개발자의 법칙 (Laws of Development)

팀 (Teams)

Conway의 법칙 (Conway's Law)

조직은 자신의 커뮤니케이션 구조를 그대로 반영하는 시스템을 설계한다.

- 소프트웨어 아키텍처는 그것을 만든 조직의 소통 구조를 자연스럽게 따라감.
- **역 Conway 전략**: 원하는 아키텍처에 맞춰 팀 구조를 먼저 재편하는 접근.

Brooks의 법칙 (Brooks's Law)

지연된 소프트웨어 프로젝트에 인력을 추가하면 오히려 더 늦어진다.

- 신규 인력 교육 및 조율에 기존 팀원의 시간이 소모되어 생산성 일시 저하.
- 커뮤니케이션 경로 증가: $\frac{n(n-1)}{2}$. 해결책은 인력 추가가 아닌 범위 조정 또는 일정 변경.

Dunbar의 수 (Dunbar's Number)

한 사람이 안정적으로 유지할 수 있는 관계의 인지적 한계는 약 150명.

- 150명을 넘으면 비공식적 소통이 한계에 도달하여 공식적 계층과 프로세스 필요.
- **Two-Pizza Team**: 5 10명의 소규모 팀이 실질적 협업에 가장 효과적.

Ringelmann 효과 (The Ringelmann Effect)

그룹 규모가 커질수록 개인의 생산성은 감소한다.

- 사회적 태만(social loafing) 현상 발생. 소규모 팀이 1인당 산출물이 높은 이유.

Price의 법칙 (Price's Law)

전체 참여자 수의 제곱근($\sqrt{n}$)에 해당하는 인원이 전체 작업의 50%를 수행한다.

- 조직이 커질수록 소수의 고성과자에 대한 의존도가 심화됨.

Putt의 법칙 (Putt's Law)

기술을 이해하는 사람은 관리하지 않고, 관리하는 사람은 기술을 이해하지 못한다.

- 기술 전문성과 관리 역할 사이의 괴리를 풍자적으로 표현.

Peter 원칙 (Peter Principle)

조직 내에서 모든 직원은 자신의 무능력 수준까지 승진하는 경향이 있다.

- 특정 역할에서 유능한 사람이 승진하여 새 역할(관리 등)에서 무능해지는 패턴.
- IC(전문가) 트랙과 매니지먼트 트랙을 분리하는 듀얼 래더 체계 필요.

Bus Factor

프로젝트가 심각한 위험에 처할 수 있는 최소 팀원 이탈 수.

- Bus Factor가 1이면 단일 장애점 존재. 지식 공유와 문서화로 이 수치를 높여야 함.

Dilbert 원칙 (Dilbert Principle)

기업은 무능한 직원을 관리직으로 승진시켜 피해를 제한하는 경향이 있다.

- 관리직이 실무에서 가장 적은 피해를 주는 자리로 여겨지는 역설적 현상.

계획 (Planning)

조기 최적화 (Premature Optimization)

조기 최적화는 모든 악의 근원이다. (Donald Knuth)

- 97%의 경우 작은 효율성은 무시해도 됨.
- **순서**: 먼저 동작하게 → 올바르게 → 필요하면 빠르게 만들 것.

Parkinson의 법칙 (Parkinson's Law)

업무는 주어진 시간을 모두 채울 때까지 확장한다.

- 짧고 명확한 마일스톤 설정이 중요. 스프린트 기반 애자일이 이에 대한 대응책.

90-90 법칙 (The Ninety-Ninety Rule)

코드의 처음 90%가 개발 시간의 90%를 차지하고, 나머지 10%가 또 다른 90%의 시간을 차지한다.

- 엣지 케이스 처리, 버그 수정 등 마지막 단계가 예상보다 훨씬 오래 걸림.

Hofstadter의 법칙 (Hofstadter's Law)

Hofstadter의 법칙을 고려하더라도 항상 예상보다 오래 걸린다.

- 버퍼를 추가해도 여전히 일정이 초과되는 소프트웨어 개발의 본질적 어려움.

Goodhart의 법칙 (Goodhart's Law)

측정 지표가 목표가 되면 더 이상 좋은 측정 지표가 아니다.

- 코드 커버리지를 KPI로 설정하면 의미 없는 테스트를 양산하는 부작용 발생.

