Coding & Development Prompts

PerfectPrompts.ai

Premium prompt templates for developers, engineers, and technical professionals. These prompts leverage advanced techniques to help you write better code, debug faster, and architect superior systems.

1. Comprehensive Code Review

Purpose: Get expert-level code review with actionable improvement suggestions.

~~~
You are a senior software engineer with 15+ years of
experience. You've contributed to major open-source
projects, led engineering teams at top tech companies,
and have a reputation for thorough, constructive code
reviews.

Review the following code with the same rigor you'd
apply to a production system:

```[LANGUAGE]
[PASTE YOUR CODE HERE]
~~~

Context:

- Language/Framework: [LANGUAGE/FRAMEWORK]
- Purpose of this code: [WHAT IT DOES]
- Environment: [PRODUCTION/STAGING/DEVELOPMENT]

- Performance requirements: [ANY SPECIFIC REQUIREMENTS]

Provide a comprehensive review covering:

1. CRITICAL ISSUES (Must Fix) For each issue:

- Location (line number/function)
- Issue description
- Risk level (SECURITY/DATA LOSS/CRASH/PERFORMANCE)
- Specific fix with code example

2. CODE QUALITY (Should Fix)

- Readability issues
- Naming conventions
- Code organization
- DRY violations
- SOLID principle violations

3. PERFORMANCE ANALYSIS

- Time complexity: O(?) with explanation
- Space complexity: O(?) with explanation
- Identified bottlenecks
- Optimization suggestions with code

4. SECURITY REVIEW

- Potential vulnerabilities
- Input validation assessment
- Data exposure risks

5. ERROR HANDLING

- Unhandled edge cases
- Error message quality
- Recovery strategy suggestions

6. TESTABILITY

- Current testability score (1-10)
- Suggestions to improve
- Critical test cases to add

7. REFACTORED VERSION Provide the improved code with comments explaining each significant change.

8. SUMMARY

- Overall quality grade (A-F)
- Top 3 priority fixes
- What's done well (positive feedback)

```
**Tips for Use:**
- Include the full context of what the code does and where it runs
- Specify your team's coding standards if you have them
- Mention any constraints (legacy system compatibility, etc.)

**Expected Output:** Multi-section code review with specific issues, fixes, and a refactored version of your code.

---

## 2. Intelligent Debugging Assistant

**Purpose:** Systematically diagnose and fix bugs with expert guidance.
```

You are a debugging expert who has solved thousands of production issues. You think systematically, form hypotheses, and never assume without evidence.

I have a bug that needs solving:

THE BUG:

- Expected behavior: [WHAT SHOULD HAPPEN]
- Actual behavior: [WHAT ACTUALLY HAPPENS]
- Error message (if any):

```
[EXACT ERROR MESSAGE]
```

CONTEXT: Code that's failing:

```
[PASTE RELEVANT CODE]
```

Stack trace (if available):

```
[PASTE STACK TRACE]
```

ENVIRONMENT:

- Language/Runtime version: [VERSION]
- Framework version: [VERSION]
- OS: [OPERATING SYSTEM]
- Dependencies that might be relevant: [LIST]

REPRODUCTION:

- Steps to reproduce: [1, 2, 3...]
- Frequency: [ALWAYS / SOMETIMES / RANDOM]
- Recent changes before bug appeared: [WHAT CHANGED]

What I've already tried:

1. [ATTEMPT 1 AND RESULT]

2. [ATTEMPT 2 AND RESULT]

Debug this systematically:

STEP 1: HYPOTHESIS FORMATION Based on the symptoms, list the top 5 likely causes ranked by probability:

1. [MOST LIKELY] - Probability: X% - Why: [REASONING]
2. ...

STEP 2: DIAGNOSTIC QUESTIONS Questions to narrow down the cause (answer if you can):

- [QUESTION 1]
- [QUESTION 2]

STEP 3: DEBUGGING STRATEGY Step-by-step debugging approach:

1. First, check [X] by doing [Y]
2. If that's not it, verify [A] by [B] ...

STEP 4: ROOT CAUSE ANALYSIS Based on available information, the most likely root cause is: [DETAILED EXPLANATION]

Evidence supporting this:

- [EVIDENCE 1]
- [EVIDENCE 2]

STEP 5: THE FIX

```
[CORRECTED CODE]
```

Explanation of why this fixes it: [REASONING]

STEP 6: VERIFICATION How to confirm the fix works:

1. [TEST STEP]

2. [TEST STEP]

STEP 7: PREVENTION To prevent this class of bug in the future:

- [CODING PRACTICE]
- [TEST TO ADD]
- [TOOLING SUGGESTION]

```
**Tips for Use:**
- Include the COMPLETE error message and stack trace
- Describe what changed recently - most bugs come from
recent changes
- List what you've already tried to avoid redundant
suggestions

**Expected Output:** Systematic bug diagnosis with
root cause analysis and verified fix.

---

## 3. Architecture Decision Advisor

**Purpose:** Get expert guidance on technical
architecture decisions.
```

You are a principal engineer who has designed systems at scale (millions of users, petabytes of data). You think in terms of tradeoffs, not perfect solutions, and you understand that context drives architecture.

Help me make an architecture decision:

THE DECISION: [DESCRIBE THE ARCHITECTURAL CHOICE YOU'RE FACING]

OPTIONS I'M CONSIDERING: Option A: [DESCRIPTION] Option B: [DESCRIPTION] Option C: [DESCRIPTION] (if applicable)

CONTEXT:

- System purpose: [WHAT IT DOES]
- Current scale: [USERS/REQUESTS/DATA VOLUME]
- Expected growth: [GROWTH PROJECTIONS]
- Team size and expertise: [DESCRIPTION]
- Timeline: [DEADLINE OR TIMEFRAME]
- Budget constraints: [IF ANY]
- Existing tech stack: [CURRENT TECHNOLOGIES]
- Non-negotiables: [HARD REQUIREMENTS]

MY CURRENT THINKING: I'm leaning toward [OPTION] because [REASONING]

Analyze this decision:

1. CLARIFYING QUESTIONS Before recommending, I'd ask:

- [QUESTION 1]
- [QUESTION 2] (Answer these if you can, or I'll proceed with assumptions)

2. TRADEOFF ANALYSIS

Factor	Option A	Option B	Option C
Complexity			
Scalability			
Maintainability			
Performance			
Cost			
Time to implement			
Team skill match			
Flexibility			

3. DEEP DIVE ON EACH OPTION

Option A: [NAME]

- How it works: [EXPLANATION]
- Best when: [SCENARIOS]
- Risks: [POTENTIAL ISSUES]
- Hidden costs: [NON-OBVIOUS CONSIDERATIONS]
- Real-world example: [WHO USES THIS AND HOW]

[Repeat for each option]

4. RECOMMENDATION Given your context, I recommend: [OPTION]

Primary reasons:

1. [REASON 1]
2. [REASON 2]
3. [REASON 3]

Key risks to mitigate:

1. [RISK] - Mitigation: [APPROACH]

5. IMPLEMENTATION GUIDANCE If you go with my recommendation:

1. Start by: [FIRST STEP]
2. Key milestones: [LIST]
3. Signals you made the right choice: [INDICATORS]
4. Signals you need to pivot: [WARNING SIGNS]

6. ALTERNATIVE PERSPECTIVE Devil's advocate - reasons to choose [OTHER OPTION]:

- [REASON 1]
- [REASON 2]

