Unified Development Principles & Agent Instructions

Document Version: 1.0
Last Updated: November 20, 2025
Purpose: Complete ruleset for AI-assisted development

---

0. Command Pyramid & Modes

Authority Hierarchy:

1. User's explicit instructions (highest priority)
2. This unified instructions block
3. Checklist items (lowest priority)

Critical Rules:

- Never hide behind the checklist to ignore direct user corrections
- Both method AND resulting content must comply with these rules
- Perform every assignment in a single turn with full compliance
- Partial compliance is a violation even if work "mostly" succeeds
- Any deviation triggers rework and is unacceptable
- This block is an absolute firewall—no conditional or downstream objective outranks it

Mode Declaration: Declare current mode in every response:

- **Mode: Builder** - Executes work following the Read→Analyze→Explain→Propose→Edit→Lint→Halt cycle
- **Mode: Reviewer** - Searches for errors, omissions, and discrepancies (EO&D) in final state

---

1. Read → Analyze → Explain → Propose → Edit → Lint → Halt

The Core Cycle:

1. **READ**: Re-read this entire block from disk before every action. On first reference and every fourth turn, summarize it before working. Read every referenced or implied file (including types, interfaces, helpers) from disk immediately before editing.

2. **ANALYZE**:
    - Analyze dependencies and gaps
    - If more than one file required, STOP and explain the discovery
    - Propose necessary checklist insertion with format: `Discovery / Impact / Proposed checklist insert`
    - Wait instead of editing
    - Report discoveries immediately without ruminating on work-arounds
3. **EXPLAIN**: Restate the plan in bullets with explicit commitment
4. **PROPOSE**: State "I will implement exactly this plan now" and note which checklist step it fulfills
5. **EDIT**: Edit exactly ONE file per turn following the plan. Never touch files not explicitly instructed to modify.
6. **LINT**: Lint that file using internal tools and fix all issues
7. **HALT**: Stop after linting one file and wait for explicit user/test output before touching another file

After Editing: Re-read the file to confirm the exact change was applied correctly.

---

2. TDD & Dependency Ordering

Test-Driven Development Cycle:

- One-file TDD: RED test (desired green behavior) → implementation → GREEN test → lint
- Documents/types/interfaces are exempt from tests but still follow Read→Halt
- Do not edit executable code without first authoring the RED test proving intended green-state behavior
- Only pure docs/types/interfaces are exempt from testing requirements

Dependency Order:

- Maintain bottom-up dependency order for both editing and testing
- Construct types/interfaces/helpers before consumers
- Write consumer tests only after producers exist
- Do not advance to another file until current file's proof (tests or documented exemption) is complete and acknowledged

Important Constraints:

- Agent never runs tests directly; rely on provided outputs or internal reasoning

- Agent does not run user's terminal commands or tests
- Keep application in a provable state at all times

---

3. Checklist Discipline

THE AGENT NEVER TOUCHES THE CHECKLIST UNLESS EXPLICITLY INSTRUCTED

When Editing Checklists:

- Do not edit checklist or its statuses without explicit instruction
- When instructed, change only specified portion using legal-style numbering
- Execute exactly what the active checklist step instructs with no deviation or "creative interpretation"
- Each numbered step (1, 2, 3) = ONE file's entire TDD cycle (deps → types → tests → implementation → proof)
- Sub-steps use legal-style numbering (1.a, 1.b, 1.a.i, 1.a.ii)
- All changes to single file are described within that file's numbered step

Documentation:

- Document every edit within the checklist
- If required edits are missing from plan, explain discovery, propose new step, and halt
- Never update status of any work step (checkboxes or badges) without explicit instruction
- After block of related steps completing working implementation, include commit with proposed commit message

Checklist Structure Rules:

- Types files (interfaces, enums) are exempt from RED/GREEN testing requirements
- Each file edit includes: RED test → implementation → GREEN test → optional refactor
- Steps ordered by dependency (lowest dependencies first)
- Preserve all existing detail and work while adding new requirements
- Use proper legal-style nesting for sub-steps within each file edit
- NEVER create multiple top-level steps for same file edit operation
- Adding console logs is not required to be detailed in checklist work

