Step-by-Step Development Plan

DevSecOps Testing Platform (1-2 Developers)

Overview

This plan is optimized for 1-2 developers building the platform over 12-18 months. Each step includes:

- **What to build**
- **Tools & technologies**
- **Concrete tasks with checkboxes**
- **Windsurf AI prompts** to accelerate development
- **Expected timeline**

Strategy: Build MVP first, then incrementally add features. Focus on automation and using existing tools/libraries.

Phase 1: Foundation & MVP (Months 1-6)

Step 1: Project Setup & Infrastructure (Week 1-2)

Goal: Set up development environment and basic infrastructure

Tasks:

- [ ] Create GitHub repository with main, develop, and feature branches
- [ ] Set up project structure (monorepo with services folder)
- [ ] Install Docker Desktop
- [ ] Create docker-compose.yml for local development
- [ ] Set up PostgreSQL container
- [ ] Set up Redis container
- [ ] Set up basic CI/CD with GitHub Actions

Tools & Technologies:

- **Version Control**: GitHub

- **Containerization**: Docker, Docker Compose
- **Database**: PostgreSQL 15
- **Cache**: Redis 7

Files to Create:

project-root/

├── .github/

│ └── workflows/

│ └── ci.yml

├── services/

│ ├── api-gateway/

│ ├── test-executor/

│ └── frontend/

├── docker-compose.yml

├── .gitignore

├── README.md

└── package.json (root)

Windsurf Prompts:

"Create a docker-compose.yml file for local development with PostgreSQL 15, Redis 7, and volume mounts for data persistence. Include environment variables for database credentials."

"Generate a GitHub Actions workflow file that runs linting and tests on every pull request to main branch."

"Create a monorepo folder structure for a microservices project with api-gateway, test-executor, and frontend services. Include package.json files with basic scripts."

Validation:

- [ ] `docker-compose up` runs without errors
- [ ] Can connect to PostgreSQL on localhost:5432
- [ ] Can connect to Redis on localhost:6379
- [ ] GitHub Actions workflow runs successfully

Step 2: Database Schema & Migrations (Week 2-3)

Goal: Design and implement core database schema

Tasks:

- [ ] Install Prisma ORM (or TypeORM)
- [ ] Design database schema for users, organizations, projects
- [ ] Create migration for initial tables
- [ ] Add test_runs and test_results tables
- [ ] Create seed data for development
- [ ] Write database connection utility

Tools & Technologies:

- **ORM**: Prisma (recommended for small teams)
- **Migration**: Prisma Migrate
- **Database Client**: Prisma Client

Windsurf Prompts:

"Create a Prisma schema file with models for User (id, email, username, passwordHash, role, createdAt, updatedAt), Organization (id, name, slug, settings, createdAt), Project (id, organizationId, name, slug, repositoryUrl, settings, createdAt), and relationships between them."

"Generate a Prisma schema for TestRun with fields: id, projectId, commitSha, branch, status, startedAt, completedAt, totalTests, passedTests, failedTests, duration, coverage percentage."

"Create a seed script using Prisma that creates a test user, organization, and sample project for development."

Files to Create:

services/api-gateway/

├── prisma/

│ ├── schema.prisma

│ ├── migrations/

│ └── seed.ts

Validation:

- [ ] `npx prisma migrate dev` creates tables successfully
- [ ] `npx prisma studio` shows all tables
- [ ] Seed script populates test data
- [ ] Can query data using Prisma Client

Step 3: API Gateway & Authentication (Week 3-5)

Goal: Build API server with JWT authentication

Tasks:

- [ ] Initialize Node.js project with TypeScript
- [ ] Set up Express.js server
- [ ] Implement JWT token generation and validation
- [ ] Create user registration endpoint
- [ ] Create login endpoint
- [ ] Create middleware for authentication
- [ ] Add rate limiting middleware
- [ ] Set up request validation (Zod)
- [ ] Add error handling middleware
- [ ] Write unit tests for auth

Tools & Technologies:

- **Runtime**: Node.js 20+
- **Framework**: Express.js or Fastify
- **Language**: TypeScript
- **Auth**: jsonwebtoken, bcrypt
- **Validation**: Zod
- **Testing**: Jest

Windsurf Prompts:

"Create an Express.js server with TypeScript that includes routes for /api/v1/auth/register, /api/v1/auth/login, and /api/v1/auth/me. Use JWT tokens stored in httpOnly cookies. Include input validation using Zod."

"Generate a middleware function that verifies JWT tokens from the Authorization header, extracts the user ID, attaches the user object to the request, and handles token expiration errors."

"Create a complete authentication system with password hashing using bcrypt, JWT token generation with 1-hour expiration and refresh tokens with 7-day expiration. Include token refresh endpoint."

"Write Jest unit tests for the authentication endpoints including successful login, invalid credentials, missing fields, and token validation."

Files to Create:

services/api-gateway/

```
├── src/
│   ├── server.ts
│   ├── routes/
│   │   └── auth.routes.ts
│   ├── middleware/
│   │   ├── auth.middleware.ts
│   │   └── rateLimit.middleware.ts
│   ├── controllers/
│   │   └── auth.controller.ts
│   ├── utils/
```

```
│   │   └── jwt.util.ts
│   └── types/
│       └── express.d.ts
├── tests/
│   └── auth.test.ts
├── package.json
└── tsconfig.json
```

Validation:

- [ ] POST /api/v1/auth/register creates a new user
- [ ] POST /api/v1/auth/login returns JWT token
- [ ] GET /api/v1/auth/me returns user info with valid token
- [ ] Rate limiting blocks excessive requests
- [ ] All tests pass

Step 4: Project & Organization Management API (Week 5-6)

Goal: CRUD operations for organizations and projects

Tasks:

- [ ] Create organization endpoints (CRUD)
- [ ] Create project endpoints (CRUD)
- [ ] Implement role-based permissions
- [ ] Add organization membership management
- [ ] Create project settings endpoints
- [ ] Add input validation for all endpoints
- [ ] Write API tests

Tools & Technologies:

- Same as Step 3 (Express.js + TypeScript)

Windsurf Prompts:

"Create Express routes for organization management: GET /api/v1/organizations (list), POST /api/v1/organizations (create), GET /api/v1/organizations/:id (get), PATCH /api/v1/organizations/:id (update), DELETE /api/v1/organizations/:id (delete). Include authentication middleware and role-based access control."

"Generate a permission middleware that checks if the authenticated user has permission to perform actions on an organization or project. Support roles: owner, admin, member, viewer."

