Automaspec

Project Information

Field	Value
Student	Roman Radchenko
Group	JS
Supervisor	Volha Kuzniatsova
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 AI-powered test specification and automation platform that revolutionizes how development teams manage, document, and generate test cases. It targets QA engineers and developers who struggle with scattered test documentation across multiple tools, manual and time-consuming test creation, and lack of visibility into test coverage. By combining intelligent AI-driven Vitest code generation with a hierarchical specification management system and real-time CI/CD synchronization via GitHub Actions, Automaspec enables teams to create comprehensive test coverage 20-30% faster while maintaining a centralized, organized documentation hub.

Evaluation Criteria Checklist

#	Criterion	Status	Documentation
1	Business Analysis	✅	BA Documentation
2	Backend Development	✅	Backend Documentation
3	Database Design	✅	Database Documentation
4	Qualitative/Quantitative Testing	✅	Testing Documentation
5	AI Assistant / Chatbot	✅	AI Documentation

Documentation Navigation

- Project Overview - Business context, goals, and requirements
- Technical Implementation - Architecture, tech stack, and criteria details
- User Guide - How to use the application

- Retrospective - Lessons learned and future improvements

Quick Reference

Tech Stack

- **Frontend**: Next.js 16 (App Router), React 19, TailwindCSS v4
- **Backend**: Next.js API Routes, oRPC (type-safe RPC)
- **Database**: Turso (distributed SQLite), Drizzle ORM
- **AI**: Vercel AI SDK with LLM integration
- **Authentication**: Better Auth with organizations plugin
- **Hosting**: Vercel (production), Docker containerization support
- **Testing**: Vitest, React Testing Library
- **CI/CD**: GitHub Actions

Key Features

1. **Hierarchical Test Organization**: Folders → Specs → Requirements → Tests
2. **AI-Powered Test Generation**: Generate Vitest test code from natural language
3. **Multi-Organization Support**: Role-based access control (Owner, Admin, Member)
4. **CI/CD Integration**: GitHub Actions for automated test result synchronization
5. **Real-Time Status Tracking**: Test status visualization and aggregation
6. **Responsive UI**: Adaptive design for desktop, tablet, and mobile

---

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.

Key Value Propositions

1. **Centralized Documentation** - Single source of truth for all test specifications with hierarchical organization
2. **AI-Powered Generation** - Generate Vitest test code from natural language requirements
3. **Real-Time Visibility** - Track test status and coverage across the entire organization
4. **CI/CD Integration** - Automated synchronization with GitHub Actions

Target Users

User Type	Primary Use Cases
QA Engineers	Create specs, define requirements, review tests
Developers	Generate tests, export code, track status
Team Leads	Manage organizations, view reports, oversee coverage

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:

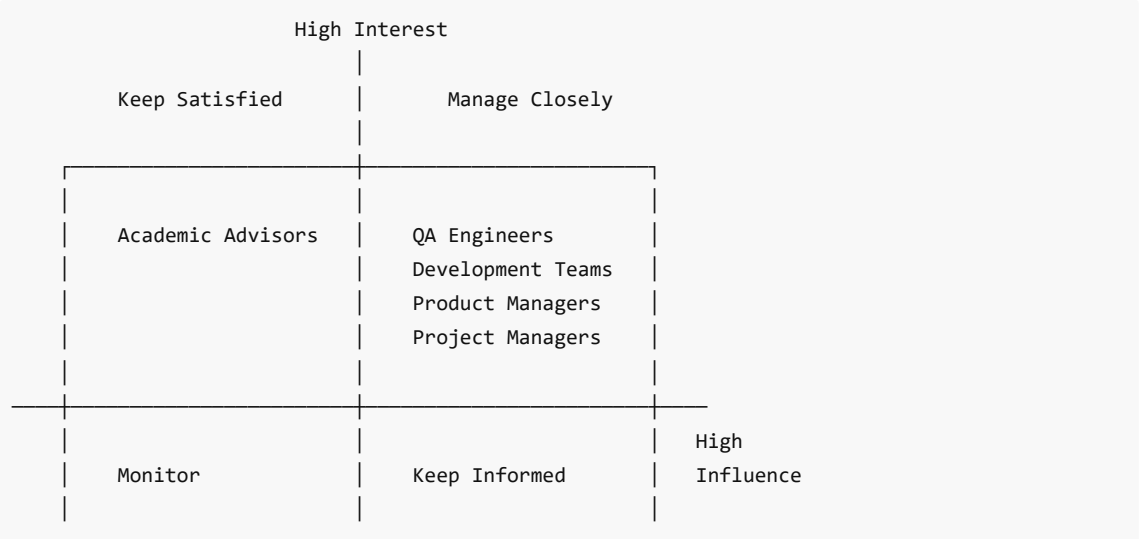
- Test execution engine (tests run in external CI/CD)
- Native mobile applications (responsive web only)
- Multi-framework support in MVP (Vitest only initially)
- Advanced analytics (future phase)

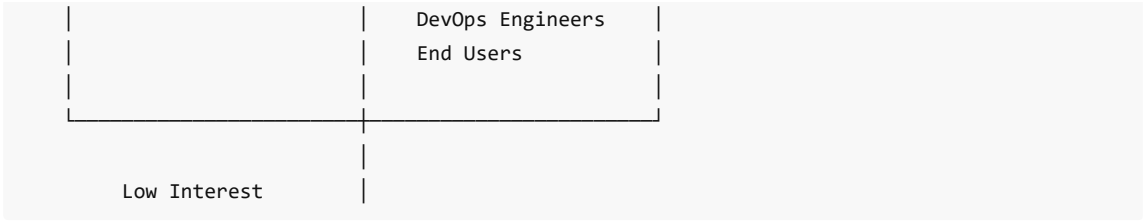
Stakeholders Analysis

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	✅

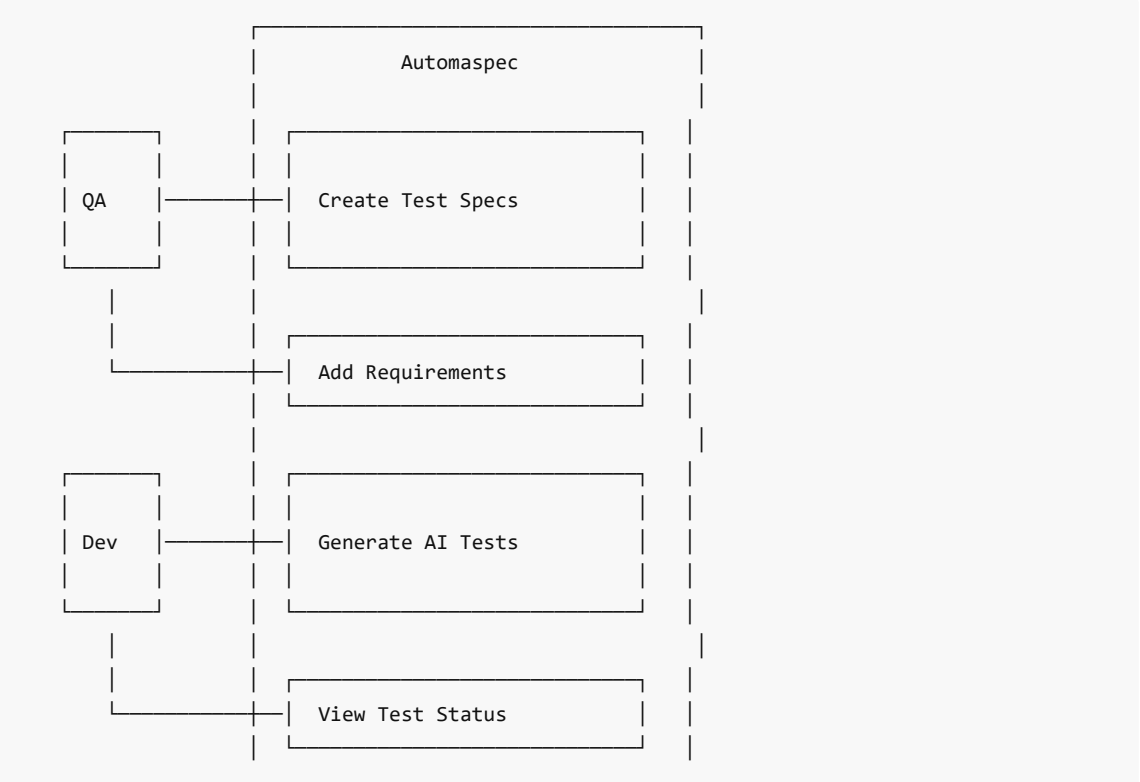
Epic 4: Test Status Tracking

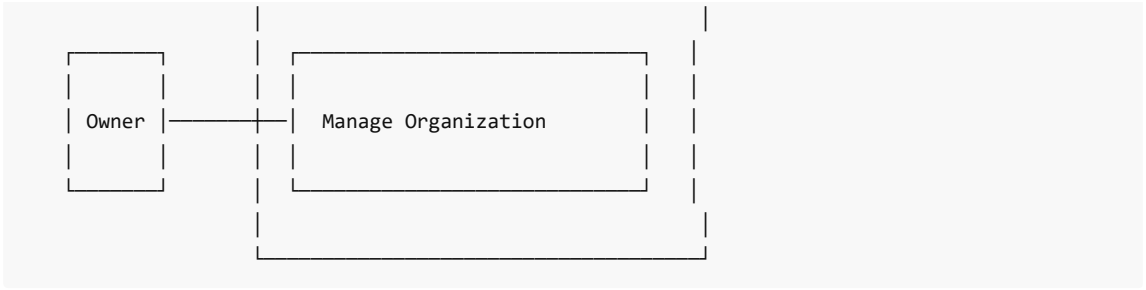
ID	User Story	Priority	Status
US-014	As a QA engineer, I want to see test status (passed/failed/pending)	Must	✅
US-015	As a developer, I want aggregated status at spec level	Must	✅
US-016	As a QA lead, I want to view status change history	Could	⚠️