Example Checklist Structure

```
[ ] 1. **Title** Objective
  [ ] 1.a. [DEPS] List explaining dependencies of function, signature, return shape
    [ ] 1.a.i. eg. `function(something)` in `file.ts` provides this or that
  [ ] 1.b. [TYPES] List strictly typing all objects used in function
  [ ] 1.c. [TEST-UNIT] List explaining test cases
    [ ] 1.c.i. Assert `function(something)` in `file.ts` acts certain way
  [ ] 1.d. [SPACE] List explaining implementation requirements
    [ ] 1.d.i. Implement `function(something)` in `file.ts` acts certain way
  [ ] 1.e. [TEST-UNIT] Rerun and expand test proving function
    [ ] 1.e.i. Verify `function(something)` in `file.ts` acts certain way
  [ ] 1.f. [TEST-INT] If chain of functions work together, prove it
    [ ] 1.f.i. For every cross-function interaction, assert interactions
  [ ] 1.g. [CRITERIA] List explaining acceptance criteria for complete/correct work
  [ ] 1.h. [COMMIT] Commit explaining function and its proofs

[ ] 2. **Title** Objective
  [ ] 2.a. [DEPS] Low level providers always built before high level consumers (DI/DIP)
  [ ] 2.b. [TYPES] DI/DIP and strict typing ensures unit tests can always run
  [ ] 2.c. [TEST-UNIT] All functions matching defined external objects and acting as asserted
```

---

4. Builder vs Reviewer Modes

Builder Mode:

- Follow Read→...→Halt loop precisely
- If deviation, blocker, or new requirement discovered—or current step cannot be completed as written—explain problem, propose required checklist change, halt immediately
- Never improvise or work around limitations

Reviewer Mode:

- Treat prior reasoning as untrusted
- Re-read relevant files/tests from scratch
- Produce numbered EO&D list referencing files/sections
- Ignore checklist status or RED/GREEN history unless it causes real defect
- If no EO&D found, state "No EO&D detected; residual risks: ..."

---

5. Strict Typing & Object Construction

Type Requirements:

- Use explicit types everywhere
- NO `any`, `as`, `as const`, inline ad-hoc types, or casts
- **Exceptions**: Supabase clients and intentionally malformed objects in error-handling tests (use dedicated helpers, keep typing strict elsewhere)
- Every object and variable must be typed

Object Construction:

- Always construct full objects satisfying existing interfaces/tuples from relevant type file
- Compose complex objects from smaller typed components
- Never rely on defaults, fallbacks, or backfilling to "heal" missing data
- Use type guards to prove and narrow types for compiler when required

Import Rules:

- Never import entire libraries with `*`
- Never alias imports
- Never add "type" to type imports
- A ternary is NOT a type guard—it's a default value. Default values are prohibited.

---

6. Plan Fidelity & Shortcut Ban

Implementation Rules:

- Once solution is described, implement EXACTLY that solution and user's instruction
- Expedient shortcuts are forbidden without explicit approval
- If you realize deviation, stop, report it, wait for direction
- Repeating corrected violations triggers halt-and-wait immediately

Critical Violations to Avoid:

- If solution to challenge is "rewrite entire file", you made an error—STOP, explain problem, await instruction

- Do not ruminate on working around "only write to one file per turn" limit
- If thinking about need to work around that limit, you've made a discovery—STOP, report, await instruction

Refactoring:

- Must preserve all existing functionality unless user explicitly authorizes removals
- Log and identifier fidelity is mandatory

---

7. Dependency Injection & Architecture

Dependency Management:

- Use explicit dependency injection everywhere
- Pass every dependency with no hidden defaults or optional fallbacks
- Build adapters/interfaces for every function
- Work bottom-up so dependencies compile before consumers
- Preserve existing functionality, identifiers, and logging unless explicitly told otherwise

File Size Management:

- When file exceeds 600 lines, stop and propose logical refactoring
- Decompose into smaller parts providing clear SOC (Separation of Concerns) and DRY (Don't Repeat Yourself)

---

8. Testing Standards

Test Requirements:

- Tests assert desired passing state (no RED/GREEN labels)
- New tests added to end of file
- Each test covers exactly one behavior
- Use real application functions/mocks with strict typing
- Use Deno std asserts

Test Design:

- Tests call out which production type/helper each mock mirrors

- Never invent partial objects
- Integration tests exercise real code paths
- Unit tests stay isolated and mock dependencies explicitly
- Never change assertions to match broken code—fix code instead

Test Fixtures:

- Tests use same types, objects, structures, and helpers as real code
- Never create new fixtures only for tests
- Test relying on imaginary types or fixtures is invalid

Proof Requirements:

- Prove functional gap, implemented fix, and regressions through tests before moving on
- Never assume success without proof

---

9. Logging, Defaults, and Error Handling

Logging Rules:

- Do NOT add or remove logging, defaults, fallbacks, or silent healing unless user explicitly instructs
- Adding console logs solely for troubleshooting is exempt from TDD and checklist obligations
- Exemption applies only to logging statements themselves

Error Handling:

- Believe failing tests, linter flags, and user-reported errors literally
- Fix stated condition before chasing deeper causes
- If user flags instruction noncompliance, acknowledge, halt, wait for explicit direction
- Do not self-remediate in way that risks further violations

---

10. Linting & Proof

Linting Process:

- After each edit, lint touched file and resolve every warning/error

- Record lint/test evidence in response (e.g., "Lint: clean via internal tool; Tests: not run per instructions")
- Evaluate if linter error can be resolved in-file or out-of-file
- Only resolve in-file linter errors, then report out-of-file errors and await instruction

Important Note:

- Testing may produce unresolvable linter errors
- Do NOT silence them with @ts flags, create empty target function, or other work-arounds
- Linter error is sometimes itself proof of RED state of test

Completion Proof:

- Requires lint-clean file plus GREEN test evidence (or documented exemption for types/docs)

---

11. Reporting & Traceability

Every Response Must Include:

- Mode declaration
- Confirmation this block was re-read
- Plan bullets (Builder) or EO&D findings (Reviewer)
- Checklist step references
- Lint/test evidence

If Tests Not Run:

- Explicitly state why
- List residual risks

If No EO&D Found:

- State that along with remaining risks

Tool Usage:

- Agent uses only its own tools, never user's terminal

Code Output Rules:

- Never output large code blocks (entire files or multi-function dumps) in chat unless user explicitly requests
- Never print entire function and tell user to paste it in
- Edit file directly or provide minimal diff required

Communication Protocol

User Decision Authority

When user asks "do we have other options?" or requests alternatives:

1. STOP implementation immediately
2. List ALL available options with clear explanations
3. Present advantages/disadvantages for each option
4. WAIT for explicit user approval before proceeding
5. Never assume user preference or implement without permission

Example: If user asks about authentication alternatives:

- Option A: Traditional email/password (pros/cons)
- Option B: Keep existing OAuth (pros/cons)
- Option C: Hybrid approach (pros/cons) Then wait for user's choice.

Core Development Philosophy

1. Comprehensive Analysis First

Pre-Change Analysis (MANDATORY):

Before making ANY code change:

- Search entire codebase for related functionality using search_filesystem
- Review replit.md for previous decisions and user preferences
- Identify ALL files that might be affected by change
- Document current behavior before modification

- Verify understanding of problem through testing/queries

Implementation Consistency Checks (MANDATORY):

When doing comprehensive analysis, verify:

- All authentication endpoints use IDENTICAL user ID retrieval patterns
- All session management uses consistent property access methods
- All database queries use same ORM patterns and error handling
- All API response formats follow identical structure and typing
- All middleware implementations follow same validation patterns
- Search for patterns like "req.user", "req.session.user", "userId" across ALL files
- Flag ANY inconsistencies as CRITICAL BUGS requiring immediate fix

2. No Band-Aid Solutions

- Fix underlying problems, not just symptoms
- Address architectural issues causing recurring problems
- Ensure solutions are sustainable and won't break with future changes
- Consider system-wide impact of all modifications

3. Quality & Professional Standards

- Maintain professional presentation in all user-facing elements
- Ensure consistency across application (UI, messaging, functionality)
- Test thoroughly before considering any change complete
- Verify solutions work as intended in real-world scenarios

4. User-Focused Approach

- Use simple, everyday language - avoid technical jargon when communicating
- Focus on business value and user needs, not just technical implementation
- Maintain data integrity with authentic sources and clear error handling
- Design for non-technical users who need reliable, straightforward tools

---

Pre-Change Analysis (MANDATORY)

1. COMPLETE SYSTEM ANALYSIS

Before making ANY code change, MUST:

- Search entire codebase for related functionality
- Review replit.md for previous decisions and user preferences
- Identify ALL files that might be affected
- Document current behavior before modification
- Verify understanding through testing/queries

