Multi-Model Methodology for Building Operational Programs

A Multi-Model AI Assembly Line for Designing, Specifying, Building, Testing, and Shipping Software

Version 1.1

June 2026

Table of Contents

Table of Contents....2
1. Executive Summary....4
 1.2 Methodology Overview....4
 1.3 Stage Summary....5
2. Workflow Diagram....6
3. AI Tool Evaluation....7
 3.1 Comparative Overview....7
 3.2 Model-Specific Notes....8
  Claude (Anthropic)....8
  ChatGPT (OpenAI GPT-4o)....8
  Gemini 1.5 Pro / Gemini 2.0 (Google)....8
  Microsoft Copilot....8
  Grok (xAI)....8
  Local LLMs (Ollama, LM Studio, Jan)....8
  IDE-Integrated Coding Tools (Cursor, GitHub Copilot)....9
  Specialized Tools Worth Monitoring....9
4. Stage-by-Stage Procedures....10
 Stage 1: Product Design & UX....10
  AI Recommendations — Stage 1....10
  Prompt Templates — Stage 1....11
 Stage 2: Requirements Specification....13
  AI Recommendations — Stage 2....13
  Prompt Templates — Stage 2....14
 Stage 3: Technical Architecture....16
  AI Recommendations — Stage 3....16
  Prompt Templates — Stage 3....18
 Stage 4: Implementation Planning....19
  AI Recommendations — Stage 4....20
  Prompt Templates — Stage 4....21
 Stage 5: Software Development....21
  AI Recommendations — Stage 5....22
  Prompt Templates — Stage 5....23
 Stage 6: Quality Assurance....24
  AI Recommendations — Stage 6....24
  Prompt Templates — Stage 6....25
 Stage 7: Security Review....25
  AI Recommendations — Stage 7....26
  Prompt Templates — Stage 7....27
 Stage 8: Documentation....27
  AI Recommendations — Stage 8....28
  Prompt Templates — Stage 8....28
 Stage 9: Consistency & Final Review....29
  AI Recommendations — Stage 9....30
  Prompt Templates — Stage 9....31
5. Operational Playbook....32
 5.1 Before You Begin....32
 5.2 Stage-by-Stage Procedures for Coordinators....32
  How to Run a Stage....32

How to Handle AI Errors........32
How to Handle Context Window Overflow........33
How to Manage Requirement Changes........33
5.3 Quality Control Procedures........33
The Rule of Two AI Models........33
The Human Approval Requirement........34
Version Control........34
5.4 Final Sign-Off Checklist........34
6. Lessons Learned and Best Practices........35
6.1 What Works........35
6.2 Common Mistakes to Avoid........35
6.3 Prompt Engineering Best Practices........35
6.4 When to Involve a Human Expert........36
7. Appendices........37
Appendix A: Glossary........37
Appendix B: Document Version History........37

1. Executive Summary

This document defines the Multi-Model Methodology for Building Operational Programs (MMMBOP) — a structured, repeatable framework for using multiple large language model (LLM) AI systems in a coordinated 'assembly line' to design, specify, build, test, and deliver software products.

The fundamental premise is that no single AI model is optimal for every task. Different models have different strengths in scope refinement versus structured specification, different context window limits, different usage quotas, and different known failure modes. Rather than betting a project on one system, MMMBOP treats AI models as specialized workers — each assigned to the stages where they perform best — supervised by a human coordinator who owns the final decisions.

This methodology is designed to be executable by a non-technical founder or project manager while producing deliverables that meet professional engineering standards. It does not require writing code directly; it requires knowing what to ask, how to validate answers, and when to move to the next stage.

1.2 Methodology Overview

The methodology consists of nine stages organized into three phases:

Phase A — Discovery & Definition (Stages 1–3)

- Stage 1: Product Design & UX
- Stage 2: Requirements Specification
- Stage 3: Technical Architecture

Phase B — Build (Stages 4–5)

- Stage 4: Implementation Planning
- Stage 5: Software Development

Phase C — Validate & Ship (Stages 6–9)

- Stage 6: Quality Assurance
- Stage 7: Security Review
- Stage 8: Documentation
- Stage 9: Consistency & Final Review

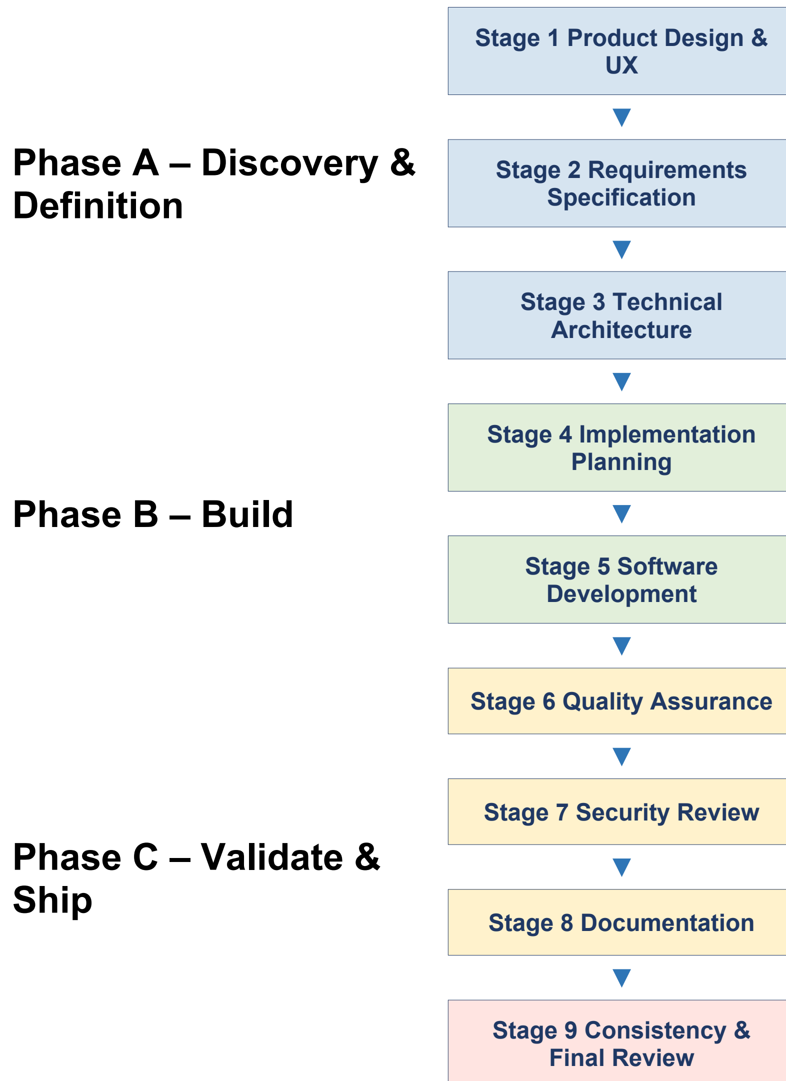
Each stage has a defined objective, explicit inputs, explicit outputs, success criteria, a decision gate that must be passed before the next stage begins, and AI model recommendations.