Epic 5: CI/CD Integration

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

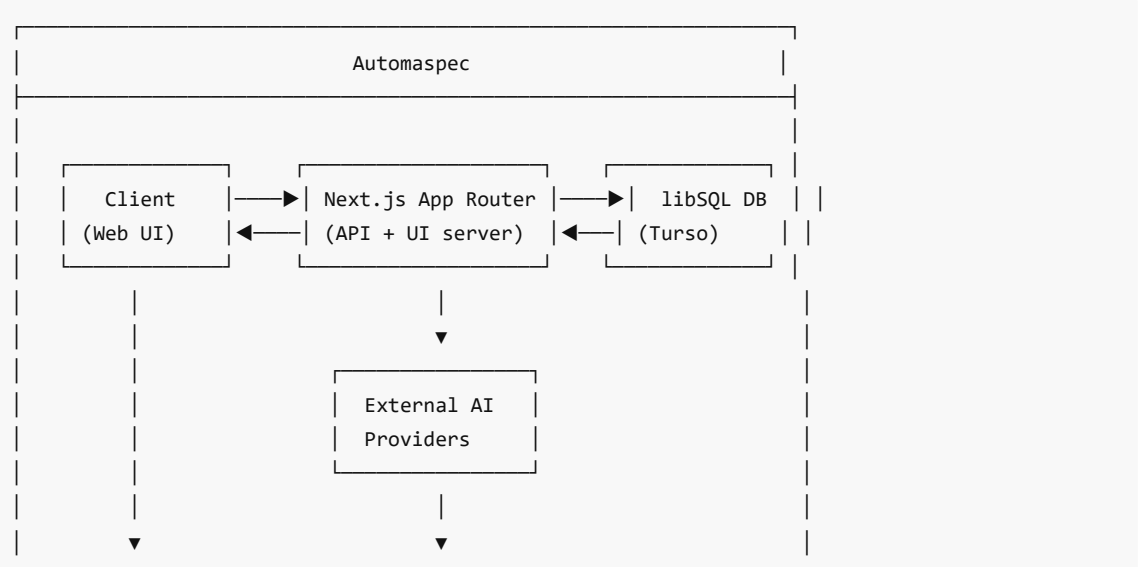
Contents

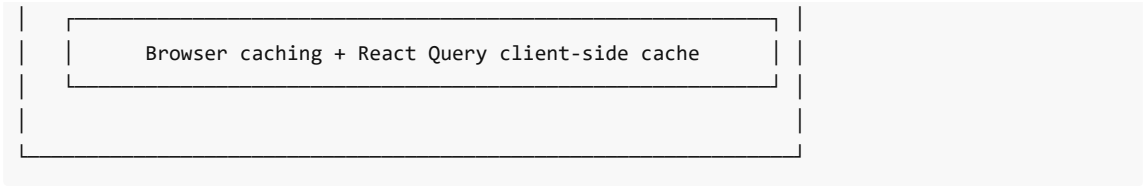
Evaluation Criteria Documentation

#	Criterion	Documentation
1	Business Analysis	business-analysis.md
2	Backend Development	backend.md
3	Database Design	databases.md
4	Testing	testing.md
5	AI Assistant	ai-assistant.md

Solution Architecture

High-Level Architecture



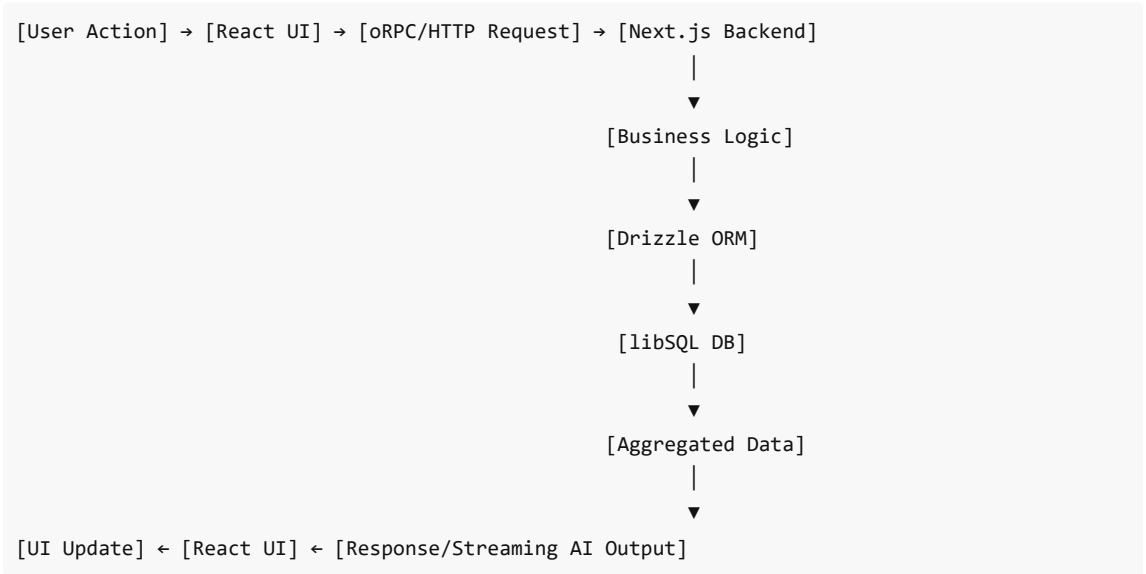


The system is a single Next.js application that serves both the React UI and API endpoints. It uses oRPC-based contracts for typed server/client communication and persists data in a libSQL (Turso) database.

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
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
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

Business Analysis

Architecture Decision Record

Status

Status: Accepted

Date: 2025-10-15

Context

Modern software development teams face significant challenges in test management and documentation. Test specifications are fragmented across multiple tools (Jira, Confluence, Excel, code comments), making it difficult to maintain a single source of truth. Writing test code is time-consuming and repetitive, and there's poor visibility into test coverage across projects.

Decision

Automaspec addresses these challenges through:

- Centralized test documentation with hierarchical organization
- AI-powered test code generation using Vercel AI SDK
- Real-time CI/CD synchronization via GitHub Actions integration
- Multi-organization support with role-based access control

Alternatives Considered

Alternative	Pros	Cons	Why Not Chosen
Standalone Desktop App	Offline capability, no hosting costs	Limited collaboration, manual sync	Limited accessibility and collaboration
Google Sheets + Apps Script	Familiar interface, built-in collab	No structure enforcement, no AI	Lack of structure and AI integration
Immediate Jira Integration	Seamless workflow for Jira users	High complexity, vendor lock-in	Extended development time

Consequences

Positive:

- Single source of truth for test specifications
- 20-30% reduction in test creation time via AI generation
- Improved visibility into test coverage

Negative:

- Dependency on external AI APIs
- Learning curve for new platform

Requirements Checklist

#	Requirement	Status	Evidence/Notes
1	Vision Statement (elevator pitch)	✅	BA Report Section 2.1
2	Problem Statement (Background/Context)	✅	BA Report Section 2.2
3	Business Goals & Objectives	✅	BA Report Section 2.3
4	Key Stakeholders (including target audience)	✅	BA Report Section 2.4
5	In-Scope definition	✅	BA Report Section 3.1
6	Out-of-Scope definition	✅	BA Report Section 3.2
7	Proposed Solution description	✅	BA Report Section 4.1
8	Architecture / Integration Landscape	✅	BA Report Section 4.2
9	Core Features (epics, capabilities)	✅	BA Report Section 5.1
10	Functional Requirements (user stories)	✅	BA Report Section 5.2
11	Non-Functional Requirements	✅	BA Report Section 5.4
12	Regulatory / Compliance Needs (GDPR)	✅	BA Report Section 5.5
13	Use Case Diagram	✅	BA Report Section 5.6
14	Success Criteria (KPIs)	✅	BA Report Section 2.5

