AI-First Developer Workflow

Potenziare lo sviluppo "da zero" con l'intelligenza artificiale

1. Opening & Mindset (5')

- **Obiettivo**: Vedere l'AI come un **partner di sviluppo**, non un sostituto.
- **Concetto chiave**: *AI-First Developer Workflow*.
- L'equazione vincente: **Tua Competenza + AI > Solo AI**.
- È richiesta competenza su due fronti:
    i. **Codice**: Architettura, debugging, design.
    ii. **AI**: Scelta del modello, context engineering, prompt design.
- Il software e la sua documentazione devono essere pensati per essere "letti" anche da un LLM.

2. Il PRD come Fondamento (7')

- Il **Product Requirements Document (PRD)** è la mappa per l'agente AI.
- Struttura raccomandata del PRD (vedi prd_template.md).
- Un buon PRD contiene:
  - **Obiettivi** chiari.
  - **User stories** definite.
  - **Specifiche tecniche** di alto livello.
  - **Criteri di accettazione**.
- **Workflow operativo**:
  - L'agente AI agisce come uno sviluppatore junior.
  - L'umano supervisiona, fa code review e merge.
  - **Checkpoint Git** diventano fondamentali per iterare in sicurezza.

3. Generare il PRD (6')

- **Fase 1: Architettura e Outline**
  - Usare un chatbot (Claude, ChatGPT) per definire la struttura iniziale.
- **Fase 2: Brainstorming e Dettagli**
  - Un template di prompt (vedi prompt_template_for_prd_generation.md)
  - **Gemini CLI** per un'analisi interattiva:
    - `gemini explain-code @path/to/file` per capire codice esistente.
    - `gemini "refactor this code to..."` per migliorare la qualità.
    - `@` per aggiungere facilmente contesto (file, documentazione).
- **Best Practice**:
  - Fornire sempre documentazione di progetto e di piattaforma.
  - Adottare il **Test-Driven Development (TDD)** per validare l'output dell'AI.

4. Cursor IDE: Dal PRD al Codice (9')

- Cursor è un IDE AI-first, fork di VS Code.
- **Code Completion**: Scrittura codice assistita e veloce.
- `Ctrl+K` : Il tuo copilota per generare o modificare blocchi di codice.
- **Modalità "Ask"**: Fai domande sul codice selezionato.
- **Modalità "Agent"**: Assegna un task complesso all'AI, che lavora in autonomia.

4.1 Cursor Rules

- **Cosa sono**: Regole statiche versionate in Git (`.cursor/rules/*.mdc`).
- **Scopo**: Garantire coerenza e rispetto degli standard del team.
- **Esempi**:
  - Stile del codice (es. "usa f-strings in Python").
  - Naming convention.
  - Passaggi obbligatori per il build.
- **Risorse**:
  - Documentazione Ufficiale
  - Repo awesome-cursor-rules

4.2 Cursor Memory

- **Cosa è**: Una memoria dinamica per-progetto (docs).
- **Come funziona**:
  - Si popola automaticamente analizzando le tue chat e azioni.
  - Ricorda preferenze, percorsi di file importanti, comandi usati.
- **Obiettivo**: Fornire un contesto sempre più ricco e personalizzato all'AI man mano che il progetto evolve.

Rules vs. Memory: Tabella Comparativa

Caratteristica	Cursor Rules (Statiche)	Cursor Memory (Dinamica)
Scopo	Guard-rail e standard fissi	Contesto operativo che evolve
Fonte	File `.mdc` scritti dall'utente	Chat, azioni e preferenze
Versioning	Sì (in Git)	No (locale al progetto)
Autore	Umano (in fase di setup)	Automatico (durante lo sviluppo)
Use Case	Architettura, stile, build	Fatti, comandi, percorsi file

Consiglio: Usa le **Rules** per la governance, lascia che **Memory** catturi il contesto del "qui e ora".

5. Agenti Autonomi (8')

- **Vantaggi**:
  - **Parallelizzazione**: Lancia più agenti su task diversi.
  - **Task Complessi**: Ideali per lavori lunghi (es. refactoring, implementazione feature).
  - **Multi-Modello**: Compara i risultati di modelli diversi per lo stesso task.
- **Cambiamento nel workflow**:
  - La complessità si sposta dalla scrittura del codice alla **revisione della Pull Request**.

5.1 Tipi di Agenti

- **Cursor Background/Web Agent**:
  - Si avvia dall'IDE su un task specifico.
  - Lavora su un branch isolato.
  - Monitorabile tramite l'interfaccia di Cursor.
  - **Costo**: Attenzione alla "Max Mode" che usa più token.
- **Google Jules (esempio interno)**:
  - **Scope**: Singola Pull Request.
  - **Attivazione**: Aggiungendo un'etichetta come `assign-to-jules` su GitHub.
  - **Modello**: Gemini 2.5 Pro.

6. Vibe Coding vs. AI-Assisted (5')

- **Vibe Coding**:
  - **Flusso**: Prompt → App Full-Stack.
  - **Ideale per**: Progetti "giocattolo", MVP, prototipi rapidi.
  - **Punto debole**: Scala difficilmente su codebase complesse.
- **AI-Assisted (su codebase estesa)**:
  - **Flusso**: Analisi codice → Applicazione regole → Agenti → Revisione.
  - **Richiede**: Digestione del contesto, checkpoint Git, revisione umana.
  - **È il workflow professionale e scalabile**.

7. Workflow End-to-End Consigliato (3')

1. **Planning**: Usa Chatbot + **Gemini CLI** per creare un `plan.md` (PRD).
2. **Commit**: `git commit -m "feat(planning): add initial plan"`
3. **Setup IDE**: In Cursor, carica le `.cursorrules` e abilita la `Memory`.
4. **Esecuzione**: Lancia un **Background Agent** su un issue specifico.
5. **Revisione**: Analizza il diff e la PR generata dall'agente.
6. **Merge**: Se valido, fai il merge e crea un nuovo checkpoint Git.
7. **Itera**: Ripeti il ciclo, aggiornando Rules e Memory se necessario.

Risorse & Q&A

- **Cursor**:
  - Homepage
  - Documentation
  - Awesome Cursor Rules
- **Google Gemini**:
  - Gemini CLI (GitHub)
- **Chatbots**:
  - Claude
  - ChatGPT

Domande?