Gilb의 법칙 (Gilb's Law)

정량화가 필요한 것은 측정하지 않는 것보다 어떤 방식으로든 측정하는 것이 낫다.

- 완벽한 측정이 불가능하더라도 대략적 측정이 무측정보다 항상 유익함.

아키텍처 (Architecture)

Hyrum의 법칙 (Hyrum's Law)

API 사용자가 충분히 많으면, 시스템의 모든 관찰 가능한 동작에 누군가 의존한다.

- 명세뿐 아니라 타이밍, 에러 메시지 형식 등 비공식적 동작도 실질적 계약이 됨.

Gall의 법칙 (Gall's Law)

작동하는 복잡한 시스템은 반드시 작동하는 단순한 시스템에서 진화한 결과이다.

- 처음부터 복잡하게 설계하면 실패할 확률이 높음. MVP 접근의 이론적 근거.

누수 추상화의 법칙 (The Law of Leaky Abstractions)

모든 비자명(non-trivial) 추상화는 어느 정도 누수가 발생한다.

- ORM이 SQL을 숨기지만 성능 문제가 생기면 결국 내부 쿼리를 확인해야 함.

Tesler의 법칙 (Tesler's Law)

모든 앱에는 제거할 수 없는 고유 복잡성이 있으며, 이동만 가능하고 제거는 불가능하다.

- 복잡성을 누가 감당할 것인가(사용자 vs 시스템)의 문제.

CAP 정리 (CAP Theorem)

분산 시스템은 일관성(C), 가용성(A), 분할 내성(P) 중 두 가지만 보장 가능하다.

- 실질적 선택은 일관성(CP) vs 가용성(AP).

Second-System 효과 (Second-System Effect)

작고 성공적인 시스템 다음에는 과도하게 설계된 비대한 후속 시스템이 뒤따른다.

- 첫 성공의 자신감으로 기능 과잉(feature creep)과 과도한 일반화 발생.

분산 컴퓨팅의 오류 (Fallacies of Distributed Computing)

네트워크는 신뢰할 수 있고, 지연 시간은 0이며, 대역폭은 무한하다는 등의 8가지 잘못된 가정.

의도하지 않은 결과의 법칙

복잡한 시스템의 한 컴포넌트를 변경하면 예측하지 못한 곳에서 부작용이 발생함.

Zawinski의 법칙 (Zawinski's Law)

모든 프로그램은 메일을 읽을 수 있을 때까지 확장을 시도한다. (기능 팽창 풍자)

품질 (Quality)

Boy Scout 규칙 (The Boy Scout Rule)

코드를 발견한 것보다 더 나은 상태로 남겨야 한다. (Robert C. Martin)

- 지속적이고 점진적인 개선(리팩토링)이 기술 부채 축적을 방지함.

Murphy의 법칙 (Murphy's Law)

잘못될 수 있는 것은 반드시 잘못된다. (방어적 프로그래밍과 장애 대비 설계의 근거)

Postel의 법칙 (Postel's Law)

자신이 하는 일에는 보수적으로, 타인으로부터 받는 것에는 관대하게. (견고성 원칙)

깨진 유리창 이론 (Broken Windows Theory)

나쁜 설계나 저품질 코드를 방치하면 추가적인 품질 저하를 유발함.

기술 부채 (Technical Debt)

코드 지름길을 택하면 미래의 시간을 빌리는 것. 이자(생산성 저하)가 발생하며 상환 계획이 필수.

Linus의 법칙 (Linus's Law)

충분한 수의 검토자가 있으면 모든 버그는 쉽게 발견된다. (오픈소스와 리뷰의 핵심)

Kernighan의 법칙 (Kernighan's Law)

디버깅은 코드를 처음 작성하는 것보다 두 배 어렵다. 가독성 높은 단순한 코드를 작성해야 하는 이유.

테스팅 피라미드 (Testing Pyramid)
단위 테스트는 많이, 통합 테스트는 중간, UI/E2E 테스트는 소수만 보유하는 전략.

살충제 역설 (Pesticide Paradox)
동일한 테스트를 반복 실행하면 효과가 감소함. 테스트 케이스를 지속적으로 업데이트해야 함.