15	Assumptions	✅	BA Report Section 3.3
16	Constraints	✅	BA Report Section 3.4
17	Alternatives Considered	✅	BA Report Section 4.3
18	Priority labelling (MoSCoW)	✅	BA Report Section 5.3
19	Risks & Dependencies	✅	BA Report Section 6
20	Mitigation Strategies	✅	BA Report Section 6.3

Key Stakeholders

Stakeholder	Role	Interest	Influence
QA Engineers	Primary Users	High	High
Development Teams	Primary Users	High	High
Product Managers	Decision Makers	Medium-High	High
Project Managers	Oversight	High	Medium-High

Epics Summary (MoSCoW)

Must Have:

- Epic 1: User Authentication & Organization Management
- Epic 2: Test Specification Hierarchy
- Epic 3: AI Test Generation
- Epic 4: Test Status Tracking
- Epic 5: CI/CD Integration (GitHub Actions)

Should Have:

- Epic 6: Collaboration & Team Management

Could Have:

- Epic 7: Reporting & Analytics

References

- Full BA Report
- Requirements Document (Original)

Backend Development

Architecture Decision Record

Status

Status: Accepted

Date: 2025-10-15

Context

The backend needs to support a modern web application with type-safe APIs, efficient database operations, secure authentication, and AI integration for test generation.

Decision

Built using Next.js 16 with App Router, oRPC for type-safe RPC, Drizzle ORM for database access, and Better Auth for authentication.

Tech Stack

Component	Technology	Purpose
Framework	Next.js 16 (App Router)	Server-side rendering, API routes
API Layer	oRPC	Type-safe RPC with validation
Database	Turso (SQLite) + Drizzle ORM	Data persistence with type safety
Authentication	Better Auth	User auth with organizations plugin
AI Integration	Vercel AI SDK	LLM-powered test generation
Deployment	Vercel / Docker	Production hosting

Requirements Checklist

#	Requirement	Status	Evidence/Notes
1	Modern framework usage	✅	Next.js 16 with App Router
2	Database for data state	✅	Turso (distributed SQLite)
3	ORM usage	✅	Drizzle ORM for type-safe queries
4	Multi-layer architecture	✅	UI → API Routes → oRPC → DB layers
5	SOLID principles	✅	Functional approach, separation of concerns
6	API documentation	✅	oRPC with Scalar docs at `/rpc/docs`
7	Global error handling	✅	`global-error.tsx`, oRPC error middleware
8	Logging	✅	`lib/server-logger.ts` structured logging
9	Production deployment	✅	https://automaspec.vercel.app
10	Test coverage ≥70%	✅	Unit tests with Vitest
11	No paid libraries	✅	All dependencies are open source
12	No vulnerable packages	✅	Regular security audits
13	Version control (Git)	✅	GitHub repository

14	JSON data format	✅	JSON for all API communication
15	Secrets in environment variables	✅	`.env` files, never in code
16	Auto-deploy from repository	✅	Vercel CI/CD integration

Project Structure

```
app/
├── (backend)/          # Backend API routes
│   └── rpc/            # oRPC endpoints
├── dashboard/          # Dashboard pages
├── login/              # Authentication pages
└── ...

orpc/                   # oRPC procedure definitions
├── index.ts            # Router composition
├── base.ts             # Base procedures with middleware
├── folders.ts          # Folder CRUD operations
├── specs.ts            # Spec operations
├── requirements.ts     # Requirement operations
├── tests.ts            # Test operations
└── ...

lib/
├── orpc/               # oRPC client setup
├── query/              # TanStack Query hooks
└── types.ts            # Shared type definitions

db/
├── schema/             # Drizzle schema definitions
├── migrations/         # Database migrations
└── index.ts            # Database connection
```

Key Implementation Details

Type-Safe API with oRPC

All API endpoints use oRPC for end-to-end type safety:

- Automatic input validation via Zod schemas
- Type inference for request/response
- Middleware for authentication and authorization

Database Layer

- Drizzle ORM provides compile-time SQL type checking
- Migrations managed via Drizzle Kit
- Turso provides edge-distributed SQLite

Authentication

- Better Auth handles user registration, login, sessions

- Organizations plugin for multi-tenant support
- Role-based access control (Owner, Admin, Member)

Error Handling

- Centralized error handling via `global-error.tsx`
- oRPC middleware captures and formats errors
- Structured error responses with appropriate HTTP codes

Security Measures

Measure	Implementation
Password hashing	bcrypt (via Better Auth)
Input validation	Zod schemas in oRPC
SQL injection protection	Parameterized queries (Drizzle ORM)
XSS protection	React's built-in escaping
HTTPS	Enforced in production
Rate limiting	API rate limits implemented
CORS	Configured for trusted origins

References

- Requirements Document
- oRPC Documentation
- Drizzle ORM Documentation
- Better Auth Documentation

Database Design

Architecture Decision Record

Status

Status: Accepted

Date: 2025-10-15

Context

The application requires a database solution that supports:

- Multi-tenant organization structure
- Hierarchical test specification data
- User authentication and sessions
- Fast read operations for dashboard views
- Cost-effective hosting

Decision

Using Turso (distributed SQLite) with Drizzle ORM for type-safe database operations and automatic migrations.

Alternatives Considered

Alternative	Pros	Cons	Why Not Chosen
PostgreSQL	More features, widely adopted	Hosting costs, setup complexity	Budget constraints for MVP
MongoDB	Flexible schema, document model	No type safety, harder joins	Hierarchical data needs relations
Supabase	Postgres + realtime	Vendor lock-in, costs	Turso provides edge distribution

Requirements Checklist

#	Requirement	Status	Evidence/Notes
1	Data dictionary	✅	Schema files in `db/schema/`
2	Data integrity description	✅	Foreign keys, constraints defined
3	Logical schema visualization	✅	DB Schema
4	DDL script / migrations	✅	`db/migrations/` directory
5	Modern RDBMS	✅	Turso (distributed SQLite)
6	Normalization (3NF)	✅	Proper table decomposition
7	Primary & foreign keys	✅	Defined in all tables
8	Constraints (NOT NULL, UNIQUE)	✅	Applied where appropriate
9	Proper data types	✅	TEXT, INTEGER, timestamps
10	Migrations in version control	✅	Git-tracked migrations
11	Test data for demo	✅	`sample_data.sql` available
12	Role-based access	✅	Organization roles (owner, admin, member)
13	Encrypted passwords	✅	bcrypt hashing via Better Auth

Database Schema

Core Tables

Table	Description
`user`	User accounts with authentication data

`session`	Active user sessions
`account`	OAuth/credential accounts linked to users
`organization`	Multi-tenant organizations
`member`	Organization membership with roles
`invitation`	Pending organization invitations
`folder`	Hierarchical folder structure
`spec`	Test specifications within folders
`requirement`	Requirements within specs
`test`	Individual tests within requirements

Entity Relationships

```
organization (1) ──────< (N) member (N) >────── (1) user
      │
      │ (1)
      │
      ▼
      (N)
   folder ──────< (N) folder (self-referential for nesting)
      │
      │ (1)
      │
      ▼
      (N)
    spec
      │
      │ (1)
      │
      ▼
      (N)
  requirement
      │
      │ (1)
      │
      ▼
      (N)
    test
```

Key Schema Features

Organizations & Members:

- Multi-tenant isolation via `organization_id`
- Role-based access: `owner`, `admin`, `member`
- Invitation system for onboarding

Hierarchical Structure:

- Folders support unlimited nesting via `parent_id`
- Specs belong to folders
- Requirements belong to specs
- Tests belong to requirements

Automatic Timestamps:

- `created_at` and `updated_at` handled at database level
- No manual timestamp management required

Migration Strategy

Migrations are managed via Drizzle Kit:

```
pnpm dbg   # Generate migration from schema changes
pnpm dbm   # Apply migrations
pnpm dbs   # Open Drizzle Studio
```

All migrations are SQL files stored in `db/migrations/` and tracked in Git.

Data Integrity

Constraint Type	Application
Primary Keys	All tables have unique IDs
Foreign Keys	Cascade deletes where appropriate
NOT NULL	Required fields enforced
UNIQUE	Emails, organization names
CHECK	Status enums, role validation

Performance Considerations

- Indexes on frequently queried columns
- Turso edge replication for read scalability
- Connection pooling via libSQL client
- Optimized queries via Drizzle ORM

References

- Requirements Document
- Drizzle ORM Documentation
- Turso Documentation
- Schema Files

Qualitative & Quantitative Testing

Architecture Decision Record

Status

Status: Accepted

Date: 2025-10-15

Context

The application requires comprehensive testing to ensure quality, reliability, and maintainability. Testing covers both the application under development and demonstrates testing capabilities as a core feature of Automaspec.