"Create project CRUD endpoints with Prisma ORM integration. Include validation for repository URL format, slug uniqueness per organization, and proper error handling."

Validation:

- [ ] Can create, read, update, delete organizations
- [ ] Can create, read, update, delete projects
- [ ] Only organization admins can modify settings
- [ ] Members can view but not edit
- [ ] All endpoints return proper error codes

Step 5: Frontend Foundation (Week 6-8)

Goal: Build React app with authentication and basic UI

Tasks:

- [ ] Initialize Vite + React + TypeScript project
- [ ] Set up Tailwind CSS
- [ ] Install and configure React Router
- [ ] Create authentication pages (login, register)
- [ ] Set up API client with Axios
- [ ] Implement JWT token storage and refresh
- [ ] Create protected route wrapper
- [ ] Build main layout with sidebar navigation
- [ ] Create organization list page
- [ ] Create project list page
- [ ] Add loading states and error handling

Tools & Technologies:

- **Build Tool**: Vite
- **Framework**: React 18
- **Language**: TypeScript
- **Styling**: Tailwind CSS
- **Routing**: React Router v6
- **HTTP Client**: Axios or TanStack Query
- **State**: Zustand or React Context

Windsurf Prompts:

"Create a Vite + React + TypeScript project with Tailwind CSS configured. Include a basic folder structure with components, pages, hooks, utils, and types folders."

"Generate a React login page with form fields for email and password, form validation, loading state, error display, and submission to /api/v1/auth/login endpoint. Style with Tailwind CSS."

"Create an Axios client with interceptors that automatically attach JWT token from localStorage to requests, handle 401 errors by refreshing the token, and redirect to login on authentication failure."

"Build a React component for a sidebar navigation with links to Dashboard, Projects, Organizations, and Settings. Include active link highlighting and a user profile dropdown."

"Create a protected route wrapper component that checks if user is authenticated, redirects to login if not, and shows a loading spinner while checking authentication status."

Files to Create:

services/frontend/

├── src/

│ ├── main.tsx

│ ├── App.tsx

│ ├── pages/

```
│   │   ├── Login.tsx
│   │   ├── Register.tsx
│   │   ├── Dashboard.tsx
│   │   ├── Projects.tsx
│   │   └── Organizations.tsx
│   ├── components/
│   │   ├── Layout.tsx
│   │   ├── Sidebar.tsx
│   │   └── ProtectedRoute.tsx
│   ├── hooks/
│   │   └── useAuth.ts
│   ├── api/
│   │   └── client.ts
│   └── types/
│       └── index.ts
├── package.json
├── vite.config.ts
└── tailwind.config.js
```

Validation:

- [ ] Can register and login
- [ ] Token stored and auto-attached to requests
- [ ] Protected routes redirect to login when not authenticated
- [ ] Can navigate between pages
- [ ] UI is responsive on mobile

Step 6: Test Executor Service - Core (Week 8-10)

Goal: Build service that executes tests and stores results

Tasks:

- [ ] Initialize Python FastAPI project
- [ ] Set up Celery for job queue
- [ ] Create endpoint to trigger test runs
- [ ] Implement Docker container executor
- [ ] Add support for Jest tests
- [ ] Parse Jest JSON results
- [ ] Store test results in database
- [ ] Upload artifacts to S3/MinIO
- [ ] Add WebSocket for real-time updates
- [ ] Write integration tests

Tools & Technologies:

- **Language**: Python 3.11+
- **Framework**: FastAPI
- **Task Queue**: Celery
- **Message Broker**: Redis
- **Container**: Docker SDK for Python
- **Storage**: MinIO (local) or AWS S3

Windsurf Prompts:

"Create a FastAPI application with an endpoint POST /api/v1/test-runs that accepts projectId, commitSha, and branch. Queue a Celery task to execute tests and return a test run ID immediately."

"Generate a Celery task that takes a test run configuration, pulls code from a Git repository, builds a Docker container with Node.js, runs Jest tests inside the container with JSON reporter, and parses the results."

"Create a Python function that parses Jest JSON output and converts it to a list of test result objects with fields: suiteName, testName, status, duration, errorMessage, errorStack."

"Build a WebSocket endpoint in FastAPI that allows clients to subscribe to test run updates. When a test completes, broadcast the result to all connected clients subscribed to that test run ID."

"Write a Python script that uses the Docker SDK to create a container from a Dockerfile, execute a command inside it with timeout, stream logs in real-time, and clean up the container after execution."

Files to Create:

services/test-executor/

```
├── app/
│   ├── main.py
│   ├── routes/
│   │   └── test_runs.py
│   ├── tasks/
│   │   ├── execute_tests.py
│   │   └── parse_results.py
│   ├── services/
│   │   ├── docker_executor.py
│   │   └── git_service.py
│   ├── parsers/
│   │   └── jest_parser.py
│   └── websocket/
│       └── test_updates.py
├── tests/
├── requirements.txt
```

├── Dockerfile

└── celery_config.py

Validation:

- [ ] Can trigger a test run via API
- [ ] Celery worker picks up the task
- [ ] Tests execute in Docker container
- [ ] Results saved to database
- [ ] WebSocket broadcasts updates
- [ ] Can view test results via API

Step 7: Test Results UI (Week 10-12)

Goal: Display test runs and results in the frontend

Tasks:

- [ ] Create test runs list page
- [ ] Create test run detail page with results table
- [ ] Add real-time updates via WebSocket
- [ ] Implement filtering and search
- [ ] Display test status badges
- [ ] Show test execution timeline
- [ ] Add ability to re-run failed tests
- [ ] Create test result detail modal

Windsurf Prompts:

"Create a React component that displays a table of test runs with columns: commit SHA, branch, status (pending/running/passed/failed), start time, duration, and pass rate. Use Tailwind CSS for styling and add status badges with appropriate colors."

"Build a React component for test run details that shows summary statistics at the top (total tests, passed, failed, skipped, duration) and a filterable table of individual test results below. Include search functionality."

"Create a WebSocket hook in React that connects to the test executor WebSocket endpoint, subscribes to test run updates, and automatically updates the UI when new results arrive. Handle connection errors and reconnection."

"Generate a React component that displays test execution progress with a progress bar, percentage complete, currently running test name, and elapsed time. Update in real-time using WebSocket data."