Lehman의 소프트웨어 진화 법칙
현실 세계를 반영하는 소프트웨어는 사용되려면 지속적으로 변경 및 진화해야 함.

Sturgeon의 법칙 (Sturgeon's Law)
모든 것의 90%는 쓸모없다. 품질에 대한 높은 기준을 유지하고 가치 있는 10%에 집중하라.

스케일 (Scale)

Amdahl의 법칙 (Amdahl's Law)
병렬화로 인한 속도 향상은 병렬화할 수 없는 작업 비율에 의해 제한된다.

Gustafson의 법칙 (Gustafson's Law)
문제 크기를 늘림으로써 병렬 처리에서 상당한 속도 향상을 달성할 수 있다.

Metcalfe의 법칙 (Metcalfe's Law)
네트워크의 가치는 사용자 수의 제곱(n^2)에 비례한다.

설계 (Design)

DRY 원칙 (Don't Repeat Yourself)
모든 지식은 단일하고 명확하며 권위 있는 하나의 표현만 가져야 한다. (중복 방지)

KISS 원칙 (Keep It Simple, Stupid)
설계와 시스템은 가능한 한 단순해야 한다. 단순함이 유지보수 비용을 낮춤.

SOLID 원칙
S(단일 책임), O(개방-폐쇄), L(리스코프 치환), I(인터페이스 분리), D(의존성 역전).

디미터 법칙 (Law of Demeter)
객체는 직접적인 친구와만 상호작용해야 함. 결합도를 낮추는 "최소 지식의 원칙".

최소 놀라움의 원칙 (Principle of Least Astonishment)
소프트웨어는 이름과 컨벤션에서 예측 가능한 동작을 해야 함. (직관적 설계)

YAGNI (You Aren't Gonna Need It)
실제로 필요하기 전까지는 기능을 추가하지 마라. 과잉 설계 방지.

의사결정 (Decisions)

Dunning-Kruger 효과
어떤 것에 대해 적게 알수록 더 자신감을 갖는 경향. 지속 학습과 자기 인식이 중요함.

Hanlon의 면도날 (Hanlon's Razor)
어리석음이나 부주의로 설명되는 것을 악의로 돌리지 마라. 팀 내 신뢰의 기반.

Occam의 면도날 (Occam's Razor)
가장 단순한 설명이 가장 정확한 경우가 많다. 복잡한 원인보다 단순한 가능성 우선 확인.

매몰 비용 오류 (Sunk Cost Fallacy)
투자한 시간 때문에 손해가 되는 선택을 유지하는 현상. 결정은 미래 가치를 기준으로 해야 함.

지도는 영토가 아니다 (The Map Is Not the Territory)
모델(UML 등)은 현실의 근사치일 뿐. 실제 시스템 동작을 관찰하며 모델을 갱신하라.

확증 편향 (Confirmation Bias)
자신의 믿음을 지지하는 정보만 선호하는 함정. 반대 증거를 적극적으로 탐색해야 함.

Hype Cycle과 Amara의 법칙
기술의 단기 효과는 과대평가하고, 장기 영향은 과소평가하는 경향.

Lindy 효과 (The Lindy Effect)
오래 사용된 기술일수록 앞으로도 계속 사용될 가능성이 높음. (Unix, SQL 등)

제1원리 사고 (First Principles Thinking)
문제를 기본 요소로 분해한 후 근본적 진실에서 해결책을 재구축하는 사고법.

역전 사고 (Inversion)
"어떻게 성공할까" 대신 "어떻게 실패할까"를 먼저 생각하여 위험 요소를 식별함.

파레토 원칙 (Pareto Principle)
문제의 80%는 원인의 20%에서 발생한다. 가장 영향력 큰 20%에 리소스를 집중하라.

Cunningham의 법칙 (Cunningham's Law)
인터넷에서 정답을 얻는 가장 좋은 방법은 질문이 아니라 틀린 답을 게시하는 것이다.

개발자 철학 (Developer Philosophy)

전면 재작성 피하기 (Avoid Ground-up Rewrites)
기술 부채로 유지가 어려워져도 전면 재작성은 최후의 수단으로 남겨야 한다.
- 복잡도 경고 신호(수정 곤란, 문서화 난항 등)를 조기에 포착해 개선할 것.
- 확장이 끝난 후에는 반드시 품질 정비와 통합 단계를 거쳐야 함.