1.3 Stage Summary

#	Stage	Primary AI(s)	Key Output	Gate Criterion
1	Product Design & UX	Claude / ChatGPT	SRS-ready design doc	User journeys validated
2	Requirements Specification	Claude	Complete SRS	All acceptance criteria defined
3	Technical Architecture	Gemini / Claude	Architecture Decision Record	Stack & APIs specified
4	Implementation Planning	Claude	Tasked implementation plan	Tasks are atomic & testable
5	Software Development	Claude / Cursor	Production code + tests	All tests pass
6	Quality Assurance	Claude / Gemini	QA report	No Sev-1 defects open
7	Security Review	Claude / GPT-4o	Security report	No critical vulns open
8	Documentation	Claude / ChatGPT	User + technical docs	Docs match code
9	Consistency & Final Review	Claude	Final sign-off report	No contradictions remain

2. Workflow Diagram

The following diagram illustrates the linear flow of the nine stages. Each stage must produce an approved output artifact before the next stage may begin. Feedback loops are permitted — a failed decision gate returns control to the responsible stage — but stages may not be skipped.



> **NOTE:** *The workflow is intentionally linear for first-pass development. Experienced teams may overlap Stages 4 and 5 for large projects, but only after Stages 1–3 are fully signed off.*

3. AI Tool Evaluation

The following section evaluates all major AI systems relevant to this workflow as of mid-2025. Capabilities, context windows, and pricing change frequently. Verify current specifications before committing to a specific model for a long-duration project.

3.1 Comparative Overview

Model	Design / Spec	Coding	Review / QA	Context Window	Best Used For
Claude	Excellent	Good	Excellent	200K tokens	Long-doc coherence, nuanced writing, following complex instructions, structured output, SRS/architecture
ChatGPT (GPT-4o)	Excellent	Excellent	Good	128K tokens	Creative brainstorming, UX ideation, broad knowledge breadth, accessible explanations
Gemini 1.5 Pro	Good	Good	Good	1M tokens	Massive codebase review, multi-file analysis, data-heavy tasks; long-context leader
Microsoft Copilot	Good	Fair	Good	~128K tokens	Office integration, enterprise Azure workflows, Teams/O365 context
Grok 3	Good	Good	Fair	~128K tokens	Recent events, scratchpad reasoning, alternative perspectives
Local LLMs (Ollama/LM Studio)	Fair	Fair	Fair	4K–32K typically	Air-gapped/confidential code, cost elimination, privacy-sensitive IP; quality varies by model
Cursor / GitHub Copilot	N/A	Excellent	Good	Varies	IDE-native code completion, inline refactors, diff-aware edits; not suited for design/spec work
Open-Source Coding Models (DeepSeek, Codestral)	Fair	Excellent	Fair	4K–32K	Low-cost code gen at scale, local deployment, boilerplate-heavy tasks

3.2 Model-Specific Notes

Claude (Anthropic)

Claude is the recommended default for specification, architecture, and review work because of its strong ability to maintain consistency across long documents, follow multi-step instructions without drift, and produce structured outputs (tables, JSON, XML) accurately. Claude's refusal to invent citations is a significant advantage for specification work where hallucinated requirements are dangerous. Claude 3.5 Sonnet offers the best balance of speed and quality for most stages; Claude 3 Opus is appropriate for highest-stakes architectural decisions.

ChatGPT (OpenAI GPT-4o)

GPT-4o is strongest in the ideation and UX design stages due to its broad knowledge base and willingness to explore alternatives rapidly. It performs well at generating multiple competing design concepts. Its main weakness in specification work is a higher tendency to add plausible-sounding but unfounded assumptions — use the anti-hallucination prompts included in Section 5 when using GPT-4o for specification tasks.

Gemini 1.5 Pro / Gemini 2.0 (Google)

Gemini's primary advantage is its one-million-token context window, which makes it the correct choice when reviewing entire codebases, comparing a full SRS against a full codebase, or any task where the input data volume exceeds Claude's or GPT-4o's window. Use Gemini for QA and consistency reviews on large projects.

Microsoft Copilot

Copilot's strongest use case within this methodology is document production inside the Microsoft Office ecosystem and for organizations already embedded in Azure. It is not recommended as a first-choice model for standalone specification or code generation due to higher observed inconsistency in long-form structured outputs.

Grok (xAI)

Grok 3 with its 'Think' mode (extended chain-of-thought reasoning) is useful as a second opinion on architectural decisions and for identifying edge cases the primary model may have missed. Its real-time web access makes it suitable for researching current library versions, known CVEs, and technology compatibility.

Local LLMs (Ollama, LM Studio, Jan)

Local models are the correct choice when source code or specifications contain commercially sensitive IP that must not be transmitted to third-party servers. Quality has improved significantly with models such as Llama 3.1 70B and DeepSeek Coder V2. Expect reduced performance on complex multi-step reasoning tasks compared to frontier API models. Always test a local model against a sample task before committing it to a stage in your workflow.

IDE-Integrated Coding Tools (Cursor, GitHub Copilot)

These tools operate inside the development environment and are optimized for file-aware code completion, refactoring, and inline editing. They are not suitable for the design, specification, or architecture stages. Cursor in particular performs well in Stage 5 (Software Development) for file-by-file implementation guided by a plan produced in Stage 4.

Specialized Tools Worth Monitoring

- Devin / SWE-agent: Autonomous software engineering agents still experimental as of mid-2025. Not recommended for production workflows without human-in-the-loop oversight at every commit.
- Sweep AI: Automated PR creation from GitHub issues. Appropriate for small, well-specified bug fixes.
- CodeRabbit: AI code review integrated into GitHub/GitLab pull requests. Appropriate as a supplement to Stage 6 (QA).

4. Stage-by-Stage Procedures

Each stage below defines the objective, inputs, outputs, success criteria, failure modes, AI recommendations, and prompt templates.

Stage 1: Product Design & UX

Stage 1: Product Design & UX	
Objective	Transform a raw problem statement into a validated product design, including user personas, user journeys, workflow definitions, screen/interaction concepts, and a documented set of edge cases ready for specification.
Inputs	Problem statement (written), target user description, any existing constraints (budget, timeline, platform), competitor references or inspiration
Outputs	Product Design Document (PDD) containing: problem statement, user personas (2–5), user journey maps, workflow diagrams (text or visual), screen/interaction wireframe descriptions, edge case catalog, open questions list
Success Criteria	All major user journeys are documented. At least three edge cases per journey are identified. A human decision-maker has reviewed and approved the PDD before Stage 2 begins.
Failure Modes	AI invents users or scenarios that do not match the actual target market. AI produces a design so broad it cannot be specified. Human reviewer skips approval and proceeds to Stage 2.
Review Checkpoint	Human decision-maker reviews PDD against original problem statement. Red-flag check: does any persona or journey feel unfamiliar or unexpected? If yes, return to AI with corrections.