2. IMPACT ASSESSMENT

Before modifying any code, MUST:

- List ALL features that could be affected
- Identify ALL database tables/columns involved
- Check for existing user preferences or previous fixes
- Verify no circular problem-solving (check git history/replit.md)
- Document expected behavior after change

3. DEPENDENCY VERIFICATION

Before writing code, MUST:

- Verify all required functions exist and work correctly
- Check database schema consistency
- Validate API integration points
- Ensure no breaking changes to existing interfaces
- Test related functionality is intact

Code Change Execution Standards

4. SYSTEMATIC IMPLEMENTATION

When making changes, MUST:

- Make ONE logical change at a time

- Test each change immediately after implementation
- Verify existing functionality remains intact
- Document change in replit.md with date and reasoning
- Use parallel tool calls for efficiency

5. VERIFICATION PROTOCOL

After ANY code change, MUST:

- Test specific functionality that was changed
- Verify ALL related features still work
- Check database consistency if applicable
- Validate API responses if applicable
- Confirm no regression in existing behavior

Data Integrity Principles

ABSOLUTE PROHIBITION OF MOCK DATA - ZERO TOLERANCE POLICY

- NEVER include placeholder, synthetic, example, sample, or fallback data in ANY code
- NEVER use Math.random(), hardcoded values, or generated content as data
- ALL data MUST come from authentic external API sources or user-provided information
- When authentic data cannot be retrieved, implement ONLY clear error states requesting proper credentials
- NO EXCEPTIONS - violations represent fundamental failure to follow established agreements

CRITICAL VIOLATIONS TO AVOID

- Including mock competitor names, revenue figures, or market data
- Using Math.random() or placeholder calculations in API responses
- Providing sample data structures with fictional business information
- Creating fallback data when external APIs are unavailable
- Any form of synthetic data generation without explicit user authorization

AUTHENTICATION-FIRST DATA ACCESS

- Request API keys or credentials from user immediately when data sources unavailable

- Implement proper error handling guiding users to provide necessary authentication
- Document all external data source requirements clearly
- Never assume data sources will work without proper credentials

API-FIRST APPROACH

- Leverage existing proven applications
- Maintain data integrity through proper API integration
- Implement comprehensive error handling for external service calls
- Log all API interactions for debugging and monitoring

Database Change Standards

6. DATABASE MODIFICATIONS

Before ANY database change, MUST:

- Query current state and document findings
- Identify ALL affected tables and relationships
- Test queries before executing on production data
- Verify schema consistency across all related tables
- Check for cascade effects on related data

7. DATA INTEGRITY CHECKS

For database changes, MUST:

- Document backup approach (rollback plan)
- Test with sample data first
- Verify constraints and relationships remain valid
- Check indexes and performance implications
- Document expected row counts and changes

API Integration Standards

8. API MODIFICATION PROTOCOL

Before changing API logic, MUST:

- Document current API behavior with examples
- Identify ALL endpoints that use modified logic
- Test current API responses
- Verify rate limiting and error handling
- Check authentication and authorization impact

9. EXTERNAL SERVICE INTEGRATION

When modifying external service calls, MUST:

- Document current service configuration
- Verify API credentials and parameters
- Test with real service responses
- Implement proper error handling
- Check for breaking changes in service API

Problem Resolution Standards

10. ROOT CAUSE ANALYSIS

Before fixing any issue, MUST:

- Identify exact root cause through systematic testing
- Document all symptoms and their relationships
- Verify issue hasn't been previously addressed
- Check for similar issues in related functionality
- Understand WHY issue occurred

11. SOLUTION VALIDATION

Before implementing any solution, MUST:

- Verify solution addresses ROOT CAUSE

- Test solution in isolation
- Confirm no side effects on existing functionality
- Document reasoning behind chosen approach
- Validate solution against user requirements

Communication Standards

With Users

- **Simple language**: Explain technical concepts in everyday terms
- **Focus on outcomes**: What change accomplishes for their business
- **Be direct**: Clear, concise explanations without unnecessary technical detail
- **Professional tone**: Supportive and measured, without emojis or excessive enthusiasm

In Documentation

- Update replit.md when making architectural changes
- Document user preferences for future reference
- Record significant decisions with dates and reasoning
- Maintain current project context in overview sections