```
**Tips for Use:**
- Be honest about your team's skills and constraints
- Include context about timeline and budget - they
matter for architecture
- Mention if you need to integrate with existing
systems

**Expected Output:** Comprehensive decision analysis
with tradeoffs, recommendation, and implementation
guidance.

---

## 4. Test Case Generator

**Purpose:** Generate comprehensive test suites for
your code.
```

You are a QA architect who believes in thorough test coverage. You write tests that catch bugs before they reach production and serve as documentation.

Generate tests for this code:

```
[PASTE CODE TO TEST]
```

CONTEXT:

- Testing framework: [JEST/PYTEST/JUNIT/RSPEC/etc.]
- Testing type needed: [UNIT/INTEGRATION/E2E/ALL]
- Coverage goal: [PERCENTAGE OR FOCUS AREAS]
- Critical paths: [MOST IMPORTANT FUNCTIONALITY]
- Known edge cases: [ANY YOU'RE AWARE OF]

Generate comprehensive tests:

1. TEST ANALYSIS Functions/methods to test:

Function	Complexity	Priority	Edge Cases

Dependencies to mock:

- [DEPENDENCY 1]: [MOCK STRATEGY]
- [DEPENDENCY 2]: [MOCK STRATEGY]

2. TEST STRUCTURE

```
// Recommended test file organization
[TEST FILE STRUCTURE]
```

3. UNIT TESTS

```
// Complete test suite

// Happy path tests
[TESTS FOR NORMAL/EXPECTED INPUTS]

// Edge case tests
[TESTS FOR BOUNDARY CONDITIONS]

// Error handling tests
[TESTS FOR ERROR SCENARIOS]

// Null/undefined/empty tests
[TESTS FOR MISSING DATA]
```

4. TEST DATA FACTORY

```
// Reusable test data generators
[FACTORY FUNCTIONS]
```

5. MOCK SETUP

```
// Mock configurations
[MOCK DEFINITIONS]
```

6. TEST MATRIX

Scenario	Input	Expected Output	Covered
[SCENARIO 1]	[INPUT]	[OUTPUT]	[YES/NO]

7. EDGE CASES CHECKLIST

- ☐ Empty inputs
- ☐ Null/undefined values
- ☐ Boundary values (min/max)
- ☐ Special characters
- ☐ Very large inputs
- ☐ Concurrent operations
- ☐ Network failures (if applicable)
- ☐ Timeout scenarios

8. COVERAGE ANALYSIS Estimated coverage: [%] Uncovered paths: [LIST] Additional tests needed: [SUGGESTIONS]

```
**Tips for Use:**
- Provide complete code including all dependencies
- Specify your preferred testing framework
- Mention any known problematic edge cases

**Expected Output:** Complete test suite with unit
tests, mocks, test data factories, and coverage
analysis.

---

## 5. Code Refactoring Advisor

**Purpose:** Get expert guidance on improving code
structure and quality.
```

You are a refactoring expert who follows Martin Fowler's patterns. You improve code structure while preserving behavior, and you know when refactoring is worth the effort.

Analyze this code for refactoring:

```
[PASTE CODE TO REFACTOR]
```

CONTEXT:

- Code status: [STABLE/ACTIVELY CHANGING/LEGACY]
- Pain points: [WHAT'S PROBLEMATIC]
- Constraints: [WHAT CAN'T CHANGE - APIs, INTERFACES, etc.]
- Time budget: [HOW MUCH TIME FOR REFACTORING]
- Test coverage: [EXISTING TEST COVERAGE %]

Provide refactoring recommendations:

1. CODE SMELL DETECTION

Smell	Location	Severity	Refactoring Pattern
[SMELL]	[LINE/FUNCTION]	[HIGH/MED/LOW]	[PATTERN NAME]

Smells identified:

- Long Method: [LOCATIONS]
- Large Class: [LOCATIONS]
- Duplicate Code: [LOCATIONS]
- Feature Envy: [LOCATIONS]
- Data Clumps: [LOCATIONS]
- Primitive Obsession: [LOCATIONS]
- Switch Statements: [LOCATIONS]
- Dead Code: [LOCATIONS]

2. REFACTORING SEQUENCE Do these in order (each should leave code working):

1. [REFACTORING 1] - Why first: [REASON] - Risk: [LOW/MED/HIGH]
2. [REFACTORING 2] - Depends on: [#1]
3. ...

3. DETAILED REFACTORING GUIDES

Refactoring 1: [NAME] Pattern: [FOWLER PATTERN NAME]

Before:

```
[BEFORE CODE]
```

After:

```
[AFTER CODE]
```

Steps:

1. [STEP - should be testable]
2. [STEP]
3. [STEP]

Test to verify: [WHAT TO TEST AFTER THIS REFACTORING]

[Continue for each refactoring]

4. DESIGN PATTERN OPPORTUNITIES This code could benefit from:

- [PATTERN 1]: [WHERE AND WHY]
- [PATTERN 2]: [WHERE AND WHY]

5. FINAL REFACTORED VERSION

```
[COMPLETE REFACTORED CODE WITH COMMENTS]
```

6. BEFORE/AFTER METRICS

Metric	Before	After	Improvement
Lines of code	[N]	[N]	[%]
Cyclomatic complexity	[N]	[N]	[%]
Functions/methods	[N]	[N]	[CHANGE]
Testability (1-10)	[N]	[N]	[+N]

7. TESTING REQUIREMENTS Tests to write/update for safe refactoring:

```
[REQUIRED TESTS]
```

```
**Tips for Use:**
- Include existing tests if you have them -
refactoring requires tests
- Mention what specifically bothers you about the code
- Be realistic about time - some refactorings are big
undertakings

**Expected Output:** Prioritized refactoring plan with
step-by-step transformations and a complete refactored
version.

---

## 6. API Design Expert

**Purpose:** Design clean, developer-friendly APIs.
```

You are an API architect who has designed APIs used by thousands of developers. You prioritize consistency, discoverability, and developer experience.

Design an API for:

PURPOSE: [WHAT THE API DOES] **CONSUMERS:** [WHO WILL USE IT - INTERNAL/EXTERNAL/BOTH] **DATA ENTITIES:** [MAIN OBJECTS/RESOURCES] **KEY OPERATIONS:** [CRUD + CUSTOM ACTIONS]

REQUIREMENTS:

- Authentication: [REQUIREMENTS]
- Rate limiting: [REQUIREMENTS]
- Response format: [JSON/XML/BOTH]
- Versioning needed: [YES/NO]

Design the API:

1. API STYLE RECOMMENDATION Recommended style: [REST/GraphQL/gRPC/HYBRID] Rationale: [WHY THIS STYLE FOR YOUR USE CASE]

2. RESOURCE DESIGN Resources identified:

Resource	Description	Parent	Operations

Relationships:

- [RESOURCE A] has many [RESOURCE B]
- [RESOURCE C] belongs to [RESOURCE D]

3. ENDPOINT SPECIFICATION

Resource: [NAME]

LIST - Get all [resources]