AI Recommendations — Stage 1

Priority	Model	Rationale	Tradeoffs / Alternatives
1st Choice	Claude	Follows multi-step instructions precisely, produces well-structured PDD output, avoids inventing facts about users	ChatGPT is preferable if you want many competing concepts to choose from. Claude is preferable if you want one well-structured design to iterate. Grok can supplement either as a critic. Gemini adds little value at this stage unless your context input is extremely large.
2nd Choice	ChatGPT (GPT-4o)	Excellent creative ideation, generates multiple competing design directions	

Priority	Model	Rationale	Tradeoffs / Alternatives
		quickly	
3rd Choice	Grok 3	Provides alternative perspectives and edge cases the primary model may miss	

Prompt Templates — Stage 1

Starter Prompt

STAGE 1 — STARTER PROMPT

```
You are a senior product designer and UX researcher.  I am building: [DESCRIBE PRODUCT IN ONE SENTENCE] The problem it solves: [DESCRIBE THE PROBLEM] Target users: [DESCRIBE WHO WILL USE THIS] Key constraints: [PLATFORM, BUDGET, TIMELINE, TECHNICAL CONSTRAINTS IF KNOWN]  Your task: 1. Define 2–4 user personas. For each: name, role, goal, key frustration, tech comfort level. 2. Write a user journey map for each persona's primary use case. Include: trigger, steps, decision points, exit points. 3. List the top 5 screens or interaction points the product needs. 4. Identify at least 3 edge cases per user journey. 5. List any open questions I must answer before requirements can be written.  IMPORTANT CONSTRAINTS: - Do not invent requirements I have not mentioned. - If you are uncertain about something, say so explicitly. Do not fill gaps with assumptions. - Flag any assumption you make with the label [ASSUMPTION]. - Do not recommend a technology stack. This stage is about users and workflows only.
```

Advanced Prompt

STAGE 1 — ADVANCED PROMPT (for complex products)

```
You are a senior product designer, UX researcher, and domain expert in [DOMAIN].  Context: - Product: [FULL DESCRIPTION] - Competitive landscape: [LIST 1–3 COMPETITORS AND WHAT THEY DO] - Non-goals: [LIST THINGS THIS PRODUCT WILL NOT DO] - Known risks: [USER ADOPTION RISKS, REGULATORY CONCERNS, ACCESSIBILITY REQUIREMENTS]  Your task: Produce a complete Product Design Document (PDD) with the following sections: 1. Problem Statement (one paragraph, precise) 2. User Personas (3–5 personas with attributes: name, role, goal, frustration, frequency of use, tech comfort, success metric) 3. User Journey Maps (primary and secondary journeys per persona; include happy path and error path) 4. Workflow Definitions (numbered steps for each primary workflow) 5. Screen Inventory (list all screens/views with one-sentence description of each) 6. Edge Case Catalog (organized by journey; minimum 3 per journey) 7. Open Questions (numbered list of decisions I must make before writing requirements) 8. Out-of-Scope Items (explicitly list what this product will NOT do)  Label every assumption [ASSUMPTION]. Label every uncertainty [UNCERTAIN]. Do not present guesses as facts.
```

Review/Checklist Prompt

STAGE 1 — REVIEW PROMPT

```
Review the following Product Design Document and act as a critical design reviewer.  [PASTE PDD HERE]  Check for: 1. Are all user personas distinct and realistic? Flag any that seem generic or implausible. 2. Does each user journey have a clear start, end, and at least one error/edge case path? 3. Are there any requirements hidden inside the design that should be explicitly called out? 4. Are there missing user types or use cases for this type of product? 5. Does the design contain any internal contradictions? 6. List any assumptions that were not labeled [ASSUMPTION] in the original document. 7. What is the single highest-risk design decision and why?  Respond with a structured critique. Do not rewrite the document — flag specific issues with section references.
```

DECISION GATE: Stage 1 → Stage 2		
Criterion	PASS: PDD reviewed and approved by human decision-maker. All open questions from Stage 1 are answered or explicitly deferred with documented rationale.	FAIL: Return to Stage 1 with reviewer feedback. Do not proceed to Stage 2 until all critical open questions are resolved.

Stage 2: Requirements Specification

Stage 2: Requirements Specification	
Objective	Convert the approved Product Design Document into a complete, numbered, unambiguous Software Requirements Specification (SRS) that can be handed to a developer or coding AI without further clarification.
Inputs	Approved Product Design Document (PDD) from Stage 1, answered open questions list, any regulatory or compliance requirements
Outputs	Software Requirements Specification (SRS) containing: functional requirements (numbered, each with acceptance criteria), non-functional requirements (performance, scalability, accessibility, compatibility), system constraints, data model sketch, assumption register, out-of-scope register
Success Criteria	Every functional requirement has at least one acceptance criterion expressed as a testable statement. No requirement is ambiguous (can be interpreted two ways). All open questions from Stage 1 are resolved. Human decision-maker has reviewed and signed off.
Failure Modes	AI invents requirements not in the PDD. Requirements are expressed as vague goals ('system should be fast') rather than measurable criteria. Conflicting requirements are present without resolution.
Review Checkpoint	Human reviewer validates every requirement against the PDD. Run the checklist prompt against the SRS. Verify that no new concepts appear in the SRS that were absent from the PDD.

AI Recommendations — Stage 2

Priority	Model	Rationale	Tradeoffs / Alternatives
1st Choice	Claude	Produces numbered, structured SRS with consistent format; strong at identifying ambiguities; follows 'no hallucination' instructions reliably	Use Claude for primary SRS generation. Use ChatGPT to generate a competing first draft if you want to compare approaches. Use Gemini as a completeness reviewer when the combined PDD + SRS exceeds Claude's context

Priority	Model	Rationale	Tradeoffs / Alternatives
			window.
2nd Choice	ChatGPT (GPT-4o)	Can rapidly generate first drafts; good at completeness checks on a draft SRS	
3rd Choice	Gemini 1.5 Pro	Suitable if PDD is very long (>50 pages); its large context window ensures the entire document is considered	

Prompt Templates — Stage 2

Starter Prompt

STAGE 2 — STARTER PROMPT

```
You are a senior business analyst and technical writer.  I will provide a Product Design Document. Your task is to convert it into a Software Requirements Specification (SRS). [PASTE APPROVED PDD HERE]  Instructions: 1. Create FUNCTIONAL REQUIREMENTS. Number each FR-001, FR-002, etc. For each, write:    - Requirement: one precise, testable sentence    - Acceptance Criteria: one or more 'Given/When/Then' statements - Priority: Must Have / Should Have / Nice to Have    - Source: reference the PDD section it came from  2. Create NON-FUNCTIONAL REQUIREMENTS (NFR-001, etc.):    - Performance targets (response time, throughput)    - Availability / uptime - Security (authentication, authorization, data protection) - Compatibility (browsers, OS, devices)    - Accessibility (WCAG level if applicable)  3. Create a CONSTRAINTS section (technology limitations, regulatory requirements, timeline) 4. Create an ASSUMPTIONS REGISTER (list every assumption the SRS depends on)  5. Create an OUT-OF-SCOPE section (copy from PDD; expand as needed)  6. Create a GLOSSARY for all domain terms used  CRITICAL: Do not add any requirement that is not traceable to the PDD. If you identify a gap, list it in an OPEN ISSUES section — do not invent a resolution. Label every uncertainty [UNCERTAIN].
```