Validation:

- [ ] Test runs list shows all runs for a project
- [ ] Can click into a test run and see all results
- [ ] Real-time updates work when tests are running
- [ ] Can filter by status (passed/failed)
- [ ] Can search test names
- [ ] Can re-run failed tests

Step 8: Code Coverage Integration (Week 12-14)

Goal: Collect and display code coverage

Tasks:

- [ ] Modify test executor to collect coverage
- [ ] Parse Istanbul/NYC coverage JSON
- [ ] Store coverage data in database
- [ ] Create coverage API endpoints
- [ ] Build coverage dashboard UI
- [ ] Show coverage trends over time
- [ ] Display file-level coverage
- [ ] Add coverage badges

Tools & Technologies:

- **Coverage Tools**: Istanbul (JavaScript), Coverage.py (Python), JaCoCo (Java)

Windsurf Prompts:

"Modify the Jest test execution task to generate coverage reports using the --coverage flag and output in JSON format. Parse the coverage-summary.json file to extract line, branch, function, and statement coverage percentages."

"Create a FastAPI endpoint GET /api/v1/projects/:projectId/coverage that returns the latest coverage data and a trend of coverage over the last 30 days. Query from PostgreSQL using Prisma."

"Build a React component that displays code coverage with a gauge chart showing overall percentage, and a table showing per-file coverage with columns: filename, lines covered, branches covered, color-coded based on thresholds."

"Create a Python function that takes a coverage JSON file, extracts file-level coverage data, and stores it in the database with relationships to the test run. Handle different coverage formats (Istanbul, Coverage.py)."

Validation:

- [ ] Coverage collected during test execution
- [ ] Coverage data stored in database
- [ ] Coverage dashboard shows current percentage
- [ ] Trend chart shows coverage over time
- [ ] Can drill down to file-level coverage

Step 9: Basic Security Scanning (Week 14-16)

Goal: Integrate OWASP Dependency Check for vulnerabilities

Tasks:

- [ ] Create security analysis service
- [ ] Integrate OWASP Dependency-Check CLI
- [ ] Parse vulnerability results
- [ ] Store vulnerabilities in database
- [ ] Create vulnerability API endpoints
- [ ] Build vulnerability dashboard UI

- [ ] Add severity filtering
- [ ] Create vulnerability detail page

Tools & Technologies:

- **Scanner**: OWASP Dependency-Check
- **Alternative**: npm audit, pip-audit

Windsurf Prompts:

"Create a Python Celery task that runs OWASP Dependency-Check on a project's dependencies, parses the XML or JSON output, extracts vulnerability information (CVE ID, severity, description, affected package), and stores it in the database."

"Generate a FastAPI endpoint GET /api/v1/projects/:projectId/vulnerabilities that returns vulnerabilities with filtering by severity (critical, high, medium, low) and pagination support."

"Build a React component that displays a vulnerability dashboard with cards showing counts by severity, a chart showing vulnerability trends, and a table of recent vulnerabilities with columns: CVE ID, severity, package, description."

"Create a Python parser for OWASP Dependency-Check JSON output that extracts CVE ID, CVSS score, severity, vulnerable dependency name and version, and recommended fix version."

Files to Create:

services/analysis-engine/

```
├── app/
│   ├── main.py
│   ├── tasks/
│   │   └── dependency_scan.py
│   ├── parsers/
│   │   └── owasp_parser.py
```

```
│   └── routes/
│       └── vulnerabilities.py
├── requirements.txt
└── Dockerfile
```

Validation:

- [ ] Can trigger security scan from UI
- [ ] Scan runs and parses results
- [ ] Vulnerabilities stored in database
- [ ] Dashboard shows vulnerability counts
- [ ] Can filter by severity
- [ ] Can view vulnerability details

Step 10: MVP Polish & Testing (Week 16-18)

Goal: Polish MVP, fix bugs, add documentation

Tasks:

- [ ] Fix all critical bugs
- [ ] Add loading states throughout UI
- [ ] Improve error messages
- [ ] Add form validation feedback
- [ ] Write user documentation
- [ ] Create API documentation with Swagger/OpenAPI
- [ ] Add health check endpoints
- [ ] Set up basic monitoring (Prometheus)
- [ ] Conduct security review
- [ ] Performance testing
- [ ] Deploy to staging environment

Windsurf Prompts:

"Generate OpenAPI/Swagger documentation for all API endpoints including request/response schemas, authentication requirements, and example requests."

"Create a comprehensive README.md for the project that includes overview, architecture diagram, setup instructions, environment variables, running locally, running tests, and deployment guide."

"Build a health check endpoint GET /health that checks database connectivity, Redis connectivity, and returns JSON with status of each service and overall system health."

"Add error boundary components to the React app that catch JavaScript errors, display user-friendly error messages, and log errors to console for debugging."

Validation:

- [ ] All features work end-to-end
- [ ] No console errors in browser
- [ ] API documentation is complete
- [ ] README has setup instructions
- [ ] Can deploy to staging
- [ ] Basic monitoring is working

Phase 2: Enhanced Features (Months 7-12)

Step 11: Add More Test Frameworks (Week 19-21)

Goal: Support pytest and Cypress

Tasks:

- [ ] Create pytest test runner plugin
- [ ] Parse pytest JSON output
- [ ] Create Cypress test runner plugin
- [ ] Parse Cypress results
- [ ] Store screenshots and videos from Cypress
- [ ] Update UI to show framework-specific info
- [ ] Test with real projects

Windsurf Prompts:

"Create a Python class for executing pytest tests in a Docker container. Run pytest with --json-report flag, parse the JSON output, and convert to standardized test result format."

"Generate a test runner plugin for Cypress that executes tests in a Docker container with Chrome, collects JSON results, uploads screenshots and videos to S3, and returns structured test results."

"Build a plugin architecture for the test executor that allows registering new test framework handlers. Each handler should implement: detect(project_path) -> bool, execute_tests(config) -> Results, parse_output(output) -> List[TestResult]."

Validation:

- [ ] Can run pytest tests successfully
- [ ] Can run Cypress E2E tests
- [ ] Screenshots and videos captured
- [ ] All results displayed in UI

Step 12: Advanced Security Scanning (Week 21-24)

Goal: Add SAST with Semgrep and secrets scanning

Tasks:

- [ ] Integrate Semgrep for SAST
- [ ] Create Semgrep rule configuration
- [ ] Parse SAST findings
- [ ] Integrate TruffleHog for secrets scanning
- [ ] Parse secrets findings
- [ ] Add security dashboard
- [ ] Implement finding deduplication
- [ ] Add false positive marking

Tools & Technologies:

- **SAST**: Semgrep
- **Secrets**: TruffleHog or GitGuardian

Windsurf Prompts:

"Create a Celery task that runs Semgrep on a project's source code, parses the JSON output to extract security findings (rule ID, severity, file, line number, code snippet, message), and stores them in the database."

