Prompt Engineering Playbook

for Software Engineers

How to Use AI Effectively for Architecture, Design, Code, and Documentation

Covering: System Design, Code Review, Technical Writing, Research, and Decision-Making

Vincent Bevia
Software Engineer | MSc, University of Liverpool | MultiSafepay, Ant Group

Part I: Technical Research Prompts

Structured prompts for evaluating technologies, validating architectural decisions, and synthesising technical knowledge. Use these before committing to a stack, adopting a new tool, or writing an RFC.

1) Technology Stack Evaluation

Analyse [TECHNOLOGY / FRAMEWORK / PLATFORM] for use in a production environment. Cover:

- Maturity: release history, stability, breaking change frequency, LTS policy
- Ecosystem: library support, tooling, IDE integration, CI/CD compatibility
- Performance: benchmarks, known bottlenecks, scaling characteristics
- Operational burden: observability, debugging, deployment complexity
- Community: contributor activity, issue response time, documentation quality

Use the latest available sources. Prioritise GitHub activity, production post-mortems, and benchmark data over marketing content. Conclude with a clear recommendation and caveats. Cite all claims.

2) Library / SDK Comparison

Compare [LIBRARY A], [LIBRARY B], and [LIBRARY C] for [USE CASE]. Provide a table covering:

- API surface: ergonomics, type safety, error handling patterns
- Dependencies: transitive dependency count, security audit history
- Performance: memory footprint, CPU overhead, benchmark results
- Maintenance: release cadence, open issues, bus factor
- Licensing: permissive vs copyleft, commercial use implications

Summarise which option fits best for different constraints (startup vs enterprise, latency-sensitive vs throughput-optimised). Flag where documentation is misleading or benchmarks are outdated. Cite sources.

3) Architecture Pattern Validation

Validate whether [ARCHITECTURE PATTERN / APPROACH] is appropriate for [CONTEXT]. Gather:

- Evidence supporting adoption: production case studies, benchmark improvements, team velocity gains
- Evidence against: failure modes, hidden complexity, operational overhead
- Scale thresholds: at what load / team size / data volume does it make sense?

- Migration cost: what does adoption require in terms of refactoring, retraining, tooling?

Prioritise real-world post-mortems and engineering blog posts over conference talks. Conclude with a verdict: premature optimisation, situationally appropriate, or industry standard. Cite all sources.

4) Deep Technical Investigation

Research and answer: [SPECIFIC TECHNICAL QUESTION]. Pull from:

- Official documentation and specifications
- Source code analysis where relevant
- GitHub issues, RFCs, and design documents
- Production incident reports and post-mortems

Explain where the documentation is incomplete or misleading. Identify edge cases that are often overlooked. Distinguish between specification behaviour and implementation behaviour. Be explicit about uncertainty. Cite all claims.

5) Developer Experience Audit

Analyse real developer feedback on [TOOL / FRAMEWORK / SERVICE] using:

- GitHub issues: common bugs, feature requests, response times
- Stack Overflow: frequently asked questions, recurring confusion
- Reddit / HackerNews: migration stories, complaints, praise
- Discord / Slack communities: pain points, workarounds, tribal knowledge

Ignore vendor marketing and curated testimonials. Identify patterns: what breaks, what's hard to debug, what causes migration regret. Cite representative examples with links.

6) Security & Dependency Risk Assessment

Assess security posture and dependency risk for [LIBRARY / FRAMEWORK / SERVICE]. Cover:

- CVE history: severity distribution, patch response time, disclosure process
- Dependency chain: transitive dependencies, known vulnerable packages
- Attack surface: input handling, serialisation, authentication mechanisms
- Compliance: SOC 2, ISO 27001, GDPR implications, data residency
- Supply chain: maintainer trust, build reproducibility, signing practices

Reference CVE databases, security advisories, and audit reports. Flag any dependencies with abandoned maintenance. Provide actionable risk mitigation if adoption proceeds. Cite all sources.

7) Trade-off Analysis for Architecture Decisions

Analyse trade-offs for: [ARCHITECTURE DECISION, e.g., “monolith vs microservices”, “SQL vs NoSQL”, “sync vs async”]. Structure as:

- Option A: strongest arguments, ideal conditions, real-world wins
- Option B: strongest arguments, ideal conditions, real-world wins
- Hidden costs: what each option makes harder over time
- Reversal cost: how painful is it to switch later?
- Team factors: skill requirements, hiring implications, onboarding complexity

Use production case studies and post-mortems, not theoretical arguments. Highlight where evidence is strong vs where it’s mostly opinion. Conclude with decision criteria, not a single recommendation. Cite everything.