Advanced Prompt

STAGE 2 — ADVANCED PROMPT

```
You are a senior business analyst, technical writer, and software architect.  CONTEXT: - Product Design Document: [PASTE OR REFERENCE PDD] - Target platform: [WEB / MOBILE / DESKTOP / API] - User volume estimate: [EXPECTED CONCURRENT USERS] - Regulatory environment: [GDPR / HIPAA / PCI-DSS / NONE / OTHER] - Integration dependencies: [LIST EXTERNAL SYSTEMS]  TASK: Produce a complete SRS with these additional sections:  7. DATA REQUIREMENTS: For each entity in the system, define:    - Entity name    - Attributes (name, type, constraints, description)    - Relationships to other entities - Retention and deletion rules  8. INTERFACE REQUIREMENTS: For each UI screen or API endpoint:    - Input data    - Output data    - Error states and messages  9. TRACEABILITY MATRIX: A table mapping each FR to the PDD section and the persona/journey that drives it  10. RISK REGISTER: For each requirement identified as high-risk, document the risk, probability, impact, and mitigation approach  Apply these quality rules without exception: - Every requirement starts with 'The system shall...' - No requirement contains the word 'user-friendly', 'fast', 'secure', or any other unmeasurable adjective without a quantified threshold - Every requirement is numbered and referenced consistently throughout the document
```

Review/Checklist Prompt

STAGE 2 — REVIEW PROMPT

```
You are a senior QA analyst reviewing an SRS for defects before it is handed to an architect.  [PASTE SRS HERE]  Check for: 1. Completeness: Are there user journeys in the PDD that have no corresponding requirements? 2. Ambiguity: List every requirement that contains unmeasurable language (e.g., 'should be fast', 'easy to use'). 3. Contradiction: Identify any pair of requirements that conflict. 4. Testability: Flag any requirement where it would be unclear how to write a passing/failing test. 5. Traceability: Are there requirements with no clear source in the PDD? 6. Hidden assumptions: List any assumption embedded in a requirement but not declared in the Assumptions Register. 7. Missing NFRs: What common non-functional requirements are absent (security, error handling, logging, audit trails)? 8. Data gaps: Are there data entities implied by requirements but not defined in the data section? Output a numbered defect list. Each entry: defect type, requirement ID, description, suggested fix.
```

DECISION GATE: Stage 2 → Stage 3		
Criterion	PASS: SRS has zero open defects rated Critical. Human decision-maker has reviewed and approved. Assumption Register is complete.	FAIL: Return to Stage 2 with defect list. Re-run review prompt after corrections.

Stage 3: Technical Architecture

Stage 3: Technical Architecture	
Objective	Select the technology stack, define the system architecture, design the database schema, specify all APIs and integrations, and address security, scalability, deployment, and observability — producing an Architecture Decision Record (ADR) that fully constrains Stage 4 planning.
Inputs	Approved SRS from Stage 2, non-functional requirements, infrastructure preferences/constraints, team skill set
Outputs	Architecture Decision Record (ADR) containing: technology stack selection with rationale, system architecture diagram (described in text), database schema, API specifications, security architecture, deployment architecture, observability plan, known risks and mitigations
Success Criteria	All NFRs from the SRS are addressed by the architecture. No major technology decision is undocumented. Security threat model is present. A developer could begin implementation with no architectural questions unanswered.
Failure Modes	AI selects a fashionable but inappropriate technology stack. Security is treated as an afterthought. Database design does not support all required data relationships. Architecture is over-engineered for the scale requirements.
Review Checkpoint	Technical human reviewer (or second AI model) evaluates: does the stack match the team's skills? Does the architecture meet every NFR? Is the security model adequate for the data sensitivity?

> **NOTE:** *Security threat modeling belongs here, not only in Stage 7. Identifying threats at architecture time is 10–100x cheaper to remediate than identifying them post-implementation.*

AI Recommendations — Stage 3

Priority	Model	Rationale	Tradeoffs / Alternatives
1st Choice	Claude	Produces comprehensive ADRs with rationale; follows structured output	For greenfield projects with well-scoped SRS, Claude alone is sufficient. For projects with complex data models, run both Claude and Gemini and

Priority	Model	Rationale	Tradeoffs / Alternatives
		instructions; reliable for security architecture reasoning	compare their database schemas. Use Grok to validate that selected library versions have no known critical CVEs before committing to the stack.
2nd Choice	Gemini 1.5 Pro / 2.0	Superior when reviewing a very large SRS to ensure architecture addresses all requirements; strong at database design for complex schemas	
3rd Choice	Grok 3	Useful for researching current library versions, known issues, and compatibility between stack components via real-time web access	

Prompt Templates — Stage 3

Starter Prompt

STAGE 3 — STARTER PROMPT

```
You are a senior software architect.  I have an approved
Software Requirements Specification. Design a technical
architecture to implement it.  [PASTE SRS HERE]  Constraints:
- Team skill set: [LIST LANGUAGES / FRAMEWORKS THE TEAM KNOWS]
- Infrastructure preference: [CLOUD PROVIDER / ON-PREM / NO
PREFERENCE] - Budget tier: [STARTUP / SMB / ENTERPRISE] -
Existing systems to integrate: [LIST]  Produce: 1. TECHNOLOGY
STACK DECISION: For each component (frontend, backend,
database, auth, hosting, CI/CD), state: chosen technology,
rationale, alternatives considered, reason alternatives were
rejected.  2. SYSTEM ARCHITECTURE: Describe the major
components and how they interact. Use clear text descriptions
of data flow. List all external integrations.  3. DATABASE
SCHEMA: For each table/collection, list fields, types,
indexes, and relationships. Use standard notation (e.g.,
'users (id PK, email UNIQUE, created_at TIMESTAMP)').  4. API
SPECIFICATION OUTLINE: For each major API group, list
endpoints, HTTP methods, request/response shapes, and
authentication requirements.  5. SECURITY ARCHITECTURE:
Authentication method, authorization model (RBAC/ABAC/etc.),
data encryption at rest and in transit, input validation
approach, secrets management.  6. DEPLOYMENT ARCHITECTURE:
Environments (dev/staging/prod), containerization strategy,
scaling approach.  7. RISK REGISTER: Top 5 architectural risks
and mitigations.  Label every assumption [ASSUMPTION]. Do not
recommend a technology you would not stake your professional
reputation on.
```

Review/Checklist Prompt

STAGE 3 — REVIEW PROMPT

```
You are a senior software architect and security engineer
reviewing an Architecture Decision Record.  [PASTE ADR HERE]
[PASTE SRS SUMMARY OR KEY NFRs HERE]  Review for: 1. NFR
Coverage: Does the architecture explicitly address every non-
functional requirement in the SRS? 2. Security gaps: Is there
a threat model? Are authentication, authorization, input
validation, and secrets management addressed? 3. Scalability:
Will the architecture support the stated user volume? What
breaks first? 4. Database: Does the schema support all data
relationships implied by the requirements? Are indexes
defined? 5. API completeness: Are all user-facing and
integration requirements servable by the described API
surface? 6. Operational gaps: Is observability (logging,
monitoring, alerting) addressed? 7. Dependency risk: Are any
selected libraries or services single points of failure or
known for instability? 8. Cost: Is there a plausible cost
estimate for the described infrastructure at target scale?
Output: numbered findings list with severity (Critical / Major
/ Minor) and recommended action.
```

DECISION GATE: Stage 3 → Stage 4		
Criterion	PASS: ADR reviewed with zero Critical findings. All SRS NFRs are traceable to the architecture. Security threat model is present.	FAIL: Return to Stage 3 with findings. Do not begin implementation planning without an approved architecture.