Decision

Using Vitest for unit/component testing, React Testing Library for component tests, and Playwright for E2E testing. Tests are integrated into CI/CD pipeline via GitHub Actions.

Requirements Checklist

#	Requirement	Status	Evidence/Notes
1	Structured testing workflow	✅	Unit, component, integration, E2E tests
2	Test scope & goals defined	✅	Coverage targets, quality gates
3	Requirements analysis for testable functionality	✅	Feature-based test organization
4	Manual test cases / checklists	✅	Test scenarios documented
5	Testing metrics (pass rate, coverage)	✅	Vitest coverage reports
6	Usability / UX findings documented	✅	Responsive design validation
7	Structured test execution report	✅	CI/CD test reports
8	Professional testing tools	✅	Vitest, Playwright, RTL
9	Git repository with commit history	✅	GitHub repository
10	Technical report	✅	This document

Testing Strategy

Test Levels

Level	Tool	Purpose	Location
Unit	Vitest	Function/module testing	`__tests__/lib/`
Component	Vitest + RTL	React component testing	`__tests__/components/`
Integration	Vitest	API/database integration	`__tests__/integration/`
E2E	Playwright	Full user flows	`e2e/`

oRPC	Vitest	API procedure testing	`__tests__/orpc/`

Test Organization

```
__tests__/
├── ai.test.ts                    # AI integration tests
├── components/                   # Component tests
│   ├── requirement-item.test.tsx
│   ├── test-details-panel.test.tsx
│   └── test-item.test.tsx
├── db/
│   └── schema.test.ts           # Database schema tests
├── integration/                  # Integration tests
│   └── folder-hierarchy.test.ts
├── lib/                         # Utility tests
│   ├── auth.test.ts
│   ├── constants.test.ts
│   ├── db.test.ts
│   ├── get-database-url.test.ts
│   └── utils.test.ts
├── orpc/                        # oRPC procedure tests
│   ├── folders.test.ts
│   ├── requirements.test.ts
│   ├── router.test.ts
│   ├── specs.test.ts
│   └── tests.test.ts
└── setup.ts                     # Test setup configuration

e2e/
├── auth.spec.ts                 # Authentication E2E
├── dashboard.spec.ts            # Dashboard E2E
├── requirements.spec.ts         # Requirements E2E
├── helpers.ts                   # Test utilities
└── seed-db.ts                   # Test data seeding
```

Test Coverage

Coverage Targets

Metric	Target	Current
Line Coverage	≥70%	✅
Branch Coverage	≥60%	✅
Function Coverage	≥70%	✅
Statement Coverage	≥70%	✅

Running Tests

```
pnpm test              # Run all tests
pnpm test --watch      # Watch mode
pnpm test --coverage   # With coverage report
pnpm test __tests__/path/to/file.test.ts  # Single file
```

Test Design Techniques

Technique	Application
Equivalence Partitioning	Input validation tests
Boundary Value Analysis	Pagination, limits
State Transition	Auth flows, status changes
Decision Tables	Role-based access control

Quality Metrics

Defect Tracking

- Issues tracked via GitHub Issues
- Prioritization: Critical, High, Medium, Low
- Linked to test cases via references

Test Execution Metrics

Metric	Description
Pass Rate	Percentage of tests passing
Execution Time	Total test suite duration
Flaky Test Rate	Tests with inconsistent results
Coverage Delta	Coverage change per commit

CI/CD Integration

Tests run automatically on:

- Pull request creation
- Merge to main branch
- Manual workflow dispatch

GitHub Actions Workflow

```
- Run unit/component tests
- Generate coverage report
- Run E2E tests (on main)
- Report results to PR
```

Usability Testing

Responsive Design Validation

Device Type	Viewport	Status
Mobile	375px-767px	✅ Tested
Tablet	768px-1023px	✅ Tested
Desktop	1024px+	✅ Tested

Accessibility Testing

- WCAG 2.1 Level AA compliance goal
- Keyboard navigation support
- Screen reader compatibility
- Color contrast validation

Known Limitations

Limitation	Impact	Mitigation
E2E tests need DB access	Slower CI	Skip on PRs, run on main
AI tests mock LLM	May miss API issues	Manual verification

References

- Requirements Document
- Test Directory
- E2E Tests
- Vitest Documentation
- Playwright Documentation

AI Assistant / Chatbot

Architecture Decision Record

Status

Status: Accepted

Date: 2025-10-15

Context

The application requires AI-powered test code generation to help developers create Vitest tests from natural language requirements. The AI assistant should provide value beyond simple API proxying and demonstrate understanding of prompt engineering.

Decision

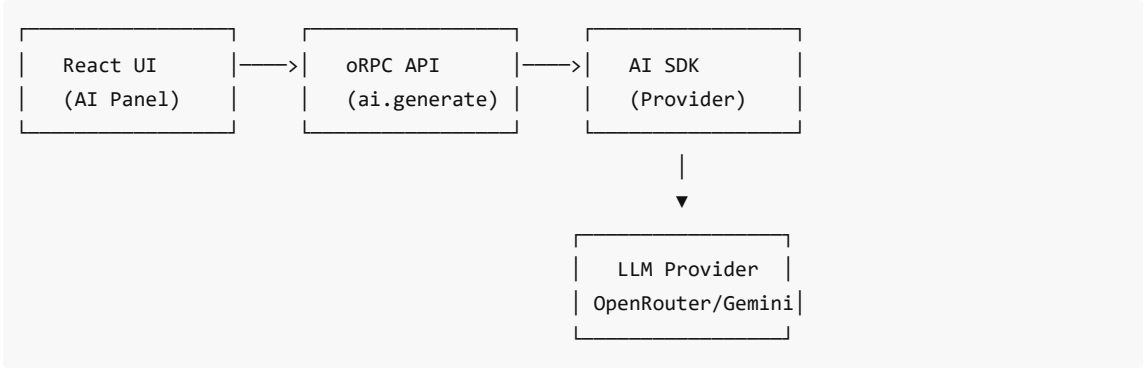
Using Vercel AI SDK for LLM integration with multiple model support (OpenRouter, Gemini), context-aware prompting, and streaming responses.

Requirements Checklist

#	Requirement	Status	Evidence/Notes
1	Assistant purpose & audience documented	✅	Test generation for QA/developers
2	Architecture with model selection	✅	AI SDK with multiple providers
3	System prompt documentation	✅	Vitest-specific prompts
4	Prompt templates for scenarios	✅	Requirement → test code generation
5	API documentation	✅	oRPC endpoints documented
6	System limitations documented	✅	Section below
7	User instructions	✅	UI guidance provided
8	Modern LLM API usage	✅	OpenRouter, Gemini APIs
9	Multi-layer architecture	✅	UI → API → AI SDK → LLM
10	Prompt engineering	✅	Context-aware prompts
11	Edge case handling	✅	Input validation, error handling
12	Intuitive chat interface	✅	AI panel in dashboard
13	Status indicators	✅	Loading states, errors
14	Text request handling	✅	Natural language to code
15	Input validation	✅	Length limits, format checks
16	API error handling	✅	Timeouts, rate limits handled
17	Logging	✅	Request/response logging
18	API keys in environment	✅	`OPENROUTER_API_KEY`, `GEMINI_API_KEY`
19	Rate limiting	✅	Request throttling implemented
20	Response time <10s	✅	Streaming for fast feedback
21	Session history	✅	Context maintained in session

Architecture

Component Diagram



Provider Configuration

Provider	API Key Env Var	Use Case
OpenRouter	`OPENROUTER_API_KEY`	Primary test generation
Gemini	`GEMINI_API_KEY`	Fallback/alternative

Prompt Engineering

System Prompt Structure

The AI assistant uses structured prompts optimized for Vitest test generation:

1. **Role Definition**: Expert Vitest test writer
2. **Context Injection**: Requirement text, spec structure, existing tests
3. **Output Format**: Valid TypeScript/Vitest code
4. **Constraints**: Follow project conventions, no external dependencies

Prompt Templates

Scenario	Template
New test from requirement	Requirement + spec context → test code
Test improvement	Existing test + feedback → improved code
Coverage expansion	Requirement + coverage gaps → additional tests

Implementation Details

AI Integration Files

```
app/
└── ai/
    └── page.tsx           # AI panel UI

orpc/
└── ai.ts                  # AI generation procedures
```

```
lib/
└── ...                    # AI utilities
```

Key Features

1. **Context-Aware Generation**
    - Includes folder/spec/requirement context
    - References existing tests in spec
    - Follows project test conventions
2. **Streaming Responses**
    - Real-time code generation display
    - Reduced perceived latency
    - Progress indication
3. **Code Quality**
    - Generated code follows Vitest patterns
    - Includes describe/it blocks
    - Proper assertions and matchers
4. **Error Handling**
    - API timeout handling
    - Rate limit response
    - Graceful degradation