90% 완성의 함정
작동하는 코드를 만드는 것은 전체 작업의 절반(50%)에 불과하다.
- 나머지 시간은 엣지 케이스 처리, 테스트, 배포, 문서화, 성능 최적화에 투입됨.
- 충분한 일정 버퍼를 두어 '동작'과 '완성' 사이의 간극을 메워야 함.

우수 사례의 자동화
반복되는 가이드라인은 문서보다 자동화된 도구(Linter, CI)로 강제하는 것이 효과적이다.
- '규칙 위반 시 빌드 실패'와 같은 환경을 구축해 실수를 원천 차단함.

극단적 데이터(Pathological Data) 고려
정상 입력(Golden Path)뿐 아니라 지연, 거대 데이터, 기괴한 문자열 등 최악을 가정하라.
- 엣지 케이스를 철저히 대비하는 것이 최종적인 코드 품질을 좌우함.

단순함의 태도
대부분의 코드는 더 단순하게 작성될 수 있다.
- 일단 동작하게 만든 뒤, 시간적 여유를 갖고 더 명확하고 간결하게 리팩터링할 것.

테스트 가능한 설계
인터페이스와 사이드 이펙트를 최소화하여 테스트 작성이 쉬운 구조를 설계하라.
- 테스트하기 어려운 코드는 대개 캡슐화가 제대로 되지 않았음을 의미함.

명백한 안전성
코드는 단지 '증명상' 문제가 없는 것을 넘어, 미래의 변화에도 '명백히 안전'해야 한다.
- 보안이나 핵심 로직은 호출 방식이 바뀌어도 견고하도록 설계해야 함.

오프바이원 에러 (Off-by-one Error) 주의
반복문이나 인덱스 계산 시 경계 조건(n vs $n-1$)에서 발생하는 실수를 항상 경계하라.

지식 공유의 단일화 (Single Source of Truth)
정보 산재를 막기 위해 공식 위키나 문서를 한 곳으로 집중 관리하라.
- 소스 컨트롤 및 코드 주석과 연동하여 문서의 최신성을 유지할 것.

재현 가능한 빌드 환경
빌드 스크립트는 단순하게 유지(`cd project; ./build`)하고 팀 전체가 동일한 환경을 공유하라.
- 모든 개발자가 빌드와 테스트 환경에 쉽게 접근할 수 있어야 함.

행동 변화를 위한 불편함
사람들이 원하는 행동을 따르게 하려면, 잘못된 방식을 사용할 때 '불편함'을 느끼게 설계하라.
- 외우게 만드는 것보다 시스템적으로 유도하는 것이 더 강력한 도구임.

성장의 법칙 (Laws of Growth)

점진적 과부하와 적응 (Adaptation)
단순한 노력의 누적이 아닌, 체계적 자극과 회복의 균형이 성장을 만든다.
- 신체가 환경 변화를 인식하고 적응할 수 있도록 꾸준한 자극을 주어야 함.

데드존(Dead Zone) 피하기
중간 강도의 애매한 연습은 시스템을 충분히 자극하지 못한다.

- 저강도·고볼륨(기반 다지기)과 고강도 인터벌(한계 돌파)을 명확히 분리할 것.

지속성의 복리 효과
매일 나타나는 것은 쉽지만, 열정이 사라진 후에도 수개월간 지속하는 것은 어렵다.
- 습관이 추진력을 얻으면 멈추는 것이 오히려 더 많은 노력을 요구하게 됨.

마일리지의 마법 (Volume matters)
정체기에 빠졌을 때 가장 확실한 해결책 중 하나는 투입하는 '양(Volume)'을 늘리는 것이다.
- 압도적인 시간 투입은 그 자체로 속도 향상과 실력의 베이스라인 이동을 가져옴.

맥락 속의 성과 평가
자신을 전문가와 비교하지 말고, 동일한 단계에 있는 코호트(Cohort) 내에서 평가하라.
- 모든 사람은 각자의 경로가 있으며, 초기의 부진이 최종적인 판단 기준이 되지는 않음.

보이지 않는 성장과 베이스라인 이동
성장은 체감되지 않으며 어느 날 갑자기 '기준선'이 이동한 결과로 드러난다.
- 어제의 '어려움'이 오늘의 '보통'이 되는 과정은 느리고 비선형적으로 일어남.