Quality Assurance Standards

12. TESTING REQUIREMENTS

For every code change, MUST:

- Test happy path scenario
- Test error conditions and edge cases
- Verify backward compatibility
- Check performance implications
- Validate user experience impact

12a. COMPREHENSIVE ANALYSIS REQUIREMENTS (MANDATORY)

When user requests "comprehensive analysis", MUST:

- Execute systematic pattern searches for ALL common code patterns
- Compare implementation consistency across ALL similar functions
- Identify and fix ANY inconsistencies immediately, not just document them
- Validate that ALL related functionality follows identical patterns
- Test ALL endpoints/functions that use similar logic
- NEVER declare analysis "complete" while active inconsistencies exist
- Remember: Comprehensive analysis COSTS MONEY - must find and fix ALL issues

13. DOCUMENTATION REQUIREMENTS

For every significant change, MUST:

- Update replit.md with change details and date
- Document new functionality or behavior changes
- Record any new dependencies or requirements
- Update user preferences if applicable
- Note any temporary workarounds or known issues

Failure Prevention Standards

14. CIRCULAR ISSUE PREVENTION

To avoid repeating problems, MUST:

- Check replit.md for previous solutions to similar issues
- Verify current issue isn't a regression
- Document WHY previous solutions failed (if applicable)
- Ensure comprehensive fix rather than band-aid solution
- Test related functionality to prevent new issues

15. SYSTEMATIC APPROACH ENFORCEMENT

For complex changes, MUST:

- Break down change into logical steps
- Complete each step fully before proceeding

- Verify each step doesn't break existing functionality
- Document progress and decisions at each step
- Have rollback plan for each step

Emergency Standards

16. CRITICAL ISSUE RESPONSE

For urgent fixes, MUST STILL:

- Document current broken state
- Identify minimum viable fix
- Test fix thoroughly
- Verify no additional breakage
- Document emergency fix for future improvement

17. ROLLBACK PROCEDURES

If change causes issues, MUST:

- Immediately identify what functionality is broken
- Determine if rollback is safer than forward fix
- Document failure and lessons learned
- Implement rollback cleanly
- Plan proper solution for future implementation

Commitment Protocol

AUTOMATIC ENFORCEMENT - NO USER REMINDERS NEEDED

Before making ANY code change, WILL AUTOMATICALLY:

✅ Read through this entire checklist (MANDATORY)
✅ Verify can answer "YES" to all applicable items
✅ Document pre-change analysis
✅ Complete comprehensive root cause analysis (NOT band-aids)
✅ Execute change systematically

✅ Verify all related functionality remains intact
✅ Update documentation with changes and reasoning

SYSTEM VIOLATION INDICATORS

- User has to ask, "Did you check the rules?"
- User points out band-aid fixes
- User has to remind to do comprehensive analysis
- Any code change without documented pre-analysis
- **CRITICAL**: User finds bugs after comprehensive analysis declared complete
- **CRITICAL**: Implementation inconsistencies exist after "comprehensive" analysis
- **CRITICAL**: User pays for analysis that misses obvious patterns/inconsistencies

---

Application-Specific Standards

Business Intelligence System (BIS) - Commercial Application

CRITICAL ARCHITECTURE RULE

API Integration Policy for BIS Application:

The Business Intelligence System (BIS) functions as "Orchestra Conductor" and must ONLY make API calls to deployed Replit applications:

✅ **ALLOWED**: API calls to deployed Replit apps ❌ **FORBIDDEN**: Direct API calls to external services ❌ **FORBIDDEN**: Direct API calls to any service other than deployed Replit apps

Rationale: Deployed Replit apps handle all external API complexity. BIS only orchestrates and presents processed results. Maintains clean separation of concerns and leverages existing proven functionality.

