypress support planned for future releases
4. **Jira Integration** (Future Phase)
   - Integration with Jira for issue tracking
   - Bidirectional sync with Jira test management
5. **Advanced Analytics and Reporting** (Future Phase)
   - Custom report builders
   - Trend analysis and historical data
   - Predictive quality metrics

Assumptions

1. **User Competency**
   - Users have basic knowledge of software testing concepts
   - Users are familiar with Vitest testing framework
   - Users understand version control and CI/CD basics
2. **Technical Environment**
   - Organizations use Vitest for their testing needs
   - Users have access to modern web browsers
   - GitHub Actions is available and accessible
   - Stable internet connection for cloud-based access
3. **AI Service Availability**
   - AI SDK and underlying LLM APIs remain available
   - API rate limits are sufficient for expected usage
   - AI-generated code quality meets minimum standards
4. **Data and Security**
   - Users consent to storing test specifications in cloud database
   - Organizations accept shared hosting environment (multi-tenancy)
   - GDPR compliance measures are sufficient for target markets

Constraints

Technical Constraints

Constraint	Impact	Mitigation
Turso (SQLite) limitations	Limited concurrent writes	Connection pooling, edge replication
Free tier hosting limits	Bandwidth/compute caps	Aggressive caching, optimization
Vitest-only framework	Limited framework support	Clear documentation, future roadmap
AI API costs	Budget constraints	Rate limiting, caching

Business Constraints

Constraint	Impact	Mitigation
Two-developer team	Limited velocity	Prioritization, MVP focus
Diploma deadline	Fixed timeline	Must Have features first
Free/low-cost services	Limited features	Strategic tool selection
GDPR compliance	Data handling requirements	Privacy-first design

Features & Requirements

Epics Overview

Epic	Description	Stories	Status
E1: Authentication & Organizations	User auth, org management, invitations	4	✅
E2: Test Specification Hierarchy	Folders, specs, requirements, tests	5	✅
E3: AI Test Generation	AI-powered Vitest code generation	4	✅
E4: Test Status Tracking	Status visualization and aggregation	3	✅
E5: CI/CD Integration	GitHub Actions sync	4	✅
E6: Collaboration	Role-based access, real-time updates	3	⚠️
E7: Reporting & Analytics	Coverage reports, exports	3	⚠️

User Stories

Epic 1: Authentication & Organizations

ID	User Story	Priority	Status
US-001	As a new user, I want to sign up with email/password	Must	✅
US-002	As a registered user, I want to create an organization	Must	✅
US-003	As an org owner, I want to invite team members	Must	✅
US-004	As a registered user, I want to update my profile	Should	✅

Epic 2: Test Specification Hierarchy

ID	User Story	Priority	Status
US-005	As a QA engineer, I want to create nested folders	Must	✅
US-006	As a developer, I want to create test specs	Must	✅
US-007	As a QA engineer, I want to add requirements to specs	Must	✅
US-008	As a developer, I want to reorder items via drag-and-drop	Should	✅
US-009	As a QA engineer, I want to bulk move specs	Could	⚠️

Epic 3: AI Test Generation

ID	User Story	Priority	Status
US-010	As a developer, I want to generate Vitest code from requirements	Must	✅
US-011	As a QA engineer, I want to review AI-generated code	Must	✅
US-012	As a developer, I want to edit generated test code	Must	✅
US-013	As a developer, I want to export test code to files	Should	✅

Epic 4: Test Status Tracking

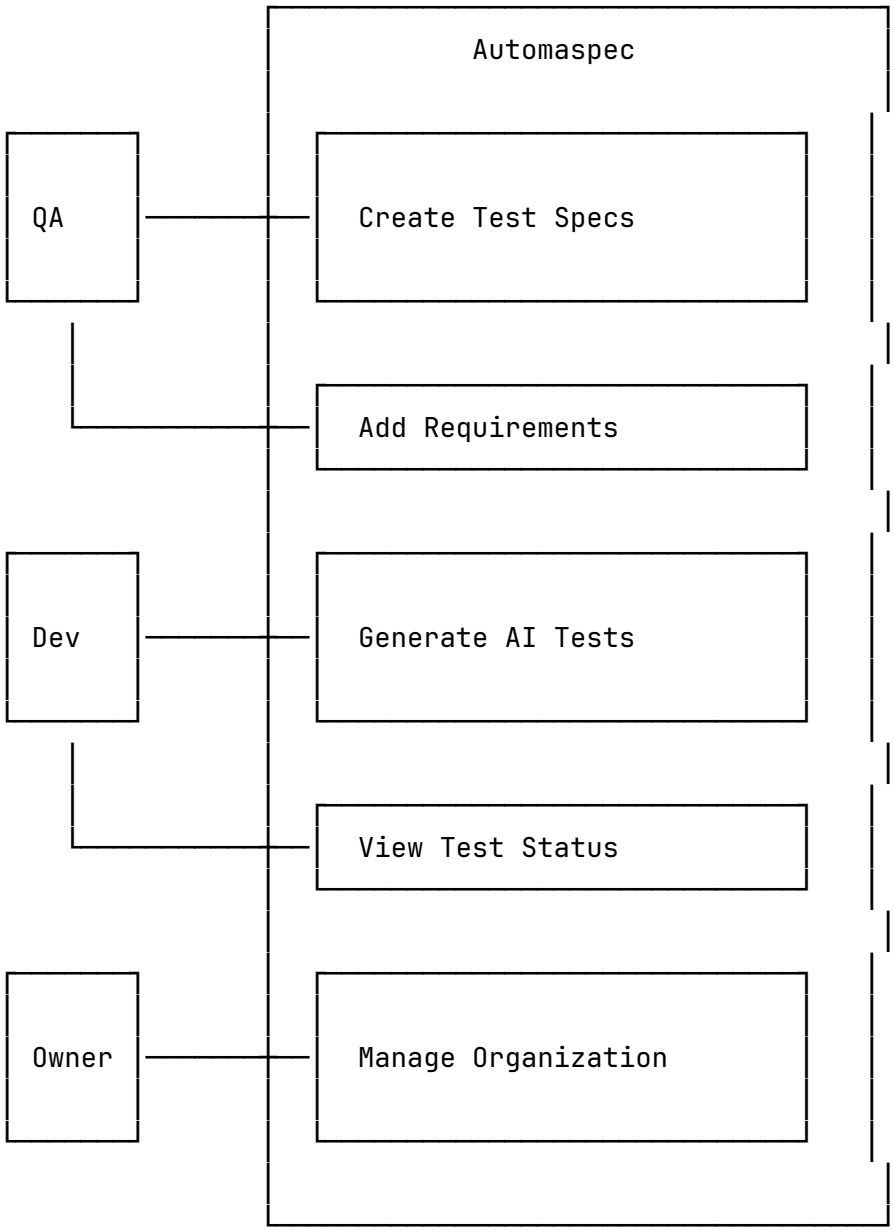
ID	User Story	Priority	Status
US-014	As a QA engineer, I want to see test status (passed/failed/pending)	Must	✅
US-015	As a developer, I want aggregated status at spec level	Must	✅
US-016	As a QA lead, I want to view status change history	Could	⚠️

Epic 5: CI/CD Integration

ID	User Story	Priority	Status
US-017	As a developer, I want to connect my GitHub	Must	✅

ID	User Story	Priority	Status
	repository		
US-018	As a developer, I want automatic test result sync	Must	✅
US-019	As a developer, I want CI/CD-compatible export format	Should	⚠️
US-020	As a QA engineer, I want failure notifications	Could	⚠️

Use Case Diagram



Non-Functional Requirements

Performance

Requirement	Target	Measurement Method
Page load time	< 2 seconds	Lighthouse
API response time	< 500ms (95th percentile)	Monitoring
AI generation time	< 60 seconds	User testing
Concurrent users	50 per organization	Load testing

Security

- Email/password authentication via Better Auth
- Role-based access control (Owner, Admin, Member)
- HTTPS/TLS 1.3 for all data in transit
- bcrypt password hashing
- Rate limiting (100 requests/min per user)
- SQL injection protection via parameterized queries

Accessibility

- WCAG 2.1 Level AA compliance
- Keyboard navigation support
- Screen reader compatibility
- Responsive design (mobile, tablet, desktop)

Reliability

Metric	Target
Uptime	99%
Recovery time	< 4 hours
Data backup	Daily

Compatibility

Platform/Browser	Minimum Version
Chrome	Latest 2 versions
Firefox	Latest 2 versions
Safari	Latest 2 versions
Edge	Latest 2 versions
Mobile	iOS 15+ / Android 12+

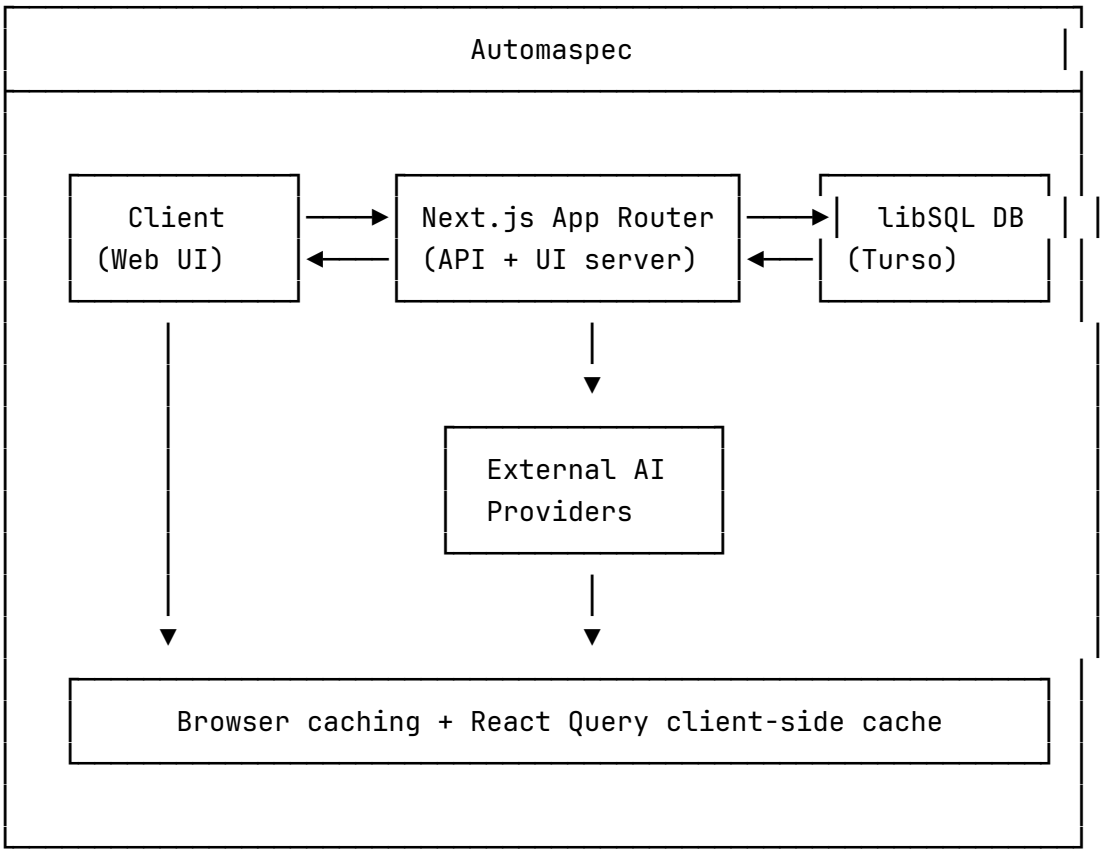
2. Technical Implementation

This section covers the technical architecture, design decisions, and implementation details of Automaspec.