Stage 4: Implementation Planning

Stage 4: Implementation Planning	
Objective	Translate the approved architecture and SRS into a sequenced list of atomic, independently testable development tasks. Each task must be small enough to be completed in a single AI coding session or a single developer sprint.
Inputs	Approved SRS, approved ADR from Stage 3
Outputs	Implementation Plan containing: ordered task list (each task has: ID, title, description, acceptance test, dependencies, estimated complexity, assigned stage/module, definition of done)
Success Criteria	All SRS functional requirements are covered by at least one task. No task is vague or oversized. Dependencies are clearly sequenced. A developer (or AI coding session) can execute each task without referring back to the architect.

Failure Modes	Tasks too large to complete in one session (AI context window overflow). Tasks that bundle unrelated concerns. Missing tasks for non-functional requirements (auth, error handling, logging). No dependency ordering.
Review Checkpoint	Cross-reference task list against SRS: every FR-xxx should appear in at least one task description.

AI Recommendations — Stage 4

Priority	Model	Rationale	Tradeoffs / Alternatives
1st Choice	Claude	Produces well-structured task breakdowns; reliable at maintaining consistent task format; good at identifying hidden dependencies	Claude is the default. If the project is so large that the full SRS + ADR + implementation plan conversation overflows Claude's context, switch to Gemini for this stage.
2nd Choice	ChatGPT (GPT-4o)	Can rapidly draft a task list; useful for brainstorming task breakdown alternatives	
3rd Choice	Gemini 1.5 Pro	Suitable for very large projects where the combined SRS + ADR exceeds Claude's context	

Prompt Templates — Stage 4

Starter Prompt

STAGE 4 — STARTER PROMPT

```
You are a senior engineering manager creating an implementation plan.  INPUTS: - SRS: [PASTE SRS OR KEY SECTIONS] - Architecture: [PASTE ADR SUMMARY] - Technology stack: [LANGUAGE / FRAMEWORK / DATABASE]  Your task: Break the product into discrete implementation tasks. For each task: - TASK-ID: (e.g., TASK-001) - Title: short imperative phrase - Description: what to build in 2–4 sentences - Acceptance Test: one or more statements of how a human can verify this task is complete - Dependencies: list TASK-IDs that must be complete before this task starts - Complexity: Small (< 2 hours) / Medium (2–8 hours) / Large (> 8 hours — should be broken down further) - Requirements covered: list FR/NFR IDs from SRS  ORDERING RULES: 1. Infrastructure and scaffolding tasks come first. 2. Authentication and authorization before any feature that requires them. 3. Database schema before any task that reads or writes data. 4. API endpoints before UI components that consume them.  Produce the complete task list. Flag any functional requirement that has no corresponding task.
```

DECISION GATE: Stage 4 → Stage 5		
Criterion	PASS: Every SRS functional requirement maps to at least one task. No Large complexity tasks remain (must be broken down). Human decision-maker has reviewed sequencing.	FAIL: Return to Stage 4. Break down Large tasks. Resolve unmapped requirements.

Stage 5: Software Development

Stage 5: Software Development	
Objective	Implement each task from the implementation plan, producing production-quality code with unit tests, following the architecture and meeting the acceptance criteria defined in Stage 4.
Inputs	Approved implementation plan, approved ADR (for stack and patterns), SRS (for acceptance criteria reference)
Outputs	Source code for all tasks, unit tests with coverage report, build instructions, dependency manifest (e.g., package.json, requirements.txt)

Success Criteria	All unit tests pass. Code builds without errors. Each completed task satisfies its acceptance test from Stage 4. No commented-out code. No TODO comments without an associated task ID.
Failure Modes	AI generates code that works in isolation but fails when integrated (interface mismatch). AI uses libraries not in the approved stack. AI produces code that compiles but does not satisfy the acceptance criteria. Context window overflow causes AI to 'forget' the architecture mid-task.
Review Checkpoint	After each TASK batch: run tests, verify acceptance criteria, check for stack compliance. Commit to version control before starting next batch.

AI Recommendations — Stage 5

Priority	Model	Rationale	Tradeoffs / Alternatives
1st Choice	Cursor / Claude (in Cursor)	Best for IDE-native development; diff-aware, file-aware; excellent for multi-file implementations	Cursor is the recommended environment for Stage 5 because it maintains file context automatically. Claude chat is preferable for planning a single complex function or debugging. Local coding models (DeepSeek, Codestral) are appropriate when code is commercially sensitive and must not leave the local environment.
2nd Choice	Claude (chat)	Best for complex logic, algorithm design, debugging sessions requiring extended reasoning	
3rd Choice	GitHub Copilot / DeepSeek Coder	Best for boilerplate generation, autocomplete-style development; lower cost at high volume	

Prompt Templates — Stage 5

Starter Prompt

STAGE 5 — STARTER PROMPT (per task)

```
You are a senior [LANGUAGE] developer.  CONTEXT: - Project: [ONE SENTENCE DESCRIPTION] - Technology stack: [FULL STACK DETAILS] - Architecture pattern: [MVC / REST / MICROSERVICES / ETC] - Task ID: [TASK-XXX] - Task title: [TITLE] - Task description: [DESCRIPTION] - Acceptance criteria: [CRITERIA FROM STAGE 4] - Depends on: [LIST COMPLETED TASK IDS]  EXISTING CODE CONTEXT: [PASTE RELEVANT EXISTING FILES OR INTERFACES]  Your task: 1. Implement the task to satisfy the acceptance criteria. 2. Write unit tests that verify each acceptance criterion. 3. Follow the existing code style and architecture pattern. 4. Do not use any library not already in the dependency manifest. 5. Do not implement anything outside the scope of this task.  After the code, provide: - A brief explanation of your implementation approach - Any edge cases you handled - Any edge cases you did NOT handle (explicitly list these) - Any assumption you made, labeled [ASSUMPTION]
```

Advanced Prompt (debugging)

STAGE 5 — DEBUGGING PROMPT

```
You are a senior [LANGUAGE] debugger.  I have the following failing test or error: [PASTE ERROR / FAILING TEST OUTPUT]  Relevant code: [PASTE CODE]  Relevant context (architecture, stack, data model): [PASTE RELEVANT CONTEXT]  Your task: 1. Identify the root cause of the failure. State your confidence (1–10). 2. Propose the minimal code change that fixes the issue without affecting other functionality. 3. Explain why the change fixes it. 4. Identify any related code that might be affected by this change. 5. If you are not confident (below 7/10), list what additional information you would need to diagnose this with higher certainty.  Do not guess. Do not rewrite code that is not related to the failure.
```

DECISION GATE: Stage 5 → Stage 6		
Criterion	PASS: All planned tasks are implemented. All unit tests pass. Code is committed to version control. Build is reproducible from a clean checkout.	FAIL: Return to Stage 5. Failing tests must be resolved. Do not carry failing tests into QA.

Stage 6: Quality Assurance

Stage 6: Quality Assurance	
Objective	Systematically verify that all implemented code meets the requirements defined in the SRS, identify defects, identify missing functionality, identify regressions, and produce a prioritized defect report.
Inputs	Source code, SRS, implementation plan, unit test results, build artifacts
Outputs	QA Report containing: test coverage summary, defect list (each with severity, steps to reproduce, expected vs. actual behavior, SRS reference), missing functionality list, regression list, test traceability matrix
Success Criteria	All Must Have requirements have a passing test. No Severity-1 (system-breaking) defects remain open. All defects have assigned severity, steps to reproduce, and SRS traceability.
Failure Modes	QA performed only by the same AI that wrote the code (conflict of interest). Severity ratings are inconsistent. Defects lack reproducible steps. Missing functionality is not identified because the AI reviewer does not have access to the SRS.
Review Checkpoint	Use a different AI model for QA review than was used for code generation. Human spot-checks at least 20% of defects for accuracy of reproduction steps.

