ChatGPT

Universal AI Coding Agent Operating Manual

Architecture & Design

1. **Outline architecture first:** Plan the system's modules, data flow, and boundaries *before* coding to guide AI and avoid chaotic integration issues [1] [2]. Design a high-level blueprint (e.g. saved as `instructions.md`) and have the AI critique it for gaps and edge cases prior to implementation.
2. **Design modular, loosely-coupled components:** Apply SOLID principles (Single-responsibility, Open/Closed, etc.) so each module has one clear purpose and minimal dependencies [3]. Use clean architecture layering to separate concerns (UI, business logic, data) for maintainability and testability [4]. This yields a scalable system with well-defined boundaries that can evolve without breaking other parts.
3. **Design for scalability:** Build with future growth in mind. Split complex services into smaller, self-contained components with well-defined interfaces [5]. Favor stateless processes and horizontal scaling strategies (e.g. load balancing, caching) so the system can handle increased load without performance loss [6]. Keep algorithms efficient and plan for data growth to ensure the application performs under pressure.

Code Quality

1. **Use consistent, descriptive naming:** Follow a naming convention (snake_case, camelCase, etc.) and use clear, pronounceable names that reveal intent [7]. For example, prefer `calculateTotalWorkdays()` over `calc()` – good names make code self-documenting. Avoid magic numbers and unclear abbreviations; every identifier should communicate purpose.
2. **Write self-documenting code:** Keep functions and classes focused and concise (e.g. limit function length ~50 lines) [8]. Code should be readable enough that comments are rarely needed. Include docstrings for public APIs and use inline comments **only** to explain non-obvious logic or decisions [9] – not to restate what the code does. Clean, expressive code and small, single-purpose functions reduce the need for continuous commentary.
3. **Refactor relentlessly:** Continuously improve code structure whenever smells emerge. If a file or function starts doing too much or you see duplicate logic, pause to refactor into simpler, modular units [10]. Regular refactoring prevents "spaghetti code" – aim for zero duplicated business logic and clear separation of concerns throughout the codebase [11]. Maintaining a tidy, well-organized codebase makes future iterations faster and safer.

User Experience

1. **Accessibility-first development:** Build UIs that work for **all** users. Use semantic HTML and proper heading structure, ensure full keyboard navigation, add alt text for images, and maintain sufficient color contrast [12]. Follow WCAG guidelines (e.g. form labels, ARIA roles for dynamic content) and test with assistive technologies. Inclusive design is not optional – it improves UX and often enhances SEO and overall quality.
2. **Progressive enhancement:** Make the app functional with just basic HTML/CSS before adding advanced features. Start with a robust HTML foundation that works on its own, then layer CSS and JavaScript enhancements on top [13] [14]. This ensures core features still work on older

devices or poor connections, improving resilience. Use feature-detection and polyfills for new APIs so the experience gracefully degrades instead of breaking.

3. **Performance budgets:** Set concrete performance targets (max page load time, bundle size, memory usage, etc.) and enforce them in development [15]. Optimize assets (minify CSS/JS, compress images) and avoid heavy computations on the main thread. Use automated tools to flag when performance limits are exceeded. By treating performance as a requirement (not an afterthought), the application stays fast and responsive for users on all devices.
4. **Graceful error handling:** Anticipate error conditions and handle them in a user-friendly way. When something goes wrong, provide clear, polite error messages that explain *what* happened and *how to fix it* [16] [17]. Do not expose raw exceptions or stack traces to users. Preserve user inputs when validation fails and guide the user to correct mistakes (e.g. inline form errors with helpful tips). In code, fail safely by catching exceptions and falling back to default behavior or error pages rather than crashing.

Security & Reliability

1. **Validate and sanitize inputs:** Treat all external data as untrusted. Implement strict input validation (whitelists of acceptable formats, length limits, etc.) and sanitize or encode outputs to prevent injection attacks [18]. Use parameterized queries for database access and avoid constructing SQL/HTML/OS commands directly from user input. Never trust client-side validation alone – always re-validate on the server.
2. **Secure authentication & access control:** Use proven libraries/frameworks for auth; do not hand-roll crypto or session logic. Enforce strong password policies and store passwords using secure hashing with salt. Implement multi-factor authentication for sensitive accounts. Throttle login attempts and use CAPTCHA or lockouts on suspicious activity to prevent brute-force attacks [19]. Every request should perform authorization checks (e.g. verify user roles/permissions) – deny by default and allow only what's necessary (principle of least privilege).
3. **Protect data privacy:** Encrypt sensitive data in transit (TLS/HTTPS everywhere) and at rest (using field or full-database encryption where applicable) [20]. Keep encryption keys and secrets out of code – use environment variables or secure key management (vaults, HSMs) [21]. Do not log personal or sensitive information (like passwords, credit card numbers). Anonymize or truncate data in logs and back-ups according to compliance standards. Regularly update dependencies to patch security vulnerabilities in libraries [22].
4. **Failure recovery and resilience:** Design systems to handle failures gracefully. Use timeouts and exponential backoff retries when calling external services to avoid hanging on slow responses. Implement circuit breakers or fallbacks (e.g. default responses or cached data) for when a dependency is down [23]. Validate assumptions with assertions and log meaningful errors if something goes wrong. Ensure there are monitoring alerts for critical failures so issues are detected and addressed promptly. Plan for disaster recovery with regular backups and the ability to restore service quickly.