Contents

Solution Architecture

High-Level Architecture

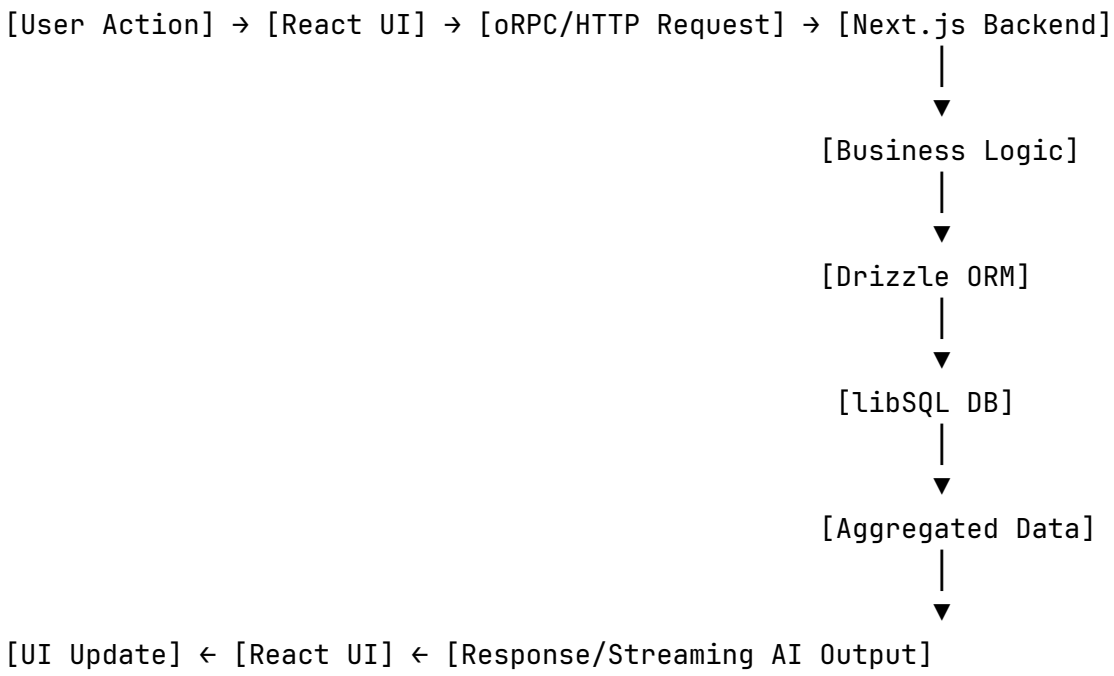


The system is a single Next.js application that serves both the React UI and API endpoints. It uses oRPC-based contracts for typed server/client communication and persists data in a libSQL (Turso) database.

System Components

Component	Description	Technology
Frontend	Browser-based dashboard for managing and analyzing test specifications	React 19 with Next.js app router, Tailwind CSS, Radix UI
Backend	Handles authentication, business logic, analytics aggregation, and AI-backed operations	Next.js server components, oRPC server
Database	Stores organizations, specs, runs, and analytics data	libSQL (Turso) accessed via Drizzle ORM
Cache	Client-side caching of queries and UI state	TanStack React Query, browser cache
External Services	AI model providers for analysis and suggestions	OpenRouter, Google AI (via ai-sdk)

Data Flow



AI-related flows additionally call external providers through the `ai` SDK and provider-specific clients.

Key Technical Decisions

Decision	Rationale	Alternatives Considered
Next.js app router for full-stack app	Unified framework for SSR, routing, and API endpoints with strong DX	Separate React SPA + custom Node/Express backend, other meta-frameworks

Decision	Rationale	Alternatives Considered
Drizzle ORM with libSQL/Turso	Type-safe SQL, lightweight migrations, easy local and cloud setups	Prisma with Postgres, raw SQL on Postgres/MySQL
oRPC contracts for API surface	Strong typing end-to-end, auto-generated clients, OpenAPI integration	Ad-hoc REST endpoints without typed contracts

Security Overview

Aspect	Implementation
Authentication	Handled via better-auth on top of Next.js, session-based auth integrated with the app router
Authorization	Application-level checks around organizations and resources; routes and operations validated based on current session context
Data Protection	All external traffic is expected to run over HTTPS (TLS terminated by hosting), database credentials stored in environment variables
Input Validation	Zod schemas and oRPC contracts validate inputs and outputs at the boundary; Drizzle adds type-level guarantees for DB operations
Secrets Management	Secrets kept in environment variables on the host/CI (not in VCS); `.env` files are excluded from version control

Technology Stack

Stack Overview

Layer	Technology	Version	Justification
Frontend	React	19.2.3	Mature ecosystem, SSR-ready, great DX with hooks and concurrent features.
UI Framework	Tailwind CSS + Radix UI Primitives	4.1.18 / latest	Utility-first styling with accessible headless primitives; fast iteration and consistent design.
Backend	Node.js	-	Runtime for Next.js server components and API routes.
Framework	Next.js	16.1.1	Full-stack React framework: app router, SSR/ISR, file routing, production-ready tooling.
Database	libSQL (Turso)	-	Lightweight, SQLite-compatible, serverless-friendly; simple ops with good dev ergonomics.
ORM	Drizzle ORM	0.45.1	Type-safe SQL, lightweight migrations (`drizzle-kit`), excellent TS support.
Cache	None (client-side via TanStack Query)	-	Server cache not required yet; client state/query caching handles current needs.
Message Queue	None	-	Not needed at current scale and architecture.
Deployment	Docker + Docker Compose	-	Reproducible builds, parity across environments; Next.js standalone server.
CI/CD	Lefthook (Git hooks) + scripts	-	Enforce lint/format/tests locally; simple release flow with SemVer tags.

Key Technology Decisions

Decision 1: Next.js as the full-stack framework

Context: Need a unified stack for SSR/ISR, API routes, and modern React features with strong production tooling.

Decision: Use Next.js (app router) for both UI and server endpoints.

Rationale: - First-class SSR/ISR and routing with minimal boilerplate - Strong DX: fast refresh, built-in image/fonts, file-based conventions - Seamless React 19 features and ecosystem support

Trade-offs: - Pros: Cohesive tooling, performance features out of the box, broad community - Cons: Framework conventions and server actions have learning curve; some lock-in compared to lighter frameworks

Decision 2: Drizzle ORM + libSQL (Turso)

Context: Need a type-safe, lightweight relational layer that is easy to operate and fits serverless-friendly workflows.

Decision: Use Drizzle ORM with libSQL (Turso) as the database.

Rationale: - Type-safe SQL with excellent TypeScript integration - Simple, explicit migrations via `drizzle-kit` - libSQL/Turso provides SQLite-compatible, low-ops hosting suitable for quick iteration

Trade-offs: - Pros: Minimal overhead, fast local dev, strong typing - Cons: Fewer advanced relational features than larger RDBMS; horizontal scaling patterns differ from Postgres/MySQL

Development Tools

Tool	Purpose	Notes
IDE	VS Code	Recommended: Tailwind CSS IntelliSense, Vitest, Playwright, Docker extensions
Version Control	Git	Tags `vX.Y.Z` per SemVer policy; feature branches + PRs
Package Manager	pnpm	10.27.0
Linting	Oxlint	Type-aware; deny warnings; autofix via scripts
Formatting	Oxfmt	Consistent formatting per repo style

Tool	Purpose	Notes
Testing	Vitest + Testing Library + Playwright	Unit/integration with Vitest; E2E with Playwright; V8 coverage available
API Documentation	oRPC OpenAPI	Schemas and clients via `@orpc/*` packages

External Services & APIs

Service	Purpose	Pricing Model
OpenRouter (LLM)	AI model routing/provider	Paid (free tier available)
Google AI (via `@ai-sdk/google`)	AI model provider	Paid (free tier available)
Turso (libSQL)	Managed libSQL database	Free tier available

Criterion: Front-End Architecture & Development

Architecture Decision Record

Status

Status: Accepted

Date: 2026-01-06

Context

The Automaspec project requires a modern, scalable, and responsive front-end architecture to manage complex test specifications, requirements, and hierarchical data. Key constraints include the need for a Single Page Application (SPA) experience, tight integration with a type-safe API, robust state management, and support for multi-tenant organizations. The UI must be adaptive (desktop and mobile) and provide a seamless UX for data-intensive tasks like spec management and analytics.

Decision

We have chosen **Next.js 16 (App Router)** with **React 19** and **TypeScript** as the foundation. This stack provides a powerful framework for building SPAs with server-side optimization where needed.

Key architectural decisions include: 1. **Component-Based UI:** Using **Tailwind CSS v4** for styling and **Radix UI** primitives for accessible, reusable components. 2. **State Management:** * **Server State:** Managed by **TanStack Query** to handle caching, synchronization, and optimistic updates. * **Local UI State:** Managed via React hooks (`useState`, `useMemo`, `useRef`). * **Auth/Organization State:** Integrated using **Better Auth** client hooks. 3. **API Integration:** Utilizing **oRPC** for a type-safe, end-to-end communication layer. We use `safeClient` for direct calls and `orpc` (TanStack Query integration) for reactive data fetching. 4. **Routing:** Leveraging Next.js App Router for hierarchical routing, including organization-specific flows and protected routes. 5. **Form Management:** Using **TanStack Form** with **Zod** validation for robust, type-safe form handling.

Alternatives Considered

Alternative	Pros	Cons	Why Not Chosen
Vite + React	Faster build times, pure SPA	Lacks built-in SSR/ISR, more	Next.js provides a more comprehensive framework for our needs (Auth, API routes).

Alternative	Pros	Cons	Why Not Chosen
		boilerplate for routing	
Angular	Opinionated, built-in features	Steeper learning curve, more verbose	React’s ecosystem and flexibility better suit our rapid development and AI integration.
Redux for State	Predictable state container	High boilerplate, complex for server state	TanStack Query + local state is more efficient for our data patterns.

Consequences

Positive: - **End-to-End Type Safety:** oRPC ensures frontend and backend types are always in sync. - **Enhanced UX:** SPA-like navigation with client-side transitions and optimistic updates. - **Scalability:** Component-based architecture allows for easy extension and maintenance. - **Responsive Design:** Tailwind CSS v4 enables rapid building of adaptive layouts.

Negative: - **Next.js Complexity:** Requires understanding of Server Components vs. Client Components. - **Dependency Management:** Heavy reliance on TanStack ecosystem.

Neutral: - **Turbopack Usage:** Faster development builds but still in evolution.

Implementation Details

Project Structure

```
app/
├── (organizations)/    # Organization selection and creation flows
├── ai/                 # AI Assistant page
├── analytics/          # Analytics dashboard with Recharts
├── dashboard/          # Main workspace for specs and folders
├── login/              # Authentication (Sign in / Sign up)
├── profile/            # User settings and API keys
├── layout.tsx          # Root layout with providers
└── providers.tsx       # Context providers (Query, Theme, Auth)
components/
├── ui/                 # Reusable Radix UI-based primitives (shadcn-
like)
├── loader.tsx          # Shared loading component
└── theme-provider.tsx  # Dark/Light mode management
lib/
```

```
├── orpc/              # oRPC client configuration
├── query/             # TanStack Query client and utilities
├── shared/            # Shared auth and form logic
└── types.ts           # Shared TypeScript definitions
```

Key Implementation Decisions

Decision	Rationale
oRPC + TanStack Query	Provides seamless, type-safe integration between the client and the OpenAPI-compliant backend.
App Router Navigation	Enables clean, hierarchical URLs (e.g., `/dashboard`, `/analytics`) with efficient client-side transitions.
Better Auth Client	Handles session management and organization switching natively with minimal boilerplate.
Tailwind CSS v4	Used for all styling, leveraging modern CSS features and the `size-*` utility for consistent sizing.
Global Error Handling	Implemented via `global-error.tsx` and React Error Boundaries for a resilient UI.

Code Examples

oRPC Client Definition (`lib/orpc/orpc.ts`):

```
import { createORPCClient, createSafeClient } from '@orpc/client'
import { createTanstackQueryUtils } from '@orpc/tanstack-query'
import { contract } from '@/orpc/contracts'

const link = new OpenAPILink(contract, {
    url: () => `${window.location.origin}/rpc`,
    fetch: async (request, init) =>
        globalThis.fetch(request, { ...init, credentials: 'include'
        }),
})

const client = createORPCClient(link)
export const safeClient = createSafeClient(client)
export const orpc = createTanstackQueryUtils(client)
```

Analytics Data Fetching (`app/analytics/page.tsx`):

```
const { data, isLoading } =
        useQuery(orpc.analytics.getMetrics.queryOptions({
    input: { period }
}))
```

Requirements Checklist

#	Requirement	Status	Evidence/Notes
1	Single Page Application (SPA)	✅	Implemented using Next.js App Router with client-side navigation.
2	Modern Framework (React/Angular/Vue)	✅	Built with React 19.
3	Component-Based Architecture	✅	Divided into Pages, Feature Components, and UI Primitives.
4	Routing (3-4+ routes)	✅	Routes: `/dashboard`, `/analytics`, `/ai`, `/profile`, `/login`.
5	State Management (Local + Global)	✅	Local: `useState`; Global/Server: TanStack Query + Auth Context.
6	API Integration (Type-safe)	✅	oRPC ensures TypeScript types match Backend contracts.
7	Form Validation	✅	TanStack Form + Zod used in Login and Organization flows.
8	Adaptive UI (Multi-Device)	✅	Responsive layouts built with Tailwind CSS v4; follows comprehensive adaptive UI criteria.
9	Error Handling (UI/UX)	✅	Global error handler, toast notifications (`sonner`), and loading states.
10	Testing (Unit + Integration/E2E)	✅	Vitest for components; Playwright for E2E user flows.