```
GET /api/v1/[resources]
Query params:
  - page (int): Page number, default 1
  - limit (int): Items per page, default 20, max 100
  - sort (string): Field to sort by
  - filter[field] (string): Filter by field value

Response: 200 OK
{
  "data": [...],
  "meta": {
    "total": 100,
    "page": 1,
    "limit": 20,
    "totalPages": 5
  }
}
```

CREATE - Create a [resource]

```
POST /api/v1/[resources]
Headers: Content-Type: application/json
Body: {
  "field1": "value",
  "field2": "value"
}

Response: 201 Created
{
  "data": { ... }
}
```

READ - Get single [resource]

```
GET /api/v1/[resources]/:id

Response: 200 OK
{
  "data": { ... }
}
```

UPDATE - Update a [resource]

```
PATCH /api/v1/[resources]/:id
Body: { fields to update }

Response: 200 OK
{
  "data": { ... }
}
```

DELETE - Delete a [resource]

```
DELETE /api/v1/[resources]/:id

Response: 204 No Content
```

[Continue for each resource and custom action]

4. REQUEST/RESPONSE SCHEMAS

```
// [Resource] Schema
{
  "id": "string (uuid)",
  "createdAt": "string (ISO 8601)",
  "updatedAt": "string (ISO 8601)",
  // ... fields
}
```

5. ERROR HANDLING

```
// Standard error response
{
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Human readable message",
    "details": [
      { "field": "email", "message": "Invalid email format" }
    ],
    "requestId": "uuid for support"
  }
}
```

HTTP Status	Code	When

400	VALIDATION_ERROR	Invalid input
401	UNAUTHORIZED	Missing/invalid auth
403	FORBIDDEN	No permission
404	NOT_FOUND	Resource doesn't exist
409	CONFLICT	Resource conflict
429	RATE_LIMITED	Too many requests
500	INTERNAL_ERROR	Server error

6. AUTHENTICATION

```
Authorization: Bearer <token>
```

- Token format: [JWT/OPAQUE]
- Token lifetime: [DURATION]
- Refresh strategy: [APPROACH]

7. RATE LIMITING Headers:

- X-RateLimit-Limit: [MAX]
- X-RateLimit-Remaining: [REMAINING]
- X-RateLimit-Reset: [TIMESTAMP]

8. VERSIONING STRATEGY Approach: [URL/HEADER] Current version: v1 Deprecation policy: [APPROACH]

9. SDK EXAMPLE How this would feel to use:

```
// Initialize client
const api = new Client({ apiKey: 'xxx' });

// List resources
const items = await api.resources.list({ page: 1,
limit: 10 });

// Create resource
const newItem = await api.resources.create({ name:
'Example' });

// Update resource
await api.resources.update('id', { name: 'Updated'
});
```

```
**Tips for Use:**
- Think from the API consumer's perspective
- Include all entities and relationships upfront
- Consider future expansion in your design

**Expected Output:** Complete API specification with
endpoints, schemas, error handling, and SDK examples.

---

## 7. Database Schema Designer

**Purpose:** Design optimized database schemas for
your application.
```

You are a database architect who designs schemas for scalability, query performance, and data integrity. You think about both immediate needs and future growth.

Design a database schema for:

APPLICATION TYPE: [DESCRIPTION] **KEY ENTITIES:** [LIST MAIN OBJECTS] **PRIMARY USE CASES:**

1. [USE CASE 1] - Read/Write: [RATIO]
2. [USE CASE 2] - Read/Write: [RATIO]
3. [USE CASE 3] - Read/Write: [RATIO]

SCALE:

- Expected records per main table: [ESTIMATES]
- Read/write ratio: [RATIO]
- Growth rate: [RATE PER MONTH/YEAR]

DATABASE: [POSTGRESQL/MYSQL/MONGODB/etc.]

Design the schema:

1. ENTITY RELATIONSHIP DIAGRAM

```
[ASCII DIAGRAM OR DESCRIPTION]

User 1---* Post
User 1---* Comment
Post 1---* Comment
...
```

2. TABLE DEFINITIONS

Table: users Purpose: [WHAT IT STORES]

```sql
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    email VARCHAR(255) NOT NULL UNIQUE,
    password_hash VARCHAR(255) NOT NULL,
    name VARCHAR(100),
    status VARCHAR(20) DEFAULT 'active' CHECK
(status IN ('active', 'inactive', 'suspended')),
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);

-- Indexes
CREATE INDEX idx_users_email ON users(email);
CREATE INDEX idx_users_status ON users(status)
WHERE status = 'active';

-- Trigger for updated_at
CREATE TRIGGER update_users_timestamp
    BEFORE UPDATE ON users
    FOR EACH ROW
    EXECUTE FUNCTION update_updated_at();
```

[Continue for each table]

3. RELATIONSHIPS

```
-- Foreign keys
ALTER TABLE posts
ADD CONSTRAINT fk_posts_user
FOREIGN KEY (user_id) REFERENCES users(id) ON
DELETE CASCADE;

-- Many-to-many junction table
CREATE TABLE post_tags (
    post_id UUID REFERENCES posts(id) ON DELETE
CASCADE,
    tag_id UUID REFERENCES tags(id) ON DELETE
CASCADE,
    PRIMARY KEY (post_id, tag_id)
);
```

4. INDEXING STRATEGY

Table	Index	Columns	Type	Rationale
users	idx_users_email	email	BTREE	Login lookups
posts	idx_posts_user_date	user_id, created_at	BTREE	User's recent posts

5. COMMON QUERIES OPTIMIZED

Query: "Get user's recent posts with comment counts"

```sql
SELECT p.*, COUNT(c.id) as comment_count
FROM posts p
LEFT JOIN comments c ON c.post_id = p.id
WHERE p.user_id = $1
GROUP BY p.id
ORDER BY p.created_at DESC
LIMIT 20;

-- Explain plan:
[EXPECTED QUERY PLAN]
```

6. NORMALIZATION ANALYSIS Current form: [3NF/BCNF] Denormalization decisions:

- [FIELD]: Denormalized because [REASON]

7. DATA TYPES DECISIONS

Field Type	Choice	Alternatives	Why
IDs	UUID	BIGINT	[REASON]
Money	DECIMAL(10,2)	FLOAT	Precision needed
Status	VARCHAR+CHECK	ENUM	[REASON]

8. MIGRATION STRATEGY

```
-- Migration: 001_initial_schema
BEGIN;

[DDL STATEMENTS]

COMMIT;

-- Rollback
BEGIN;

[ROLLBACK STATEMENTS]

COMMIT;
```

9. SCALING CONSIDERATIONS

- Partitioning strategy: [IF NEEDED]
- Sharding key: [IF NEEDED]
- Read replica strategy: [RECOMMENDATION]

10. SEED DATA

```
-- Development seed data
INSERT INTO users (email, password_hash, name)
VALUES
    ('test@example.com', 'hash', 'Test User');
```

```
**Tips for Use:**
- Describe your most common queries
- Include expected data volumes
- Mention any compliance requirements (GDPR, etc.)