System Limitations

Limitation	Description
Vitest only	Currently generates Vitest tests only
No execution	Does not run generated tests
Context window	Limited by LLM token limits
Quality variance	AI output quality may vary
API dependency	Requires external LLM API access

Usage Guidelines

For Users

1. Provide clear, specific requirements
2. Review generated code before accepting
3. Edit as needed for edge cases
4. Verify test assertions match expectations

For Developers

1. API keys stored in environment variables only
2. Rate limiting prevents abuse

3. Logging captures generation metrics
4. Error boundaries handle failures gracefully

Metrics & Monitoring

Metric	Description
Generation time	Time from request to complete response
Token usage	Tokens consumed per generation
Success rate	Percentage of successful generations
User acceptance	Rate of accepted vs rejected code

Security Measures

Measure	Implementation
API key protection	Environment variables only
Input sanitization	Zod validation
Output validation	Code structure verification
Rate limiting	Per-user request limits
Logging	No sensitive data in logs

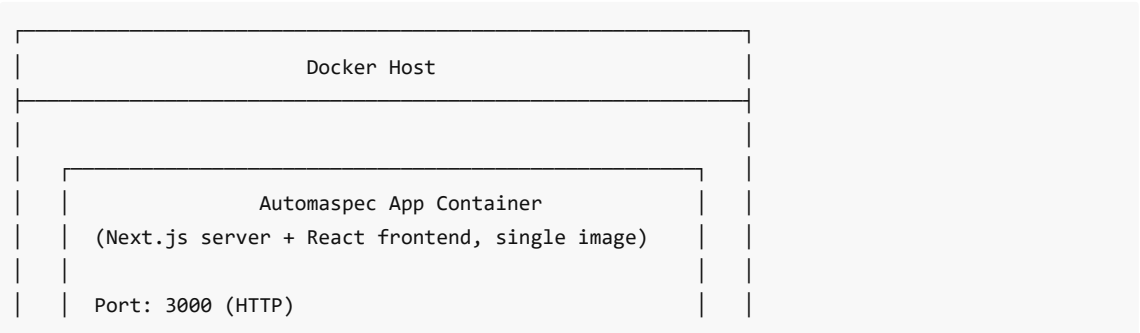
References

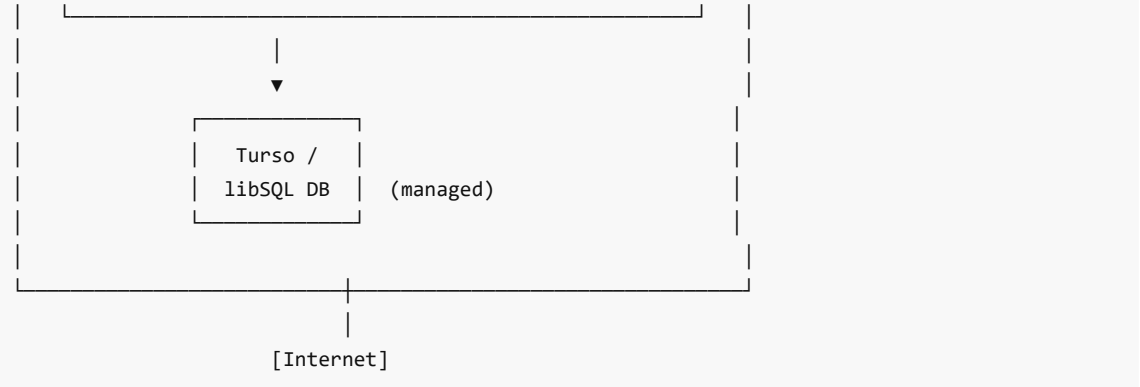
- Requirements Document
- Vercel AI SDK
- AI Implementation
- oRPC Procedures

Deployment & DevOps

Infrastructure

Deployment Architecture





The app is packaged as a single Next.js standalone container (runner stage). It connects over the internet to a managed libSQL (Turso) database.

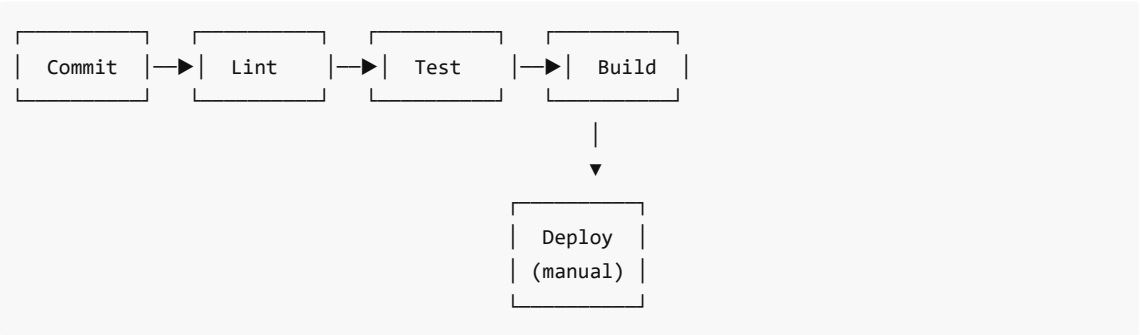
Environments

Environment	URL	Branch
Development	http://localhost:3000	feature/*
Production	[Deployed Docker host URL]	main

There is no dedicated staging environment; production images are built from main and tagged using semantic versioning.

CI/CD Pipeline

Pipeline Overview



Automated CI can run linting, tests, and Next.js builds, while deployment is performed manually using Docker images and docker-compose.

Pipeline Steps

Step	Tool	Actions
Build	Node.js + pnpm + Next.js	Install dependencies and run `pnpm build`
Lint	Oxlint	Run `pnpm lint` to enforce code quality
Format	Oxfmt	Run `pnpm format` for consistent formatting

Test	Vitest + Playwright	Run `pnpm test` and `pnpm test:e2e`
Docker Image	Docker	Build standalone Next.js image via Dockerfile
Deploy	Docker Compose	Start/upgrade app using `docker-compose.prod.yml`

Pipeline Configuration

Example GitHub Actions workflow sketch:

```yaml
name: CI

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 24
          cache: pnpm
      - run: corepack enable pnpm
      - run: pnpm install --frozen-lockfile
      - run: pnpm lint
      - run: pnpm test
      - run: pnpm build
```

Environment Variables

Variable	Description	Required	Example
`DATABASE_URL`	libSQL/Turso database URL used by Drizzle	Yes	`libsql://<host>?authToken=...`
`DATABASE_AUTH_TOKEN`	Token for Turso/libSQL access	Yes	`***` (stored in secrets)
`NEXT_PUBLIC_DATABASE_URL`	Public base URL for DB (if exposed to client)	No	`https://...`
`NODE_ENV`	Environment mode	Yes	`development` / `production`
`PORT`	Application port	No	`3000`

Secrets are stored outside the repo (for example, Docker host environment, docker-compose `.env`, or CI secret store). `.env` is not committed.

How to Run Locally

Prerequisites

- Node.js 24+
- pnpm (managed via corepack)
- Docker (optional, for container-based run)
- Access to a libSQL/Turso database (or compatible local instance)

Setup Steps

```
# 1. Clone repository
git clone <repository-url>
cd automaspec

# 2. Install dependencies
corepack enable pnpm
pnpm install --frozen-lockfile

# 3. Configure environment variables
cp .env.example .env   # if available, otherwise create .env
# Set DATABASE_URL, DATABASE_AUTH_TOKEN, and other required vars

# 4. Run database migrations (if needed)
pnpm dbup

# 5. Start development server
pnpm dev
```

Docker Setup (Alternative)

```
# Development Docker Compose
pnpm docker:dev:build
pnpm docker:dev:up

# Stop containers
pnpm docker:dev:down
```

For production-like runs on a server, use:

```
pnpm docker:prod:build
pnpm docker:prod:up
```

Verify Installation

After starting the app (locally or via Docker):

1. Open http://localhost:3000
2. The Automaspec dashboard should load and display seeded or empty analytics/views

3. API and backend health can be inferred from successful page loads; a dedicated health endpoint can be added if required

Monitoring & Logging

Aspect	Tool	Dashboard URL
Application Logs	Pino (structured logs), Docker logs	n/a (view via `docker logs` or hosting provider)
Error Tracking	Not yet integrated	-
Performance	Browser devtools, custom analytics	-
Uptime	Not yet integrated	-

