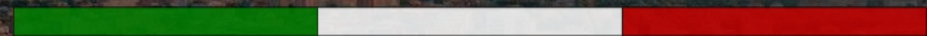


DELPHIDAY



italian conference

Spec-Driven Development

in Delphi, con BMAD e Claude Code



Claudio Piffer

PC Soft di Piffer Claudio

@ claudio.piffer@gmail.com

https://github.com/claudio-piffer

linkedin.com/in/piffer-claudio-3599a16a

DELPHIDAY

italian conference

9-10 Giugno 2026
Piacenza



OPEN-SOURCE PROJECTS



<https://github.com/claudio-piffer>



<https://github.com/claudio-piffer/IAMClient4D>

DELPHIDAY

italian conference

9-10 Giugno 2026
Piacenza





AGENDA

1. **SDD: cos'è, da dove veniamo e perché conta**
2. **Il panorama dei framework, con focus su BMAD**
3. **Quale framework per quale linguaggio (focus Delphi)**
4. **Dentro BMAD: le 4 fasi e il context engineering**
5. **Demo dal vivo: un'app Delphi con BMAD e Claude Code**
6. **Lezioni, trappole e Q&A**



SDD: cos'è e da dove veniamo

1



Il problema: il «vibe coding»

- Gli agenti AI producono codice **plausibile** in pochi minuti — ma che **drifta dall'intento** originale.
- Inventano API inesistenti e la qualità si degrada man mano che il progetto cresce.
- Il codice **passa i test**, eppure viola l'architettura o introduce anti-pattern di sicurezza.
- Il disallineamento lo scopri giorni dopo, quando è già in produzione.

In una riga: il problema non è la velocità, è l'assenza di una fonte di verità condivisa.

Perché conta: i numeri

9,8-42,1%

del codice generato dagli
LLM
vulnerabile nei benchmark

10.000+

nuovi problemi di sicurezza
al mese

10x

più problemi
in 6 mesi



Cos'è lo SDD (e da dove nasce)

- La **specifica è la fonte di verità**; il codice diventa un artefatto generato e verificato, non più il documento primario.
- Le spec SDD **si «eseguono» come gate di validazione**, non si leggono soltanto.
- Non è un'idea nuova: radici nei **metodi formali**, nel **BDD (Behavior-Driven Development)** e nell'**API design-first**.
- Nuovo nel 2026: l'infrastruttura AI è **matura abbastanza da renderlo pratico**.



Lo spettro di rigore

- **Spec-first** — scrivi prima la spec, poi generi il codice. Il livello più leggero.
- **Spec-anchored** — la spec resta agganciata al codice e lo vincola lungo tutto il ciclo. (È dove si colloca BMAD.)
- **Spec-as-source** — la spec è la sorgente primaria; il codice è interamente derivato. Il livello più rigoroso.

Scegli il livello in base a rischio e contesto, non per moda.



SDD vs TDD vs vibe coding

- **Vibe coding** — veloce ma fragile: nessun vincolo, l'intento si perde.
- **TDD (Test-Driven Development — sviluppo guidato dai test)** — eccellente sul «come» (la correttezza), ma non cattura il «cosa» (l'intento).
- **SDD (Spec-Driven Development)** — allinea prima il «cosa», poi lascia che test e codice ne discendano.

Non sono in conflitto: lo SDD ingloba la disciplina del TDD, non la sostituisce.



Perché ora — e dove NON serve

- «**The spec is the prompt**»: gli agenti AI ora reggono spec strutturate e le usano come istruzioni operative.
- Feedback loop più corti: meno cicli «rigenera tutto da capo» rispetto al vibe coding.
- Pratica ancora in assestamento — ThoughtWorks parla di «semantic diffusion». Non è una bacchetta magica.
- Antipattern: **over-spec** (specificare troppo) e approccio «big-bang» su task piccoli o esplorativi.



Il panorama dei framework

2



La mappa dei framework SDD

I principali framework Spec-Driven Development oggi

- **GitHub Spec Kit** — approccio «constitution-driven», indipendente dall'IDE.
- **Superpowers** — plugin per Claude Code/Codex ecc che unisce skill e TDD (Test-Driven Development).
- **AWS Kiro e Google Antigravity** — IDE agentici (Amazon e Google) con SDD integrato.
- **OpenSpec, Tessl** — alternative emergenti, leggere o sperimentali.
- **BMAD (BMad Method)** — il più strutturato: orchestrazione multi-agente sull'intero ciclo. È quello su cui ci concentriamo.



I framework SDD a confronto

Framework	Approccio	Forte quando	Token / costo
Spec Kit	Constitution-driven, IDE-agnostic	Team nuovi a SDD; portabilità	Leggero
Superpowers	Plugin Claude Code: skill + TDD	SDD+TDD	Medio
GSD	Meta-prompting, context engineering	Claude Code; team piccoli	Basso
OpenSpec	Change-mgmt brownfield (delta)	Modifiche piccole e iterative	Molto basso
Kiro	IDE agentico: spec → design → task	Stack AWS; tutto integrato	Assorbito nell'IDE
Tessl	Spec-as-source (beta)	Anti-allucinazioni API	n.d. (beta)
BMAD	Multi-agente, SDLC completo	Enterprise; compliance; ruoli	Alto



Token, costi e portabilità

Quanto consumano

- BMAD: consuma parecchi token
- OpenSpec e GSD: una frazione di quel costo (leggeri, single-assistant o context engineering).
- Kiro: costo assorbito nell'IDE. Spec Kit: leggero, dipende dall'agente che colleghi.