**Expected Output:** Complete database schema with
DDL, indexes, query optimization, and scaling
considerations.

---

## 8. Performance Optimizer

**Purpose:** Identify and fix performance bottlenecks
in your code.
```

You are a performance engineering expert who has optimized systems handling millions of requests. You profile before optimizing and measure after.

Analyze and optimize this code for performance:

```
[PASTE YOUR CODE]
```

CONTEXT:

- Current performance: [METRICS IF KNOWN - response time, throughput, etc.]
- Target performance: [DESIRED METRICS]
- Bottleneck symptoms: [WHAT'S SLOW]
- Scale: [REQUESTS/SECOND, DATA VOLUME, etc.]
- Environment: [HARDWARE, CLOUD, etc.]
- Acceptable tradeoffs: [MEMORY vs SPEED, READABILITY vs PERFORMANCE]

Provide performance analysis and optimization:

1. PERFORMANCE PROFILE

- Time complexity: O([?]) - [EXPLANATION]
- Space complexity: O([?]) - [EXPLANATION]
- Hot paths: [MOST EXPENSIVE OPERATIONS]
- I/O operations: [DATABASE, NETWORK, FILE]

2. BOTTLENECK ANALYSIS

Bottleneck 1: [LOCATION]

- Impact: [QUANTIFIED IF POSSIBLE]
- Root cause: [WHY IT'S SLOW]
- Fix: [SOLUTION]

Bottleneck 2: [LOCATION] ...

3. OPTIMIZATION STRATEGIES

Quick Wins (Implement Now):

1. [OPTIMIZATION] - Expected improvement: [X%]

```
// Before
[SLOW CODE]

// After
[FAST CODE]
```

Medium Effort:

1. [OPTIMIZATION] - Expected improvement: [X%]
   - Implementation approach: [STEPS]
   - Tradeoffs: [CONSIDERATIONS]

Major Refactoring:

1. [OPTIMIZATION] - Expected improvement: [X%]
    - Why it's worth it: [JUSTIFICATION]
    - Implementation plan: [STEPS]

4. ALGORITHM IMPROVEMENTS Current: [ALGORITHM] - O([COMPLEXITY]) Better: [ALGORITHM] - O([COMPLEXITY])

```
// Improved algorithm
[CODE]
```

5. CACHING STRATEGY What to cache:

Data	Cache Duration	Invalidation Strategy

```
// Caching implementation
[CODE]
```

6. DATABASE OPTIMIZATIONS (if applicable)

- Query optimization: [SUGGESTIONS]
- Index additions: [SUGGESTIONS]
- Connection pooling: [CONFIGURATION]

7. ASYNC/PARALLEL OPPORTUNITIES Operations that can run in parallel:

```
// Parallelized version
[CODE]
```

8. FULLY OPTIMIZED CODE

```
[COMPLETE OPTIMIZED VERSION WITH COMMENTS]
```

9. BENCHMARKING CODE

```
// Benchmark to verify improvements
[BENCHMARK CODE]
```

10. MONITORING RECOMMENDATIONS Metrics to track:

- [METRIC 1]: Alert if [THRESHOLD]
- [METRIC 2]: Alert if [THRESHOLD]

```
**Tips for Use:**
- Include actual performance numbers if you have them
- Specify your constraints - sometimes memory is
cheap, sometimes CPU is
- Mention if this is a hot path or rarely-called code

**Expected Output:** Comprehensive performance
analysis with optimized code and benchmarks.

---

## 9. Documentation Generator

**Purpose:** Generate professional documentation for
your code.
```

You are a technical writer who creates documentation that developers actually want to read. You balance completeness with clarity.

Generate documentation for:

```
[PASTE CODE/API/LIBRARY]
```

DOCUMENTATION TYPE: [README/API DOCS/GUIDE/REFERENCE/ALL] **TARGET AUDIENCE:** [BEGINNER/INTERMEDIATE/EXPERT DEVELOPERS] **PROJECT CONTEXT:** [WHAT THIS IS PART OF]

Generate documentation:

1. README.md

~~~~markdown
# [Project Name]

[One-paragraph description - what it does and why
it matters]

## Features

- [Feature 1]
- [Feature 2]
- [Feature 3]

## Quick Start

[30-second setup to get running]

\`\`\`[LANGUAGE]
[MINIMAL WORKING EXAMPLE]
\`\`\`

## Installation

\`\`\`bash
[INSTALLATION COMMANDS]
\`\`\`

### Prerequisites
- [Requirement 1]
- [Requirement 2]

## Usage

### Basic Usage
\`\`\`[LANGUAGE]
[BASIC EXAMPLE WITH COMMENTS]
\`\`\`
~~~~

````
### Advanced Usage
\`\`\`[LANGUAGE]
[ADVANCED EXAMPLE]
\`\`\`

## API Reference

### `functionName(param1, param2)`

[Description]

**Parameters:**
| Name | Type | Required | Description |
|------|------|----------|-------------|
| param1 | string | Yes | [Description] |

**Returns:** `ReturnType` - [Description]

**Example:**
\`\`\`[LANGUAGE]
[USAGE EXAMPLE]
\`\`\`

[Continue for each public function/class]

## Configuration

| Option | Type | Default | Description |
|--------|------|---------|-------------|
| [option] | [type] | [default] | [description] |

## Error Handling

| Error | Cause | Solution |
|-------|-------|----------|
| [ErrorType] | [Why it happens] | [How to fix] |

## FAQ
````

```
**Q: [Common question]**
A: [Answer]

## Troubleshooting

### [Problem]
**Symptom:** [What you see]
**Solution:** [How to fix]

## Contributing

[How to contribute - brief guidelines]

## License

[License info]
```

2. INLINE DOCUMENTATION

```
[CODE WITH ADDED DOCSTRINGS/COMMENTS]
```

3. EXAMPLES DIRECTORY

```
// examples/basic.js
[COMMENTED EXAMPLE]

// examples/advanced.js
[COMMENTED EXAMPLE]
```

```
**Tips for Use:**
- Include real, working code examples
- Write for someone who's never seen your code
- Include common mistakes and how to avoid them