"Generate a Python script that runs TruffleHog to scan a Git repository for secrets, parses the output to extract secret type, file location, and the secret itself (masked), and saves findings to database."

"Build a React dashboard component for security findings that shows tabs for different categories (SAST, SCA, Secrets), displays count badges, and a table of findings with severity indicators and remediation suggestions."

Validation:

- [ ] SAST scans detect code issues
- [ ] Secrets scanning finds API keys
- [ ] Findings categorized correctly
- [ ] Can mark false positives
- [ ] Security dashboard shows all findings

Step 13: Log Management (Week 24-27)

Goal: Collect and search logs

Tasks:

- [ ] Set up Elasticsearch
- [ ] Install Filebeat or Fluentd for log collection
- [ ] Create log ingestion API
- [ ] Parse and index logs
- [ ] Build log search API
- [ ] Create log viewer UI
- [ ] Add filtering and search
- [ ] Implement log retention policies

Tools & Technologies:

- **Search Engine**: Elasticsearch
- **Log Shipper**: Filebeat or Fluentd
- **Alternative**: Loki (simpler, from Grafana)

Windsurf Prompts:

"Create a FastAPI endpoint POST /api/v1/logs/search that accepts Elasticsearch query DSL, searches logs, and returns results with pagination. Support filtering by level, service, timestamp range."

"Generate a React component for log viewing that displays logs in a table with columns: timestamp, level, service, message. Include a search bar, level filter dropdown, and date range picker. Use Tailwind CSS."

"Write a Python function that takes raw log strings, parses them into structured format (timestamp, level, service, message, metadata), and indexes them in Elasticsearch with proper field mappings."

"Create a log retention policy script that runs daily, deletes logs older than 30 days from Elasticsearch, and logs the deletion statistics."

Validation:

- [ ] Logs being collected from all services
- [ ] Logs indexed in Elasticsearch
- [ ] Can search logs in UI
- [ ] Filtering works correctly
- [ ] Old logs are deleted automatically

Step 14: Metrics & Monitoring (Week 27-30)

Goal: Collect metrics and create dashboards

Tasks:

- [ ] Set up Prometheus for metrics
- [ ] Add Prometheus client to all services

- [ ] Expose metrics endpoints
- [ ] Set up Grafana for visualization
- [ ] Create system health dashboard
- [ ] Create test execution metrics dashboard
- [ ] Set up alerting rules
- [ ] Add custom metric collection API

Tools & Technologies:

- **Metrics**: Prometheus
- **Visualization**: Grafana
- **Instrumentation**: Prometheus client libraries

Windsurf Prompts:

"Add Prometheus instrumentation to a FastAPI application. Include metrics for request count, request duration histogram, active connections, and custom business metrics like test_runs_total."

"Create a Grafana dashboard JSON configuration that displays API request rate, average response time, error rate percentage, and active test runs over time. Include panels for each metric."

"Generate Prometheus alerting rules that trigger when: API error rate > 5%, test execution queue has > 100 jobs waiting, database connection pool is exhausted, or any service is down."

"Build a React component that fetches metrics from Prometheus HTTP API, displays current values with trend indicators (up/down arrows), and renders line charts for the last 24 hours."

Validation:

- [ ] Metrics being collected from all services
- [ ] Grafana dashboards showing metrics
- [ ] Alerts trigger correctly
- [ ] Custom metrics can be added
- [ ] Metrics visible in UI

Step 15: Quality Gates (Week 30-32)

Goal: Implement configurable quality gates

Tasks:

- [ ] Create quality gate configuration API
- [ ] Build quality gate evaluation engine
- [ ] Evaluate gates on test completion
- [ ] Store gate results
- [ ] Create quality gate UI
- [ ] Add gate status to test runs
- [ ] Implement blocking/warning modes
- [ ] Add gate configuration presets

Windsurf Prompts:

"Create a quality gate evaluation engine that takes test run results and a quality gate configuration (conditions like coverage > 80%, no critical vulnerabilities, test pass rate > 95%) and returns pass/fail with detailed results for each condition."

"Generate a React component for configuring quality gates with a form that allows adding multiple conditions. Each condition should have: metric (dropdown), operator (>, <, =), threshold (number input), and severity (blocking/warning)."

"Build a FastAPI endpoint POST /api/v1/quality-gates/:gateId/evaluate that takes a test run ID, evaluates all conditions against the test results, and returns detailed pass/fail status with reasons."

"Create a visual quality gate status component in React that displays pass/fail status with a checkmark or X icon, lists all conditions with their results, and shows which conditions failed with specific values."

Validation:

- [ ] Can create and configure quality gates
- [ ] Gates evaluated automatically after test runs
- [ ] Failed gates shown prominently in UI

- [ ] Can set blocking vs. warning gates
- [ ] Gate history tracked over time

Step 16: Basic AI Features (Week 32-36)

Goal: Integrate Claude API for test failure explanations

Tasks:

- [ ] Set up Claude API client
- [ ] Create AI service
- [ ] Implement test failure explanation
- [ ] Add code context extraction
- [ ] Build AI recommendation UI
- [ ] Add caching for similar failures
- [ ] Implement rate limiting
- [ ] Add user feedback mechanism

Tools & Technologies:

- **AI API**: Claude (Anthropic)
- **SDK**: Anthropic Python/JavaScript SDK

Windsurf Prompts:

"Create a Python function that takes a failed test (name, error message, stack trace, code snippet) and uses Claude API to generate a human-friendly explanation of why the test failed and suggest possible fixes."

"Generate a FastAPI endpoint POST /api/v1/ai/explain-failure that accepts a test result ID, extracts relevant context (test code, error, recent changes), calls Claude API, caches the response in Redis, and returns the explanation."

"Build a React component that displays AI-generated test failure explanations with an expandable card showing: test name, explanation text, suggested fixes (bullet points), and thumbs up/down feedback buttons."

"Create a caching strategy for AI responses that hashes the test error signature (error type + message + stack trace pattern), checks Redis cache before calling Claude API, and stores responses with 7-day expiration."