Portabilità

- Producono Markdown/JSON: le spec migrano tra tool con poco attrito.



Come scegliere: le 5 domande

Cinque domande per scegliere il framework giusto

- **Tipo di progetto?** Greenfield (da zero) o brownfield (codice esistente)?
- **Dimensione del team?** Un solo sviluppatore o un team che ha bisogno di coordinamento?
- **Compliance?** Servono tracciabilità e vincoli formali (settori regolati)?
- **Budget di token?** Quanto puoi spendere in chiamate ai modelli?
- **Modifiche piccole e iterative** o progettazione di sistemi grandi?



Zoom su BMAD

BMAD = «Build More Architect Dreams»

- Licenza **MIT** (open source, uso libero anche commerciale).
- Orchestra **12+ agenti specializzati** su tutto il ciclo di vita del software.
- Versione **v6.8.0** (mag 2026), ~48.500 stelle e 5.600+ fork su GitHub: comunità ampia e attiva.
- Tra gli assistenti supportati, **Claude Code è quello consigliato**.



Le 4 fasi di BMAD

Quattro fasi, ognuna alimenta la successiva

- **Analysis (Analisi)** — dalle idee grezze ai requisiti e ai vincoli.
- **Planning (Pianificazione)** — dai requisiti a Epic e User Story.
- **Solutioning (Progettazione)** — dalle storie al progetto tecnico (architettura, API).
- **Implementation (Implementazione)** — dal progetto al codice, una storia alla volta.

Il principio: l'output di ogni fase è il vincolo di input della successiva.



Agenti e handoff «file-based»

Agenti specializzati per ruolo

- Ruoli: **PM (Product Manager)**, **Architect**, **UX**, **Dev (Developer)**, **QA (Quality Assurance)**, **Scrum Master** e altri.
- **Handoff via file**: ogni agente legge l'output del precedente e scrive il proprio. Il passaggio di consegne è un documento su disco.
- Ne risulta una catena tracciabile, dai requisiti fino alla consegna.
- **«Party Mode»**: più agenti nella stessa sessione che discutono insieme un problema.



La «costituzione» e le novità v6

project-context.md: la costituzione del progetto

→ Raccoglie le regole e convenzioni che gli agenti devono rispettare, sempre.

Novità della versione 6

→ **Skill «sharded»** (frammentate): l'agente carica solo il contesto che gli serve, non tutto.

→ **.decision-log**: traccia le decisioni e il «perché» tra un workflow e l'altro.

Perché conta in Delphi: è uno stack di nicchia per gli LLM, quindi contratto e regole esplicite pesano ancora di più.



Installare e aggiornare BMAD

Installazione

- Requisito: **Node.js 20+** (consigliato 20.12+).
- Un comando per tutto: **npx bmad-method install** (primo install, upgrade).
- Dai prompt scegli il **modulo (bmm)** e il **tool (claude-code)**; le skill finiscono in `.claude/skills/`.

Aggiornare e canali di rilascio

- **Ri-lancia lo stesso comando**: rileva e installa la nuova versione stabile.
- Anteprema: **npx bmad-method@next** per l'ultima prerelease; `@<versione>` per fissarne una precisa.



Comandi BMAD (1/2): pianificazione

Parti sempre da qui

→ **/bmad-help** — ispeziona il progetto e ti dice il comando successivo da lanciare.

Pianificazione (Fasi 1–3)

→ **bmad-product-brief** → **bmad-prd** (PRD = Product Requirements Document, documento dei requisiti)

→ **bmad-ux** (spec di esperienza utente) → **bmad-create-architecture** (architettura + ADR)

→ **bmad-create-epics-and-stories** — scompone il PRD in Epic e User Story.



Comandi BMAD (2/2): il dev loop

Implementazione (Fase 4)

- **bmad-sprint-planning** — pianifica lo sprint e gli stati delle storie.
- Ciclo per ogni storia: **bmad-create-story** → **bmad-dev-story** → **bmad-code-review**.
- **bmad-retrospective** — chiude l'epic con una retrospettiva.

Scorciatoia

- **quick-dev** (o /quick-spec) — dev-loop compresso per modifiche piccole e circoscritte.

Nota: i nomi cambiano tra versioni — verifica con /bmad-help o in .claude/skills/.



Greenfield & Brownfield: la «manopola»

BMAD non è una sequenza fissa: è un menù di skill che si attivano in misura diversa

- La manopola regola **quante fasi attraversi** e **quanto le validi**.
- **Greenfield** = progetto nuovo da zero.
- **Brownfield** = progetto esistente da estendere o correggere.
- **Differenza chiave**: il brownfield parte SEMPRE documentando e investigando il codice esistente, prima di specificare il nuovo.