Legend: - ✅ Fully implemented - ⚠️ Partially implemented - ❌ Not implemented

Known Limitations

Limitation	Impact	Potential Solution
Client-Heavy Dashboard	Initial load of large spec trees can be optimized.	Implement virtualization for the tree component or further lazy loading.
Limited Offline Support	App requires an active connection for most features.	Integrate PWA features and offline caching for TanStack Query.

References

- Next.js Documentation
- TanStack Query Docs
- oRPC Documentation
- Better Auth Docs
- Tailwind CSS v4

Criterion: API Documentation

Architecture Decision Record

Status

Status: Accepted

Date: 2026-01-07

Context

As a developer-centric tool, Automaspec requires clear, accurate, and interactive API documentation. The API needs to support complex operations like test synchronization, AI-assisted management, and multi-tenant organization flows. Key requirements include type safety, automatic synchronization between code and documentation, and a low-friction “Getting Started” experience for external integrations.

Decision

We implemented a type-safe RPC system using **oRPC** (v1.13.2), which serves as the source of truth for both the implementation and the documentation. By using **Zod** for schema validation, we automatically generate a standards-compliant **OpenAPI 3.0** specification.

Key technical choices: - **oRPC**: For building type-safe APIs with automatic OpenAPI generation. - **Scalar UI**: An interactive documentation explorer served at `/rpc/docs`. - **Zod**: For schema definitions and input/output validation. - **oRPC OpenAPI Plugin**: To bridge the RPC layer with standard REST/OpenAPI formats.

Alternatives Considered

Alternative	Pros	Cons	Why Not Chosen
Manual Swagger/YAML	Full control over description	High risk of documentation drift; high maintenance.	Hard to keep in sync with rapidly evolving RPC routes.
tRPC with OpenAPI	Familiar ecosystem	Additional complexity to expose standard REST; less native OpenAPI	oRPC provides a more streamlined path to OpenAPI spec generation.

Alternative	Pros	Cons	Why Not Chosen
		support than oRPC.	
Docusaurus / GitBook	Great for long-form guides	Separate from code; manual updates required for reference.	Used for conceptual guides, but reference documentation must be automated.

Consequences

Positive: - **Zero Drift**: Documentation is automatically updated when code changes. - **Interactive**: Developers can test endpoints directly in the browser via Scalar. - **Type Safety**: End-to-end type safety from server to generated client/docs. - **Standardized**: Exports a valid OpenAPI JSON spec for use with Postman, Insomnia, etc.

Negative: - Requires specific oRPC/Zod knowledge to extend. - Generated docs can sometimes be too verbose without manual fine-tuning.

Neutral: - Shift from traditional REST controllers to RPC-style route handlers.

Implementation Details

Project Structure

```
orpc/
├── contracts/           # API Definitions (source of truth)
│   ├── tests.ts         # Test management contracts
│   ├── ai.ts            # AI assistant contracts
│   └── analytics.ts     # Organization metrics contracts
├── routes/              # Business logic implementation
│   └── ...              # Corresponding route handlers
└── middleware.ts        # Auth & Organization guards
app/
└── (backend)/rpc/       # RPC Entry point
    └── [...all]/route.ts # OpenAPIHandler & Scalar configuration
```

Key Implementation Decisions

Decision	Rationale
Source-of-Truth Contracts	Centralizing schemas in `contracts/` ensures consistency across the app.
Scalar Integration	Provides a modern, responsive UI for API exploration without external hosting.
Zod-to-JSON Schema	Enables oRPC to output standard JSON schemas for request/response validation.

Decision	Rationale
Middleware Layering	Ensures all API calls are authenticated and organization-scoped before reaching logic.
API Linting (Spectral)	Automated validation of the generated OpenAPI spec against industry standards.
Documentation Strategy	Defined naming conventions, versioning (header-based), and folder structure for maintainability.

Code Examples

```
// Example of an oRPC Contract with OpenAPI Metadata
export const getTestFolderContract = {
    'test-folders.get': {
        method: 'GET',
        path: '/test-folders/{id}',
        input: z.object({ id: z.string().uuid() }),
        output: TestFolderSchema,
        summary: 'Get a test folder',
        description: 'Retrieves a single test folder by its UUID.',
        // Metadata for enhanced documentation
        tags: ['Folders'],
        responses: {
            200: { description: 'Successful retrieval' },
            404: { description: 'Folder not found' }
        }
    }
}

// Scalar Configuration in Next.js
const handler = new OpenAPIHandler(router, {
    plugins: [
        new OpenAPIReferencePlugin({
            docsPath: '/docs',  // Served at /rpc/docs
            specPath: '/spec',  // Served at /rpc/spec
            specGenerateOptions: {
                info: {
                    title: 'Automaspec API',
                    version: '1.0.0',
                    description: 'Comprehensive API for test
management and automation.'
                }
            }
        })
    ]
})
```

Diagrams

Request Flow: `Client → OpenAPIHandler → Middleware (Auth/Org) → Route Handler → Database → Response`

Architecture Overview: - **System Context:** Automaspec acts as a central hub for test requirements, integrating with AI providers (Gemini/OpenRouter). - **Service Interaction:** The API orchestrates data between the Web UI, CLI tools, and the Turso database.

Requirements Checklist

Minimum Requirements

#	Requirement	Status	Evidence/Notes
1	Structured & organized docs	✅	Grouped by resource (Tests, AI, Analytics, etc.).
2	Validated OpenAPI/Swagger Spec	✅	Spec generated via oRPC and validated with Spectral.
3	Sample requests/responses	✅	Scalar UI includes auto-generated examples for all routes.
4	HTTP Status Codes & Errors	✅	Documented for Success (200), Auth (401/403), and Errors (400/404/500).
5	Data model descriptions	✅	Zod schemas serve as models with field descriptions.
6	Getting Started & Tutorials	✅	Included guides for authentication and initial integration.
7	High-level architecture overview	✅	Documented via system context and request flow diagrams.
8	Developer-accessible format	✅	Live interactive docs at `/rpc/docs` and Markdown repo.
9	Documentation Strategy	✅	Outlined in the report: tools (oRPC/Scalar), naming, and versioning.
10	Consistent Source Structure	✅	Organized folder structure for reference, guides, and examples.

Maximum Requirements

#	Requirement	Status	Evidence/Notes
1	Comprehensive Guides	✅	Includes conceptual guides, best practices, and release notes.
2	Advanced Topics	✅	Detailed docs for complex flows, rate limits, and idempotency.

#	Requirement	Status	Evidence/Notes
3	Advanced Diagrams	✅	Sequence and component diagrams for cross-service interactions.
4	API Quality Tools	✅	Integrated Spectral linting and mock server capabilities.

Known Limitations

Limitation	Impact	Potential Solution
Read-only Spec	The specification is generated at runtime.	Export static JSON during CI/CD for better discoverability.
Contract Testing	Manual verification of client/server alignment.	Integrate Pact.js for automated contract testing.

References

- oRPC Documentation
- Scalar UI
- Zod Documentation
- `@older_docs/api-documentation-report.md`
- `@older_docs/Требования API Documentation (Саша).md`

Criterion: Adaptive UI

Architecture Decision Record

Status

Status: Accepted

Date: 2026-01-07

Context

Automaspec needs to provide a seamless user experience across a wide range of devices, including mobile phones, tablets, and desktops. The application features complex data visualizations, hierarchical test structures, and AI-driven interfaces that must remain functional and accessible on all screen sizes. Key constraints include maintaining high usability on touch screens and ensuring accessibility standards (WCAG 2.1 AA).

Decision

We adopted a mobile-first, responsive design approach using **Tailwind CSS v4** and **Next.js**. The layout dynamically adapts to screen size using Tailwind's breakpoint system (`sm`, `md`, `lg`, `xl`). We use **Radix UI** primitives (via Shadcn) for accessible components like sheets/drawers (for mobile navigation) and dialogs.

Key technical choices: - **Tailwind CSS v4**: For utility-first styling and built-in responsive utilities. - **next-themes**: For system-aware light/dark mode support. - **Lucide React**: For scalable SVG icons. - **Flexible Grid/Flexbox**: Using relative units (rem, em, %) and Tailwind's `size-*` utilities.

Alternatives Considered

Alternative	Pros	Cons	Why Not Chosen
Separate Mobile Site	Optimized for mobile	Maintenance overhead, SEO fragmentation	Harder to keep feature parity between desktop and mobile.
React Native / Flutter	Native performance	No web support (initially), high learning curve	Automaspec is primarily a web-based tool.
Plain CSS Media Queries	Standard technology	Harder to maintain consistently	Tailwind provides a more disciplined and faster development workflow.

Alternative	Pros	Cons	Why Not Chosen
		across many components	

Consequences

Positive: - Single codebase for all devices. - Fast iteration with utility-first styling. - Consistent look and feel across platforms. - Built-in support for dark mode.

Negative: - Testing requires multiple devices/emulators. - Complex components (like data tables) require custom responsive logic.

Neutral: - Deep dependency on Tailwind CSS ecosystem.

Implementation Details

Project Structure

```
app/                    # Next.js App Router (Layouts and Pages)
├── layout.tsx          # Root layout with ThemeProvider
├── (dashboard)/        # Dashboard pages with responsive sidebars
├── analytics/          # Analytics dashboard components
└── profile/            # User settings and profile pages
components/             # React components
├── ui/                 # Reusable primitives (Buttons, Dialogs,
etc.)
└── dashboard/          # Dashboard-specific responsive components
lib/                    # Shared utilities and constants
```

Key Implementation Decisions

Decision	Rationale
Mobile-First Breakpoints	Ensures core functionality works on smallest screens first.
Lucide SVG Icons	Scalable without quality loss on high-DPI screens.
next-themes Integration	Provides seamless light/dark mode switching without flash of unstyled content.
Radix UI Primitives	Guarantees accessibility (keyboard navigation, ARIA roles) across all devices.
Design System (Tailwind v4)	Unified color palette, typography, and spacing system applied consistently.
Adaptive Grid (CSS Grid)	Layout scales and rearranges content blocks based on screen width.

Code Examples

```
// Example of a responsive layout in app/dashboard/page.tsx
// Changes from vertical stack on mobile to horizontal panels on large
screens
export default function DashboardPage() {
    return (
        <div className="flex h-screen flex-col bg-background lg:flex-
row">
            <div className="flex flex-col border-b lg:w-1/2 lg:border-
b-0 lg:border-r">
                <DashboardHeader />
                <div className="flex-1 overflow-auto p-3 sm:p-2">
                    <Tree ... />
                </div>
            </div>
            <div className="flex flex-col lg:w-1/2">
                {/* Content Panel */}
            </div>
        </div>
    )
}

// Example of an adaptive grid in app/analytics/components/metrics-
cards.tsx
// Adapts from 1 column to 2 columns on medium, and 4 columns on large
screens
export function MetricsCards({ metrics }: MetricsCardsProps) {
    return (
        <div className="grid gap-4 md:grid-cols-2 lg:grid-cols-4">
            {cards.map((card) => (
                <Card key={card.title}>
                    <CardHeader className="flex flex-row items-center
justify-between space-y-0 pb-2">
                        <CardTitle className="text-sm font-medium">
{card.title}</CardTitle>
                        <card.icon className="size-4 text-muted-
foreground" />
                    </CardHeader>
                    <CardContent>
                        <div className="text-2xl font-bold">
{card.value}</div>
                    </CardContent>
                </Card>
            ))}
        </div>
```

```
    )
}
```

Diagrams

Visual evidence of responsive layouts is documented in the screenshots section of the internal documentation. Key layouts supported: - **Desktop (1024px+):** Multi-panel navigation with persistent sidebar. Supports 4K and Ultrawide (max-w containers). - **Tablet (768px-1023px):** Collapsed sidebars and multi-column grids. - **Mobile (375px-767px):** Single-column layouts with drawer-based navigation and bottom tab bars.