Files to Create:

services/ai-service/

```
├── app/
│   ├── main.py
│   ├── routes/
│   │   └── ai.py
│   ├── services/
│   │   ├── claude_client.py
│   │   └── context_extractor.py
│   └── utils/
│       └── cache.py
├── requirements.txt
└── Dockerfile
```

Validation:

- [ ] Can request explanation for failed test
- [ ] Explanation is helpful and accurate
- [ ] Similar failures use cached responses
- [ ] Rate limiting prevents API abuse
- [ ] Can provide feedback on explanations

Step 17: Natural Language Queries (Week 36-40)

Goal: Allow querying data with natural language

Tasks:

- [ ] Build NLP query parser
- [ ] Create query intent classifier
- [ ] Implement query-to-SQL converter
- [ ] Add query-to-Elasticsearch converter
- [ ] Create conversational interface UI
- [ ] Add query suggestions
- [ ] Implement query history
- [ ] Add example queries

Windsurf Prompts:

"Create a function that uses Claude API to convert natural language queries like 'show me all failed tests in the last week' to PostgreSQL queries. Provide the database schema as context and ask Claude to generate safe, parameterized SQL."

"Build a conversational interface component in React that looks like a chat interface. Users type natural language questions, the system responds with data in table or chart format, and maintains conversation context."

"Generate a query parser that extracts intent (search, aggregate, compare, trend) and entities (time range, test status, project name, severity) from natural language, then constructs appropriate database or Elasticsearch queries."

"Create a query suggestion system that shows example queries like: 'What are the flakiest tests?', 'Show critical vulnerabilities added this week', 'Compare coverage between main and develop branches'."

Validation:

- [ ] Can ask questions in natural language
- [ ] System understands common queries
- [ ] Results displayed appropriately
- [ ] Conversation context maintained
- [ ] Query suggestions are helpful

Step 18: Notification System (Week 40-42)

Goal: Send alerts via email, Slack, webhooks

Tasks:

- [ ] Create notification service
- [ ] Implement email notifications
- [ ] Add Slack integration
- [ ] Add webhook support
- [ ] Create notification rules UI
- [ ] Implement notification templates
- [ ] Add notification preferences per user
- [ ] Test all notification channels

Tools & Technologies:

- **Email**: SendGrid or AWS SES
- **Slack**: Slack API
- **Webhooks**: Simple HTTP POST

Windsurf Prompts:

"Create a notification service in Python that can send notifications via multiple channels (email, Slack, webhook). Include methods: send_email(), send_slack_message(), send_webhook(). Use a factory pattern to select the appropriate channel."

"Generate a FastAPI endpoint POST /api/v1/notifications/rules that allows creating notification rules like: when test failure rate > 10%, notify via Slack; when critical vulnerability found, notify via email."

"Build a React component for managing notification preferences where users can toggle notifications on/off for different event types (test failures, security findings, coverage drops) and select their preferred channel (email, Slack, webhook)."

"Create Slack message templates for different events: test run completed (with pass/fail status), critical vulnerability detected (with severity and CVE ID), coverage dropped below threshold."

Validation:

- [ ] Email notifications sent correctly
- [ ] Slack messages appear in channels
- [ ] Webhooks trigger correctly
- [ ] Users can configure preferences
- [ ] Notifications not too noisy

Step 19: Dashboard Builder (Week 42-44)

Goal: Allow creating custom dashboards

Tasks:

- [ ] Create dashboard configuration storage
- [ ] Build widget library (charts, tables, metrics)
- [ ] Implement drag-and-drop dashboard editor
- [ ] Add dashboard sharing
- [ ] Create dashboard templates
- [ ] Implement data refresh
- [ ] Add export to PDF/image
- [ ] Build dashboard gallery

Windsurf Prompts:

"Create a React dashboard builder component using react-grid-layout for drag-and-drop. Include a widget library sidebar with available widgets (line chart, bar chart, metric card, table), and allow users to drag widgets onto the canvas."

"Generate a dashboard configuration API that stores dashboard layouts as JSON (widget types, positions, sizes, data queries). Include endpoints for: create, update, delete, list, and clone dashboards."

"Build a widget component system in React where each widget type (chart, table, metric) has a configuration modal for selecting data source, time range, filters, and display options. Use Recharts for visualizations."

"Create dashboard templates for common use cases: Test Execution Overview (test run trends, pass rate, duration), Security Dashboard (vulnerability counts by severity, recent findings), Code Quality (coverage trends, complexity metrics)."

Validation:

- [ ] Can create custom dashboards
- [ ] Drag-and-drop works smoothly
- [ ] Widgets display data correctly
- [ ] Dashboards can be shared
- [ ] Templates available for quick start

Step 20: Performance Optimization & Scale Prep (Week 44-48)

Goal: Optimize for production scale

Tasks:

- [ ] Add database indexes for slow queries
- [ ] Implement Redis caching for API responses
- [ ] Optimize frontend bundle size
- [ ] Add lazy loading for routes
- [ ] Implement pagination for large lists
- [ ] Set up database connection pooling
- [ ] Add CDN for static assets
- [ ] Optimize Docker images
- [ ] Add horizontal scaling support
- [ ] Load test the system

Tools & Technologies:

- **Load Testing**: k6 or Apache JMeter
- **Profiling**: Node.js profiler, Python cProfile
- **Monitoring**: Prometheus + Grafana

Windsurf Prompts:

"Analyze the PostgreSQL slow query log and suggest indexes to add. For the test_results table with frequent queries on (test_run_id, status), create appropriate composite indexes."

"Implement a Redis caching layer for the API Gateway that caches GET responses for 5 minutes. Include cache invalidation when related data is updated (e.g., invalidate project cache when project is updated)."

"Optimize the React application bundle by code-splitting routes, lazy loading components, and removing unused dependencies. Use Vite's build analyzer to identify large chunks and suggest improvements."

"Create a k6 load testing script that simulates 100 concurrent users creating test runs, viewing results, and searching logs. Measure response times, error rates, and identify bottlenecks."

"Add database connection pooling to the API Gateway using pg-pool with min 10, max 50 connections. Implement connection retry logic and timeout handling."

Validation:

- [ ] API response times under 200ms (p95)
- [ ] Frontend loads in under 3 seconds
- [ ] Can handle 100 concurrent test runs
- [ ] Database queries optimized
- [ ] No memory leaks under load

Phase 3: Production Ready (Months 13-18)

Step 21: Enterprise Features (Week 49-52)

Goal: Add SSO, RBAC, audit logging

Tasks:

- [ ] Implement SAML 2.0 SSO
- [ ] Add OAuth2 SSO (Google, GitHub)
- [ ] Enhance RBAC with custom roles
- [ ] Implement fine-grained permissions
- [ ] Add comprehensive audit logging
- [ ] Create audit log viewer
- [ ] Implement IP allowlisting
- [ ] Add session management

Tools & Technologies:

- **SSO**: passport-saml (Node.js)
- **OAuth**: passport-oauth2

Windsurf Prompts:

"Implement SAML 2.0 authentication in Express.js using passport-saml. Include endpoints for: SSO login redirect, ACS (Assertion Consumer Service) callback, SLO (Single Logout), and metadata endpoint."