---

3. User Guide

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

Contents

Getting Started

System Requirements

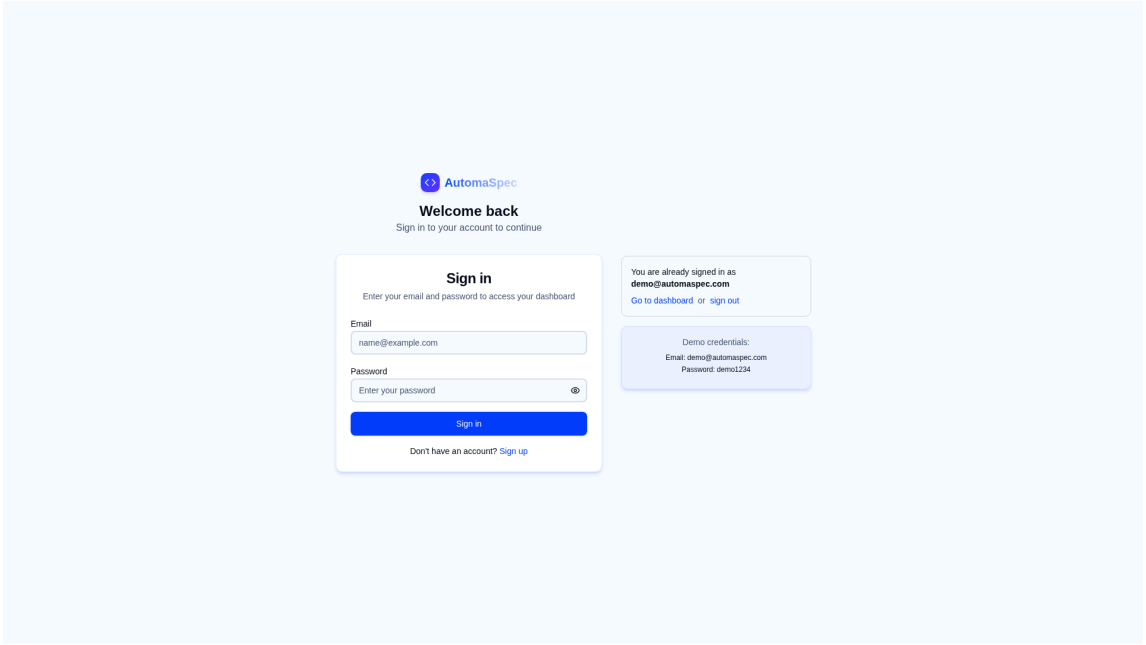
Requirement	Minimum	Recommended
Browser	Chrome 90+, Firefox 88+, Safari 14+, Edge 90+	Latest version
Screen Resolution	1280x720	1920x1080
Internet	Required	Stable broadband
Device	Desktop, Tablet, or Mobile	Desktop for full experience

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

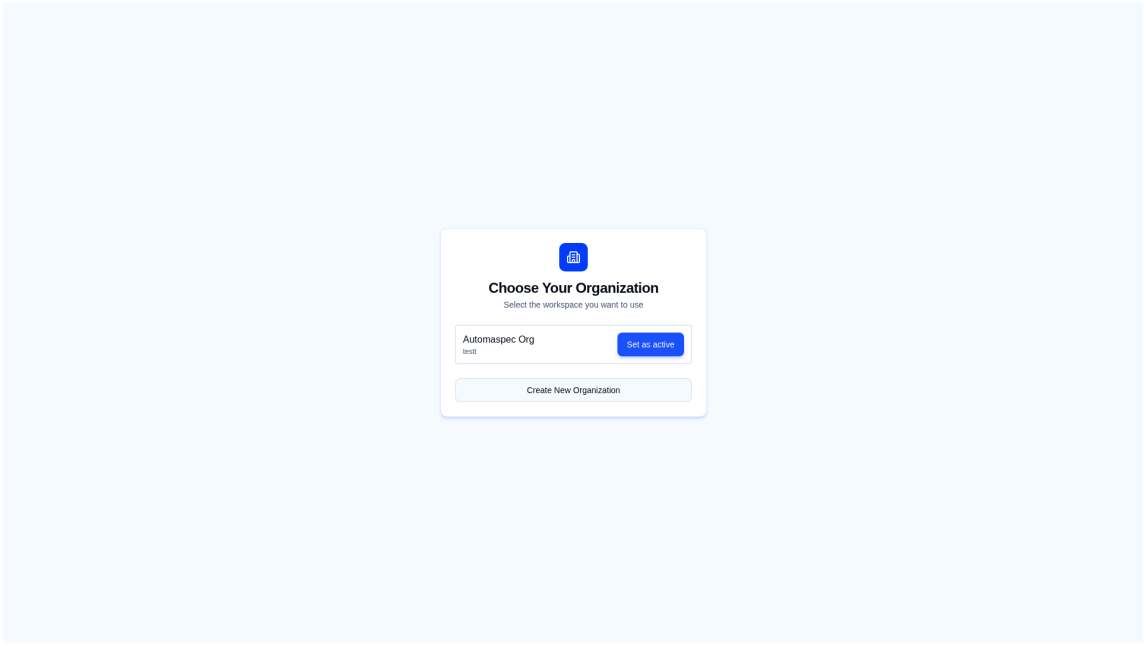
First Launch

Step 1: Registration/Login



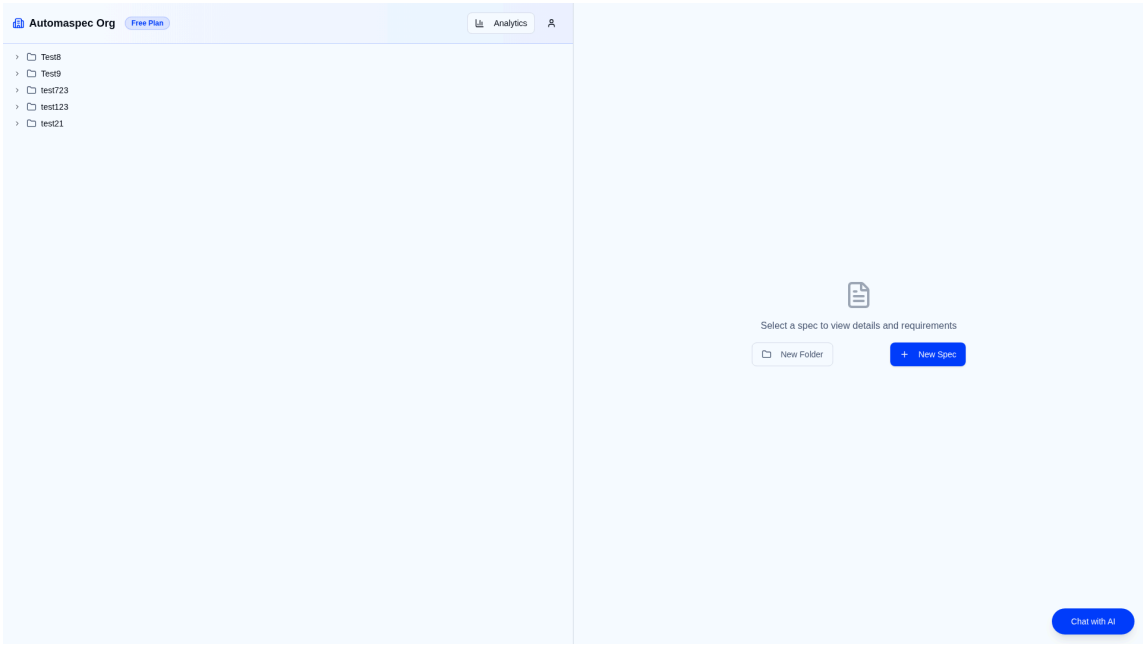
1. Click **"Sign In"** or **"Get Started"** on the landing page
2. For new users: Enter your email and create a password (minimum 8 characters)
3. For existing users: Enter your credentials and click **"Sign In"**
4. You will be redirected to the organization selection page

Step 2: Organization Setup



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



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
Create a folder	Click **"+ New Folder"** in the left panel, enter name
Create a test spec	Select a folder, click **"+ New Spec"**, enter details
Add a requirement	Open a spec, click **"+ Add Requirement"**
Generate AI test	Select a requirement, click **"Generate with AI"**
View analytics	Click **"Analytics"** in the navigation menu
Invite team member	Go to Profile → Organization → Invite Members

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
`Ctrl/Cmd + N`	New spec/folder (context dependent)
`Ctrl/Cmd + S`	Save current changes
`Escape`	Close modal/panel
`Tab`	Navigate between fields