과도한 측정 경계
지표에 과도하게 집착하면 불안과 오판을 부른다.
- 전략을 선택했다면 매일 일희일비하지 말고, 간헐적인 점검을 통해 방향성을 확인하라.

기본기의 정밀함 (Master the Basics)
화려한 기술보다 소수의 기본 원칙을 극도로 정밀하고 높은 강도로 반복하는 것이 핵심이다.
- 프로그래밍에서는 인프라 이해, 성능 직관, 복잡도 관리 등이 이에 해당함.

바람직한 어려움 (Desirable Difficulty)
성장은 100% 확신할 수 없는 불확실성의 경계, 즉 '약간 벅찬' 과제를 수행할 때 일어난다.
- 너무 편안하면 정체되고, 너무 패닉 상태면 배우지 못한다. 경계선에서의 노력이 필요함.

점을 연결하기 (Connecting the Dots)
과거의 사소하고 무관해 보이는 경험들이 나중에 예상치 못한 방식으로 연결되어 가치를 만든다.
- 지금 하는 모든 학습과 경험이 미래의 자산이 될 것임을 신뢰하라.

집중과 여백 (Focus and Margin)

집중의 본질은 '중단'
집중의 실질적 의미는 더 많은 일을 하는 것이 아니라, 대부분의 일을 멈추는 것이다.
- 중요하지 않은 일을 멈출 때 비로소 핵심적인 업무에 투입할 에너지와 여백이 생김.

30% 개선의 법칙
모든 것을 1%씩 개선하려는 시도를 멈추면, 가장 중요한 한 가지를 30% 개선할 여백이 확보된다.
- 자원을 분산시키지 말고 잠재적 상승 여력이 가장 큰 단 하나의 기회에 집중하라.

무화과나무의 비유 (The Fig Tree)
모든 선택지를 쥐려다 아무것도 결정하지 못하면, 모든 기회는 결국 썩어서 떨어지고 만다.
- 모든 것을 가질 수는 없으며, 진정으로 중요한 몇 가지를 위해 나머지를 포기해야 함.

만족한 고객보다 '광팬'
모든 고객을 만족시키려 하기보다, 절대 이탈하지 않고 입소문을 내는 '광팬'을 만드는 데 집중하라.
- 부적합한 고객을 거절할 때 핵심 고객에게 더 깊이 몰입할 여백이 생김.

약점 보완보다 강점 레버리지
모든 약점을 보완하려는 시도를 멈추고, 자신의 강점을 극대화하는 데 에너지를 사용하라.

통제보다 자율 (Margin for Autonomy)
모든 세부 사항을 통제하려 하지 마라. 팀이 자율적으로 성장하고 본인만이 해야 할 일에 집중할 여백을 확보하라.

몰입(Flow)을 위한 단절
주기적으로 이메일이나 소셜 미디어를 확인하는 습관을 멈추고 창의적 생산성을 위한 여백을 확보하라.
- 잦은 맥락 전환은 깊은 사고와 몰입 상태로의 진입을 방해함.

단 하나의 지표 (North Star Metric)
모든 지표를 쫓는 것을 멈추면, 회사를 변화시킬 단 하나의 결정적인 지표에 집중할 여백이 생긴다.

Hell Yeah or No
우선순위가 하나뿐일 때 결정은 단순해진다. "Hell Yeah!"가 아니라면 그것은 "No"다.
- 지평선의 산봉우리처럼 명확한 목표 하나가 모든 선택의 기준이 되어야 함.

실무적 조언 (Practical Advice)

총을 고쳐라 (Fix the Gun)
팀에서 반복적으로 실수가 발생하는 지점이 있다면, 그 실수를 유발하는 '시스템'을 고쳐야 한다.
- 단순히 실수를 지적하는 것보다 시스템을 개선해 실수를 원천 차단하는 것이 훨씬 효과적임.

품질과 속도의 균형 (Quality vs. Speed)
"지금 상황에서 버그를 출시해도 괜찮은가?"라고 자문하고, 위험도에 따라 개발 공정의 엄격함을 조절하라.
- 치명적이지 않은 웹 앱은 빠르게 출시하고 고치는 것이, 완벽을 위해 시간을 끄는 것보다 낫다.