**Expected Output:** Complete documentation package
ready for publication.

---

## 10. Code Translation/Migration

**Purpose:** Convert code between languages or
frameworks while maintaining functionality.
```

You are a polyglot developer fluent in multiple programming languages. You understand the idioms of each language and translate code to feel native, not mechanical.

Translate this code:

FROM:

```
[PASTE SOURCE CODE]
```

TO: [TARGET_LANGUAGE/FRAMEWORK]

REQUIREMENTS:

- Preserve functionality: [EXACT BEHAVIOR / EQUIVALENT BEHAVIOR]
- Target version: [LANGUAGE VERSION]
- Style guide: [ANY SPECIFIC STYLE TO FOLLOW]
- Dependencies available: [LIBRARIES YOU CAN USE]

Provide translation:

1. LANGUAGE MAPPING

Source Concept	Target Equivalent	Notes
[CONCEPT]	[EQUIVALENT]	[DIFFERENCES]

2. DEPENDENCY MAPPING

Source Library	Target Alternative	Feature Parity
[LIBRARY]	[EQUIVALENT]	[FULL/PARTIAL]

3. TRANSLATED CODE

```
[TRANSLATED CODE WITH COMMENTS EXPLAINING NON-
OBVIOUS TRANSLATIONS]
```

4. IDIOM ADJUSTMENTS Changes made to follow target language idioms:

Source Pattern:

```
[ORIGINAL PATTERN]
```

Target Idiom:

```
[IDIOMATIC VERSION]
```

Explanation: [WHY THIS IS MORE IDIOMATIC]

5. GOTCHAS Behavioral differences to be aware of:

1. [DIFFERENCE 1]: [HOW IT AFFECTS CODE]
2. [DIFFERENCE 2]: [HOW IT AFFECTS CODE]

6. TESTING EQUIVALENCE Tests to verify the translation maintains behavior:

```
[EQUIVALENT TEST CASES]
```

7. MIGRATION CHECKLIST

- [ ] All functions translated
- [ ] Error handling equivalent
- [ ] Types properly mapped
- [ ] Tests passing
- [ ] Performance acceptable
- [ ] Dependencies resolved

```
**Tips for Use:**
- Specify both source and target language versions
- Mention if you need exact behavior or can accept
equivalent behavior
- Note any libraries available in the target
environment

**Expected Output:** Idiomatic translated code with
mapping explanations and verification tests.

---

## 11. Security Audit

**Purpose:** Identify security vulnerabilities in your
code.
```

You are a security engineer who specializes in application security. You think like an attacker but provide constructive remediation guidance.

Audit this code for security vulnerabilities:

```
[PASTE CODE TO AUDIT]
```

CONTEXT:

- Application type: [WEB/API/MOBILE/DESKTOP]
- Data sensitivity: [PUBLIC/INTERNAL/CONFIDENTIAL/REGULATED]
- Authentication method: [HOW USERS LOG IN]
- Deployment: [WHERE THIS RUNS]
- Known concerns: [SPECIFIC AREAS YOU'RE WORRIED ABOUT]

Provide security audit:

1. VULNERABILITY SUMMARY

Severity	Count	Categories
Critical	[N]	[TYPES]
High	[N]	[TYPES]
Medium	[N]	[TYPES]
Low	[N]	[TYPES]

2. CRITICAL VULNERABILITIES

Vuln-001: [NAME]

- Location: [FILE:LINE]
- CWE: [CWE NUMBER]
- OWASP: [OWASP CATEGORY]
- Description: [WHAT'S WRONG]
- Attack scenario: [HOW AN ATTACKER COULD EXPLOIT THIS]
- Impact: [WHAT COULD HAPPEN]

- Fix:

```
// Vulnerable
[BAD CODE]

// Fixed
[SECURE CODE]
```

[Continue for each vulnerability]

3. INPUT VALIDATION AUDIT

Input Point	Current Validation	Recommendation
[INPUT]	[CURRENT]	[RECOMMENDED]

4. AUTHENTICATION/AUTHORIZATION REVIEW

- [ ] Password storage: [ASSESSMENT]
- [ ] Session management: [ASSESSMENT]
- [ ] Access controls: [ASSESSMENT]
- [ ] Token handling: [ASSESSMENT]

5. DATA PROTECTION REVIEW

- [ ] Encryption at rest: [ASSESSMENT]
- [ ] Encryption in transit: [ASSESSMENT]
- [ ] PII handling: [ASSESSMENT]
- [ ] Logging sensitive data: [ASSESSMENT]

6. SECURE CODE CHECKLIST

- [ ] SQL injection protected
- [ ] XSS protected
- [ ] CSRF tokens implemented

- [ ] Rate limiting in place
- [ ] Error messages don't leak info
- [ ] HTTPS enforced
- [ ] Headers properly configured

7. REMEDIATION PRIORITY

1. [VULNERABILITY] - Fix immediately: [WHY]
2. [VULNERABILITY] - Fix this sprint: [WHY]
3. [VULNERABILITY] - Plan for: [WHY]

8. ADDITIONAL RECOMMENDATIONS Security improvements beyond fixing vulnerabilities:

- [RECOMMENDATION 1]
- [RECOMMENDATION 2]

```
**Tips for Use:**
- Include all relevant code, not just the suspicious parts
- Mention your deployment environment - it affects risk
- Note any compliance requirements (PCI, HIPAA, etc.)

**Expected Output:** Comprehensive security audit with vulnerabilities, fixes, and prioritized remediation plan.

---

## 12. Code Learning Explainer

**Purpose:** Get patient, thorough explanations of code for learning.
```

You are a patient programming teacher who explains code in a way that builds understanding. You use analogies, break down complexity, and check comprehension.

Explain this code to me:

```
[PASTE CODE YOU WANT TO UNDERSTAND]
```

MY BACKGROUND:

- Experience level: [BEGINNER/INTERMEDIATE/ADVANCED]
- Languages I know: [LIST]
- Specific confusion: [WHAT'S CONFUSING ME]
- Learning goal: [WHAT I WANT TO UNDERSTAND]

Explain this code:

1. THE BIG PICTURE In plain English, this code: [SIMPLE EXPLANATION]

Real-world analogy: [RELATABLE COMPARISON]

2. KEY CONCEPTS Before diving in, you need to understand:

Concept: [NAME]

- What it is: [EXPLANATION IN SIMPLE TERMS]
- Why it exists: [THE PROBLEM IT SOLVES]
- Analogy: [REAL-WORLD COMPARISON]

```
// Simple example
[MINIMAL CODE DEMONSTRATING CONCEPT]
```

[Continue for each foundational concept]

3. LINE-BY-LINE WALKTHROUGH

```
// Line 1: [CODE]
// What it does: [EXPLANATION]
// Why it's written this way: [REASONING]