AI Recommendations — Stage 6

Priority	Model	Rationale	Tradeoffs / Alternatives
1st Choice	Claude	Reliable, structured defect reporting; follows SRS traceability instructions; consistent severity rating	Always use a model different from the one that generated the code for primary QA. This reduces the risk that the reviewing model rationalizes its own errors. When the codebase is large, Gemini's million-token context makes it the superior choice.
2nd Choice	Gemini 1.5 Pro	Best choice when the full codebase plus SRS exceeds 50K tokens; can hold entire project context	
3rd Choice	ChatGPT (GPT-4o)	Useful secondary reviewer; catches different categories of defects than Claude	

Prompt Templates — Stage 6

Starter Prompt

STAGE 6 — QA REVIEW PROMPT

```
You are a senior QA engineer performing a code review against a requirements document.  SRS (or key sections): [PASTE SRS]  Code to review: [PASTE CODE OR DESCRIBE FILE STRUCTURE]  Unit test results: [PASTE TEST OUTPUT]  Your task: 1. For each functional requirement (FR-xxx), state: PASS / FAIL / NOT TESTED — and explain why. 2. List all defects found. For each:    - Defect ID (BUG-001, BUG-002, etc.)    - Severity: Critical (system fails) / Major (feature fails) / Minor (cosmetic or edge case)    - Requirement reference (FR-xxx or NFR-xxx)    - Steps to reproduce    - Expected behavior    - Actual behavior    - Suggested fix (optional) 3. List any requirements for which no code implementation exists. 4. List any code functionality for which there is no corresponding requirement (scope creep). 5. List any unit tests that are testing the wrong thing or provide false assurance.  Do not rationalize failures as acceptable. Report what you find.
```

DECISION GATE: Stage 6 → Stage 7		
Criterion	PASS: No Critical or Major defects remain open. All Must Have requirements show PASS status. Defect list has been reviewed and prioritized by a human.	FAIL: Return to Stage 5 for defect fixes. Re-run QA review after each fix batch.

Stage 7: Security Review

Stage 7: Security Review	
Objective	Perform a systematic security audit of the implemented code, architecture, and configuration. Identify vulnerabilities, authentication and authorization weaknesses, data protection gaps, and compliance risks. Produce a prioritized remediation plan.
Inputs	Source code, ADR, SRS (NFR security requirements), deployment configuration, database schema
Outputs	Security Audit Report containing: threat model validation, OWASP Top 10 review, authentication/authorization findings, data protection findings, dependency vulnerability scan results, compliance gap analysis (if applicable), remediation plan with priority

Success Criteria	All Critical and High severity vulnerabilities are remediated or have an accepted risk with documented rationale. OWASP Top 10 review is complete. Dependency scan shows no known Critical CVEs.
Failure Modes	Security review is superficial (general advice not tied to specific code). AI confabulates CVEs or vulnerability details. Authentication review does not examine actual token handling code. Dependency scan is not performed against actual package manifest.
Review Checkpoint	Human security reviewer (or a second AI model) cross-checks: does the threat model match the actual deployment architecture? Are any high-severity findings suspiciously absent for this type of application?

AI Recommendations — Stage 7

Priority	Model	Rationale	Tradeoffs / Alternatives
1st Choice	Claude	Thorough, structured security analysis; strong at reviewing authentication/authorization logic; good at compliance gap analysis	Run both Claude and ChatGPT independently on the same security review prompt and compare findings. Use Grok to validate CVE claims made by either model against live vulnerability databases. Never rely on a single AI model for security review of a production system.
2nd Choice	ChatGPT (GPT-4o)	Strong OWASP knowledge; useful second reviewer; provides different remediation recommendations	
3rd Choice	Grok 3	Real-time web access allows checking current CVE databases and library vulnerability disclosures	

Prompt Templates — Stage 7

Starter Prompt

STAGE 7 — SECURITY REVIEW PROMPT

```
You are a senior application security engineer.  Application
type: [WEB APP / MOBILE APP / API / DESKTOP APP]
Authentication method: [JWT / SESSION COOKIE / OAUTH / OTHER]
User data stored: [DESCRIBE PII, FINANCIAL DATA, HEALTH DATA,
ETC.] Regulatory requirements: [GDPR / HIPAA / PCI-DSS / NONE]
Code / architecture to review: [PASTE CODE, SCHEMA, OR
ARCHITECTURE DESCRIPTION]  Perform a security review covering:
1. AUTHENTICATION: Are tokens properly generated, stored,
transmitted, and expired? 2. AUTHORIZATION: Is every endpoint
protected? Is role-based access control correctly implemented?
3. INPUT VALIDATION: Is all user input validated and sanitized
before processing or storage? 4. INJECTION RISKS: SQL
injection, command injection, XSS, SSRF. Is parameterized
querying used? 5. DATA PROTECTION: Is sensitive data encrypted
at rest? In transit? Are secrets in environment variables? 6.
DEPENDENCY RISKS: List any libraries with known
vulnerabilities (state your confidence in this claim). 7.
ERROR HANDLING: Do error messages leak sensitive information?
8. LOGGING: Are security events (failed logins, privilege
escalations) logged? 9. CSRF / CORS: Are cross-origin policies
correctly configured? 10. COMPLIANCE: Does the implementation
meet stated regulatory requirements?  For each finding: -
Severity: Critical / High / Medium / Low - Description of the
vulnerability - Location in code (if applicable) - Recommended
remediation - Confidence in this finding (1–10) — if below 7,
explain why  Do not make up CVEs or vulnerability details. If
you are uncertain, say so.
```

DECISION GATE: Stage 7 → Stage 8		
Criterion	PASS: All Critical and High severity security findings are remediated. Dependency scan shows no Critical CVEs. Threat model is documented.	FAIL: Return to Stage 5 for security remediation. Do not ship code with open Critical security findings.

Stage 8: Documentation

Stage 8: Documentation

Objective	Produce user-facing and technical documentation that accurately reflects the implemented system. Documentation must be written after the code is stable, not during development, to prevent describing planned rather than actual behavior.
Inputs	Final source code, SRS, ADR, QA-signed-off build
Outputs	User documentation (user guide or README), API documentation (if applicable), developer setup guide, architecture summary, deployment runbook
Success Criteria	User documentation covers all Must Have features. API documentation matches all endpoints in code. Developer can set up a working development environment using only the setup guide. No documentation references features that do not exist.
Failure Modes	Documentation is written from the SRS rather than the actual code (describes planned not actual behavior). AI invents API endpoint parameters or return values. Documentation is missing error handling instructions.
Review Checkpoint	Human spot-check: follow the setup guide from scratch on a clean machine. Verify at least three API endpoint descriptions against actual code.