8) Technical Decision Summary

Summarise research to support a decision on [TECHNOLOGY CHOICE / ARCHITECTURE QUESTION]. Structure as:

- Known facts: what is well-established and uncontroversial
- Open questions: what is still unclear or context-dependent
- Key risks: what could go wrong, how likely, how detectable
- Recommended next steps: spikes, prototypes, proof-of-concepts, or decisions

Avoid speculation. Flag where engineering judgement is required vs where data exists. Cite all sources. This should be usable as input to an ADR or RFC.

Part II: Engineering-Specific Prompts

Targeted prompts for common engineering tasks: architecture, code review, debugging, and system design. These are designed for senior-level, production-aware outputs.

1) System Architecture Analysis

Analyse the architecture for [SYSTEM / SERVICE / FEATURE] using sources from documentation, design patterns, and production case studies. Break this into:

- Component boundaries and responsibilities
- Data flow and communication patterns
- Scalability characteristics and bottlenecks
- Failure modes and resilience patterns
- Operational concerns (monitoring, deployment, maintenance)

Avoid marketing language. Structure with clear headings and finish with trade-offs and recommendations. Cite relevant patterns, papers, or production examples.

2) Code Review Assistant

Review the following code as a senior engineer. Focus on:

- Correctness: logic errors, edge cases, off-by-one errors
- Security: input validation, injection risks, authentication/authorization
- Performance: algorithmic complexity, unnecessary allocations, N+1 queries
- Maintainability: naming, structure, separation of concerns
- Testing: missing test cases, testability concerns

Prioritise issues by severity. For each issue, explain the problem, why it matters, and suggest a fix. Do not nitpick style unless it affects readability significantly.

3) Debugging and Root Cause Analysis

Help me debug the following issue: [DESCRIBE SYMPTOM, ERROR MESSAGE, OR UNEXPECTED BEHAVIOUR]. Based on the information provided:

- List the most likely root causes, ranked by probability
- For each hypothesis, explain what evidence would confirm or rule it out
- Suggest specific diagnostic steps (logs to check, queries to run, tests to write)
- Identify what additional information you need to narrow down the cause

Be systematic. Avoid jumping to conclusions. Flag uncertainty.

4) API Design Review

Review the following API design (REST / GraphQL / gRPC / SDK) as a senior engineer. Evaluate:

- Consistency: naming conventions, resource structure, error handling
- Usability: discoverability, documentation needs, common use cases
- Evolvability: versioning strategy, breaking change risk, extensibility
- Performance: payload size, pagination, caching opportunities
- Security: authentication, authorization, rate limiting, input validation

Provide specific recommendations with rationale. Reference industry standards and best practices where applicable.

5) Database Schema Review

Review the following database schema as a senior engineer. Analyse:

- Normalisation: appropriate level, denormalisation trade-offs
- Indexing: missing indexes, over-indexing, composite index opportunities
- Data integrity: constraints, foreign keys, validation rules
- Query patterns: likely slow queries, N+1 risks, join complexity
- Scalability: partitioning needs, sharding considerations, growth projections

Provide specific recommendations. Flag potential migration challenges for suggested changes.

6) Performance Analysis

Analyse performance for [SYSTEM / ENDPOINT / FEATURE]. Based on the provided metrics, traces, or code:

- Identify the primary bottleneck (CPU, memory, I/O, network, external dependency)
- Estimate the impact of the bottleneck on latency and throughput
- Suggest optimisations ranked by effort vs impact
- Identify what additional profiling or instrumentation would help

Be specific about expected gains. Avoid premature optimisation — focus on measurable impact.

7) Migration Planning

Help me plan a migration from [CURRENT STATE] to [TARGET STATE]. Cover:

- Pre-migration assessment: dependencies, risks, blockers
- Migration strategy: big bang vs incremental, feature flags, dual-write
- Rollback plan: how to revert if issues arise

- Testing strategy: what to validate before, during, and after
- Communication plan: stakeholders to notify, runbook for the team

Prioritise risk mitigation. Identify what could go wrong and how to detect it early.

8) Incident Post-Mortem Assistant

Help me write a blameless post-mortem for the following incident: [DESCRIBE INCIDENT]. Structure as:

- Summary: what happened, impact, duration
- Timeline: key events in chronological order
- Root cause analysis: contributing factors, 5 whys
- What went well: detection, response, communication
- What could be improved: gaps in monitoring, process, tooling
- Action items: specific, assigned, time-bound improvements

Focus on systemic improvements, not individual blame. Prioritise action items by impact and effort.