I comandi sono slash command (es. /bmad-prd); molte skill rispondono anche alla frase naturale. (opz.) = passo opzionale · [loop] = ripetuto per ogni story.



Quadro d'insieme

Scenario	Quando usarlo	Artefatti	Fasi attive
Greenfield complesso	Prodotto nuovo, scope incerto, team, rischio alto	12+	discovery + spec + validazione + exec + QA
Greenfield medio	Progetto nuovo serio, scope già chiaro	~6	spec core + readiness + sprint
Greenfield basso	Prototipo o feature singola, idea già chiara	0-1	quick-dev diretto
Brownfield complesso	Evoluzione grande su legacy critico	12+	document + investigate + tutto
Brownfield medio	Nuova feature su progetto esistente	~6	document + spec core
Brownfield basso	Bugfix o piccola modifica mirata	0	investigate + quick-dev



Greenfield — complesso (full)

Prodotto nuovo serio: scope incerto, team, rischio alto — tutta la profondità di BMAD

- **Analysis (opz.):** /bmad-brainstorming · market-research · domain-research · prfaq
- **Brief & Planning:** /bmad-product-brief → /bmad-prd → /bmad-prd (validate) · /bmad-ux (technical-research opz.)
- **Solutioning:** /bmad-create-architecture · generate-project-context · create-epics-and-stories · check-implementation-readiness (shard / spec / advanced-elicitation opz.)
- **Implementation:** /bmad-sprint-planning → [loop] create-story → dev-story → checkpoint-preview → code-review · sprint-status · qa-generate-e2e-tests
- **Chiusura:** /bmad-correct-course (opz.) · /bmad-retrospective

21 passi: la manopola al massimo. Validazioni e QA in ogni fase.



Greenfield — medio (standard)

Scope già chiaro — il percorso di BudgetGo (demo): equilibrio rigore / velocità

- **Planning:** /bmad-product-brief → /bmad-prd · /bmad-ux
- **Solutioning:** /bmad-create-architecture · generate-project-context · create-epics-and-stories
- **Gate:** /bmad-check-implementation-readiness
- **Implementation:** /bmad-sprint-planning → [loop] create-story → dev-story → code-review · sprint-status

~6 artefatti core: niente discovery, niente review avversariale. Consigliato per la demo dal vivo.



Brownfield — complesso (full)

Evoluzione importante su legacy critico — prima il contesto dell'esistente, poi il nuovo

- **Esistente (sempre 1°):** /bmad-document-project · /bmad-investigate [area / bug] · generate-project-context
- **Discovery (opz.):** /bmad-brainstorming · /bmad-technical-research
- **Brief & Planning:** /bmad-product-brief → /bmad-prd → /bmad-prd (validate) · /bmad-ux
- **Solutioning:** /bmad-create-architecture (integrazione col legacy) · create-epics-and-stories · advanced-elicitation → check-implementation-readiness (shard-doc opz.)
- **Implementation:** /bmad-sprint-planning → [loop] create-story → dev-story → checkpoint-preview → code-review · sprint-status

La differenza dal greenfield è l'avvio: document + investigate. Chiusura: correct-course / retrospective.



Brownfield — medio (standard)

Nuova feature su progetto esistente di dimensioni medie

- **Esistente:** /bmad-document-project · /bmad-generate-project-context
- **Planning:** /bmad-prd · /bmad-ux (opz., se tocca la UI)
- **Solutioning:** /bmad-create-architecture (delta) · /bmad-create-epics-and-stories
- **Gate & Implementation:** /bmad-check-implementation-readiness · /bmad-sprint-planning → [loop] create-story → dev-story
→ code-review

~6 artefatti: si documenta l'esistente per dare contesto all'agente, poi il core delle specifiche.



Lean (basso): dritti al codice

Prototipo, feature singola o bugfix con idea già chiara — poco o nessun artefatto

- **Greenfield basso:** /bmad-prd (opz., versione leggera) → /bmad-quick-dev
- **Brownfield basso:** /bmad-document-project (opz., se manca contesto) → /bmad-investigate [bug / area] → /bmad-quick-dev

Si salta la specifica formale: codice allineato ai pattern del progetto. In dubbio: /bmad-help.



Framework per linguaggio

3



BMAD è un «orchestration layer» (e non è il solo)

Uno strato che sta SOPRA il coding agent e gli impone un metodo

- Non rimpiazza Claude Code: lo **guida** — pianifica prima di scrivere, task atomici, contesto fresco per fase, verifica con criteri.
- È una **famiglia**: oltre a BMAD ci sono Superpowers, GSD, ecc — filosofie diverse, stesso scopo.
- Quasi tutti **agnostici rispetto al tool**: funzionano con Claude Code, Cursor, Codex... perché sono file di testo, non plugin proprietari.
- **Il tratto comune**: gate obbligatori tra le fasi — non scrivi codice prima dei test, non fai merge senza review.

BMAD è il più completo della famiglia: copre l'intero ciclo, dall'idea alla storia implementata.