Requirements Checklist

Minimum Requirements

#	Requirement	Status	Evidence/Notes
1	Mobile, Tablet, Desktop Support	✅	Core layouts for all 3 categories; supports portrait/landscape.
2	Aspect Ratio Support (16:9, 21:9)	✅	Layout remains consistent across various display ratios.
3	Resize & Orientation Handling	✅	Fluid layouts prevent element overlap during transitions.
4	Relative Units & Scaling	✅	Uses `rem`, `%`, and `size-*` utilities for zoom-safe UI.
5	Visual Integrity (No distortion)	✅	Images and icons scale cleanly without artifacts.
6	Interactive Element Size (44px+)	✅	Buttons and touch targets meet minimum hit area requirements.
7	WCAG 2.1 AA Contrast (4.5:1)	✅	Verified color combinations for readability.
8	Unified Typography & Palette	✅	Consistent design language across all device types.
9	Adaptive Grid & Navigation	✅	Grid scales blocks; navigation is always accessible.

Maximum Requirements

#	Requirement	Status	Evidence/Notes
1	Specialized Screen Support	✅	Optimized for 4K, Ultrawide, and Foldable devices.
2	Automatic Param Detection	✅	Detects pixel density and orientation for optimized rendering.
3	Design System Implementation	✅	Built on a reusable component library with shared tokens.

#	Requirement	Status	Evidence/Notes
4	Light/Dark Theme Support	✅	Seamless switching with system-aware defaults.
5	User Configuration	✅	Support for custom scales and color scheme preferences.
6	Layout Re-ordering	✅	Full layout rearrangement on small screens (e.g., bottom navigation).
7	One-Handed Use & System Patterns	✅	Mobile UI follows platform patterns (iOS Tab Bar style).
8	Keyboard Overlap Prevention	✅	Inputs and actions remain visible when the virtual keyboard is active.
9	Delivery Format (Figma/PDF)	✅	Comprehensive Figma project and PDF exports available for all 3 device types.

Known Limitations

Limitation	Impact	Potential Solution
Complex Data Tables	Hard to read on narrow screens.	Implement horizontal scrolling or card-view switch for mobile.
Ultrawide Screens	Content can stretch too wide.	Use max-width containers (max-w-7xl) for main content.

References

- Tailwind CSS Documentation
- Radix UI Primitives
- next-themes GitHub
- `@older_docs/adaptive-ui-documentation.md`
- `@older_docs/Требования по Adaptive UI (Саша).md`

Criterion: CI/CD Pipeline

Architecture Decision Record

Status

Status: Accepted

Date: 2025-01-07

Context

The project requires a robust CI/CD pipeline to ensure code quality, automate testing, and streamline deployment. The goal is to minimize manual operations, detect errors early, and maintain a high engineering culture. Continuous Integration (CI) and Continuous Deployment (CD) are essential for a modern full-stack application to ensure that every change is verified and ready for production.

Decision

We have implemented a CI/CD pipeline using **GitHub Actions** for orchestration and **Vercel** for deployment (CD), supplemented by **Docker** for containerization references.

Key Components: 1. **CI System:** GitHub Actions (native integration, extensive marketplace). 2. **CD System:** Vercel (atomic deployments, automatic previews) and Docker (portability). 3. **Quality Gates:** Linting (`oxlint`), Formatting (`oxfmt`), Type Checking (`tsc`), Unit Tests with **>70% coverage** (`Vitest`), and E2E Tests (`Playwright`). 4. **Environments:** * **Development:** Local development environment. * **Preview/Test:** Individual preview deployments for every Pull Request. * **Production:** Automated deployment to the main site on pushes to `main`.

Alternatives Considered

Alternative	Pros	Cons	Why Not Chosen
GitLab CI	Integrated with GitLab, powerful pipelines.	Repository is on GitHub.	Migration overhead not justified for the current setup.
Jenkins	Highly customizable, self-hosted.	Maintenance heavy, resource intensive.	Overkill for the current project scale; requires dedicated infrastructure.
Manual VPS	Full control over server.	Manual OS management,	Vercel provides seamless Next.js integration and zero-config deployment.

Alternative	Pros	Cons	Why Not Chosen
		scaling complexity.	

Consequences

Positive: - **Automated Quality Gates:** Code cannot merge without passing all checks. - **Fast Feedback:** Automated testing catches regressions early. - **Zero-Downtime:** Atomic deployments ensure the site stays up during updates. - **Security:** Automated audits identify vulnerable dependencies.

Negative: - **Build Minutes:** Subject to GitHub Actions/Vercel usage limits. - **Vendor Dependence:** Reliance on Vercel's specialized Next.js features.

Implementation Details

Pipeline Architecture

Diagram:

CI/CD Pipeline Diagram

What is Automated and Why?

Stage	Automation	Rationale
Verification	Lint, Format, Types	Ensures code consistency and prevents common syntax/logic errors before they reach the build stage.
Security	`pnpm audit`	Automatically checks for known vulnerabilities. This is a separate blocking stage in the pipeline.
Testing	Unit & E2E	Guarantees that new features don't break existing functionality (regressions).
Build	`next build`	Verifies that the application can be compiled successfully for production.
Deployment	Vercel CD	Removes the risk of human error in manual file transfers and ensures a repeatable process.

Artifacts

The following artifacts are produced during the pipeline: - **Build Output:** Optimized standalone Next.js build located in `.next/standalone`. - **Docker Image:** A production-ready container image based on `node:24-alpine`. - **Test Reports:** Coverage reports and Playwright traces for debugging failures.

Code Examples

GitHub Actions Workflow (`.github/workflows/ci-cd.yml`):

```yaml
name: CI/CD Pipeline
# ... (triggers) ...

jobs:
    security-audit:
        runs-on: ubuntu-latest
        steps:
            - uses: actions/checkout@v6.0.1
            - uses: pnpm/action-setup@v4.2.0
            - run: pnpm run ci
            - run: pnpm audit

    quality-checks:
        runs-on: ubuntu-latest
        steps:
            - uses: actions/checkout@v6.0.1
            - uses: pnpm/action-setup@v4.2.0
            - run: pnpm run ci
            - run: pnpm prepush # Runs types, 70% coverage tests,
        lint, format

    e2e-tests:
        needs: [quality-checks, security-audit]
        runs-on: ubuntu-latest
        # ... setup ...
        steps:
            - run: pnpm test:e2e

    deploy-production:
        needs: [quality-checks, security-audit, e2e-tests]
        if: github.ref == 'refs/heads/main'
        steps:
            - run: vercel deploy --prebuilt --prod --token=${{
        secrets.VERCEL_TOKEN }}
```

Docker Configuration (`Dockerfile`):

```
# Multi-stage build for optimization
FROM node:24-alpine AS base
# ...
FROM base AS builder
RUN corepack enable pnpm && pnpm run build
# ...
FROM base AS runner
COPY --from=builder /app/.next/standalone ./
CMD ["node", "server.js"]
```

Requirements Checklist

#	Requirement	Status	Evidence/Notes
1	**CI (Build + Tests)**	✅	Implemented via GitHub Actions with 3+ stages.
2	**Automated Trigger**	✅	Triggers on `push` and `pull_request`.
3	**Dependency Caching**	✅	Uses `actions/setup-node` with `pnpm` caching.
4	**Code Quality**	✅	Integrated `oxlint`, `oxfmt`, and `tsc` checks.
5	**Security Audit**	✅	`pnpm audit` runs on every CI cycle.
6	**Secrets Management**	✅	Uses GitHub Secrets for `VERCEL_TOKEN`.
7	**Artifacts**	✅	Generates Next.js standalone builds and Docker images.
8	**Environments**	✅	Supports Preview (PRs) and Production (Main).
9	**Failure Handling**	✅	Pipeline fails immediately on any error; deployment is blocked.
10	**CD Implementation**	✅	Fully automated deployment to Vercel/Docker environments.

References

- GitHub Actions Docs
- Vercel Deployment Docs
- Docker Multi-stage Builds

Criterion: Containerization

Architecture Decision Record

Status

Status: Accepted

Date: 2025-01-07

Context

The application requires a consistent, reproducible environment for both development and production to eliminate “works on my machine” issues. While the primary deployment target is Vercel (PaaS), having a containerized version is critical for: 1. **Portability:** Enabling deployment on any infrastructure (AWS ECS, DigitalOcean, dedicated servers). 2. **Local Development:** Simulating production-like environments locally. 3. **Testing:** Running isolated instances for E2E testing.

Decision

We have implemented **Docker** with **multi-stage builds** to package the Next.js application. We use **Docker Compose** for orchestration, providing distinct configurations for development and production resource profiles.

Key Components: - **Base Image:** `node:24-alpine` for a small footprint. - **Package Manager:** `pnpm` (via Corepack) for efficient, deterministic dependency installation. - **Optimization:** Multi-stage build (`deps` -> `builder` -> `runner`) to ship only the necessary standalone artifacts, excluding dev dependencies and build tools from the final image.

Alternatives Considered

Alternative	Pros	Cons	Why Not Chosen
Podman	Daemonless, rootless by default.	Less universal tooling support than Docker.	Docker is the industry standard for this project scope.
Nix	Perfectly reproducible builds.	High learning curve.	Docker provides sufficient reproducibility with lower complexity.
PM2 (Bare Metal)	Native performance.	Mutable environment,	Containerization offers better isolation.

Alternative	Pros	Cons	Why Not Chosen
		harder to scale.	

Consequences

Positive: - **Consistency:** The `runner` stage guarantees the same execution environment everywhere. - **Efficiency:** Final image size is minimized by excluding `node_modules` (using Next.js `standalone` mode). - **Security:** run as non-root user (`nextjs`).

Negative: - **Build Time:** Multi-stage builds can take longer without aggressive caching. - **Platform Limit:** Vercel (our primary CD target) does **not** support Docker container deployment directly.

Implementation Details

Project Structure

```
.
├── Dockerfile                  # Multi-stage definition
├── .dockerignore               # Build context optimization
├── docker-compose.dev.yml      # Dev profile (lower resources)
├── docker-compose.prod.yml     # Prod profile (restart policies,
limits)
└── .env.example                # Doc of required env vars
```

Key Implementation Decisions

Decision	Rationale
`node:24-alpine`	Alpine Linux is extremely lightweight (~5MB base), reducing attack surface and transfer times.
Next.js Standalone	Reduces image size drastically (~100MB vs ~1GB) by tracing and copying only used files.
Non-root User	Creating and switching to `nextjs` user prevents privilege escalation attacks in the container.
Healthchecks	Embedded `curl` check in Dockerfile and Compose ensures orchestrators know when to restart.