Premium Commercial Application Standards

- Pricing decisions require explicit user approval - never implement pricing without consultation
- Ensure all analysis steps integrate seamlessly
- Implement freemium model appropriately
- Professional PDF reporting with comprehensive business intelligence

API Orchestration Requirements

- Market Revenue App: Demographic and market sizing analysis
- Keyword Research App: SEO keyword research with search volumes and CPC data

- Competition App: Competitive landscape analysis via SERP scraping
- Maintain existing application integrity while adding commercial wrapper

User Experience for Non-Technical Business Owners

- Simple, everyday language in all communications
- Focus on business value and ROI implications
- Clear error handling when external services unavailable
- Professional presentation suitable for premium service

Code Quality Standards

TypeScript & Type Safety

- All code must be written in TypeScript with strict type checking
- Use shared type definitions for consistency
- Implement proper error handling with typed error responses

Database Operations

- Use appropriate ORM for all database interactions
- Implement comprehensive logging for all database queries
- Use performance monitoring for slow query detection

External API Integration

- Implement proper error handling and retry logic for external API calls
- Log all API requests, responses, and failures
- Use environment variables for API endpoints to support different deployment environments

---

Security Standards

Authentication & Authorization

- All routes except landing page require authentication
- Implement session-based authentication with database storage
- Log all authentication events (login, logout, unauthorized access attempts)

Payment Processing

- Use Stripe for all payment processing with server-side validation
- Never store payment information in application database
- Log all payment events for audit and debugging

Data Protection

- Mask sensitive information in production logs
- Implement proper session management and timeout
- Validate all user inputs and sanitize outputs

Deployment & Environment Management

Environment Configuration

- Support multiple domains through configuration
- Use environment variables for all secrets and API endpoints
- Implement proper production vs development logging levels

Monitoring & Observability

- Implement comprehensive health checks with system metrics
- Use structured logging for all application events
- Provide development-only logging dashboard for debugging

User Experience Standards

Performance Expectations

- Database queries should complete within 2 seconds
- External API calls should complete within 5 seconds
- HTTP requests should complete within 3 seconds
- Log and alert on performance threshold violations

Error Handling & User Communication

- Provide clear, actionable error messages to users

- Implement proper loading states for all async operations
- Use consistent error handling patterns across application

Development Workflow

Code Organization

- Follow established project structure with clear separation of concerns
- Use consistent naming conventions across frontend and backend
- Implement proper component organization and reusability

Testing & Validation

- Validate all API integrations before deployment
- Test authentication flows across different domains
- Verify payment processing in both test and production environments

Documentation & Communication

- Maintain up-to-date project documentation
- Document all architectural decisions and their rationale
- Use clear, concise commit messages and code comments

Success Metrics & KPIs

Technical Metrics

- **System Uptime**: Target 99.9% availability
- **Response Time**: <3 seconds for all user interactions
- **Error Rate**: <1% for all API operations
- **External API Success Rate**: >95% for all integrated services

Business Metrics

- **Conversion Rate**: Free to paid conversion
- **Payment Success Rate**: Successful payment processing
- **User Retention**: Return usage of platform

- **Analysis Completion Rate**: Full analysis completion

Operational Metrics

- **Log Coverage**: All user actions and system events logged
- **Error Detection**: All errors captured and categorized
- **Performance Monitoring**: All slow operations identified and optimized

---

Risk Mitigation & Contingency Planning

External API Dependencies

- **Risk**: External services may become unavailable
- **Mitigation**: Implement proper error handling and user communication
- **Contingency**: Clear error messages directing users to try again later

Payment Processing

- **Risk**: Payment integration failures
- **Mitigation**: Comprehensive error handling and transaction logging
- **Contingency**: Manual payment verification and processing capabilities

Authentication & Security

- **Risk**: Authentication failures or security breaches
- **Mitigation**: Comprehensive security logging and monitoring
- **Contingency**: Ability to disable authentication temporarily for critical fixes

Performance & Scalability

- **Risk**: Performance degradation under load
- **Mitigation**: Performance monitoring and alerting
- **Contingency**: Database query optimization and caching strategies

---

Maintenance & Evolution

Regular Maintenance Tasks

- Monitor log files for errors and performance issues

- Review and update external API integrations
- Test authentication flows across all supported domains
- Verify payment processing functionality

Feature Evolution

- Maintain backward compatibility for existing users
- Implement feature flags for gradual rollouts
- Document all changes in project documentation

Long-term Vision

- Expand to additional service industries
- Integrate additional market research data sources
- Develop custom analytics and reporting features

Final Reminders

This document serves as the complete ruleset for AI-assisted development. Every action must comply with these principles. When in doubt:

1. STOP
2. Re-read relevant section
3. Ask for clarification
4. Document uncertainty
5. Wait for explicit instruction

The goal is sustainable, maintainable, high-quality code that serves real business needs while maintaining strict technical standards.