AI Recommendations — Stage 8

Priority	Model	Rationale	Tradeoffs / Alternatives
1st Choice	Claude	Produces clear, well-structured technical prose; follows precise instructions; accurate at extracting API documentation from code	Use Claude for developer documentation and API reference generation. Use ChatGPT if your user-facing documentation needs a warmer, consumer-friendly tone. For confidential internal documentation, use a local model.
2nd Choice	ChatGPT (GPT-4o)	Excellent at adapting tone for user-facing documentation (accessible, friendly); good at writing tutorials	
3rd Choice	Local LLMs	Appropriate if documentation must remain confidential; quality varies by model	

Prompt Templates — Stage 8

Starter Prompt

STAGE 8 — DOCUMENTATION PROMPT

```
You are a senior technical writer.  I will provide the final, stable source code and architecture description. Your task is to write documentation for this system.  CODE / ARCHITECTURE: [PASTE FINAL CODE OR DESCRIBE KEY COMPONENTS]  AUDIENCE: [DEVELOPERS / END USERS / SYSTEM ADMINISTRATORS]  Produce: 1. README (for developers): Project description, prerequisites, installation steps, configuration, running tests, deployment 2. USER GUIDE (for end users): Feature descriptions keyed to actual implemented functionality only. Do not describe features that are not in the code. 3. API REFERENCE (if applicable): For each endpoint: method, path, description, request parameters, request body, response body, error codes, example request, example response  CRITICAL RULES: - Describe only what the code actually does. Do not describe what the SRS intended. - If you are uncertain about the behavior of a piece of code, say [VERIFY THIS] rather than guessing. - Do not invent example values for API responses — note where example values must be filled in by a human.
```

DECISION GATE: Stage 8 → Stage 9		
Criterion	PASS: All documentation deliverables are complete. Setup guide has been validated on a clean machine. No [VERIFY THIS] flags remain unresolved.	FAIL: Return to Stage 8. Resolve all verification flags before proceeding.

Stage 9: Consistency & Final Review

Stage 9: Consistency & Final Review	
Objective	Perform a cross-artifact consistency review to identify contradictions, requirement drift, undocumented changes, and any remaining gaps between the specification, code, tests, and documentation. Produce a final sign-off report.
Inputs	All deliverables from Stages 1–8: PDD, SRS, ADR, Implementation Plan, Source Code, QA Report, Security Report, Documentation
Outputs	Final Consistency Report: list of all contradictions found with resolution, requirement drift log, final traceability matrix (SRS requirement → code module → test → documentation), sign-off checklist
Success Criteria	No Critical contradictions remain. All Must Have requirements trace from SRS through code through test through documentation. Human decision-maker has signed the final sign-off checklist.

Failure Modes	Review is superficial because the AI does not have enough context to compare all artifacts. Contradictions are rationalized rather than flagged. Sign-off is performed without human review.
Review Checkpoint	This stage requires a human reviewer to read the final consistency report and make a go/no-go decision. AI cannot substitute for human accountability at this gate.

AI Recommendations — Stage 9

Priority	Model	Rationale	Tradeoffs / Alternatives
1st Choice	Gemini 1.5 Pro / 2.0	Million-token context allows the entire artifact set to be loaded simultaneously for genuine cross-artifact comparison	The key constraint for this stage is context window. If your combined deliverables exceed Claude's context, Gemini is the correct primary choice. For smaller projects, Claude produces more actionable, structured reports.
2nd Choice	Claude	Best choice when total artifact volume is under 150K tokens; produces structured, well-organized consistency reports	
3rd Choice	ChatGPT (GPT-4o)	Useful secondary reviewer; different model catches different classes of inconsistencies	

Prompt Templates — Stage 9

Starter Prompt

STAGE 9 — CONSISTENCY REVIEW PROMPT

```
You are a senior systems architect performing a final
consistency review before product release.  I am providing you
with the complete set of project deliverables.  PRODUCT DESIGN
DOCUMENT: [PASTE OR SUMMARIZE] SOFTWARE REQUIREMENTS
SPECIFICATION: [PASTE OR SUMMARIZE] ARCHITECTURE DECISION
RECORD: [PASTE OR SUMMARIZE] IMPLEMENTATION PLAN: [PASTE OR
SUMMARIZE] QA REPORT SUMMARY: [PASTE] SECURITY REPORT SUMMARY:
[PASTE] DOCUMENTATION SUMMARY: [PASTE]  Your task: 1.
CONTRADICTION SCAN: Identify every case where two or more
documents make conflicting statements about the same topic. 2.
REQUIREMENT DRIFT: Identify requirements that were present in
the SRS but are absent or altered in the code/documentation.
3. SCOPE CREEP: Identify functionality in the code that has no
corresponding requirement in the SRS. 4. DOCUMENTATION GAPS:
Identify SRS requirements that have no documentation coverage.
5. TEST GAPS: Identify SRS requirements that have no test
coverage. 6. FINAL TRACEABILITY: Produce a matrix: SRS
Requirement ID → Implementation → Test → Documentation
(PRESENT / ABSENT for each).  For each issue found: - Issue ID
(CON-001, etc.) - Severity: Critical / Major / Minor -
Description - Affected documents - Recommended resolution  Be
thorough. A missed contradiction at this stage becomes a
production defect.
```

DECISION GATE: Stage 9 → SHIP		
Criterion	PASS: Final Consistency Report shows zero Critical issues. Human decision-maker has signed the final sign-off checklist. All Must Have requirements show PRESENT in the traceability matrix.	FAIL: Return to the affected stage for each open Critical issue. Reassemble deliverables and re-run Stage 9 review.

5. Operational Playbook

This section is written for a non-technical project coordinator or founder. It assumes no software engineering background. It assumes access to at least one of the AI tools described in Section 3.

5.1 Before You Begin

Complete the following before starting Stage 1:

☐	Write a one-paragraph problem statement. Be specific about what problem exists and for whom.
☐	Identify who will be the human decision-maker (the person who approves each stage gate). This is usually the founder or product owner.
☐	Set up accounts for at least two AI tools (recommended: Claude and one other).
☐	Create a project folder. You will save a document from every stage.
☐	Decide how you will store your work: a cloud folder (Google Drive, Dropbox), a git repository, or a shared drive.
☐	Identify any non-negotiable constraints: platform, budget, timeline, regulations, existing technology.

5.2 Stage-by-Stage Procedures for Coordinators

How to Run a Stage

1. Copy the starter prompt for the stage from Section 4.
2. Fill in all [BRACKETED PLACEHOLDERS] with your project's information.
3. Paste the approved deliverable from the previous stage where indicated.
4. Submit the prompt to the recommended AI model.
5. Read the output. Flag anything that surprises you or seems wrong.
6. Run the review/checklist prompt with the AI output pasted in.
7. Review the checklist output. If defects are found, return to the AI with corrections.
8. When the output is satisfactory, save it to your project folder with the stage number and date.
9. Review the decision gate criteria. If all criteria are met, proceed to the next stage.
10. If any gate criterion is not met, return to the current stage and do not proceed.

How to Handle AI Errors

When an AI produces output that seems wrong, use the following escalation steps:

- Step 1: Re-read the output carefully. Is the AI actually wrong, or did you misunderstand?
- Step 2: Ask the AI to explain its reasoning: 'Why did you include X? What is your basis for that claim?'
- Step 3: If the AI cannot justify a claim, discard that specific output and note it as a gap.
- Step 4: Try the same prompt in a second AI model. Compare outputs.
- Step 5: If both models agree on something suspicious, research the claim independently before accepting it.
- Step 6: For high-stakes decisions (technology choice, security architecture, legal compliance), engage a human expert regardless of AI confidence.