"Create a comprehensive audit logging system that records all user actions (login, create/update/delete operations) with: user ID, IP address, user agent, action type, resource type, resource ID, before/after values, and timestamp."

"Build a fine-grained permission system where permissions are defined as: resource:action (e.g., project:read, test-run:create). Store role-permission mappings in database and implement a fast permission check using Redis cache."

"Generate a React component for viewing audit logs with filtering by: user, action type, date range, and resource. Display in a table with expandable rows showing detailed change history (before/after diff)."

Validation:

- [ ] Can login with Google/GitHub
- [ ] SAML SSO works with test IdP
- [ ] Custom roles can be created
- [ ] All actions are audit logged
- [ ] Audit logs are searchable

Step 22: Advanced Observability (Week 52-56)

Goal: Distributed tracing and APM

Tasks:

- [ ] Set up Jaeger for distributed tracing
- [ ] Instrument services with OpenTelemetry

- [ ] Add trace propagation between services
- [ ] Create service dependency map
- [ ] Build trace viewer UI
- [ ] Implement span correlation with logs
- [ ] Add performance profiling
- [ ] Create SLO tracking

Tools & Technologies:

- **Tracing**: Jaeger or Tempo
- **Instrumentation**: OpenTelemetry
- **APM**: OpenTelemetry + Grafana

Windsurf Prompts:

"Add OpenTelemetry instrumentation to FastAPI and Express.js services. Configure automatic instrumentation for HTTP requests, database queries, and Redis operations. Export traces to Jaeger."

"Create a service dependency map component in React that visualizes microservice calls based on trace data. Show services as nodes, requests as edges, with edge thickness representing call volume."

"Build a trace viewer UI in React that displays distributed traces with a waterfall chart showing span timing, parent-child relationships, and span details (service, operation, duration, tags, logs)."

"Generate a Python script that analyzes Jaeger traces to calculate p50, p95, p99 latencies for each service endpoint and identifies slow operations that exceed SLO thresholds."

Validation:

- [ ] Traces collected from all services
- [ ] Can view full request trace in UI
- [ ] Service dependencies visualized
- [ ] Can identify performance bottlenecks
- [ ] Logs correlated with traces

Step 23: Container Security Scanning (Week 56-58)

Goal: Scan Docker images for vulnerabilities

Tasks:

- [ ] Integrate Trivy for container scanning
- [ ] Scan images in CI/CD pipeline
- [ ] Store container vulnerabilities
- [ ] Create container security dashboard
- [ ] Add image signing verification
- [ ] Implement policy enforcement
- [ ] Add base image recommendations

Tools & Technologies:

- **Scanner**: Trivy
- **Registry**: Docker Hub or AWS ECR
- **Signing**: Docker Content Trust or Cosign

Windsurf Prompts:

"Create a GitHub Actions workflow that runs Trivy to scan Docker images for vulnerabilities before pushing to registry. Fail the build if critical vulnerabilities are found."

"Generate a Python script that uses Trivy JSON output to parse container vulnerabilities, extract CVE details, affected packages, and severity, then stores them in the database linked to the Docker image SHA."

"Build a container security dashboard in React showing: number of vulnerable images, vulnerabilities by severity, most vulnerable base images, and a list of images with their security scores."

"Create a policy engine that evaluates container images against security policies (e.g., no critical vulnerabilities, no outdated base images, no secrets in layers) and blocks deployment if policies fail."

Validation:

- [ ] Docker images scanned automatically
- [ ] Vulnerabilities stored and displayed
- [ ] Dashboard shows security posture
- [ ] Policies enforced in CI/CD
- [ ] Can track remediation progress

Step 24: Compliance & Reporting (Week 58-60)

Goal: Generate compliance reports

Tasks:

- [ ] Create report templates (SOC2, GDPR)
- [ ] Implement report generation service
- [ ] Add scheduled report generation
- [ ] Build report viewer UI
- [ ] Add PDF export
- [ ] Create compliance dashboard
- [ ] Implement compliance scoring
- [ ] Add evidence collection

Windsurf Prompts:

"Create a report generation service that produces compliance reports in HTML and PDF formats. Include sections for: executive summary, security findings, test coverage, vulnerabilities by severity, and compliance controls met."

"Generate a Python script using ReportLab or WeasyPrint that converts compliance report data (JSON) into a professional PDF with charts, tables, company branding, and section navigation."

"Build a compliance dashboard in React showing: overall compliance score (0-100), control categories (access control, encryption, logging, testing), status indicators (met/partial/not met), and trend over time."

"Create a scheduled job system using Celery that generates compliance reports weekly, sends them to configured recipients via email, and archives them in S3 for audit purposes."

Validation:

- [ ] Can generate SOC2 compliance report
- [ ] Reports include all required data
- [ ] PDF export looks professional
- [ ] Compliance score calculated correctly
- [ ] Reports sent automatically

Step 25: Multi-language Test Support (Week 60-63)

Goal: Support Java, Go, Ruby tests

Tasks:

- [ ] Add JUnit (Java) test runner
- [ ] Add Go test runner
- [ ] Add RSpec (Ruby) test runner
- [ ] Create language detection system
- [ ] Add language-specific coverage
- [ ] Update UI for language-specific info
- [ ] Test with real projects

Windsurf Prompts:

"Create a Python test runner plugin for JUnit that builds a Docker container with Java and Maven, executes tests, parses the JUnit XML report, and extracts test results, including test classes, methods, and execution times."

"Generate a Go test runner plugin that executes 'go test -json' in a Docker container, parses the JSON output line-by-line, and converts Go test events (run, pass, fail, skip) to standardized test results."

"Build an automatic language detection function that inspects a project directory and determines the primary language based on: presence of package.json (JavaScript), pom.xml (Java), go.mod (Go), Gemfile (Ruby), requirements.txt (Python)."

"Create a test framework registry system where each framework registers its detector, executor, and parser. When a test run is triggered, iterate through detectors to find matching frameworks and use the appropriate executor."