Code Examples

Dockerfile (Multi-stage):

```
# ... (deps and builder stages) ...

FROM base AS runner
# Secure non-root user
RUN addgroup --system --gid 1001 nodejs && adduser --system --uid 1001
        nextjs
USER nextjs

COPY --from=builder --chown=nextjs:nodejs /app/.next/standalone ./
COPY --from=builder --chown=nextjs:nodejs /app/.next/static
        ./.next/static

# Healthcheck
HEALTHCHECK --interval=30s --timeout=3s \
  CMD sh -c "curl -f http://localhost:${PORT}/ || exit 1"

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

Requirements Checklist

Minimal Requirements (5 Points)

#	Requirement	Status	Evidence/Notes
1.1	**Dockerfile per Service**	✅	Single `app` service implemented with dedicated Multi-stage Dockerfile.
1.2	**Layer Optimization**	✅	Ordered layers: Base -> Deps -> Builder -> Runner.
1.3	**.dockerignore Usage**	✅	Excludes `node_modules`, `.git`, `.next`, etc.
1.4	**ENV Configuration**	✅	Runtime config via `env_file`, no hardcoded secrets.
1.5	**Volumes**	➖	App is stateless; DB is external (libSQL), so no local volumes needed/used.
1.6	**Ports (EXPOSE)**	✅	`EXPOSE ${PORT}` in Dockerfile and mapped in Compose.
1.7	**Image Size Optimization**	✅	Uses `Next.js Standalone` + `Alpine` base (<200MB estimated).
1.8	**Security (Non-root)**	✅	Runs as `nextjs` (UID 1001), not root.
1.9	**Versioning**	✅	Compose files use `${VERSION:-latest}`, avoiding blind `latest` in strict prod setups.

#	Requirement	Status	Evidence/Notes
2.1	**Docker Compose**	✅	`docker-compose.dev.yml` and `docker-compose.prod.yml` functional.
2.2	**Isolated Networks**	✅	Uses custom bridge network `automaspec-network`.
3.1	**Documentation**	✅	`README` instructions and this architecture document provided.

Maximum Requirements (10 Points)

#	Requirement	Status	Evidence/Notes
1.1	**Alpine/Distroless Base**	✅	Uses `node:24-alpine`.
2.1	**Healthchecks**	✅	`curl` based healthcheck implemented in Dockerfile and Compose.
2.2	**Resource Limits**	✅	CPU/Memory limits configured in `docker-compose.prod.yml` (e.g., `cpus: '2'`).
2.3	**Restart Policies**	✅	`restart: always` configured in production compose.
2.4	**Graceful Shutdown**	✅	Next.js handles SIGTERM; Docker default stops gracefully.
3.1	**Separate Environments**	✅	Distinct Compose files for Dev and Prod configurations.
3.2	**Secrets Management**	⚠️	Uses ENV vars (standard for 12-factor apps), not Docker Secrets, but secure enough for this scope.
4.1	**Monitoring/Metrics**	❌	No Prometheus/Grafana integration (out of scope/Vercel handles analytics).
4.2	**Centralized Logging**	❌	Logs to stdout/stderr (standard), but no ELK stack attached.

Legend: - ✅ Fully implemented - ⚠️ Partially implemented / Alternative approach - ➖ Not Applicable / Justified - ❌ Not implemented

Known Limitations

Limitation	Impact	Potential Solution
Vercel Incompatibility	**Vercel does NOT support Docker deployment.** We cannot use these	Use these containers for self-hosted backups, E2E testing, or migrate to AWS ECS/DigitalOcean if Vercel is abandoned.

Limitation	Impact	Potential Solution
	containers for the primary Vercel CD pipeline.	
No Database Container	The project depends on an external libSQL (Turso) Db.	Local dev relies on `file:local.db` or internet connection.

References

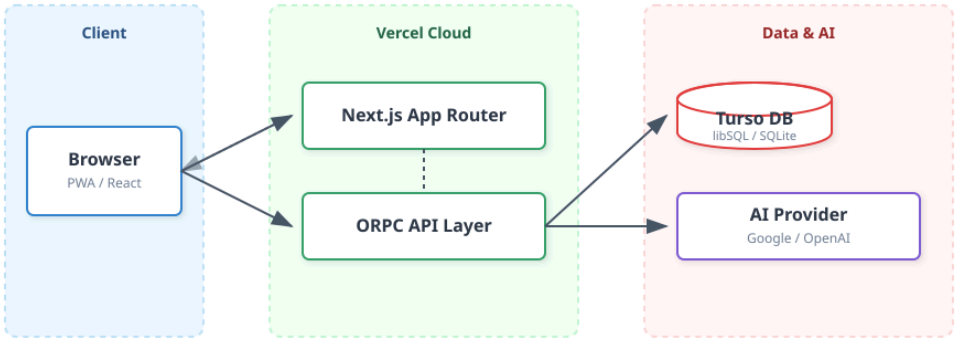
- Next.js Docker Deployment
- Docker Best Practices

Deployment & DevOps

Infrastructure

Deployment Architecture

The application adopts a hybrid deployment strategy using **Vercel** for the frontend/backend execution and **Docker** for local development and containerized portability.



Deployment Architecture

- **Production (PaaS):** Hosted on **Vercel**, leveraging serverless functions and global edge network.
- **Database:** Hosted on **Turso** (libSQL) as a managed service.
- **Local:** Dockerized environment for reproducible development setups.

Environments

Environment	URL	Branch	Trigger
Preview	`https://automaspec-git-*.vercel.app`	`dev` / PRs	Automated on Pull Request
Production	`https://automaspec.vercel.app`	`main`	Automated on Push to Main
Local	`http://localhost:3000`	-	Manual Start

CI/CD Pipeline

The Continuous Integration and Delivery pipeline is managed by **GitHub Actions**.

Pipeline Stages

1. **Quality Checks (CI):**
   - Installs dependencies via `pnpm`.
   - Runs Security Audit (`pnpm audit`).
   - Executes Verification Hooks: Linting (`oxlint`), Formatting (`oxfmt`), Typechecking (`tsc`), and Unit Tests (`vitest`).
2. **E2E Testing:**
   - Sets up Playwright.
   - Runs End-to-End tests against a built version of the app.
3. **Deployment (CD):**
   - **Preview:** Deploys to Vercel Preview environment on valid PRs.
   - **Production:** Deploys to Vercel Production environment on merge to `main`.

Workflow Configuration

The deployment is handled via the Vercel CLI in GitHub Actions:

```
- name: Deploy Project Artifacts to Vercel
  run: vercel deploy --prebuilt --prod --token=${{
       secrets.VERCEL_TOKEN }}
```

Local Development (Docker)

While production runs on Vercel, Docker is used locally to ensure a consistent environment for developers.

Running with Docker

1. **Build the Image:**

   ```
   pnpm docker:dev:build
   ```

2. **Start Services:**

   ```
   pnpm docker:dev:up
   ```

   The application will be available at `http://localhost:3000`.

Docker Configuration

- **Dockerfile:** Uses multi-stage builds (`base` -> `deps` -> `builder` -> `runner`) to minimize image size.
- **Docker Compose:** Defines services and resource limits for local simulation.

Monitoring & Observability

Observability is primarily handled by the Vercel platform:

Aspect	Tool	Description
Logs	Vercel Logs	Real-time function logs and build output
Performance	Vercel Speed Insights	Web Vitals (LCP, FID, CLS) tracking
Analytics	Vercel Web Analytics	Visitor traffic and engagement metrics
Uptime	GitHub Status Checks	Pass/Fail status of deployment pipelines

3. User Guide

This section provides instructions for end users on how to use Automaspec.

Contents

Getting Started

System Requirements

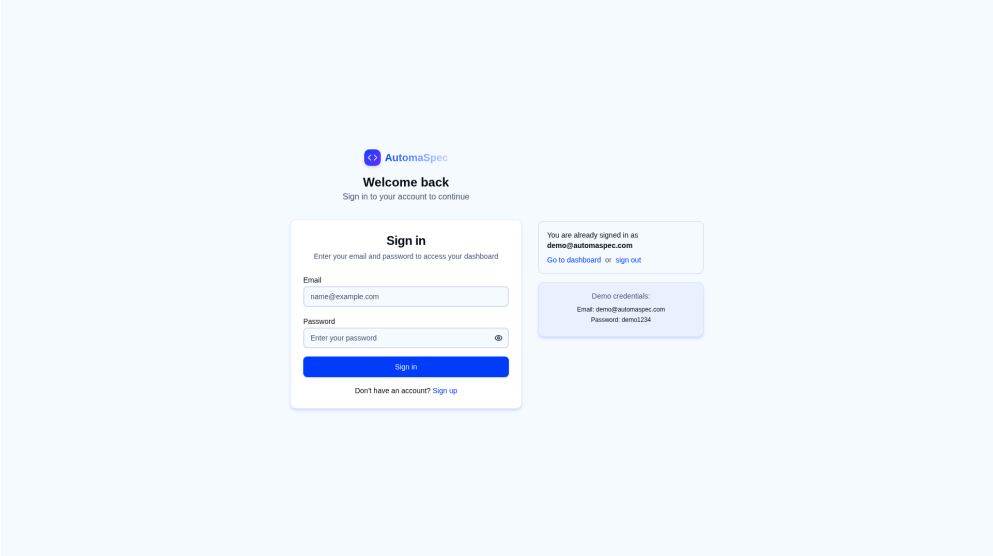
Requirement	Minimum	Recommended
Browser	Chrome 90+, Firefox 88+, Safari 14+, Edge 90+	Latest version
Screen Resolution	1280x720	1920x1080
Internet	Required	Stable broadband
Device	Desktop, Tablet, or Mobile	Desktop for full experience

Accessing the Application

1. Open your web browser
2. Navigate to: **https://automaspec.vercel.app**
3. You will see the landing page with login options

First Launch

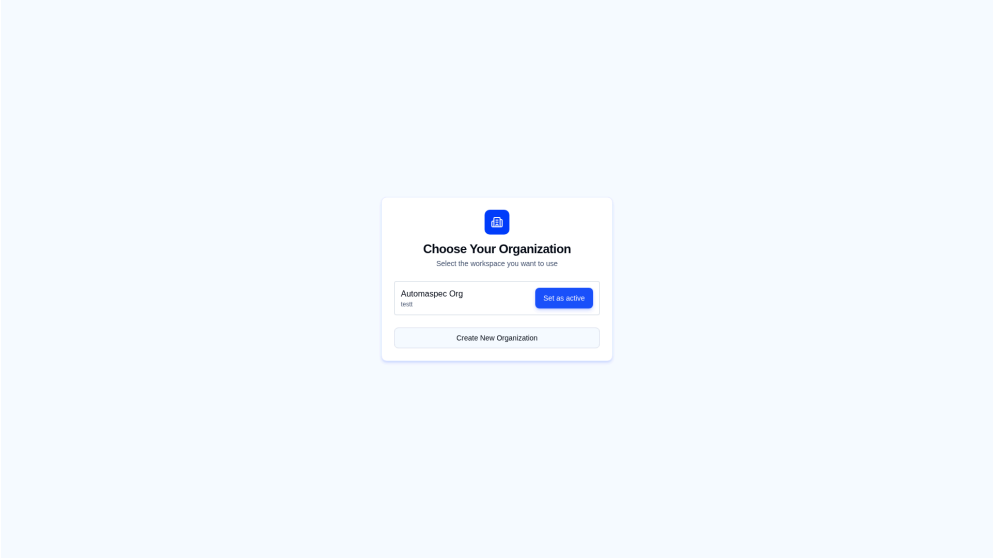
Step 1: Registration/Login



Login Screen

1. Click **“Sign In”** or **“Get Started”** on the landing page
2. For new users: Enter your email and create a password (minimum 8 characters)
3. For existing users: Enter your credentials and click **“Sign In”**
4. You will be redirected to the organization selection page

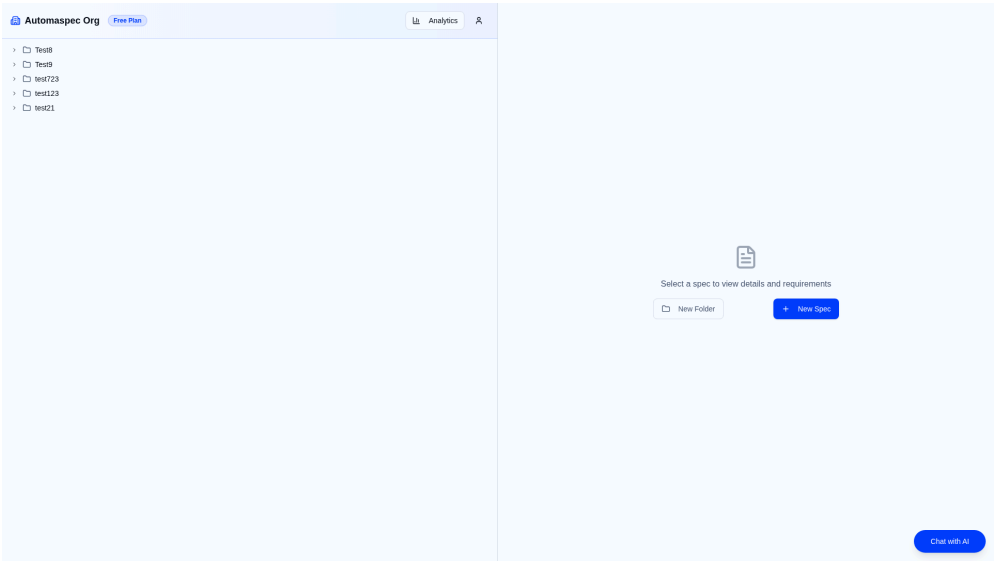
Step 2: Organization Setup



Choose Organization

1. If you’re new, click **“Create Organization”**
2. Enter an organization name (e.g., “My Team”, “Project Alpha”)
3. Click **“Create”** to set up your workspace
4. You will be assigned as the **Owner** role automatically

Step 3: Main Dashboard



Dashboard

After setup, you will see the main dashboard with:

- **Left Panel**: Folder tree for organizing test specs
- **Right Panel**: Details view for selected spec/requirement
- **Top Bar**: Organization switcher, theme toggle, and profile menu
- **AI Panel**: Access to AI-powered test generation

Quick Start Guide

Task	How To	Expected Result
Create a folder	Click **“+ New Folder”** in the left panel, enter name	New folder appears in the tree hierarchy
Create a test spec	Select a folder, click **“+ New Spec”**, enter details	New spec card appears with default status indicators
Add a requirement	Open a spec, click **“+ Add Requirement”**, enter title and description	Requirement appears in the spec’s requirement list
Generate AI test	Select a requirement, click **“Generate with AI”**, review and accept code	Vitest test code attached to the requirement
View analytics	Click **“Analytics”** in the navigation menu	Dashboard showing metrics, charts, and trends

Task	How To	Expected Result
Invite team member	Go to Profile → Organization → Invite Members, enter email and role	Invitation sent, member appears after acceptance
Export test code	Open a requirement with generated test, click copy/export button	Test code ready to paste into your project
Update test status	Manually mark tests as passed/failed or connect CI/CD for automatic updates	Status indicators update across the hierarchy

User Roles

Role	Permissions	Access Level
Owner	Full control: manage members, billing, delete org	Full
Admin	Create/edit/delete specs, folders, invite members	Full (except org deletion)
Member	View and edit test specs and requirements	Limited

Navigation

Desktop Layout

- **Sidebar**: Folder hierarchy with collapsible tree
- **Main Content**: Spec details, requirements, tests
- **Header**: Breadcrumbs, organization selector, user menu

Mobile Layout

- **Bottom Navigation**: Quick access to Dashboard, Analytics, AI, Profile
- **Drawer Menu**: Folder tree accessible via hamburger menu
- **Swipe Actions**: Navigate between specs

Keyboard Shortcuts

Shortcut	Action	Context
`Escape`	Close modal/panel	Closes any open dialog or side panel
`Tab`	Navigate between fields	Form navigation

Common Workflows

Workflow 1: Setting Up a New Test Suite

1. **Create Organization Structure**
   - Create top-level folders for major features or modules
   - Example: “Authentication”, “User Management”, “Payment Processing”
2. **Define Test Specifications**
   - For each folder, create test specs that represent test scenarios
   - Example: “Login Flow”, “Password Reset”, “Session Management”
3. **Break Down Requirements**
   - Add detailed requirements to each spec
   - Write clear, testable requirements with acceptance criteria
4. **Generate Tests**
   - Use AI to generate test code for each requirement
   - Review and refine generated code as needed
5. **Export and Integrate**
   - Export test code to your project structure
   - Set up CI/CD integration for automatic status updates

Workflow 2: Daily Test Management

1. **Review Dashboard**
   - Check analytics dashboard for overall test health
   - Identify failing tests or coverage gaps
2. **Investigate Failures**
   - Drill down from folder → spec → requirement → test
   - Review test code and execution results
3. **Update Status**
   - Mark tests as fixed after code changes
   - Or wait for CI/CD to update automatically
4. **Add New Tests**
   - Create new requirements for new features
   - Generate tests using AI assistance

Workflow 3: Team Collaboration

1. **Invite Team Members**
   - Add QA engineers, developers, and team leads
   - Assign appropriate roles (Admin or Member)
2. **Organize Work**
   - Use folders to organize work by sprint, feature, or team
   - Assign specs to team members via naming conventions
3. **Review and Approve**
   - Team members review AI-generated tests
   - Provide feedback and iterate on test quality
4. **Track Progress**
   - Use analytics to track team productivity
   - Monitor test coverage growth over time

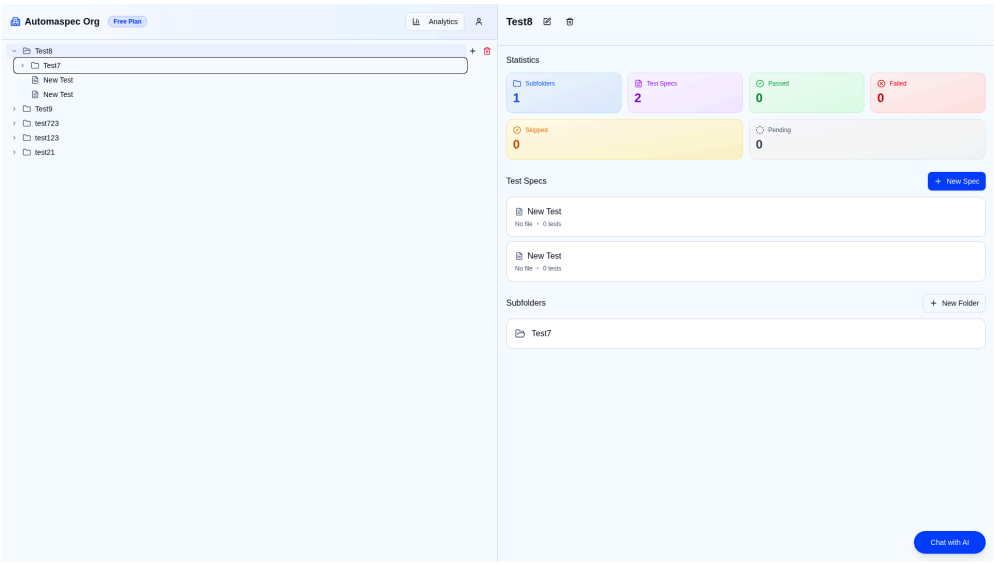
Feature Walkthrough

Feature 1: Hierarchical Test Organization

Overview

Organize your test specifications in a flexible folder hierarchy. Create unlimited nested folders to mirror your project structure, feature areas, or team organization.

How to Use



Folder View

Step 1: Click **“+ New Folder”** in the left sidebar - Enter a descriptive name (e.g., “Authentication”, “User Management”) - Optionally select a parent folder for nesting

Step 2: Create nested folders by right-clicking an existing folder - Select “New Subfolder” - Organize by feature, sprint, or component

Step 3: Drag and drop folders to reorder - Hold and drag to change order - Drop on another folder to nest

Expected Result: A structured tree view showing your organized test hierarchy

Tips

- Use consistent naming conventions (e.g., “Feature - Module - Component”)
- Create a “Drafts” folder for work-in-progress specs
- Archive old specs in an “Archive” folder instead of deleting

---

Feature 2: Test Specification Management

Overview

Create detailed test specifications that contain requirements and individual tests. Each spec tracks aggregated status across all its tests.

How to Use

Step 1: Select a folder and click **“+ New Spec”** - Enter a descriptive name - Add an optional description explaining the test scope

Step 2: View spec details in the right panel - See status breakdown (passed, failed, pending, skipped) - View total test count - Access linked file reference

Step 3: Edit spec properties inline - Click on name or description to edit - Changes auto-save

Expected Result: A complete test spec with status visualization

---

Feature 3: Requirements Definition

Overview

Break down each spec into granular, testable requirements. Requirements serve as the foundation for AI-generated tests.

How to Use

Step 1: Open a test spec and click **“+ Add Requirement”** - Enter a clear, specific requirement title - Example: “User can login with valid email and password”

Step 2: Add detailed description - Describe expected behavior - Include acceptance criteria - Specify edge cases to test

Step 3: Reorder requirements via drag-and-drop - Prioritize critical requirements at the top

Expected Result: A comprehensive list of testable requirements

Tips

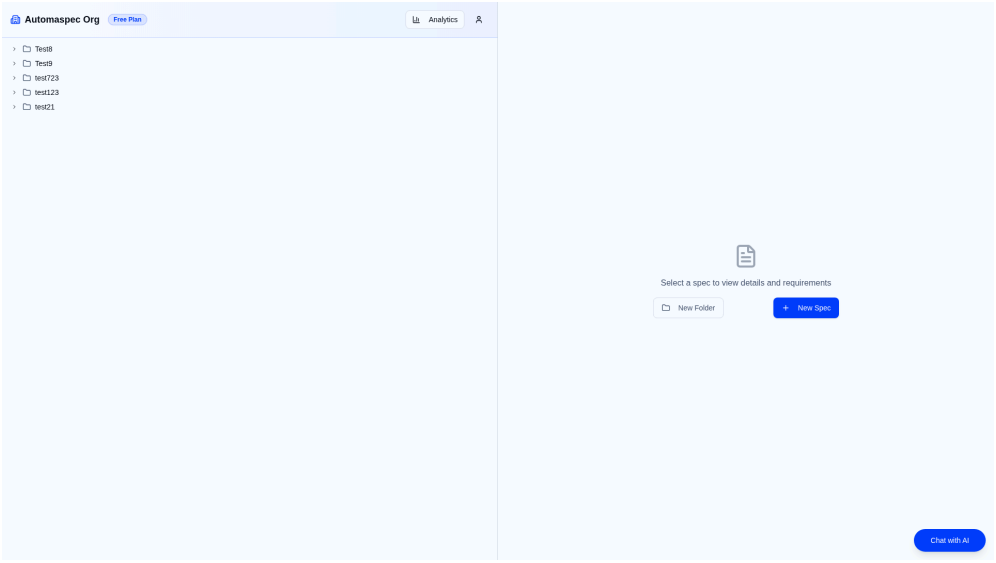
- Write requirements in user story format: “As a [user], I want [action], so that [benefit]”
- Keep requirements atomic (one testable thing per requirement)
- Include both positive and negative test scenarios

Feature 4: AI-Powered Test Generation

Overview

Generate Vitest test code from natural language requirements using AI. The system understands your context and produces ready-to-use test code.

How to Use



AI Panel

Step 1: Select a requirement and click **“Generate with AI”** - The AI panel opens on the right side - Your requirement context is automatically included

Step 2: Review the streaming response - Watch as test code generates in real-time - Code appears with syntax highlighting

Step 3: Accept, edit, or regenerate - Click **“Accept”** to save the generated code - Edit inline if adjustments are needed - Click **“Regenerate”** for a different approach

Expected Result: Valid Vitest test code attached to your requirement

Tips

- Provide detailed requirement descriptions for better results
- Review generated assertions for correctness
- Combine AI generation with manual refinement for best quality

---

Feature 5: Test Status Tracking

Overview

Track test execution status across your entire test suite. Status aggregates from individual tests up through requirements and specs.

How to Use

Step 1: View status indicators on specs and folders - Green: All tests passing - Red: One or more tests failing - Yellow: Tests pending or skipped

Step 2: Drill down to identify issues - Click on a spec to see requirement-level breakdown - Click on a requirement to see individual test results

Step 3: Update status manually or via CI/CD sync - Individual tests can be marked manually - Connect GitHub Actions for automatic updates

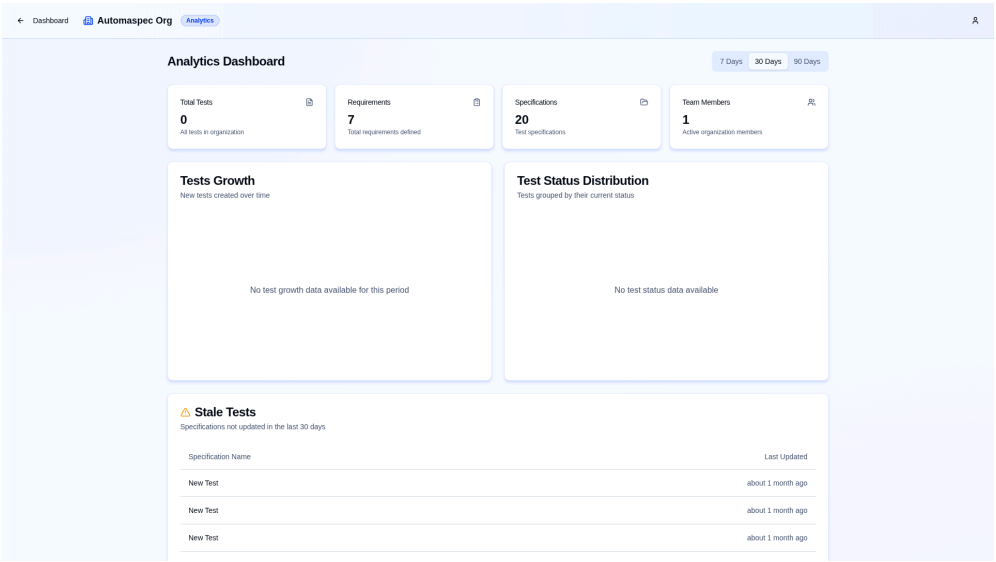
Expected Result: Real-time visibility into test health

Feature 6: Analytics Dashboard

Overview

Visualize your test coverage and health metrics with interactive charts and statistics.