Il fattore decisivo

Quanto l'LLM conosce il tuo linguaggio conta più del framework

Linguaggi ben supportati

- **C#/.NET, JavaScript/TypeScript, Python, Dart/Flutter, Go** — molti esempi nel training, test maturi: l'AI sbaglia poco.

Linguaggi di nicchia

- **Delphi, COBOL, ecc** — pochi dati: l'AI «allucina» di più, quindi serve contesto esplicito.



GitHub Spec Kit

- **Meglio con:** linguaggi mainstream e team multi-IDE; è agnostico
- **Delphi:** usabile — la constitution accetta le tue convenzioni, ma resta leggero
- **Pro:** portabile, battle-tested, la partenza più semplice
- **Contro:** poca struttura multi-agente; debole sulle micro-modifiche



Superpowers

- **Meglio con:** .NET, Angular, Flutter, Python — dove il TDD vola
- **Delphi:** punto debole — il loop TDD (DUnitX) è poco coperto dagli LLM
- **Pro:** plugin per Claude Code, zero config, TDD + review a subagent
- **Contro:** poco personalizzabile per convenzioni di nicchia



BMAD

- **Meglio con:** qualunque linguaggio, soprattutto nicchia ed enterprise
- **Delphi:** la scelta migliore — costituzione + agenti custom per DMVC/FMX/FireDAC
- **Pro:** massimo contesto e controllo; handoff tracciabili; brownfield
- **Contro:** più pesante (setup, token) e curva di apprendimento



Gli altri, in breve

- **GSD:** come Superpowers ma più lean; Delphi: stesso limite sul TDD
- **Kiro:** mainstream nel suo IDE su AWS; Delphi: nessun supporto dedicato
- **Tessl:** forte dove ci sono molte librerie note; Delphi: registry povero
- **OpenSpec:** ottimo per brownfield in ogni linguaggio; Delphi: valido sul legacy



Per Delphi: il verdetto

Vincitore: BMAD

- Dà più contesto e controllo proprio dove l'AI sa poco — ed è il caso di Delphi.

Tre regole pratiche

- **Usa un modello forte** (es. Opus) per le fasi di ragionamento.
- **Parti da moduli greenfield ben delimitati** (da zero, con confini chiari).
- **Investi su spec e review**: l'esecuzione automatica è il punto debole, non la pianificazione.

In una riga: il framework è l'impalcatura, non il carburante.



BMAD completo vs light

Stesso metodo, dose diversa

Tieni (versione light)

- Il «perché» e le ambiguità risolte; specifica della feature, piano, task piccoli.
- Decision log, review umana, build/test come gate di validazione.

Lasci fuori (quando il lavoro è delimitato)

- **PRD enterprise** (personas, mercato, roadmap, metriche); UX spec completa; epic/story mapping; sprint planning.



Quando completo, quando light

BMAD light

- Feature e refactoring delimitati; demo e formazione.
- Esempi: estrarre logica da una form, centralizzare query, aggiungere retry/timeout su una chiamata REST.

BMAD completo

- Nuovo prodotto, migrazione o modernizzazione di un legacy.
- Più team, dominio ambiguo, requisiti di compliance.

La regola: più il lavoro è grande e incerto, più giri la manopola verso il completo.



Il project-context.md per Delphi

La «costituzione» del progetto, in concreto

- Stack e versioni: **Delphi 12/13, VCL/FMX, DMVCFramework, FireDAC, DUnitX.**
- Memoria e ownership: **try/finally**; rispetta chi possiede l'oggetto (chi alloca, libera).
- Non modificare i file visuali **.dfm/.fmx**; non sostituire la libreria dati concordata.
- **Mai inventare API Delphi**: se incerto, fermati e dichiara «NON SO».
- **Build e test obbligatori** come gate prima di considerare fatto un task.



Regole vs verifiche

Due livelli, due posti

- **Le regole** stanno in CLAUDE.md / project-context.md: l'agente dovrebbe rispettarle.
- **Le verifiche** stanno negli hook e negli script: non sono opzionali, si attivano da sole.

Hook utili per Delphi

- Blocca l'azione se il diff tocca file .dfm/.fmx non autorizzati.
- Pre-commit: **build Win64 e test DUnitX; stop se falliscono.**

Verità scomoda: nessun file garantisce obbedienza — la review umana resta.



Quanto costa davvero

Il prompt «economico» che costa caro

- «Rifattorizza questa form» sembra gratis, ma porta **rework**: rollback, fix della build, fix a runtime.
- Investire token **prima** sul contesto costa meno del rework **dopo**.

Quale modello usare

- Lavoro quotidiano: **modello bilanciato**. Architettura, refactoring critico, legacy complesso: **modello forte (Opus)**.
- Task leggeri e riepiloghi: **modello economico/veloce**.



Dentro BMAD: Context Engineering

4



Context Engineering, non un code generator

La tesi del metodo BMAD

- La maggior parte degli assistenti AI «fallisce al terzo giorno»: dimentica il perché del codice scritto il primo giorno.
- BMAD non è un generatore di codice: è una pipeline agile di **context engineering** (ingegneria del contesto).
- Costruisce e blocca progressivamente il contesto attraverso 4 fasi distinte.
- L'output di una fase diventa il limite di input rigido della successiva.