Validation:

- [ ] Can run JUnit tests
- [ ] Can run Go tests
- [ ] Can run RSpec tests
- [ ] Language detected automatically
- [ ] Coverage works for all languages

Step 26: Test Flakiness Detection (Week 63-65)

Goal: Identify and track flaky tests

Tasks:

- [ ] Track test history over multiple runs
- [ ] Calculate flakiness score
- [ ] Identify flaky tests
- [ ] Create flaky test dashboard
- [ ] Add flakiness indicators in UI
- [ ] Implement automatic retry logic
- [ ] Create flakiness report

Windsurf Prompts:

"Create a SQL query that calculates test flakiness score by analyzing the last 100 runs of each test. A test is flaky if it fails intermittently (passes some times, fails others without code changes). Return tests with flakiness > 0.1."

"Generate a Python function that analyzes test history and detects flaky tests using these criteria: test has passed and failed in the last 30 days, failure rate between 5-95%, no consistent error pattern."

"Build a flaky tests dashboard in React showing: count of flaky tests, most flaky tests (sorted by flakiness score), trend over time, and impact metrics (developer time wasted, false alarms)."

"Implement automatic retry logic for tests that executes failed tests up to 3 times. If a test passes on retry, mark it as 'flaky-pass'. Track retry statistics and alert when a test becomes consistently flaky."

Validation:

- [ ] Flaky tests identified correctly
- [ ] Flakiness score makes sense
- [ ] Dashboard shows flaky tests
- [ ] Auto-retry works for flaky tests
- [ ] Can track flakiness over time

Step 27: Infrastructure as Code Scanning (Week 65-67)

Goal: Scan Terraform/CloudFormation

Tasks:

- [ ] Integrate Checkov for IaC scanning
- [ ] Add Terraform scanning
- [ ] Add CloudFormation scanning
- [ ] Add Kubernetes manifest scanning
- [ ] Store IaC findings
- [ ] Create IaC security dashboard
- [ ] Add policy-as-code rules

Tools & Technologies:

- **IaC Scanner**: Checkov or tfsec
- **Alternative**: Terrascan

Windsurf Prompts:

"Create a Celery task that runs Checkov on Terraform files, parses the JSON output to extract policy violations (check ID, severity, resource, file path, line number), and stores findings in the database."

"Generate a Python parser for Checkov output that categorizes findings by cloud provider (AWS, GCP, Azure), resource type (S3, EC2, IAM), and compliance framework (CIS, PCI-DSS, HIPAA)."

"Build an IaC security dashboard in React showing: misconfigurations by severity, top violated policies, resources with most issues, compliance by framework, and remediation guidance for each finding."

"Create custom Checkov policies for organization-specific requirements like: all S3 buckets must have encryption, EC2 instances must use approved AMIs, IAM policies must not allow * actions."

Validation:

- [ ] Terraform files scanned successfully
- [ ] Misconfigurations detected
- [ ] Findings categorized correctly
- [ ] Dashboard shows IaC security posture
- [ ] Custom policies can be added

Step 28: API Rate Limiting & Quotas (Week 67-68)

Goal: Implement per-user/org quotas

Tasks:

- [ ] Implement token bucket rate limiting
- [ ] Add per-user quotas (API calls, test runs)
- [ ] Create quota tracking system
- [ ] Build quota dashboard
- [ ] Add quota alerts
- [ ] Implement quota enforcement
- [ ] Create upgrade prompts

Windsurf Prompts:

"Implement a token bucket rate limiter in Redis that allows: 100 requests per minute per user, with burst capacity of 150. Return rate limit headers (X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset) in API responses."

"Create a quota system that tracks usage per organization: API calls, test runs, storage used. Store monthly quotas in database and current usage in Redis. Implement middleware that checks quotas before processing requests."

"Build a usage dashboard in React showing: current month's usage vs. quota for each metric (API calls, test runs, storage), usage trends over time, projected end-of-month usage, and upgrade options if nearing limits."

"Generate alert notifications when organizations reach 80% and 95% of their quota. Include current usage, quota limit, days remaining in billing period, and link to upgrade plan."

Validation:

- [ ] Rate limiting works correctly
- [ ] Quotas enforced for all resources
- [ ] Users see usage in dashboard
- [ ] Alerts sent when approaching limits
- [ ] Graceful error messages when exceeded

Step 29: Documentation & Tutorials (Week 68-70)

Goal: Complete user and developer docs

Tasks:

- [ ] Write comprehensive user guide
- [ ] Create API documentation with examples
- [ ] Write integration guides for CI/CD
- [ ] Create video tutorials
- [ ] Write troubleshooting guide
- [ ] Document architecture decisions
- [ ] Create contribution guide

- [ ] Build documentation site

Tools & Technologies:

- **Docs**: Docusaurus or VitePress
- **API Docs**: Swagger/OpenAPI
- **Videos**: Loom or OBS Studio

Windsurf Prompts:

"Create a Docusaurus documentation site structure with sections: Getting Started, User Guide, API Reference, Integration Guides, Troubleshooting, Architecture, and Contributing. Include sidebar navigation and search."

"Generate comprehensive API documentation for all endpoints using OpenAPI 3.0 specification. Include: endpoint descriptions, request/response schemas, authentication requirements, example requests with curl, and response codes."

"Write a step-by-step integration guide for GitHub Actions that shows: setting up API token, adding workflow file, configuring test execution, viewing results in the platform, and setting up quality gates."

"Create a troubleshooting guide covering common issues: connection failures, test execution errors, authentication problems, performance issues. For each issue, include: symptoms, possible causes, and solutions with code examples."

Validation:

- [ ] Documentation is complete and clear
- [ ] All features are documented
- [ ] API examples work correctly
- [ ] Integration guides tested
- [ ] Search works in docs site

Step 30: Beta Testing & Final Polish (Week 70-72)

Goal: Beta testing with real users

Tasks:

- [ ] Recruit 5-10 beta users
- [ ] Create beta onboarding process
- [ ] Set up feedback collection system
- [ ] Monitor beta usage closely
- [ ] Fix critical bugs found
- [ ] Address user feedback
- [ ] Optimize based on real usage
- [ ] Prepare for public launch

Windsurf Prompts:

"Create a user onboarding flow that guides new users through: creating their first organization, adding a project, connecting a repository, running their first test, and viewing results. Include interactive tooltips and progress indicators."

"Build a feedback widget in React that allows users to report bugs, request features, or provide feedback from anywhere in the app. Include: feedback type selector, description textarea, screenshot capture, and email notification to team."