How to Use



Analytics

Step 1: Navigate to **Analytics** in the top menu

Step 2: View key metrics: - Total specs, requirements, and tests - Pass/fail rates - Coverage trends over time

Step 3: Filter by time period - Select 7 days, 30 days, or custom range - Compare across periods

Expected Result: Data-driven insights into your testing efforts

---

Feature 7: Team Collaboration

Overview

Invite team members to your organization with role-based access control.

How to Use

Step 1: Go to **Profile → Organization Settings**

Step 2: Click **“Invite Members”** - Enter email addresses - Select role (Admin or Member)

Step 3: Manage existing members - View all members and their roles - Change roles or remove members as needed

Expected Result: Collaborative workspace with proper access controls

---

Feature 8: Dark/Light Theme

Overview

Switch between dark and light themes based on preference or system settings.

How to Use

Step 1: Click the theme toggle in the header (sun/moon icon)

Step 2: Choose your preference: - Light mode - Dark mode - System (follows OS preference)

Expected Result: Consistent theme across all pages

FAQ & Troubleshooting

Frequently Asked Questions

General

Q: What is Automaspec?

A: Automaspec is an AI-powered test specification and automation platform that helps QA engineers and developers organize test documentation, generate test code using AI, and track test coverage in a centralized location.

Q: Is Automaspec free to use?

A: Yes, Automaspec offers a free tier that includes all core features. Premium plans may be introduced in the future for advanced features and higher usage limits.

Q: Which testing frameworks are supported?

A: Currently, Automaspec generates test code for **Vitest**. Support for Jest, Playwright, and Cypress is planned for future releases.

Account & Access

Q: How do I reset my password?

A: Click “Forgot Password” on the login page. Enter your email address, and you’ll receive a password reset link. The link expires after 24 hours.

Q: Can I belong to multiple organizations?

A: Yes, you can be a member of multiple organizations. Use the organization switcher in the header to switch between them. Your role may differ across organizations.

Q: How do I leave an organization?

A: Go to Profile → Organization Settings → Leave Organization. Note: Organization owners cannot leave without transferring ownership first.

Features

Q: How does AI test generation work?

A: When you click “Generate with AI”, your requirement text and context are sent to our AI service (powered by OpenRouter/Gemini). The AI analyzes your requirement and generates Vitest test code that you can review, edit, and save.

Q: Is my test data secure?

A: Yes. All data is transmitted over HTTPS and stored in an encrypted database. We do not share your test specifications with third parties. AI generation requests are processed in real-time and not stored by the AI provider.

Q: Can I export my test code?

A: Yes. Each test includes a code viewer with a copy button. You can also export entire specs as files for integration into your codebase.

Troubleshooting

Common Issues

Problem	Possible Cause	Solution
Page won’t load	Cache issue	Clear browser cache and refresh
Can’t login	Wrong credentials	Use “Forgot Password” to reset
AI generation fails	Rate limit exceeded	Wait a few minutes and try again
Changes not saving	Network issue	Check internet connection; changes auto-retry
Slow performance	Large dataset	Try filtering or using search

Error Messages

Error Code/Message	Meaning	How to Fix
“Session expired”	Your login session has timed out	Click “Login” to sign in again
“Rate limit exceeded”	Too many requests in short time	Wait 1-2 minutes before retrying
“Permission denied”	You don’t have access to this resource	Contact your organization admin

Error Code/Message	Meaning	How to Fix
“AI service unavailable”	AI provider is temporarily down	Try again in a few minutes
“Invalid input”	Form validation failed	Check all required fields are filled

Browser-Specific Issues

Browser	Known Issue	Workaround
Safari	Cookies blocked in private mode	Use normal browsing mode
Firefox	Strict tracking protection blocks auth	Add site to exceptions
Mobile browsers	Keyboard covers input fields	Scroll down or use landscape mode

Getting Help

Self-Service Resources

- This Documentation
- API Documentation
- GitHub Repository

Contact Support

Channel	Best For
GitHub Issues	Bug reports, feature requests
Email	Account issues, security concerns

Reporting Bugs

When reporting a bug, please include:

1. **Steps to reproduce** - What actions lead to the issue?
2. **Expected behavior** - What should happen?
3. **Actual behavior** - What actually happens?
4. **Screenshots** - If applicable
5. **Browser/Device info** - Browser name, version, operating system

Submit bug reports at: GitHub Issues

Tips for Best Experience

1. **Use Chrome or Firefox** for best compatibility
2. **Keep browser updated** to the latest version
3. **Disable ad blockers** if experiencing login issues
4. **Use descriptive names** for folders and specs
5. **Write detailed requirements** for better AI generation
6. **Save work regularly** (though changes auto-save)

4. Retrospective

This section reflects on the project development process, lessons learned, and future improvements.

What Went Well ✅

Technical Successes

- **Type-safe full-stack**: Using oRPC with Zod provided end-to-end type safety from database to UI, catching errors at compile time
- **Modern React patterns**: React 19 with Next.js App Router delivered excellent performance with Server Components
- **Drizzle ORM**: Type-safe SQL with simple migrations made database work predictable and maintainable
- **AI integration**: Vercel AI SDK provided clean abstractions for LLM integration with streaming support
- **Tailwind CSS v4**: Utility-first styling accelerated UI development significantly

Process Successes

- **Iterative development**: Building features incrementally allowed for continuous testing and refinement
- **Code quality automation**: Lefthook pre-commit hooks enforced linting/formatting consistently
- **Component-based architecture**: Reusable UI primitives (via Shadcn/Radix) sped up development
- **Documentation as code**: Keeping docs in Markdown alongside code improved maintainability

Personal Achievements

- Deepened understanding of modern full-stack TypeScript development
- Gained practical experience with AI/LLM integration in production applications
- Improved skills in responsive design and accessibility
- Learned effective patterns for type-safe API design

What Didn’t Go As Planned ⚠️

Planned	Actual Outcome	Cause	Impact
Multi-framework support	Vitest only	Time constraints, complexity	Medium

Planned	Actual Outcome	Cause	Impact
Jira integration	Not implemented	Scope prioritization	Low
Advanced analytics	Basic metrics only	Feature prioritization	Low
Mobile native app	Responsive web only	Resource constraints	Low

Challenges Encountered

1. **AI Generation Quality**
   - Problem: Initial AI-generated tests had inconsistent quality
   - Impact: Required multiple iterations of prompt engineering
   - Resolution: Developed refined prompts with context injection and example patterns
2. **Database Migration Complexity**
   - Problem: Schema changes required careful migration handling with Turso
   - Impact: Some deployment delays during development
   - Resolution: Adopted stricter migration practices and testing workflow
3. **Authentication Edge Cases**
   - Problem: Better Auth organization plugin had undocumented behaviors
   - Impact: Extra time debugging session handling
   - Resolution: Deep-dived into source code and created custom middleware

Technical Debt & Known Issues

ID	Issue	Severity	Description	Potential Fix
TD-001	Large spec trees performance	Medium	Tree rendering slows with 100+ items	Implement virtualization (react-window)
TD-002	No offline support	Low	App requires active connection	Add PWA features and offline caching
TD-003	Limited test history	Low	Only current status stored	Add test run history table
TD-004	No undo/redo	Low	Accidental deletions are permanent	Implement soft delete and undo system

Code Quality Areas for Improvement

- Some components could be further decomposed for better reusability
- Test coverage could be expanded for edge cases in AI generation
- Error boundary handling could be more granular

Future Improvements (Backlog)

High Priority

1. **Multi-Framework Support**
   - Description: Add Jest, Playwright, and Cypress code generation
   - Value: Broader user adoption, framework flexibility
   - Effort: Medium (2-3 weeks per framework)
2. **CI/CD Result Sync**
   - Description: Automatic test status updates from GitHub Actions runs
   - Value: Real-time visibility without manual updates
   - Effort: Medium (webhook handling, parsing logic)

Medium Priority

3. **Version History**
   - Description: Track changes to specs and requirements over time
   - Value: Audit trail, ability to revert changes
4. **Bulk Operations**
   - Description: Select and move/copy multiple specs at once
   - Value: Improved efficiency for large reorganizations

Nice to Have

5. Custom report generation with PDF export
6. Integration with Jira/Linear for issue linking
7. Real-time collaborative editing (like Google Docs)
8. AI-powered test suggestions based on code changes

Lessons Learned

Technical Lessons

Lesson	Context	Application
Start with type safety	oRPC caught many bugs at compile time	Always choose typed solutions over dynamic
Invest in dev experience	Lefthook, Drizzle Studio saved hours	Good tooling pays dividends quickly
AI prompts need iteration	First attempts at test generation were poor	Budget time for prompt engineering
Mobile-first is worth it	Responsive design was straightforward	Always start with smallest viewport

Process Lessons

Lesson	Context	Application
Scope ruthlessly	Many nice-to-haves were cut	Define MVP early and stick to it
Document decisions	ADR format proved valuable	Record why, not just what
Test early	Late-discovered bugs were costly	Write tests alongside features

What Would Be Done Differently

Area	Current Approach	What Would Change	Why
Planning	Feature-based roadmap	More user story driven	Better alignment with actual needs
Database	Single SQLite database	Consider Postgres for scale	More features, better tooling
Testing	Manual + unit tests	More E2E tests earlier	Catch integration issues sooner
AI	Single provider	Multi-provider from start	Better fallback and cost optimization

Personal Growth

Skills Developed

Skill	Before Project	After Project