Il tratto distintivo è la rigidità dei limiti tra le fasi, non la velocità.



Le 4 fasi e i 4 ruoli

Una catena di contesto a valle

- **Analysis — Business Analyst:** idee grezze → requisiti; fissa vincoli, persona, compliance
- **Planning — Product Manager:** requisiti → Epic e User Story; il contesto si restringe
- **Solutioning — Architect:** story + vincoli → technical design (data model, API, struttura)
- **Implementation — Developer:** riceve il blueprint esatto e scrive il codice di una story

Il Developer riesce perché l'Architect ha spianato la strada, e l'Architect perché l'Analyst ha mappato il territorio



La guida intelligente: bmad-help

Non devi memorizzare comandi

- **npx bmad-method install** crea due cartelle: `_bmad` (il cervello) e `_bmad-output` (gli asset generati, isolati dal codice)
- `bmad-help` non è un manuale statico: ispeziona lo stato del workspace
- Guarda cosa c'è nelle cartelle, capisce a che punto sei e prescrive il comando successivo
- Ti dà la singola istruzione per il tuo contesto, non la lista di tutti i comandi

Il framework si auto-documenta a runtime



Analysis: brainstorming e PRFAQ

L'AI come coach, non come oracolo

- **Brainstorming:** attrito cognitivo strutturato. Protocolli anti-bias + domain shifting, un passo alla volta
- **Product Brief:** discovery morbido e collaborativo, quando la direzione è chiara
- **PRFAQ (Working Backwards):** press release + FAQ; stress-test, l'AI diventa stakeholder scettico

Il Brief crea consenso; il PRFAQ demolisce le assunzioni deboli



Planning: blindare il PRD

Solo Cosa e Perché, mai il Come

- L'agente PM (Product Manager) traduce la vision in **PRD.md (Product Requirements Document)**, la source of truth (fonte di verità) del progetto.
- **FR (Functional Requirements — requisiti funzionali)**: cosa deve fare il sistema.
- **NFR (Non-Functional Requirements — requisiti non funzionali)**: vincoli operativi (performance, compliance, disponibilità).
- Se il PRD nomina un database o una tecnologia specifica, il processo ha fallito.

È un confine rigido: se una feature non è nel PRD, gli agenti non la costruiscono.



Solutioning: architettura e gate check

Prevenire i conflitti tra agenti

- Agenti in parallelo hanno **context window** (finestre di contesto) isolate: senza architettura condivisa fanno scelte tecniche contrastanti.
- bmad-create-architecture genera gli **ADR (Architecture Decision Record — registri delle decisioni architetturali)**: scelte tecniche definitive.
- Gli ADR non sono log storici per umani: sono un meccanismo di controllo attivo passato a ogni agente.
- **Implementation Readiness (verifica di prontezza)**: l'Architect valida ogni story vs architettura e blocca i prerequisiti mancanti.

Vincoli programmabili che fanno agire modelli non deterministici come un team coeso.



Implementation: il build cycle

Una story alla volta, con disciplina

- **bmad sprint planning** → `sprint-status.yaml`: ogni story con stato pending / in progress / done
- **Scrum Master (Bob)**: `bmad create story` — prende la prima pending, scrive il file della story (criteri, vincoli, test)
- **Developer (Amelia)**: `bmad dev story` — context ristretto alla story, non esce dallo scope
- **Code review**: quality gate vs i criteri della story; solo se passa, la story diventa done

La forza è la **separation of concerns**: **pianifica, esegui e revisiona come step isolati**



Gli strumenti per il contesto

Oltre le 4 fasi: quattro meccanismi chiave

- **Quick Dev:** per modifiche a «blast radius zero» (raggio d'impatto nullo — un solo componente/funzione). Comprime l'intento, esegue, si auto-revisiona.
- **project-context.md:** la «costituzione» — Tech Stack preciso + Critical Rules (le regole non ovvie, non «scrivi clean code»).
- **Distillation & Sharding:** compressione lossless (senza perdita) in bullet densi + frammentazione in file indicizzati, contro la «context blindness» (cecità da contesto).
- **Adversarial review, Pre-mortem, Party Mode:** attrito sintetico per far emergere i difetti.



Un caso istruttivo: lo sharding deprecato

bmad-shard-doc — oggi deprecato (e è una buona notizia)

- **Cosa faceva:** spezzava un documento markdown grande (es. un PRD da 50k token) in file più piccoli per intestazione, per non saturare la finestra di contesto.
- **Perché è deprecato:** le finestre di contesto sono cresciute e i tool supportano i sottoprocessi — spezzare a mano non serve quasi più.
- **L'alternativa di oggi:** documento intero. I workflow usano un «dual discovery»: cercano prima il file intero, poi l'eventuale versione frammentata.
- **Resta utile solo se:** noti che la tua combinazione modello/tool non riesce a caricare tutti i documenti necessari.

La lezione: i workaround di ieri (sharding) diventano inutili man mano che gli LLM migliorano.