// Line 2: [CODE]
// What it does: [EXPLANATION]
// This connects to line 1 by: [CONNECTION]
```

4. DATA FLOW Watch how data moves through this code:

```
Input: [START]
   |
Step 1: [TRANSFORMATION]
   |
Step 2: [TRANSFORMATION]
   |
Output: [END]
```

5. MENTAL MODEL Think of this code like: [EXTENDED ANALOGY]

6. COMMON MISTAKES When writing code like this, beginners often:

1. [MISTAKE 1] - Why it's wrong: [EXPLANATION]
2. [MISTAKE 2] - Why it's wrong: [EXPLANATION]

7. TRY IT YOURSELF To test your understanding:

Exercise 1 (Easy): [DESCRIPTION] Hint: [GUIDANCE]

Exercise 2 (Medium): [DESCRIPTION] Hint: [GUIDANCE]

8. RELATED CONCEPTS Now that you understand this, you should learn:

1. [CONCEPT]: [WHY IT'S RELATED]
2. [CONCEPT]: [WHY IT'S RELATED]

9. QUESTIONS? If any part is still unclear, ask about:

- [POTENTIALLY CONFUSING PART 1]
- [POTENTIALLY CONFUSING PART 2]

```
**Tips for Use:**
- Be honest about what you don't understand
- Mention languages you already know for better analogies
- State your learning goal clearly

**Expected Output:** Patient, thorough explanation with analogies and exercises.

---

## 13. Git Commit Message Generator

**Purpose:** Create meaningful, standardized commit messages.
```

You are an experienced developer who writes clear, informative commit messages following conventional commit standards.

Generate a commit message for these changes:

CHANGES MADE:

```
[PASTE YOUR GIT DIFF OR DESCRIBE CHANGES]
```

CONTEXT:

- Type of change: [FEAT/FIX/DOCS/STYLE/REFACTOR/TEST/CHORE]
- Scope (optional): [AFFECTED COMPONENT/MODULE]
- Breaking change: [YES/NO]

- Related issue/ticket: [NUMBER IF ANY]

Generate commit message:

CONVENTIONAL COMMIT FORMAT:

```
<type>(<scope>): <description>

[body - what and why, not how]

[footer - breaking changes, issue references]
```

RECOMMENDED COMMIT MESSAGE:

```
[COMPLETE COMMIT MESSAGE]
```

ALTERNATIVE OPTIONS:

1. Concise version:

```
[SHORTER VERSION]
```

2. Detailed version:

```
[MORE DETAILED VERSION]
```

IF CHANGES SHOULD BE SPLIT: This looks like multiple logical changes. Consider:

1. Commit 1: `[MESSAGE]`
2. Commit 2: `[MESSAGE]`

```
**Tips for Use:**
- Include the actual diff for more accurate messages
- Mention if this is part of a larger feature
- Note any breaking changes explicitly

**Expected Output:** Well-formatted commit message
following conventional commits standard.

---

## 14. Code Interview Problem Solver

**Purpose:** Practice coding interview problems with
expert-level solutions.
```

You are a senior engineer who has conducted 500+ coding interviews and knows what distinguishes good solutions from great ones.

Help me solve this interview problem:

PROBLEM: [PASTE THE PROBLEM STATEMENT]

CONSTRAINTS:

- Time limit: [IF ANY]
- Space limit: [IF ANY]
- Input size: [RANGES]

MY ATTEMPT (optional):

```
[MY SOLUTION IF I HAVE ONE]
```

Solve this comprehensively:

1. PROBLEM UNDERSTANDING In my own words: [RESTATE THE PROBLEM]

Key insights:

- [INSIGHT 1]
- [INSIGHT 2]

Questions I'd ask the interviewer:

- [CLARIFYING QUESTION 1]
- [CLARIFYING QUESTION 2]

2. EXAMPLE WALKTHROUGH Input: [EXAMPLE] Step 1: [ACTION] Step 2: [ACTION] Output: [RESULT]

Edge cases to consider:

- [EDGE CASE 1]
- [EDGE CASE 2]

3. APPROACH ANALYSIS

Approach 1: Brute Force

- Time: O([?])
- Space: O([?])
- Idea: [DESCRIPTION]
- Why not ideal: [LIMITATIONS]

Approach 2: Optimized

- Time: O([?])
- Space: O([?])
- Idea: [DESCRIPTION]
- Key insight: [WHAT MAKES THIS BETTER]

Approach 3: Optimal (if different)

- Time: O([?])
- Space: O([?])

- Idea: [DESCRIPTION]

4. OPTIMAL SOLUTION

```
[CLEAN, INTERVIEW-READY CODE WITH COMMENTS]
```

5. COMPLEXITY ANALYSIS Time: O([?]) because [REASONING] Space: O([?]) because [REASONING]

6. TEST CASES

```
// Test cases to verify solution
assert(solution([INPUT]) === [OUTPUT]); // Normal case
assert(solution([INPUT]) === [OUTPUT]); // Edge case
assert(solution([INPUT]) === [OUTPUT]); // Large input
```

7. FOLLOW-UP QUESTIONS The interviewer might ask:

- "What if [CONSTRAINT CHANGE]?" - [HOW SOLUTION CHANGES]
- "How would you scale this?" - [ANSWER]

8. INTERVIEW TIPS When presenting this solution:

- Start by: [WHAT TO SAY FIRST]
- Clarify: [IMPORTANT POINTS]
- Avoid: [COMMON MISTAKES]

```
**Tips for Use:**
- Include the complete problem statement with
constraints
- Share your attempt even if incomplete - feedback is
valuable
- Mention the company/role if targeting a specific
interview style

**Expected Output:** Comprehensive solution with
multiple approaches, complexity analysis, and
interview presentation tips.

---

## 15. Technical Specification Writer

**Purpose:** Create detailed technical specifications
for features.
```

You are a technical program manager who writes specs that engineering teams can implement without ambiguity.

Write a technical specification for:

FEATURE: [WHAT YOU'RE BUILDING] **CONTEXT:** [WHY IT'S NEEDED] **STAKEHOLDERS:** [WHO CARES ABOUT THIS] **TIMELINE:** [WHEN IT'S NEEDED]

Generate technical specification:

Technical Specification: [FEATURE NAME]

Metadata

- Author: [NAME]
- Status: Draft
- Created: [DATE]
- Last Updated: [DATE]
- Reviewers: [NAMES]

1. Overview

1.1 Problem Statement

[What problem are we solving]

1.2 Goals

- [GOAL 1]
- [GOAL 2]

1.3 Non-Goals

- [EXPLICITLY OUT OF SCOPE 1]
- [EXPLICITLY OUT OF SCOPE 2]

1.4 Success Metrics

Metric	Current	Target	How Measured

[METRIC]	[VALUE]	[VALUE]	[METHOD]

2. Background

[Context needed to understand this feature]

3. Detailed Design

3.1 Architecture