Next.js App Router	Beginner	Advanced
AI/LLM Integration	Beginner	Intermediate
Type-safe APIs (oRPC)	None	Intermediate
Drizzle ORM	Beginner	Advanced
Responsive Design	Intermediate	Advanced

Key Takeaways

1. **Type safety is non-negotiable** - The investment in typed APIs and database queries prevented countless bugs
2. **AI is a tool, not magic** - LLM integration requires careful prompt design and quality validation
3. **User feedback is invaluable** - Early testing revealed UX issues that weren't obvious during development
4. **Start simple, iterate** - MVP features shipped faster than planned "perfect" features

---

Retrospective completed: January 7, 2026

Glossary

Core Terms

Term	Definition
Test Spec	A test specification document containing one or more requirements and their associated tests
Requirement	A specific, testable condition or capability within a test spec
Test	An individual test case implementation that verifies a requirement
Folder	A hierarchical container for organizing test specs (supports unlimited nesting)
Organization	A multi-tenant workspace that isolates test data between teams

Technical Terms

Term	Definition
AI SDK	Vercel’s unified SDK for interacting with multiple LLM providers
Better Auth	Open-source authentication solution used for user management
Drizzle ORM	TypeScript ORM for type-safe SQL database queries
LLM	Large Language Model (e.g., GPT-4, Claude, Gemini)
oRPC	Type-safe RPC framework for Next.js API routes
Turso	Distributed SQLite database platform (libSQL)
Vitest	Fast unit testing framework for Vite-based projects

Acronyms

Acronym	Full Form	Description
API	Application Programming Interface	Standard interface for software communication
CI/CD	Continuous Integration / Continuous Deployment	Automated build, test, and deployment pipeline
CRUD	Create, Read, Update, Delete	Basic data operations
ER	Entity Relationship	Database modeling approach

Acronym	Full Form	Description
FAQ	Frequently Asked Questions	Common user queries
GDPR	General Data Protection Regulation	EU data privacy law
JWT	JSON Web Token	Token-based authentication standard
MVP	Minimum Viable Product	Initial product version with core features
ORM	Object-Relational Mapping	Database abstraction layer
RBAC	Role-Based Access Control	Permission system based on user roles
REST	Representational State Transfer	API architectural style
RPC	Remote Procedure Call	API communication pattern
SPA	Single Page Application	Web app architecture
SSR	Server-Side Rendering	HTML generated on server
TDD	Test-Driven Development	Development methodology
UI/UX	User Interface / User Experience	Design disciplines
WCAG	Web Content Accessibility Guidelines	Accessibility standards

User Roles

Role	Definition
Owner	Full control over organization, can delete org, manage all members
Admin	Can manage specs, folders, and invite members (cannot delete org)
Member	Can view and edit test specs and requirements

Test Statuses

Status	Definition
passed	Test executed successfully
failed	Test execution resulted in an error or assertion failure
pending	Test is defined but not yet implemented or run
skipped	Test was intentionally skipped during execution
missing	Expected test does not exist

Domain-Specific Terms

Testing

Term	Definition
Test Coverage	Percentage of code or requirements covered by tests
Test Suite	Collection of related tests
Assertion	Statement that verifies expected behavior
Mocking	Simulating external dependencies in tests
E2E Testing	End-to-end testing of complete user flows

AI Integration

Term	Definition
Prompt Engineering	Designing effective prompts for LLM interactions
Context Window	Maximum tokens an LLM can process in one request
Token	Unit of text processed by LLMs (roughly 4 characters)
Streaming	Receiving LLM responses incrementally as generated
RAG	Retrieval-Augmented Generation (extending LLM with external data)

API Reference

This document provides a reference for the application’s API endpoints, which are defined using ORPC contracts. The API is Type-Safe and validated using Zod schemas.

Account

> Contracts for user account data management.

Export Account Data

- **Method:** `GET`
- **Path:** `/account/{userId}`
- **Summary:** Export account data
- **Description:** Export the current user account data including profile and organization memberships.
- **Input:** `{ userId: string }`
- **Output:** `{ user: { id, name, email, ... }, memberships: [{ organizationId, role, ... }] }`

Delete Account

- **Method:** `DELETE`
- **Path:** `/account/{userId}`
- **Summary:** Delete account
- **Description:** Permanently delete the current user account.
- **Input:** `{ userId: string }`
- **Output:** `{ success: boolean }`

AI

> AI assistant integration endpoints.

Chat with AI

- **Method:** `POST`
- **Path:** `/ai/chat`
- **Summary:** Chat with AI
- **Description:** Send chat messages to the AI assistant and receive a streamed or blocked response.
- **Input:** `AIChatRequestSchema` (Message history, model params)
- **Output:** `AIChatResponseSchema` (AI Generation)

Analytics

> Organization metrics and statistics.

Get Metrics

- **Method:** `GET`
- **Path:** `/analytics/metrics`
- **Summary:** Get analytics metrics
- **Description:** detailed analytics metrics for the active organization dashboard.
- **Input:** `AnalyticsMetricsInputSchema`
- **Output:** `AnalyticsMetricsOutputSchema` (Counts, graphs, trends)

Tests

> Core domain logic for Test Management (Folders, Specs, Requirements, Tests).

Test Folders

Get Folder - `GET /test-folders/{id}` - Get a single test folder by ID.

List Folders - `GET /test-folders` - List folders in the organization, optionally filtered by `parentFolderId`.

Get Children - `GET /test-folders/{folderId}/children` - Get direct children (sub-folders or specs) of a specific folder. Supports optional `depth` parameter.

Find by Name - `GET /test-folders/find` - Find a folder by its exact name.

Create/Update Folder - `POST /test-folders/{id}` - Upsert a test folder definition.

Edit Folder - `PATCH /test-folders/{id}` - Partially update folder fields (name, description, order).

Delete Folder - `DELETE /test-folders/{id}` - Delete a folder (and cascade delete contents).

Test Specs

Get Spec - `GET /test-specs/{id}` - Get a single test spec file definition.

List Specs - `GET /test-specs` - List specs, optionally filtered by `folderId`.

Create/Update Spec - `PUT /test-specs/{id}` - Upsert a test spec reference.

Edit Spec - `PATCH /test-specs/{id}` - Partially update spec fields.

Delete Spec - `DELETE /test-specs/{id}` - Delete a spec reference.

Test Requirements

List Requirements - `GET /test-requirements` - List requirements, filtered by `specId`.

Create/Update Requirement - `PUT /test-requirements/{id}` - Upsert a business requirement.

Edit Requirement - `PATCH /test-requirements/{id}` - Partially update requirement details.

Replace Spec Requirements - `PUT /test-specs/{specId}/requirements` - Bulk replace all requirements for a specific spec (Sync mode).

Delete Requirement - `DELETE /test-requirements/{id}` - Remove a specific requirement.

Tests (Executions)

List Tests - `GET /tests` - List test executions, filtered by `requirementId`.

Create/Update Test - `PUT /tests/{id}` - Upsert a test execution record.

Edit Test - `PATCH /tests/{id}` - Update test status or code snippet.

Delete Test - `DELETE /tests/{id}` - Remove a test execution record.

Sync Report - `POST /tests/sync-report` - Upload a Vitest/Playwright JSON report to automatically update test statuses.

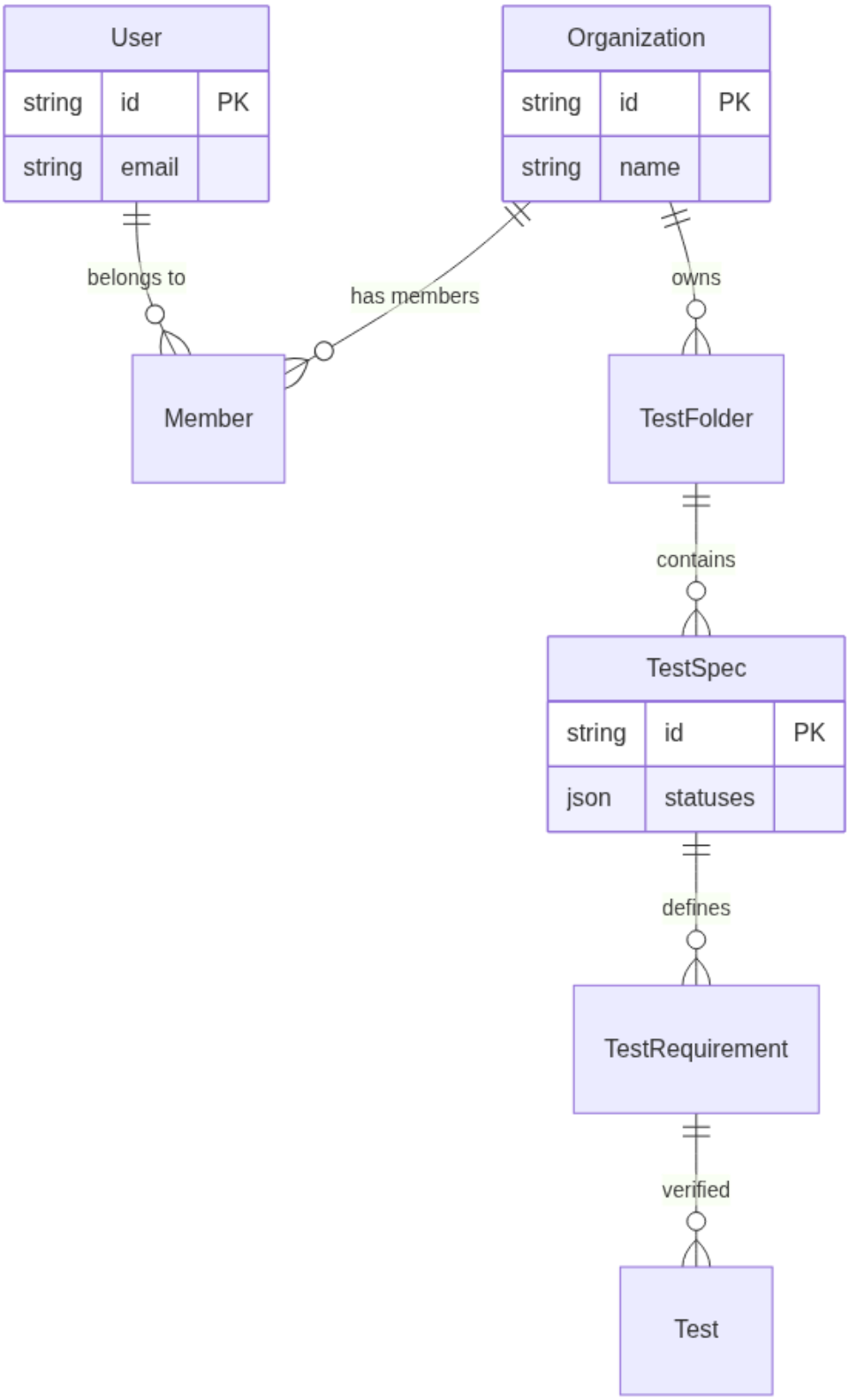
Get Report - `GET /tests/report` - Retrieve the latest stored test report.

Database Schema

Overview

The application uses **SQLite** (via LibSQL/Turso) as its relational database, with **Drizzle ORM** for type-safe schema definition and query building. The schema is modularized into two main domains: **Authentication/Organization** and **Testing Core**.

Entity Relationship Diagram



Database Schema

Tables Reference

Authentication & Organization (`db/schema/auth.ts`)

`user`

Core user entity. - `id` (text, PK): Unique identifier - `name` (text): Display name - `email` (text, unique): User email - `emailVerified` (boolean): Verification status - `image` (text): Avatar URL - `createdAt`, `updatedAt` (timestamp)

`session`

Active user sessions. - `id` (text, PK) - `userId` (text, FK -> user.id): Owner - `token` (text, unique): Session token - `expiresAt` (timestamp): Expiration time - `ipAddress`, `userAgent` (text): Audit info - `activeOrganizationId` (text): Context for current session

`account`

OAuth accounts linked to users (Google, GitHub, etc.). - `id` (text, PK) - `userId` (text, FK -> user.id) - `providerId` (text): e.g., “google” - `accountId` (text): Provider-specific ID - `accessToken`, `refreshToken` (text): OAuth tokens

`organization`

Tenancy unit. - `id` (text, PK) - `name` (text) - `slug` (text, unique): URL-friendly identifier - `plan` (text): Subscription plan (default: ‘free’) - `metadata` (text): JSON metadata

`member`

Linking table between Users and Organizations. - `id` (text, PK) - `organizationId` (text, FK -> organization.id) - `userId` (text, FK -> user.id) - `role` (text): ‘owner’, ‘member’, etc.

`invitation`

Pending organization invites. - `id` (text, PK) - `email` (text): Invitee email - `organizationId` (text, FK -> organization.id) - `inviterId` (text, FK -> user.id) - `status` (text): ‘pending’, ‘accepted’ - `role` (text): Role to be assigned

`apiKey`

API Access keys for external integrations. - `id` (text, PK) - `userId` (text, FK -> user.id) - `key` (text): Hashed key - `permissions` (text): Scopes - `rateLimitEnabled` (boolean)

Testing Core (`db/schema/tests.ts`)

`test_folder`

Hierarchical organization for test specs. - `id` (text, PK) - `name` (text) - `organizationId` (text, FK -> organization.id) - `parentFolderId` (text, nullable): For subfolders - `order` (integer): Sorting order

`test_spec`

A file or module containing requirements. - `id` (text, PK) - `name` (text) - `fileName` (text): Source file link - `folderId` (text, FK -> test_folder.id) - `organizationId` (text, FK -> organization.id) - `statuses` (json): Aggregated stats (passed, failed counts) - `numberOfTests` (integer)

`test_requirement`

Specific business requirement to be tested. - `id` (text, PK) - `name` (text): Requirement title - `description` (text) - `specId` (text, FK -> test_spec.id) - `order` (integer)

`test`

Individual test execution implementation. - `id` (text, PK) - `requirementId` (text, FK -> test_requirement.id) - `status` (text): Test result status - `framework` (text): ‘playwright’, ‘jest’, etc. - `code` (text): Test code snippet