Bonus: Meta Prompt for Engineering-Aware AI Assistant

Embed this as a system prompt for AI tools used by your engineering team:

> *You are a senior software engineer and system architect. You prioritise correctness, security, performance, and maintainability. You avoid hype and marketing claims. You distinguish between best practices, opinions, and hard requirements. You call out uncertainty and trade-offs. You structure answers with clear reasoning, concrete examples, and actionable recommendations. You cite sources when making factual claims.*

Part III: Prompt Engineering Playbook

1. Why Engineers Need Prompt Engineering

Software engineering sits at the intersection of:

- Correctness (logic, edge cases, security)
- Performance (latency, throughput, resource usage)
- Maintainability (readability, testing, documentation)
- Production reality (operations, monitoring, incident response)

Generic AI prompts produce shallow answers, outdated patterns, and stackoverflow-grade solutions that may not work in production.

Goal: Make AI behave like a senior engineer who has seen production failures, not a tutorial writer.

2. Core Principles for Engineering Prompts

2.1 Specify the Context

Always provide: language/framework version, scale (users, requests, data size), constraints (latency requirements, compliance, team size), and what you've already tried.

✗ Bad: "How do I cache database queries?"

✓ Good: "How should I cache PostgreSQL queries in a Python/FastAPI service handling 10k RPM with p99 latency requirements of 100ms? We're considering Redis but open to alternatives."

2.2 Ask for Trade-offs, Not "Best Practice"

In engineering there is no perfect solution — only trade-offs. Always include: "Explain the trade-offs and when each approach is appropriate."

2.3 Request Production Awareness

Ask explicitly for failure modes, operational concerns, and what can go wrong. Tutorial code and production code are different.

2.4 Force Evidence and Sources

For factual claims, always add: "Cite documentation, benchmarks, or production case studies. Flag where information is uncertain or evolving."

3. Prompt Patterns (Reusable Templates)

Pattern 1 – Architecture Review

Use when designing or reviewing system architecture.

> You are a senior software architect. Analyse the architecture for [SYSTEM]. Structure the answer as: Component boundaries, Data flow, Scalability characteristics, Failure modes, Operational concerns. Call out trade-offs. Cite relevant patterns or production examples.

Pattern 2 – Code Review

Use for thorough code review.

> Review this code as a senior engineer focused on production readiness. Check for: Correctness, Security, Performance, Maintainability, Testing gaps. Prioritise by severity. For each issue: explain the problem, why it matters, and suggest a fix.

Pattern 3 – Technology Evaluation

Use before committing to a technology choice.

> Compare [OPTION A], [OPTION B], [OPTION C] for [USE CASE]. Evaluate: Capabilities, Learning curve, Ecosystem maturity, Performance, Operational burden. Provide a decision matrix. Cite benchmarks and production case studies. Flag unknowns.

Pattern 4 – Failure Mode Analysis

Use to anticipate production failures.

> Analyse failure modes for [SYSTEM / FEATURE]. For each potential failure: Technical root cause, User impact, Detection strategy (metrics, alerts), Mitigation pattern. Prioritise by likelihood and severity.

Pattern 5 – Design Document Review

Use before circulating a design doc.

> Review this design document as a senior engineer. Identify: Unclear requirements, Missing alternatives considered, Gaps in failure handling, Security concerns, Operational blind spots, Missing success metrics. Prioritise feedback by impact.

4. High-Value Prompts for Day-to-Day Work

Architecture & Design

- “What are the trade-offs between microservices and a modular monolith for a team of 5 engineers building [USE CASE]?”
- “How should I design the retry and circuit breaker strategy for [EXTERNAL DEPENDENCY]?”

Code & Implementation

- “Review this function for edge cases, error handling, and potential race conditions.”

- “How can I refactor this code to be more testable without changing its behaviour?”

Debugging & Operations

- “Given this error trace and these metrics, what are the most likely root causes?”
- “What metrics and alerts should I set up to detect [FAILURE MODE] early?”

Documentation & Communication

- “Help me write an ADR (Architecture Decision Record) for choosing [OPTION] over [ALTERNATIVES].”

5. What NOT to Ask AI

Avoid vague questions:

- ✗ “Is this code good?”
- ✗ “What’s the best database?”
- ✗ “How do I scale?”

Instead, ask:

- ✓ “Review this code for security vulnerabilities and performance issues.”
- ✓ “Compare PostgreSQL, MongoDB, and DynamoDB for [SPECIFIC USE CASE] with [SPECIFIC REQUIREMENTS].”
- ✓ “How do I scale [SPECIFIC COMPONENT] from 1k to 100k requests per second? What are the bottlenecks?”