Comandi BMAD — Analysis & Planning

Usati nella demo

- **bmad-help** — punto d'ingresso: ispeziona il workspace e prescrive il comando giusto
- **/bmad-product-brief [CB]** — Analysis: brief esecutivo (visione, problema, perché ora); l'agente fa pushback
- **/bmad-prd [PRD]** — Planning: PRD formale con FR/NFR. Solo Cosa e Perché, mai tecnologie
- **/bmad-ux [CU]** — Planning: spec di esperienza per la UI (qui minimale)

Fasi 1–2: dalle idee ai requisiti. Il contesto si restringe a ogni passo.



Comandi BMAD — Solutioning

Usati nella demo

- **/bmad-create-architecture [CA]** — architettura + ADR. Qui le scelte tecnologiche SONO ammesse
- **/bmad-create-epics-and-stories [CE]** — scompone il PRD in epiche e user story, incrociate con l'architettura
- **/bmad-check-implementation-readiness [IR]** — gate check: valida le storie vs architettura, blocca i prerequisiti mancanti

Carrellata

- **/bmad-index-docs** — indice di navigazione degli artefatti di planning

Fase 3: da requisiti a progetto tecnico. Gli ADR tengono coerenti gli agenti.



Comandi BMAD — Implementation

Usati nella demo

- **/bmad-sprint-planning [SP]** — crea sprint-status.yaml: storie con stato pending / in progress / done
- **/bmad-create-story [CS]** — lo Scrum Master scrive il file della storia (criteri, vincoli, test)
- **/bmad-dev-story [DS]** — il Developer implementa, con contesto ristretto alla storia
- **/bmad-code-review [CR]** — quality gate vs i criteri: solo se passa, la storia è done

Carrellata

- **/bmad-quick-dev** — dev-loop compresso per modifiche a blast radius zero (un component/function)
- **/bmad-correct-course** — riallinea il piano quando emerge un cambiamento

Fase 4: una storia alla volta. Separa pianificare, eseguire, revisionare.



Demo: BudgetGo

5



BudgetGo: cos'è e perché questo esempio

Un esempio piccolo ma vero, scelto per mostrare il metodo — non per stupire

- **Cosa fa:** un mini gestore di spese con avviso di sforamento budget. REST (DMVCFramework) + client FMX, SQLite via FireDAC.
- **Scopo:** vedere il flusso SDD completo su Delphi — dalla spec al codice testato — senza la complessità di un gestionale reale.
- **Perché proprio questo:** ha UNA regola di business chiara (soglie 80%/100%) — perfetta da specificare, implementare e soprattutto TESTARE dal vivo.
- **Cosa copre:** PRD → architettura/ADR → epiche/storie → una storia end-to-end (avviso sforamento) con dev + code review + test.

Volutamente piccolo: l'obiettivo è che seguiate il METODO, non che ammiriate l'app.



Architettura target

I quattro pezzi del sistema

- **Client FMX (FireMonkey)** — UI multiplatforma (Win64/Android/iOS).
- **Servizio REST** — server console DMVCFramework su :8080.
- **Persistenza** — FireDAC + SQLite (tabelle expenses, budgets).
- **Contratto** — OpenAPI: definisce gli endpoint condivisi tra client e server.

Il dominio (le regole di budget) vive nel service layer, separato dalla UI.



01 - Product brief: le 3 ambiguità

Un brief «completo»... non lo è mai

- «Budget mensile» - mese di calendario o ultimi 30 giorni?
- Rimborsi e importi negativi - ammessi? Contano nel totale?
- Avviso di sforamento - solo al superamento o anche vicino alla soglia?

Far emergere queste domande è il momento «aha» di SDD



Fin dove arriva BMAD

Il percorso completo (a scopo didattico)

- **Analysis:** brainstorming, product brief, ricerca
- **Planning:** PRD, epics, stories
- **Solutioning:** architettura, UX spec
- **Implementation:** dev stories, code review, retrospettiva
- Agenti specializzati (PM, Architect, SM, Dev) che si passano gli artefatti

Lo mostro pre-generato: ne rigenero un pezzo dal vivo



...e perché qui torniamo al light

Per questa slice

- 1 feature, 1 sviluppatore, scope delimitato
- Epics e stories qui sarebbero gusci vuoti: struttura senza informazione

La regola: la manopola

- Più il lavoro è grande, incerto o condiviso → più giri verso il completo
- Vi ho mostrato la macchina: ora sapete cosa c'è dietro la manopola

Torniamo al dev loop: spec → task → codice → review



Code review

Un agente dedicato rivede il codice generato

- È il pattern più sottovalutato dello SDD: **chi controlla il lavoro**.
- Verifica l'aderenza al contratto, alle regole di business e alle convenzioni.



Lezioni, trappole e Q&A

6



Cosa funziona, cosa no

Funziona

- Feedback loop più corti del vibe coding, senza il peso del waterfall.
- Codice che sopravvive in produzione, allineato all'intento.

Funziona meno

- Task piccoli o esplorativi: la specifica può diventare un peso.
- Stack di nicchia (come Delphi): meno esempi, serve più contesto esplicito.