톱 갈기 (Sharpen the Saw)
도구를 잘 다루는 것은 엔지니어의 핵심 역량이다.
- 주요 단축키, 셸, 브라우저 개발 도구, 타이핑 속도 등은 거의 항상 연마할 가치가 있음.

우발적 복잡성 식별
어려움을 단순하게 설명할 수 없다면, 그것은 '우발적 복잡성'일 가능성이 크다.
- 본질적인 복잡성을 다루기 전에, 구조적 복잡성부터 해결해 문제를 단순하게 만들어라.

근본 원인 해결 (One Layer Deeper)
표면적인 증상(null 체크 추가 등)만 고치지 말고, 한 층 더 깊이 파고들어 버그의 근본 원인을 해결하라.
- 근본 원인을 고치면 시스템 전체가 깨끗해지고 유사한 버그가 다시 발생하지 않음.

히스토리의 가치 (Git History)
재현하기 힘든 버그일수록 커밋 히스토리를 파고들어라.
- `git bisect` 등을 활용해 버그가 발생하기 시작한 정확한 시점과 코드를 찾는 것이 직관보다 정확함.

나쁜 코드의 피드백 (Perfect is the Enemy)
완벽한 코드는 피드백을 주지 않지만, 조금 부족하더라도 빠르게 짠 코드는 학습과 방향성을 제공한다.
- 모든 모범 사례를 따르느라 시간을 허비하기보다, 핵심 가치를 빠르게 검증하는 쪽을 택하라.

디버깅을 쉽게 만들기
소프트웨어를 디버깅하기 쉽게 만드는 데 시간을 투자하라.
- 재현과 설정에 걸리는 시간이 전체의 50%를 넘는다면, 디버깅 툴이나 환경을 개선해야 함.

질문하기의 기술
한 시간 넘게 혼자 고민하기보다, 시스템을 잘 아는 동료에게 질문하라.
- 스스로 답을 찾을 수 있다는 것이 명백한 경우를 제외하면, 질문은 팀 전체의 속도를 높여줌.

배포 주기를 북극성으로
빠르고 자주 배포할 수 있는 환경을 구축하는 것이 프로젝트 성공의 핵심이다.
- 느린 CI, 까다로운 린터 등 배포 속도를 늦추는 요소는 장애만큼이나 심각하게 다뤄야 함.

생산성의 법칙 (Laws of Productivity)

올바른 것을 만들기 (Knowing What to Build)
잘못된 것을 빨리 만드는 것은 생산적이지 않다.
- 고객의 요구, 타 팀의 제약 사항, 과거의 실패 사례를 먼저 파악하는 것이 속도보다 중요함.

더 적은 일을 하기
최고의 생산성은 일을 빨리 끝내는 것이 아니라, 하지 않아도 되는 일을 찾아 없애는 것이다.
- 가치가 낮은 '바쁜 업무(Busy work)'나 프로세스를 조정하여 실질적인 임팩트에 집중하라.

도구의 응답 속도
에디터, Git, 빌드 시스템의 지연 시간은 단순히 시간 낭비를 넘어 개발자의 '집중력'을 깨뜨린다.
- 1초의 지연이 반복되면 문맥 전환(Context Switch) 비용으로 인해 막대한 손실이 발생함.

집단적 지식의 유지
10x 개발자는 대개 코드베이스를 가장 잘 아는 사람이다.
- Bus Factor를 1보다 크게 유지하고, 소유권이 너무 자주 바뀌지 않도록 지식을 공유하라.

명확한 경계와 문서화
깔끔한 인터페이스와 양질의 문서는 개발자가 시스템 전체를 연구하는 시간을 획기적으로 줄여준다.
- 문서 한 페이지가 누락되면 수백 명의 개발자 시간을 낭비하게 될 수 있음.

조력자로서의 인프라
인프라는 장애물이 아니라 도움이 되어야 한다.
- “우리 인프라에서는 안 됩니다”라는 말이 나오지 않도록, 실제 유스케이스에 밀착된 설계를 지향하라.

기술 부채의 노출 영역 축소
기술 부채를 줄이면 코드를 변경할 때 이해해야 할 ‘노출 영역’이 최소화된다.
- 부채 상환 프로젝트는 중간에 포기하지 말고 반드시 완료해야 시스템이 더 나빠지지 않음.