6. Safe Use Guidelines

6.1 Never Trust AI for Security-Critical Decisions

AI is a research accelerator, not a security auditor. Cryptography, authentication, and authorization code must be reviewed by experts and tested thoroughly.

6.2 Never Paste Secrets

Do not paste: API keys, passwords, tokens, production connection strings, PII, or customer data. Use redacted or synthetic examples.

6.3 Verify Before Shipping

AI-generated code is a starting point, not a final solution. Test it, review it, understand it. If you can’t explain why it works, don’t ship it.

6.4 Treat AI Output as a Junior Engineer’s Work

It might be correct, it might be clever, but it needs review. Check edge cases, error handling, and production concerns.

7. Example: Real Engineering Prompt

> You are a senior backend engineer. Review the following service architecture: Context: E-commerce checkout service, 50k orders/day peak, p99 latency requirement 500ms, PostgreSQL for persistence, Redis for caching, RabbitMQ for async processing. Analyse: Single points of failure, Scalability bottlenecks, Data consistency risks, Failure modes and recovery strategies, Missing observability. Structure as an architecture review document. Prioritise findings by risk. Suggest concrete improvements with trade-offs.

8. How This Improves Engineering Outcomes

Teams using structured prompt engineering report:

- Better architecture discussions (trade-offs surfaced early)
- Faster code reviews (AI catches obvious issues before human review)
- Fewer production incidents (failure modes considered upfront)
- Clearer documentation (structured templates, consistent quality)
- Faster onboarding (institutional knowledge captured in prompts)

9. Team Prompt Library (Recommended)

Create a shared internal doc: “Approved Prompts for Engineering” — reviewed by senior engineers, used across the team. This becomes institutional knowledge instead of ad-hoc prompting.

Include prompts for:

- Code review
- Architecture review
- Design doc feedback
- Incident post-mortems
- Technology evaluation

10. Executive Summary

Prompt engineering is not about “using AI to code faster.” It is about using AI to think more rigorously about architecture, trade-offs, and failure modes — before mistakes reach production.

Part IV: AI Prompts for Technical Writing and Documentation

A practical collection of editorial prompts for architecture documents, RFC-style specs, design docs, internal wikis, ADRs, and technical books.

1. Document Redundancy and Clarity

Purpose: Remove repetition and clarify ideas without changing the core meaning.

When to use: After drafting a design doc or spec, to eliminate redundant passages and improve readability.

> Act as a senior technical editor for my document. Task: Review the provided document and identify repeated or overlapping expressions of the same ideas. Remove unnecessary repetition and blend duplicated passages into a single, clearer expression. Strict rules: Do NOT add new ideas or arguments. Do NOT remove any idea or meaning. Do NOT change the document's focus or intent. All original ideas must remain present, but expressed once, clearly and precisely. Allowed edits: Rewriting for clarity, sentence restructuring, paragraph merging, removing duplicated wording. Not allowed: Conceptual reframing, new transitions that introduce interpretation, deleting nuance. Output: (1) Revised document, (2) Change log (brief bullets describing what was merged or de-duplicated). Input: [PASTE DOCUMENT HERE]

2. Content Integration

Purpose: Integrate new content into existing material without creating redundancy.

When to use: When you have additional notes, research, or sections to incorporate.

> Act as a senior technical writer with expertise in software engineering and system architecture. Your task is to integrate this content into the proper section without adding redundancy. Synthesize and analyze the material. Decide where to place it as a section or subsection. Maintain consistency in terminology and formatting with the existing document. Here is the new content: [PASTE NEW CONTENT HERE]

3. Reference-Grade Section Generation

Purpose: Generate publication-quality content on specific technical topics.

When to use: When you need to create a comprehensive section from notes, tickets, or research.

> Act as a senior technical writer with expertise in software engineering and system architecture. Synthesize, analyze, and generate reference-grade material that sounds assertive, descriptive, and forward-looking, not defensive. Add it as a section to my document. Here is the source material: [PASTE SOURCE MATERIAL HERE]

4. Red-Pen Technical Review

Purpose: Technical accuracy check with minimal scope changes.

When to use: Before you circulate a design doc or use it as a reference.

> Review this document with a “red pen” only. Make zero scope additions. Fix only what is wrong, unclear, or technically inaccurate. Flag uncertainties instead of guessing. Return: (1) Clean edited document, (2) Precise change log, (3) Technical issues list (only if needed). Here is the content: [PASTE CONTENT HERE]

5. Publication-Ready Editor

Purpose: Final comprehensive review before sharing.

When to use: As the last step before publishing to a wiki or sending to stakeholders.