```
[DIAGRAM]
```

3.2 Data Model

```
[SCHEMA/TYPES]
```

3.3 API Changes

```
[NEW/MODIFIED ENDPOINTS]
```

3.4 User Flow

1. User does [ACTION]
2. System responds with [RESPONSE]
3. ...

3.5 Error Handling

Scenario	Error	User Message	Recovery
[SCENARIO]	[CODE]	[MESSAGE]	[ACTION]

4. Implementation Plan

4.1 Phases

Phase	Scope	Duration	Dependencies
1	[SCOPE]	[TIME]	[DEPS]

4.2 Tasks

- [ ] [TASK 1] (estimate: X days)
- [ ] [TASK 2] (estimate: X days)

4.3 Testing Strategy

- Unit tests: [APPROACH]
- Integration tests: [APPROACH]
- E2E tests: [APPROACH]
- Manual testing: [APPROACH]

5. Risks and Mitigations

Risk	Probability	Impact	Mitigation

[RISK]	[H/M/L]	[H/M/L]	[APPROACH]

6. Security Considerations

[Security implications and how they're addressed]

7. Monitoring and Alerting

[What to monitor and when to alert]

8. Rollout Plan

- ☐ Feature flag: [FLAG NAME]
- ☐ Rollout stages: [PERCENTAGES AND TIMING]
- ☐ Rollback plan: [PROCEDURE]

9. Open Questions

- [QUESTION 1]
- [QUESTION 2]

10. Appendix

[Additional diagrams, research, references]

```
**Tips for Use:**
- Include enough context for someone new to understand
- Be explicit about what's NOT included
- List open questions - it's okay not to have all
answers

**Expected Output:** Comprehensive technical
specification ready for team review.

---

## 16. Microservices Design

**Purpose:** Design scalable microservices
architecture.
```

You are a distributed systems architect who has designed microservices at scale. You understand the tradeoffs between monoliths and microservices.

Design a microservices architecture for:

SYSTEM: [WHAT YOU'RE BUILDING] **CURRENT STATE:** [MONOLITH/GREENFIELD/PARTIAL] **SCALE REQUIREMENTS:** [USERS/REQUESTS/DATA] **TEAM:** [SIZE AND STRUCTURE]

Design the architecture:

1. SERVICE IDENTIFICATION Based on domain analysis:

Service	Responsibility	Data Owned	Team Owner
[NAME]	[DOES WHAT]	[ENTITIES]	[TEAM]

2. SERVICE BOUNDARIES

```
[ASCII DIAGRAM OF SERVICES AND THEIR RELATIONSHIPS]
```

3. COMMUNICATION PATTERNS

From	To	Pattern	Protocol	Data
[SVC]	[SVC]	Sync/Async	REST/gRPC/Events	[PAYLOAD]

4. DATA MANAGEMENT

Service	Database	Why This DB	Shared Data Strategy
[SVC]	[DB TYPE]	[REASON]	[APPROACH]

5. CROSS-CUTTING CONCERNS

Authentication: [APPROACH - e.g., API Gateway + JWT]

Service Discovery: [APPROACH - e.g., Kubernetes DNS, Consul]

Configuration: [APPROACH - e.g., ConfigMaps, Vault]

Logging: [APPROACH - e.g., Structured JSON, ELK]

Monitoring: [APPROACH - e.g., Prometheus + Grafana]

Tracing: [APPROACH - e.g., Jaeger, Zipkin]

6. RESILIENCE PATTERNS

- Circuit breakers: [WHERE AND HOW]
- Retries: [POLICY]
- Timeouts: [DEFAULTS]
- Bulkheads: [ISOLATION STRATEGY]

7. DEPLOYMENT STRATEGY

```
# Example Kubernetes deployment
[K8S MANIFESTS]
```

8. MIGRATION PATH (if from monolith) Phase 1: [FIRST SERVICE TO EXTRACT] Phase 2: [NEXT SERVICES] Phase 3: [COMPLETION]

9. RISKS AND MITIGATIONS

Risk	Impact	Mitigation
[RISK]	[IMPACT]	[APPROACH]

```
**Tips for Use:**
- Be honest about team size - affects service count
- Include current system state for migration planning
- Mention any existing infrastructure constraints

**Expected Output:** Complete microservices
architecture design with service boundaries,
communication patterns, and deployment strategy.

---

## 17. CI/CD Pipeline Designer

**Purpose:** Design robust continuous integration and
deployment pipelines.
```

You are a DevOps engineer who has built CI/CD pipelines for organizations of all sizes. You balance thoroughness with speed.

Design a CI/CD pipeline for:

PROJECT TYPE: [WEB APP/API/LIBRARY/MOBILE] **TECH STACK:** [LANGUAGES/FRAMEWORKS] **HOSTING:** [AWS/GCP/AZURE/VERCEL/etc.] **TEAM SIZE:** [NUMBER] **CURRENT MATURITY:** [NONE/BASIC/ADVANCED]

Design the pipeline:

1. PIPELINE OVERVIEW

```
[ASCII DIAGRAM]
Push --> Build --> Test --> Deploy
```

2. TRIGGER STRATEGY

Branch	Trigger	Actions
main	Push	[ACTIONS]
develop	Push	[ACTIONS]
feature/*	Push	[ACTIONS]
PR	Open/Update	[ACTIONS]

3. STAGES

Stage: Build

```
# GitHub Actions / GitLab CI / etc.
[CONFIGURATION]
```

- Duration target: [TIME]
- Caching strategy: [WHAT TO CACHE]

Stage: Test

```
[CONFIGURATION]
```

- Unit tests
- Integration tests
- Coverage threshold: [%]
- Parallelization: [STRATEGY]

Stage: Security

```
[CONFIGURATION]
```

- SAST scanning
- Dependency scanning
- Secret detection

Stage: Deploy

```
[CONFIGURATION]
```

- Environment promotion: dev - staging - prod
- Rollback strategy: [APPROACH]

4. ENVIRONMENT STRATEGY

Environment	Branch	Auto-deploy	Approval
Development	develop	Yes	No
Staging	main	Yes	No
Production	main	No	Yes

5. QUALITY GATES Pipeline blocks if:

- [ ] Tests fail
- [ ] Coverage below [X%]
- [ ] Security vulnerabilities found
- [ ] Linting errors
- [ ] Build size exceeds [X MB]

6. COMPLETE CONFIGURATION

```
# .github/workflows/ci.yml (or equivalent)
[FULL CONFIGURATION FILE]
```

7. SECRETS MANAGEMENT

Secret	Where Stored	How Accessed
[SECRET]	[LOCATION]	[METHOD]

8. MONITORING

- Build notifications: [SLACK/EMAIL/etc.]
- Deployment tracking: [TOOL]
- Performance monitoring: [APPROACH]

9. OPTIMIZATION TIPS

- Reduce build time by: [SUGGESTIONS]
- Reduce costs by: [SUGGESTIONS]
- Improve reliability by: [SUGGESTIONS]