"Generate an analytics dashboard for monitoring beta usage: active users per day, features used, error rates, average session duration, and funnel analysis (signup → first project → first test run)."

"Create a bug triage system that categorizes issues by: severity (critical/high/medium/low), affected users, frequency, and impact. Automatically prioritize critical bugs affecting multiple users."

Validation:

- [ ] Beta users onboarded successfully
- [ ] Collecting regular feedback
- [ ] Critical bugs fixed within 24 hours
- [ ] Usage analytics look healthy
- [ ] Ready for public launch

Deployment & Infrastructure Setup

Kubernetes Deployment (Optional - If using K8s)

Tasks:

- [ ] Create Kubernetes cluster (EKS, GKE, or AKS)
- [ ] Set up Helm charts for each service
- [ ] Configure ingress controller
- [ ] Set up secrets management
- [ ] Configure horizontal pod autoscaling
- [ ] Set up persistent volumes
- [ ] Configure monitoring and logging
- [ ] Set up backup and disaster recovery

Windsurf Prompts:

"Create a Helm chart for the API Gateway service including: deployment with 3 replicas, service, horizontal pod autoscaler (CPU > 70%), configmap for environment variables, and secret for database credentials."

"Generate Kubernetes manifests for the entire platform including: namespace, deployments for all services, services, ingress with TLS, persistent volume claims for databases, and configmaps for configuration."

"Create a backup strategy for Kubernetes that: daily backups of PostgreSQL using pg_dump to S3, Redis RDB snapshots, Elasticsearch snapshots, and retention policy of 30 days."

Simple Docker Compose Deployment (Recommended for small teams)

Tasks:

- [ ] Create production docker-compose.yml
- [ ] Set up reverse proxy (nginx)
- [ ] Configure SSL certificates (Let's Encrypt)
- [ ] Set up automated backups
- [ ] Configure log rotation
- [ ] Set up monitoring
- [ ] Create deployment scripts

- [ ] Document deployment process

Windsurf Prompts:

"Create a production-ready docker-compose.yml that includes: API gateway, test executor, analysis engine, AI service, PostgreSQL with volume, Redis with persistence, Elasticsearch, nginx reverse proxy with SSL, and Prometheus + Grafana."

"Generate an nginx configuration for reverse proxying the platform with: SSL termination, rate limiting, gzip compression, caching of static assets, websocket support, and security headers."

"Create a backup script that runs daily via cron: dumps PostgreSQL to S3, backs up Redis RDB file, exports Elasticsearch indices, and retains last 30 days of backups. Include restore instructions."

"Write a deployment script (deploy.sh) that: pulls latest Docker images, runs database migrations, gracefully restarts services with zero downtime, runs health checks, and rolls back if health checks fail."

Tools & Technologies Summary

Core Stack

- **Backend API**: Node.js + TypeScript + Express.js
- **Test Executor**: Python + FastAPI + Celery
- **Frontend**: React + TypeScript + Vite + Tailwind CSS
- **Database**: PostgreSQL + Prisma ORM
- **Cache/Queue**: Redis
- **Search**: Elasticsearch
- **Storage**: MinIO (local) or S3 (production)

Additional Services

- **Metrics**: Prometheus + Grafana
- **Tracing**: Jaeger or Tempo

- **AI**: Claude API (Anthropic)
- **Security Scanners**: OWASP Dependency-Check, Semgrep, TruffleHog, Trivy, Checkov

Development Tools

- **IDE**: VSCode with Windsurf AI extension
- **Version Control**: GitHub
- **CI/CD**: GitHub Actions
- **Containers**: Docker + Docker Compose
- **Testing**: Jest, pytest
- **API Testing**: Postman or Insomnia

Deployment Options

- **Simple**: Docker Compose on VPS (DigitalOcean, Linode)
- **Advanced**: Kubernetes (EKS, GKE, AKS)
- **Serverless**: AWS Lambda + API Gateway (for API only)

Time Estimates

MVP (Months 1-6):

- Core test execution: 6-8 weeks
- Basic security scanning: 2-3 weeks
- UI foundation: 4-5 weeks
- Code coverage: 2 weeks
- Polish: 2-3 weeks

Enhanced Features (Months 7-12):

- Additional test frameworks: 3 weeks
- Advanced security: 3 weeks
- Log management: 3-4 weeks
- Metrics: 3-4 weeks
- Quality gates: 2 weeks
- Basic AI: 4 weeks
- NLP queries: 4 weeks
- Notifications: 2 weeks
- Dashboard builder: 2 weeks
- Performance optimization: 4 weeks

Production Ready (Months 13-18):

- Enterprise features: 3 weeks
- Advanced observability: 4 weeks
- Container security: 2 weeks
- Compliance: 2 weeks
- Multi-language support: 3 weeks
- Flakiness detection: 2 weeks
- IaC scanning: 2 weeks
- Rate limiting: 1 week
- Documentation: 2 weeks
- Beta testing: 2 weeks

Total: 12-18 months with 1-2 developers

Success Tips for Small Teams

1. **Use Windsurf AI aggressively**: Let AI write boilerplate code, tests, and documentation
2. **Leverage existing tools**: Don't build from scratch what already exists
3. **Focus on MVP**: Ship early, iterate based on feedback
4. **Automate everything**: CI/CD, testing, deployments, backups
5. **Keep it simple**: Docker Compose over Kubernetes initially
6. **Document as you go**: Future you will thank present you
7. **Use managed services**: Managed databases, managed Kubernetes when possible
8. **Monitor from day one**: Catch issues before users report them
9. **Regular demos**: Show progress weekly to stay motivated
10. **Don't over-engineer**: Build for today's scale, not tomorrow's

---

Daily Workflow

Morning (9 AM - 12 PM):

1. Check monitoring dashboards for issues
2. Review and respond to user feedback
3. Pick highest priority task from backlog
4. Write code using Windsurf AI assistance
5. Write tests for new code
6. Submit PR

Afternoon (1 PM - 5 PM):

1. Code review (if 2 developers)
2. Continue feature development
3. Update documentation
4. Fix bugs from testing
5. Deploy to staging
6. End of day: commit and push

Weekly:

- Monday: Sprint planning, prioritize week's tasks
- Wednesday: Mid-week sync, adjust priorities
- Friday: Demo completed work, retrospective

This plan is designed to be flexible. Adjust timelines based on your actual velocity and priorities. The key is consistent progress, not perfect adherence to the schedule.