> Act as a senior technical editor. Thoroughly proofread the provided document to identify and correct grammatical, syntactical, or typographical errors. Review and validate technical details. Maintain consistency in terminology and formatting. Flag uncertainties as “Needs verification.” DO NOT ADD NEW CONTENT. Reorder sections only if clearly necessary for logical consistency. Result should be: Status: APPROVED FOR PUBLICATION. Here is the content: [PASTE CONTENT HERE]

6. Multi-Document Coherence Review

Purpose: Ensure multiple documents work together as a coherent whole.

When to use: When you have several related design docs, ADRs, or specs.

> Review the following related documents together (not in isolation). Ensure they read as parts of a single, well-structured body of work, with consistent tone, technical depth, terminology, and narrative flow. Identify inconsistencies in tone, phrasing, or terminology. Smooth transitions. Flag repeated introductions of the same concept and terminology variations. Allowed: Harmonizing terminology, consolidating duplicated explanations, minor rewording. Not allowed: Adding new sections, changing core arguments, removing necessary detail. Output: (1) Revised documents, (2) Consistency notes, (3) Status: APPROVED FOR PUBLICATION. Input: [PASTE DOCUMENTS HERE]

7. Quick-Start Editing Session

Purpose: Kick off a focused editing session with clear objectives.

When to use: Any time you start editing a non-trivial document.

> I want to start editing a document. Document: [PASTE CONTENT] Focus: <structure / tightening / clarity / cutting repetition / diagrams / etc.> Constraints: <any limits, e.g., “keep length similar”, “highlight weak areas first”> Review the document, identify issues, and propose improvements. Wait for my approval before applying changes.

8. Architecture Decision Record (ADR) Generator

Purpose: Generate a well-structured ADR from a decision discussion.

When to use: After making an architecture decision, to document it properly.

> Generate an Architecture Decision Record (ADR) for the following decision: [DESCRIBE THE DECISION AND CONTEXT] Structure as: - Title: Short descriptive title - Status: Proposed / Accepted / Deprecated / Superseded - Context: What is the issue we're deciding? - Decision: What is the change we're proposing? - Alternatives Considered: What options were evaluated? - Consequences: What are the trade-offs? What becomes easier or harder? Be specific about trade-offs. Don't oversell the chosen option.

9. RFC / Design Doc Scaffold

Purpose: Generate a structured starting point for a design document.

When to use: When starting a new design doc from scratch.

> Generate a design document scaffold for: [FEATURE / SYSTEM / CHANGE] Include sections for: - Summary (1-2 sentences) - Motivation (why is this needed?) - Goals and Non-Goals - Proposed Solution (high-level) - Detailed Design - Alternatives Considered - Security Considerations - Operational Considerations - Open Questions - Timeline / Milestones Provide guiding questions for each section to help the author fill it in. Don't generate fake content — provide structure and prompts.

Best Practices for Technical Documentation

General Guidelines

- Provide complete text when possible. Partial snippets make edits less coherent.
- Be explicit about constraints: what must not change (terminology, diagrams, examples).
- Review changes before accepting: especially technical claims and edge cases.
- Use iterative refinement: separate passes for structure, clarity, and technical accuracy.
- Maintain a glossary: define preferred terms and keep them consistent.

Practical Sequencing Strategy

Phase 1 – Draft: Quick-Start Editing → Identify major issues

Phase 2 – Structure: Redundancy & Clarity → Remove repetition

Phase 3 – Content: Content Integration → Add and merge new material

Phase 4 – Technical: Red-Pen Review → Verify technical accuracy

Phase 5 – Polish: Publication-Ready → Final approval

Phase 6 – Coherence: Multi-Document Review → Align related documents

Appendix: Why This Matters

Structured prompt engineering:

- Turns AI from “generic assistant” into a domain-aware collaborator
- Forces outputs to be production-grade, not tutorial-grade
- Reduces hallucinations by anchoring on constraints and trade-offs
- Aligns with engineering reality (production concerns, security, operations)

This is the difference between:

“How do I cache database queries?” → Generic tutorial answer

“You are a senior engineer. Design a caching strategy for [CONTEXT] with [CONSTRAINTS]. Explain trade-offs...” → Production-ready guidance

“Review my code” → Surface-level comments

“Review this code for security, performance, and production readiness. Prioritise by severity...” → Actionable, prioritised feedback

The goal is not to make AI do your job. The goal is to make AI help you think more rigorously — about architecture, trade-offs, failure modes, and documentation — before mistakes reach production.

About the Author

For more technical content on software engineering, system architecture, and practical engineering guides, visit:

https://corebaseit.com/