```
**Tips for Use:**
- Specify your CI/CD platform preference
- Include your current infrastructure
- Mention any compliance requirements

**Expected Output:** Complete CI/CD pipeline
configuration with all stages, quality gates, and
deployment strategy.

---

## 18. Error Message Decoder

**Purpose:** Understand and fix cryptic error
messages.
```

You are a developer support expert who has seen every error message. You decode cryptic errors and provide clear solutions.

Decode this error:

ERROR MESSAGE:

```
[PASTE THE FULL ERROR MESSAGE]
```

CONTEXT:

- Language/Framework: [DETAILS]
- What I was trying to do: [ACTION]
- Code that triggered it:

```
[RELEVANT CODE]
```

Decode and solve:

1. ERROR TRANSLATION In plain English: [WHAT THIS ERROR ACTUALLY MEANS]

Technical meaning: [PRECISE EXPLANATION]

2. ROOT CAUSE This error occurs because: [EXPLANATION]

Common triggers:

1. [TRIGGER 1]
2. [TRIGGER 2]
3. [TRIGGER 3]

3. SOLUTION

Quick Fix:

```
[IMMEDIATE SOLUTION]
```

Proper Fix:

```
[BETTER LONG-TERM SOLUTION]
```

Why the proper fix is better: [EXPLANATION]

4. RELATED ERRORS You might also see:

- [RELATED ERROR 1]: [BRIEF EXPLANATION]
- [RELATED ERROR 2]: [BRIEF EXPLANATION]

5. PREVENTION To avoid this error in the future:

- [ ] [PRACTICE 1]
- [ ] [PRACTICE 2]

6. DEBUGGING COMMANDS Useful commands to investigate:

```
[COMMANDS]
```

```
**Tips for Use:**
- Include the COMPLETE error message
- Mention what you were trying to do
- Include the code that triggered the error

**Expected Output:** Clear explanation with immediate
fix and prevention strategies.

---

## 19. Code Snippet Library Builder

**Purpose:** Create reusable code snippets for common
patterns.
```

You are a senior developer who maintains a library of battle-tested code snippets. Your snippets are clean, well-documented, and production-ready.

Create a code snippet for:

PATTERN: [WHAT THE SNIPPET DOES] **LANGUAGE:** [PROGRAMMING LANGUAGE] **USE CASE:** [WHEN TO USE THIS] **REQUIREMENTS:** [ANY SPECIFIC REQUIREMENTS]

Create the snippet:

1. SNIPPET

```
/**
 * [BRIEF DESCRIPTION]
 *
 * @param {[TYPE]} [PARAM] - [DESCRIPTION]
 * @returns {[TYPE]} [DESCRIPTION]
 * @example
 * [USAGE EXAMPLE]
 */
[CODE]
```

2. USAGE EXAMPLES

Basic usage:

```
[BASIC EXAMPLE]
```

Advanced usage:

```
[ADVANCED EXAMPLE]
```

3. VARIATIONS

Variation 1: [FOR SPECIFIC USE CASE]

```
[VARIATION CODE]
```

Variation 2: [FOR DIFFERENT USE CASE]

```
[VARIATION CODE]
```

4. GOTCHAS

- [GOTCHA 1]: [HOW TO AVOID]
- [GOTCHA 2]: [HOW TO AVOID]

5. TESTS

```
[TEST CASES FOR THE SNIPPET]
```

6. RELATED SNIPPETS You might also need:

- [RELATED PATTERN 1]
- [RELATED PATTERN 2]

```
**Tips for Use:**
- Be specific about the use case
- Mention any performance requirements
- Include edge cases you need to handle

**Expected Output:** Production-ready snippet with
documentation, examples, and tests.

---

## 20. Legacy Code Modernization Planner

**Purpose:** Create a plan to modernize legacy code
safely.
```

You are a software architect who specializes in legacy system modernization. You've migrated systems without breaking production.

Plan modernization for:

LEGACY SYSTEM:

```
[PASTE LEGACY CODE OR DESCRIBE SYSTEM]
```

CURRENT STATE:

- Age: [YEARS OLD]
- Tech stack: [OLD TECHNOLOGIES]
- Pain points: [WHAT'S PROBLEMATIC]
- Dependencies: [EXTERNAL SYSTEMS]

TARGET STATE:

- Target stack: [NEW TECHNOLOGIES]
- Timeline: [AVAILABLE TIME]
- Risk tolerance: [LOW/MEDIUM/HIGH]

Create modernization plan:

1. ASSESSMENT

- Code health score: [1-10]
- Technical debt level: [LOW/MEDIUM/HIGH/CRITICAL]
- Modernization complexity: [SIMPLE/MODERATE/COMPLEX]

2. MODERNIZATION STRATEGY Recommended approach: [BIG BANG/STRANGLER FIG/HYBRID] Rationale: [WHY THIS APPROACH]

3. PHASE PLAN

Phase 1: [NAME] (Duration: [TIME])

- Goal: [OBJECTIVE]

- Scope: [WHAT'S INCLUDED]
- Risk: [LEVEL]
- Rollback plan: [HOW TO REVERT]
- Success criteria: [HOW TO KNOW IT WORKED]

Tasks:

1. [TASK] - [ESTIMATE]
2. [TASK] - [ESTIMATE]

[Continue for each phase]

4. CODE TRANSFORMATIONS

Pattern: [OLD PATTERN] to [NEW PATTERN]

Before:

```
[OLD CODE]
```

After:

```
[NEW CODE]
```

Steps to transform:

1. [STEP]
2. [STEP]

[Continue for major patterns]

5. DATA MIGRATION

Data	Current Location	Target Location	Strategy

[DATA]	[OLD]	[NEW]	[APPROACH]

6. TESTING STRATEGY

- Characterization tests: [FOR EXISTING BEHAVIOR]
- Parallel running: [IF APPLICABLE]
- Canary deployment: [APPROACH]

7. RISK REGISTER

Risk	Probability	Impact	Mitigation
[RISK]	[H/M/L]	[H/M/L]	[APPROACH]

8. SUCCESS METRICS

Metric	Before	After Target
Deployment frequency	[VALUE]	[TARGET]
Lead time	[VALUE]	[TARGET]
Error rate	[VALUE]	[TARGET]

9. GO/NO-GO CHECKLIST Before each phase:

- ☐ [CHECKPOINT 1]
- ☐ [CHECKPOINT 2]

```
**Tips for Use:**
- Be honest about the current state - sugarcoating
leads to bad plans
- Include all integration points
- Specify your risk tolerance

**Expected Output:** Comprehensive modernization plan
with phased approach and risk management.

---

*These prompts are designed to be customized to your
specific needs. Replace the bracketed placeholders
with your actual information for best results.*

*Pro tip: Chain prompts together for complex tasks -
use the Code Review output to inform the Refactoring
prompt, then use the Test Generator for the refactored
code.*
```