I test nello SDD: due strade

La specifica si verifica: il test è parte del metodo, non un'aggiunta

- **QA integrato (bmad-qa-generate-e2e-tests):** incluso, genera test API ed E2E dal codice, percorso felice + casi limite critici. Si lancia a epica completata.
- **Test Architect — TEA (modulo a parte):** agente esperto + 9 workflow per la strategia di test, priorità per rischio (P0-P3), quality gate, tracciabilità.
- **Regola d'oro:** valida con un modello DIVERSO da quello che ha generato — evita il bias di auto-conferma.

Inizia dal QA integrato; aggiungi TEA quando ti servono strategia, gate o tracciabilità.



Sicurezza e quality gate

Chiudere il cerchio sul codice «plausibile ma vulnerabile»

- Ricordate il dato d'apertura: una quota importante del codice generato dall'AI è vulnerabile. Lo SDD risponde con i gate.
- Un **gate** è un controllo che blocca: build e test devono passare, la review deve approvare, prima che una storia sia «fatta».
- I gate possono essere **verifiche deterministiche** (hook, CI) o **umane** (la review) — non dipendono dalla buona volontà del modello.
- Con TEA: **NFR assessment (performance, sicurezza) e release gate** per decisioni go/no-go basate sui dati.



Sicurezza del codice: requisito, non ripensamento

Nel vibe coding la sicurezza arriva dopo. Nello SDD nasce nella spec.

- Nel PRD come **NFR (requisiti non funzionali)**: validazione input, query parametrizzate, gestione segreti, autorizzazione.
- Nelle storie come **criteri di accettazione**: «l'endpoint rifiuta input non valido» diventa una condizione verificabile, non una speranza.
- In verifica: **test di sicurezza + NFR assessment del Test Architect (TEA)** valutano esplicitamente la robustezza.

Shift-left: la sicurezza del codice si scrive prima, si verifica durante — non si rincorre dopo.



Sicurezza dell'architettura: scelte negli ADR

Le decisioni di sicurezza strutturali sono decisioni di architettura

- Negli **ADR (Architecture Decision Record)**: dove sta il confine di fiducia, come si autentica, dove si validano i dati, cosa è esposto.
- Gli ADR sono vincoli ATTIVI passati a ogni agente: nessuno «dimentica» la regola di sicurezza a metà implementazione.
- **Esempio dal mondo reale:** autenticazione (Keycloak/Entra), token JWT, middleware su /api/* — sono ADR, non scelte nella testa del singolo.

Il check di Implementation Readiness valida le storie contro l'architettura, sicurezza inclusa.



Caso reale: l'automazione spinta (e i suoi limiti)

Un loop che implementa un'intera epica da solo

- Idea: uno script legge `sprint-status.yaml`, prende la prossima storia, lancia `create-story` → `dev-story` → `code-review` → `commit`, e ripete per tutte.
- **Risultato:** una web app non banale (login, salvataggio automatico, sync multi-dispositivo) costruita in ~2 ore.
- **Il problema:** per automatizzare, erano stati **tolti i test dal loop**. Esito: un bug silenzioso (errore in creazione cliente) scoperto solo a mano.

La morale: senza il gate di verifica, la velocità produce debito. Non è colpa dell'LLM — è il gate mancante.



Farlo bene: il loop semi-autonomo con gate

Automatizzare l'orchestrazione, NON i controlli

- **A cosa serve:** uno script chiude un'epica storia per storia — create-story → dev-story → test → review → commit — togliendo l'attrito del lancio manuale.
- **Come si usa:** legge sprint-status.yaml, lancia Claude Code headless (claude -p), una sessione fresca per storia (niente drift).
- **I gate restano:** test falliti o review critica → il loop si FERMA. L'epica la chiudi tu.
- **Risparmio token:** modello per fase — Opus dove serve giudizio (review), più economico dove è meccanico (dev-story su spec blindata).

Vantaggi: meno ripetizione, ritmo costante. **Svantaggi:** costo token, fragile su storie incerte, va sorvegliato.



I gate del loop e il ruolo dell'umano

Tre gate, due automatici e uno «assistito»

- **Gate test (deterministico):** build + DUnitX. Lo script guarda l'exit code; se $\neq 0$ il loop si FERMA. Non dipende dall'AI.
- **Gate review (assistito):** l'agente emette REVIEW_VERDICT: PASS/FAIL. Su FAIL il loop si ferma — l'AI segnala, l'umano decide.
- **Gate chiusura (umano):** lo script non marca mai l'epica «done». L'OK finale è sempre tuo.

I compiti dell'umano che monitora

- Decidere sul FAIL della review; verificare l'app dal vivo; marcare «done» e fare push; (opz.) lanciare la retrospettiva.



Versionamento e tracciabilità

Gli artefatti vivono in Git: l'AI diventa auditabile

- PRD, architettura, ADR, decision-log, storie: tutti **file di testo versionati in Git**, non documenti opachi.
- Sono **diffabili e revisionabili in una pull request**: vedi cosa è cambiato nell'intento, non solo nel codice.
- La **tracciabilità**: da una riga di codice risali alla storia, alla decisione, al requisito che l'ha generata.