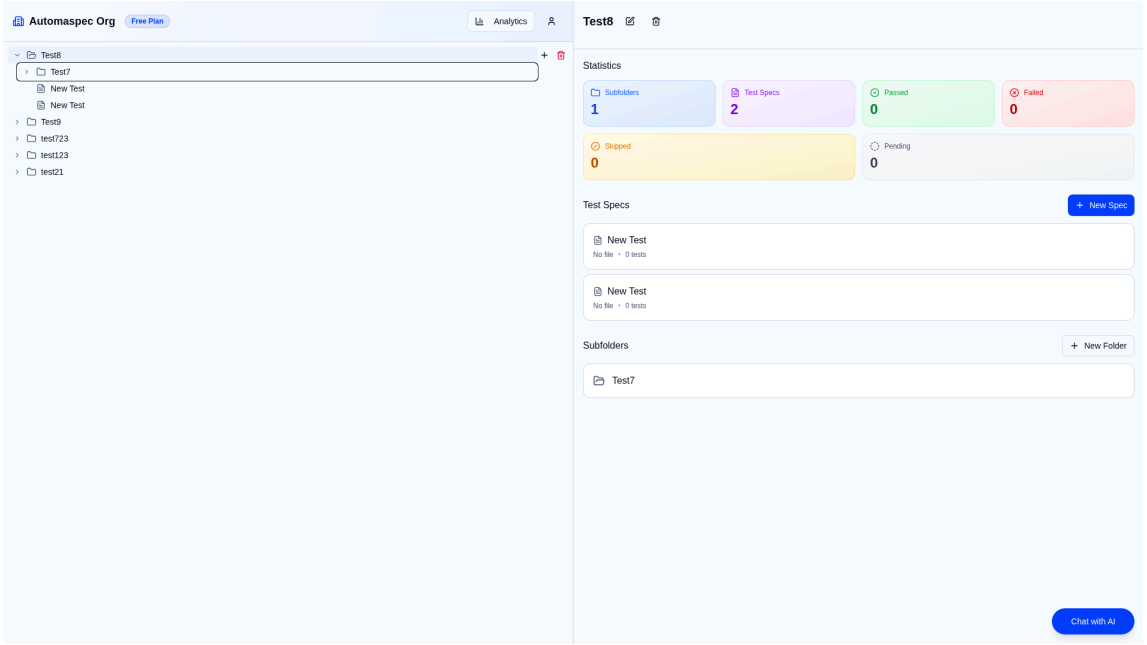
Feature Walkthrough

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



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

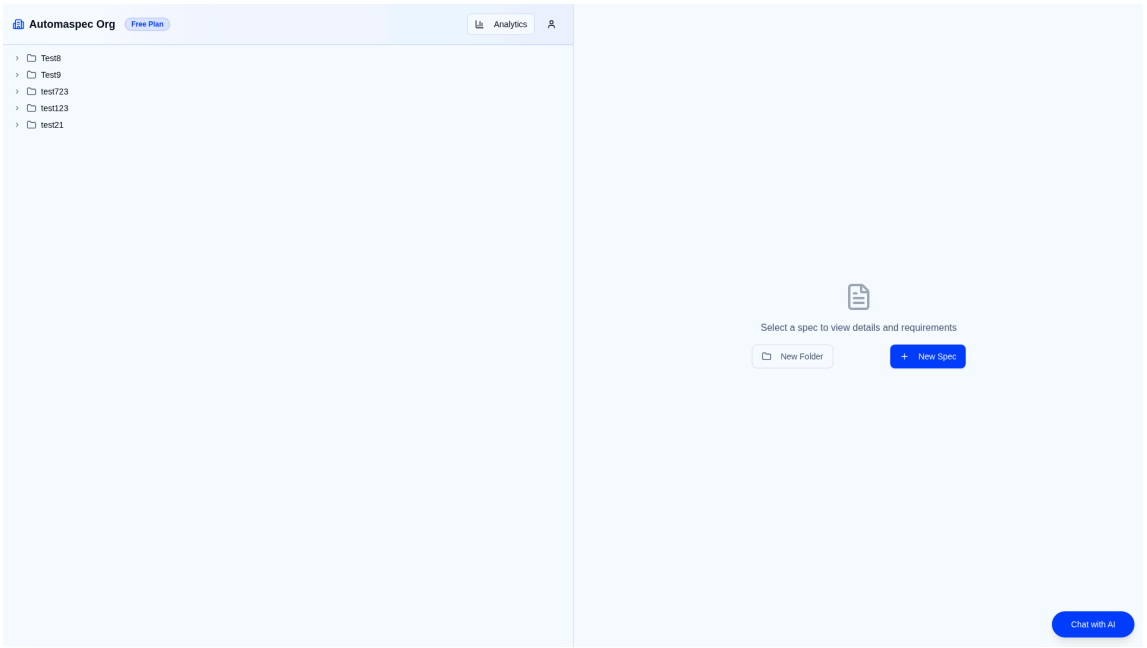
- Write requirements in user story format: "As a [user], I want [action], so that [benefit]"
- Keep requirements atomic (one testable thing per requirement)
- Include both positive and negative test scenarios

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



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

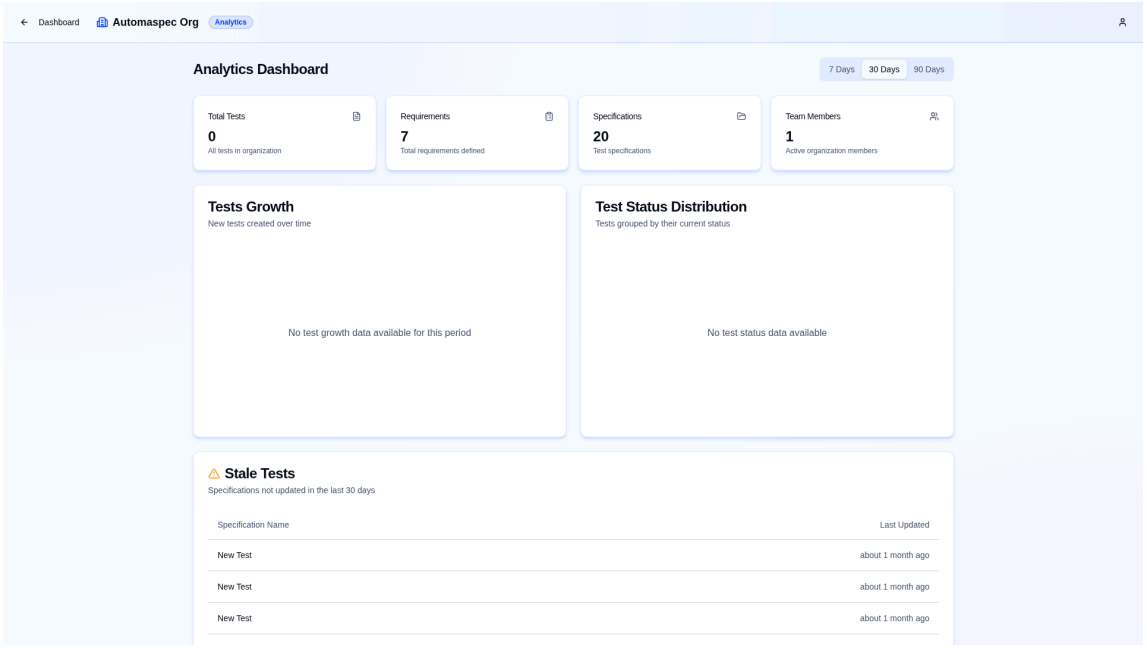
Expected Result: Real-time visibility into test health

Feature 6: Analytics Dashboard

Overview

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

How to Use



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
"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
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
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

Overview

Attribute	Value
Base URL	`https://automaspec.vercel.app/rpc`
Authentication	Session-based (cookies)
Format	JSON
Documentation	Interactive Docs

The Automaspec API is built using **oRPC** which provides type-safe RPC endpoints. Full interactive documentation is available at `/rpc/docs` powered by Scalar UI.

Authentication

All API endpoints require authentication via session cookies. Users must be logged in through the web interface or use API keys for programmatic access.