낮은 실패율 유지
빌드, 테스트, 배포의 실패는 개인뿐 아니라 시스템 소유 팀 전체의 시간을 낭비하게 만든다.
- 실패 확률을 낮추는 것이 곧 팀 전체의 생산성 향상으로 직결됨.

실용적인 관행
환경이 프로토타이핑을 방해한다면 가장 생산적인 접근 방식도 막히게 된다.
- 도구가 사용하기 쉬워야 측정과 데이터 기반 의사결정이 활발해짐.

정신적 RAM(Focus) 보호
회의, 인터럽트, 미해결 질문들은 개발자의 ‘정신적 RAM’을 점유하여 집중력을 저하시킨다.
- 한 주에 너무 많은 프로젝트를 동시에 생각하게 하는 것은 생산성을 해치는 행위임.

프로젝트 완수의 가치
50% 구축된 기능은 50%의 생산성이 아니라 0%의 생산성이다.
- 프로젝트가 완료되기 전에 우선순위를 계속 변경하면 팀의 생산성은 바닥으로 떨어짐.

톱날 갈기 (Sharpen the Saw)
나무를 빨리 잘라야 할 때는 톱날을 가는 것부터 시작하라.
- 도구 연마와 문서 작성에 들이는 몇 시간은 회사 전체의 수천 시간을 절약할 수 있음.

AI 시대의 비판적 사고 (Critical Thinking in the AI Era)

AI는 ‘인턴’이다
AI의 답변이나 코드는 전지적 존재의 정답이 아니라, 검증이 필요한 ‘인턴의 초안’으로 다루어야 한다.
- “누가 답했는가”보다 “무엇을 근거로 말했는가”를 따지고, 사람이 최종 책임을 져야 함.

진짜 문제 정의 (What)
솔루션으로 달려가기 전에 진짜 풀어야 할 문제가 무엇인지 명확히 규정하라.
- 표면적인 요구사항에 매몰되지 말고, 목표가 무엇인지 5W1H로 체계적으로 점검할 것.

맥락과 환경의 인식 (Where)
한 환경에서 잘 작동하는 해결책이 다른 곳에서는 부작용을 낳을 수 있음을 인지하라.
- 해결책이 적용될 위치(클라이언트, 서버, DB 등)와 파급 범위를 미리 상상해야 함.

분석의 깊이 선택 (When)
응급 처치가 필요한 상황과 근본 원인 분석이 필요한 상황을 구분하라.
- 시간 압박 속에서도 나중에 반드시 재검토해야 할 지점을 표시해두는 엄밀함이 필요함.

5 Whys와 근본 원인 (Why)
표면적인 증상에 안주하지 말고, ‘왜’를 반복하여 문제의 핵심에 도달하라.
- 확증 편향에 빠지지 않도록 다른 가능한 원인이 있는지 끊임없이 의심해야 함.

증거 기반의 사고 (How)
주장보다 근거, 직감보다 실험과 측정 결과를 우선시하라.
- AI가 그럴듯한 답변을 내놓을수록 로그, 테스트, 재현 실험을 통한 직접 검증이 중요해짐.

성급한 도약 방지 (Plunging-in Bias)
충분한 데이터 수집과 문제 정의 없이 바로 코딩에 뛰어드는 편향을 경계하라.
- 결론을 뒷받침하는 구체적인 증거를 모으는 데 더 많은 시간을 할당할 것.

집단 사고의 회피
다양한 관점을 의도적으로 끌어들여 반대 의견이나 다른 가능성을 탐색하라.
- “새 눈”으로 문제를 보게 함으로써 객관성을 높이고 데이터와 가정의 타당성을 검증함.

프리모텀 (Premortem)
프로젝트가 실패했다고 가정하고 그 이유를 미리 적어보며 리스크를 식별하라.
- 계획 단계에서 간과했던 숨은 가정과 부작용을 드러내는 데 효과적임.

인간 고유의 장점 유지
AI가 초안을 만들더라도 “옳은 문제를, 옳은 이유로, 옳은 방식으로 푸는 것”은 인간의 몫이다.
- 겸손한 호기심과 비판적 사고를 유지하는 팀이 AI 시대에도 좋은 결과를 낸다.