> **WARNING:** *Never accept AI output that contains specific numerical claims (e.g., 'this algorithm is O(n log n)', 'this library has a 99.9% uptime SLA') without verifying the claim from the original source. AI models are reliable at structure and reasoning, but can confabulate specific facts.*

How to Handle Context Window Overflow

When a project grows large, a single AI conversation may not be able to hold all the context needed. Signs of overflow include: the AI contradicts what it said earlier in the same conversation; the AI ignores instructions it followed earlier; the AI seems to forget the project context.

- Option 1: Start a new conversation and paste a project summary before the current task.
- Option 2: Switch to Gemini 1.5 Pro, which has a much larger context window.
- Option 3: Break the task into smaller sub-tasks, each of which fits within one conversation.

How to Manage Requirement Changes

Requirements will change. The following rules prevent silent corruption of the project:

11. Every change must be documented in a change log before implementation begins.
12. Assess the impact: which stages are affected by this change?
13. Update each affected document before proceeding.
14. If a change affects Stages 1–3 (design, requirements, architecture), all downstream stages must be re-reviewed for impact.
15. Never implement a requirement change without first updating the SRS. The SRS is the single source of truth.

5.3 Quality Control Procedures

The Rule of Two AI Models

For any decision that has significant consequences (architecture, security, database design),

always obtain a second opinion from a different AI model. Different models have different failure modes. Agreement between two independent models increases confidence; disagreement flags a topic requiring human judgment.

The Human Approval Requirement

AI models can produce output but they cannot take responsibility. A human must review and approve every stage gate. This is not optional. The human approver is responsible for verifying that the output is sensible, complete, and consistent with the project's actual goals.

Version Control

Every approved stage deliverable must be saved with a version number and date. If a deliverable is revised, save the revision as a new version. Never overwrite the previously approved version. This creates an audit trail that allows you to identify when and why the project deviated from its original design.

5.4 Final Sign-Off Checklist

Before declaring the project ready to ship, the decision-maker must complete the following:

☐	Stage 1: Product Design Document is approved and saved.
☐	Stage 2: Software Requirements Specification is approved and saved.
☐	Stage 3: Architecture Decision Record is approved and saved.
☐	Stage 4: Implementation Plan is approved and saved.
☐	Stage 5: All planned tasks are implemented, all tests pass, code is in version control.
☐	Stage 6: QA Report shows no open Critical or Major defects.
☐	Stage 7: Security Report shows no open Critical or High vulnerabilities.
☐	Stage 8: All documentation deliverables are complete and validated.
☐	Stage 9: Final Consistency Report shows no open Critical issues.
☐	Stage 9: Final traceability matrix shows PRESENT for all Must Have requirements.
☐	Human decision-maker has personally reviewed at least the QA and Security reports.
☐	At least one AI model other than the primary development model has reviewed the final codebase.
☐	Deployment runbook has been tested in a staging environment.
☐	All team members (or contractors) have been briefed on what was built and what was not built.

6. Lessons Learned and Best Practices

6.1 What Works

- Treating AI output as a first draft, not a final answer, consistently produces better outcomes than accepting AI output uncritically.
- Using the same structured format (numbered requirements, consistent IDs) across all documents makes cross-stage traceability dramatically easier.
- Running the review/checklist prompt in a fresh conversation — without the generation context — catches more defects than reviewing within the same conversation.
- Saving approved deliverables to a shared folder before proceeding to the next stage eliminates the most common cause of downstream confusion.
- For code generation, providing the AI with the exact interface it must implement (function signatures, API contracts) before asking it to write code reduces integration errors significantly.

6.2 Common Mistakes to Avoid

- Skipping the Stage 1 human approval gate. The most expensive mistakes in this workflow originate from undetected misalignment in the product design. An hour of review at Stage 1 saves weeks of rework at Stage 5.
- Using the same AI conversation for both generation and review of the same artifact. A model will almost always rationalize its own errors when asked to review them. Always use a separate conversation, and ideally a separate model, for review.
- Accepting AI security recommendations without verification. AI models are trained on general security knowledge but may be unaware of specific vulnerabilities in specific library versions. Always verify security findings against current CVE databases.
- Treating the SRS as a living document that can be updated informally. The SRS is the contract between what was agreed and what was built. Informal changes introduce drift that causes the QA and consistency reviews to find false positives.
- Asking an AI to complete multiple stages in one prompt. Each stage has a distinct objective and requires the output of the previous stage as input. Combining stages produces outputs that satisfy neither stage completely.

6.3 Prompt Engineering Best Practices

- Always specify the AI's role in the first line of the prompt ('You are a senior software architect'). This conditions the model's vocabulary, level of detail, and assumptions.
- Always include explicit negative instructions ('Do not invent requirements not in the provided PDD'). Positive instructions alone are insufficient.

- Always request that the AI label its assumptions and uncertainties. Models that are instructed to flag uncertainty produce more reliable outputs than those asked to produce confident answers.
- For long documents, provide a summary at the top and the full document below. AI models attend more reliably to the beginning of their context than the middle.
- When a prompt produces a wrong answer, do not simply ask the AI to try again. Explain specifically what was wrong and why. Vague feedback produces only superficially different wrong answers.

6.4 When to Involve a Human Expert

AI tools are powerful accelerators, not replacements for human expertise in the following situations:

- Legal and regulatory compliance: AI can identify relevant regulations but cannot provide legal advice. Any compliance claim must be verified by a qualified professional.
- Security architecture for sensitive data: AI can identify common vulnerabilities, but penetration testing by a qualified security professional is required before handling healthcare, financial, or authentication data at scale.
- Novel technical decisions: When no model can provide a confident, consistent answer on a technical question, the question requires human expert judgment.
- Stakeholder communication: AI can help draft communications, but relationship-sensitive decisions (pricing, vendor selection, partner agreements) must be owned by a human.
- Final accountability: The human decision-maker is always responsible for the shipped product. AI tools shift work; they do not shift responsibility.

7. Appendices

Appendix A: Glossary

ADR — Architecture Decision Record. A document that records a significant architectural decision, the context in which it was made, and the rationale for the choice.

Acceptance Criteria — A set of conditions that a software feature must meet to be accepted as complete. Typically expressed as Given/When/Then statements.

Context Window — The maximum amount of text an AI model can process in a single conversation. Text beyond this limit is not considered by the model.

Decision Gate — A formal checkpoint between workflow stages. The project does not proceed to the next stage until all gate criteria are met.

Functional Requirement (FR) — A statement of what the system must do. Numbered sequentially (FR-001, FR-002, etc.).

Hallucination — The production by an AI model of factually incorrect or invented information stated with apparent confidence. A significant risk in specification and architecture work.

Non-Functional Requirement (NFR) — A constraint on how the system must perform (speed, security, availability) rather than what it must do.

PDD — Product Design Document. The output of Stage 1. Describes users, journeys, workflows, and edge cases.

Requirement Drift — The gradual, untracked divergence of implemented functionality from the original specification. The primary target of the Stage 9 consistency review.

SRS — Software Requirements Specification. The output of Stage 2. The complete, numbered, testable statement of what the system must do.

Traceability Matrix — A table that maps each requirement to its design origin, implementation location, test, and documentation. The primary tool for detecting requirement drift.

Appendix B: Document Version History

Version	Date	Description
1.0	June 2026	First publication of Multi-Model Methodology for Building Operational Programs
1.1	June 2026	Renamed, simplified, cleared up, edits...