Development Workflow

1. **Version control hygiene:** Maintain a clean, traceable Git history. Commit early and often in small chunks with descriptive commit messages [24]. Use feature branches and pull requests for collaboration and code review. Never let large, un-reviewed changes pile up – frequent commits make it easier to pinpoint bugs (via `git bisect`) and rollback if needed. Ensure that any AI-generated changes are reviewed just like human code to maintain quality and consistency.
2. **Automated testing culture:** Write tests to cover all critical logic and edge cases. Adopt the testing pyramid – many fast unit tests, a moderate number of integration tests, and a few end-to-end tests for UI or system flows [25]. Aim for tests to run on each build. Prefer deterministic,

isolated tests (use mocking or test data) so failures pinpoint real issues. Treat warnings or flaky tests as issues to fix immediately. A code change is not “done” until it’s covered by appropriate tests.

3. **Continuous integration & delivery:** Integrate early, integrate often. Every commit should trigger an automated build and test run. Keep the CI pipeline fast and green – fix broken builds immediately. Include static code analysis and security scanning in the pipeline to catch problems early [26] . Ensure the code can be deployed in a reproducible way (infrastructure as code, containerization, etc.). Practice continuous delivery: the main branch should always be in a deployable state, and releases to production should be routine and automated after passing all checks.
4. **Instrumentation and observability:** Build in logging and monitoring from the start to support debugging and maintenance. Use structured logs (e.g. JSON) with consistent fields (timestamps, request IDs, user IDs) so they can be aggregated and searched easily [27] . Implement application metrics (latency, throughput, error rates) and set up dashboards/alerts for abnormal values. Utilize distributed tracing for complex, microservices-based systems to follow request flows. **Do not** log sensitive data [28] , but do log enough context to diagnose issues. This observability ensures you can detect problems and understand system behavior in production.
5. **Rigorous code reviews and retrospectives:** Treat every code generation or change as if coming from a junior developer – review it thoroughly [29] [30] . Check for security issues (e.g. injection flaws, use of hard-coded secrets) [31] , performance concerns (inefficient algorithms, N+1 database queries) [32] , and correctness (off-by-one errors, missing error handling) [33] . Provide constructive feedback and ensure the code meets the project’s style and quality standards before approval. Periodically reflect on what went well or wrong in development (including AI assistant interactions) to continuously improve the workflow without slowing down the creative coding “flow”.

Each rule above serves as a guardrail for AI-assisted development – ensuring that the code you generate is not only functional but also clean, secure, and maintainable by human standards. By adhering to these guidelines, AI coding agents can support rapid, exploratory coding **and** uphold professional engineering best practices by default, striking the right balance between creative velocity and technical rigor [34] [35] .

---

[1] [2] [8] [9] [24] [29] [30] [31] [32] [33] [34] [35] AI Agent Best Practices: 12 Lessons from AI Pair Programming for Developers | Forge Code
https://forgecode.dev/blog/ai-agent-best-practices/

[3] SOLID Principles with Real Life Examples - GeeksforGeeks
https://www.geeksforgeeks.org/system-design/solid-principle-in-programming-understand-with-real-life-examples/

[4] Complete Guide to Clean Architecture - GeeksforGeeks
https://www.geeksforgeeks.org/system-design/complete-guide-to-clean-architecture/

[5] [6] Designing for scalability: Principles every engineer should know
https://www.statsig.com/perspectives/designing-for-scalability-principles

[7] The Ultimate Guide to Self Documenting Code: Write Code That Speaks For Itself | DocuWriter.ai
https://www.docuwriter.ai/posts/ultimate-guide-self-documenting-code

[10] [11] Best Practices I Learned for AI Assisted Coding | by Claire Longo | Medium
https://statistician-in-stilettos.medium.com/best-practices-i-learned-for-ai-assisted-coding-70ff7359d403

[12] 10 Web Accessibility Guidelines for Developers
https://daily.dev/blog/10-web-accessibility-guidelines-for-developers

13 14 Building a robust frontend using progressive enhancement - Service Manual - GOV.UK
https://www.gov.uk/service-manual/technology/using-progressive-enhancement

15 Performance budgets - Performance | MDN
https://developer.mozilla.org/en-US/docs/Web/Performance/Guides/Performance_budgets

16 17 Error Messages Designing and UX 101 - Usersnap Blog
https://usersnap.com/blog/error-messages-best-practices/

18 19 20 21 22 OWASP Explained: Secure Coding Best Practices
https://blog.codacy.com/owasp-top-10

23 27 28 Debugging in Production: Leveraging Logs, Metrics and Traces - DevOps.com
https://devops.com/debugging-in-production-leveraging-logs-metrics-and-traces/

25 Your Most Comprehensive Guide for Modern Test Pyramid in 2025
https://fullscale.io/blog/modern-test-pyramid-guide/

26 Top 8 CI/CD Best Practices for Your DevOps Team’s Success
https://middleware.io/blog/ci-cd-best-practices/