Per l'azienda: l'AI smette di essere una «scatola nera» e diventa governabile e verificabile.



Le trappole da evitare

Quattro errori comuni

- **Over-spec** (specificare troppo) e rilasci «big-bang» tutto-in-una-volta.
- **Spec che divergono dal codice**: la specifica va mantenuta viva, non scritta e dimenticata.
- **Fiducia cieca** nell'output dell'agente, senza review.
- **Confondere «più rigore» con «più valore»**: il rigore giusto dipende dal contesto.



Lo SDD sposta il giudizio umano

Vince l'architetto con senso del prodotto, non il «miglior battitore»

- Il valore si sposta a un piano di controllo più alto.
- Conta pensare in sistemi e comunicare l'intento con chiarezza.

La spec è il punto in cui le tue decisioni diventano eseguibili.



La retrospettiva di epica

Chiudere un'epica imparando qualcosa

- **Cos'è:** un workflow che, nei panni dello Scrum Master, rivede un'epica completata — cosa ha funzionato, cosa no, quali pattern sono emersi.
- **A cosa serve:** estrae le lezioni e le trasforma in informazioni utili per la pianificazione delle epiche successive (miglioramento continuo).
- **Cosa produce:** un documento di retrospettiva e l'aggiornamento dello sprint-status; le decisioni confluiscono nella memoria del progetto.

Come si invoca

- **bmad-retrospective** — a epica completata, in una finestra di contesto fresca.

È la «manopola»: su un'epica piccola è facoltativa; su progetti veri è ciò che dà memoria al lungo periodo.



Lo SDD non finisce con la 1.0

Non è waterfall: è un ciclo che si ripete

- Chiusa la 1.0, nascono nuove esigenze. Lo SDD non riparte da capo: riprende dagli artefatti esistenti.
- Lo stato vive nei **file** (PRD, epiche, storie, decision-log), non nella sessione: il progetto si riapre, non si ricostruisce.
- Le epiche chiuse restano come **storia**; le nuove (EPIC-02, EPIC-03...) nascono accanto, non al posto loro.

Due percorsi, secondo la «manopola»: micro-modifica (breve) o evoluzione importante (completa). Le vediamo entrambe.



Micro-modifica: il percorso breve

Quando il cambiamento è piccolo e circoscritto

- Esempi: un nuovo campo, una query da correggere, un timeout da aggiungere, un bugfix. Una sola unità, nessun effetto a catena.

Comandi (BMAD v6.8 — Quick Flow)

- **bmad-help** — riapro e mi oriento.
- **bmad-quick-dev** — comprime spec ed esecuzione in un solo workflow: descrivo l'intento, l'agente implementa e si auto-revisiona.

Salta architettura ed epiche: sproporzionato a una modifica minima.



Evoluzione importante: da 1.0 a 2.0

Quando arriva un blocco nuovo o cambia l'architettura

- **bmad-help** — riapro il progetto; rileva le epiche 1.0 chiuse.
- **bmad-correct-course** — aggiorna PRD e piano con il cambiamento (gestisce i cambi di scope, senza ripartire da zero).
- **bmad-create-architecture** — SOLO se la 2.0 tocca l'architettura (nuovo endpoint, nuova tabella, nuova integrazione).
- **bmad-create-epics-and-stories** — nascono EPIC-02, EPIC-03... accanto alle vecchie.
- **bmad-check-implementation-readiness** — gate: valida il backlog 2.0 prima di scrivere codice.
- Poi il dev loop: **bmad-sprint-planning** → **create-story** → **dev-story** → **code-review**.



E se parto da un progetto esistente? (brownfield)

Brownfield: codice che esiste già, con i suoi pattern e il suo debito

- Passo zero — **bmad-generate-project-context**: scansiona il codice e ne deduce stack, pattern e convenzioni, così gli agenti seguono ciò che c'è invece di inventare.
- Per codebase complessi — **bmad-document-project**: documenta lo stato reale del progetto (utile se la doc manca o è obsoleta).

Poi due approcci secondo la dimensione

- **Document-First** (progetti piccoli): prima documenti tutto, poi il PRD.
- **PRD-First** (codebase grandi): prima il PRD del miglioramento, poi documenti solo le aree toccate.

Da qui si converge sul flusso noto: **architettura** → **epiche/storie** → **dev loop**.




Risorse e link

Per approfondire

- **BMAD-METHOD** — github.com/bmad-code-org/BMAD-METHOD
- **GitHub Spec Kit, AWS Kiro, GSD, Superpowers, OpenSpec** — i punti di partenza alternativi.
- **Paper «Spec-Driven Development» (Piskala, gen 2026)** (<https://arxiv.org/pdf/2602.00180>)
- **Slide e codice della demo**

An aerial photograph of a historic Italian city, likely Florence, showing a dense cluster of buildings with terracotta roofs. A river, the Arno, flows through the center of the city. In the foreground, a large square (Piazza del Duomo) is visible, featuring a prominent building with a series of arches. The image is overlaid with a semi-transparent dark layer containing text.

DELPHIDAY


italian conference

GRAZIE!