Session-Based Auth

```
Cookie: session=<session_token>
```

API Key Auth (for integrations)

```
Authorization: Bearer <api_key>
```

Core Endpoints

Folders

List Folders

```
GET /rpc/folders.list
```

Query Parameters:

Parameter	Type	Required	Description
organizationId	string	Yes	Organization UUID

Response:

```
{
  "data": [
    {
      "id": "uuid",
      "name": "Feature Tests",
```

```
      "parentFolderId": null,
      "order": 0
    }
  ]
}
```

Create Folder

```
POST /rpc/folders.create
```

Request Body:

```
{
  "name": "New Folder",
  "organizationId": "uuid",
  "parentFolderId": null
}
```

Test Specs

List Specs

```
GET /rpc/specs.list
```

Query Parameters:

Parameter	Type	Required	Description
folderId	string	Yes	Parent folder UUID

Create Spec

```
POST /rpc/specs.create
```

Request Body:

```
{
  "name": "Login Tests",
  "folderId": "uuid",
  "organizationId": "uuid"
}
```

Get Spec

```
GET /rpc/specs.get
```

Query Parameters:

Parameter	Type	Required
id	string	Yes

Requirements

List Requirements

```
GET /rpc/requirements.list
```

Query Parameters:

Parameter	Type	Required
specId	string	Yes

Create Requirement

```
POST /rpc/requirements.create
```

Request Body:

```
{
  "name": "User can login with valid credentials",
  "description": "Verify that...",
  "specId": "uuid"
}
```

Tests

List Tests

```
GET /rpc/tests.list
```

Query Parameters:

Parameter	Type	Required
requirementId	string	Yes

Create Test

```
POST /rpc/tests.create
```

Request Body:

```
{
  "requirementId": "uuid",
  "status": "pending",
  "framework": "vitest",
  "code": "describe('Login', () => { ... })"
}
```

Update Test Status

```
POST /rpc/tests.updateStatus
```

Request Body:

```
{
  "id": "uuid",
  "status": "passed"
}
```

AI Generation

Generate Test Code

```
POST /rpc/ai.generate
```

Request Body:

```
{
  "requirementId": "uuid",
  "context": "Additional context..."
}
```

Response (Streaming):

```
{
  "code": "describe('Feature', () => {\n  it('should...', () => {\n    // Generated test\n
})\n})"
}
```

Analytics

Get Metrics

```
GET /rpc/analytics.getMetrics
```

Query Parameters:

Parameter	Type	Required
organizationId	string	Yes
period	string	No (default: "7d")

Error Responses

Status Code	Description
400	Bad Request - Invalid input
401	Unauthorized - Not logged in
403	Forbidden - No permission
404	Not Found - Resource doesn't exist
429	Too Many Requests - Rate limited
500	Internal Server Error

Error Response Format:

```
{
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid input",
    "details": { ... }
  }
}
```

Rate Limiting

Tier	Requests/Minute	Requests/Day
Free	60	1,000
Pro	300	10,000

OpenAPI Specification

Full OpenAPI 3.0 specification available at:

- **Interactive Docs**: https://automaspec.vercel.app/rpc/docs
- **JSON Spec**: https://automaspec.vercel.app/rpc/spec

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**.

Attribute	Value
Database	SQLite (LibSQL/Turso)
ORM	Drizzle ORM
Migrations	Drizzle Kit

Entity Relationship Diagram

```
erDiagram
    User ||--o{ Account : "has"
    User ||--o{ Session : "has"
    User ||--o{ Member : "belongs to"
    User ||--o{ APIKey : "owns"

    Organization ||--o{ Member : "has members"
    Organization ||--o{ Invitation : "sends"
    Organization ||--o{ TestFolder : "owns"
    Organization ||--o{ TestSpec : "owns"

    User ||--o{ Invitation : "invites"

    TestFolder ||--o{ TestSpec : "contains"
    TestFolder ||--o{ TestFolder : "parent"

    TestSpec ||--o{ TestRequirement : "defines"
    TestRequirement ||--o{ Test : "verified by"

    User {
        string id PK
        string email
        string name
        boolean emailVerified
    }

    Organization {
        string id PK
        string name
        string slug
        string plan
    }

    TestSpec {
        string id PK
```

```
        string name
        json statuses
        integer numberOfTests
    }

    TestRequirement {
        string id PK
        string name
        string description
    }

    Test {
        string id PK
        string status
        string framework
        string code
    }
```

Tables Reference

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

`user`

Core user entity.

Column	Type	Constraints	Description
id	text	PK	Unique identifier
name	text	NOT NULL	Display name
email	text	NOT NULL, UNIQUE	User email
emailVerified	boolean	NOT NULL	Email verification status
image	text	NULLABLE	Avatar URL
createdAt	timestamp	NOT NULL	Creation time
updatedAt	timestamp	NOT NULL	Last update time

`session`

Active user sessions.

Column	Type	Constraints	Description
id	text	PK	Unique identifier
userId	text	FK → user.id	Session owner
token	text	NOT NULL, UNIQUE	Session token
expiresAt	timestamp	NOT NULL	Expiration time

ipAddress	text	NULLABLE	Client IP
userAgent	text	NULLABLE	Browser/client info
activeOrganizationId	text	NULLABLE	Current org context

`account`

OAuth/credential accounts linked to users.

Column	Type	Constraints	Description
id	text	PK	Unique identifier
userId	text	FK → user.id	Account owner
providerId	text	NOT NULL	e.g., "credential", "google"
accountId	text	NOT NULL	Provider-specific ID
accessToken	text	NULLABLE	OAuth access token
refreshToken	text	NULLABLE	OAuth refresh token
password	text	NULLABLE	Hashed password (credential auth)

`organization`

Multi-tenant organization unit.

Column	Type	Constraints	Description
id	text	PK	Unique identifier
name	text	NOT NULL	Organization name
slug	text	UNIQUE	URL-friendly identifier
logo	text	NULLABLE	Logo URL
plan	text	NOT NULL, DEFAULT 'free'	Subscription plan
metadata	text	NULLABLE	JSON metadata

`member`

Linking table between Users and Organizations.

Column	Type	Constraints	Description
id	text	PK	Unique identifier
organizationId	text	FK → organization.id	Organization
userId	text	FK → user.id	User
role	text	NOT NULL, DEFAULT 'member'	Role: 'owner', 'admin', 'member'

`invitation`

Pending organization invites.

Column	Type	Constraints	Description
id	text	PK	Unique identifier
email	text	NOT NULL	Invitee email
organizationId	text	FK → organization.id	Target organization
inviterId	text	FK → user.id	Who sent the invite
role	text	NULLABLE	Role to be assigned
status	text	NOT NULL, DEFAULT 'pending'	Status: 'pending', 'accepted'
expiresAt	timestamp	NOT NULL	Invitation expiration

`apiKey`

API access keys for external integrations.

Column	Type	Constraints	Description
id	text	PK	Unique identifier
name	text	NULLABLE	Key name
key	text	NOT NULL	Hashed key value
userId	text	FK → user.id	Key owner
enabled	boolean	DEFAULT true	Is key active
rateLimitEnabled	boolean	DEFAULT true	Rate limiting on/off
rateLimitMax	integer	DEFAULT 10	Max requests per window
permissions	text	NULLABLE	JSON permission scopes

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

`test_folder`

Hierarchical organization for test specs.

Column	Type	Constraints	Description
id	text	PK	Unique identifier
name	text	NOT NULL	Folder name
description	text	NULLABLE	Optional description
parentFolderId	text	NULLABLE	Parent folder for nesting

organizationId	text	FK → organization.id	Owner organization
order	integer	NOT NULL, DEFAULT 0	Sorting order
createdAt	text	NOT NULL	Creation timestamp
updatedAt	text	NOT NULL	Last update timestamp

`test_spec`

A test specification containing requirements.

Column	Type	Constraints	Description
id	text	PK	Unique identifier
name	text	NOT NULL	Spec name
fileName	text	NULLABLE	Source file reference
description	text	NULLABLE	Optional description
statuses	json	NOT NULL	Aggregated status counts
numberOfTests	integer	NOT NULL, DEFAULT 0	Total test count
folderId	text	FK → test_folder.id	Parent folder
organizationId	text	FK → organization.id	Owner organization

`test_requirement`

Specific business requirement to be tested.

Column	Type	Constraints	Description
id	text	PK	Unique identifier
name	text	NOT NULL	Requirement title
description	text	NULLABLE	Detailed description
order	integer	NOT NULL, DEFAULT 0	Sorting order
specId	text	FK → test_spec.id	Parent spec

`test`

Individual test implementation.

Column	Type	Constraints	Description
id	text	PK	Unique identifier
status	text	NOT NULL	Test result: passed, failed, pending, skipped
framework	text	NOT NULL	Framework: vitest, jest, playwright

code	text	NULLABLE	Generated test code
requirementId	text	FK → test_requirement.id	Parent requirement

Relationships

Relationship	Type	Description
User → Session	One-to-Many	User can have multiple sessions
User → Member	One-to-Many	User can belong to multiple orgs
Organization → Member	One-to-Many	Org has multiple members
Organization → TestFolder	One-to-Many	Org owns folders
TestFolder → TestFolder	One-to-Many	Folder nesting (self-referential)
TestFolder → TestSpec	One-to-Many	Folder contains specs
TestSpec → TestRequirement	One-to-Many	Spec defines requirements
TestRequirement → Test	One-to-Many	Requirement has tests

Migrations

Migrations are managed via Drizzle Kit and stored in `db/migrations/`.

Command	Description
`pnpm dbg`	Generate migration from schema changes
`pnpm dbm`	Apply migrations to database
`pnpm dbs`	Open Drizzle Studio
`pnpm dbup`	Generate + apply migrations

Seeding

Test data can be loaded via:

```
sqlite3 db/local.db < sample_data.sql
```
