David Soden

AI-Assisted Coding

Course Syllabus

David Soden

This course teaches how to build real software with AI assistance, not demos that fall apart in production. It runs from the tools and mental models through architecture, data, security, and delivery, then lands in a hands-on build: a support-ticket helpdesk app taken from web app to native desktop.

AI-Assisted Coding – 101

- LLM vs Harness
- Global vs Project level configurations
- Skills, the universal prompt that acts like a function
- Skills resources (skills.sh)
- Plugins: custom code plus Skills
- Agent vs Agent orchestration

AI Orchestration Frameworks

- What these frameworks are and the problem they solve (ad-hoc prompting into a repeatable process)
- BMAD: a development methodology as agents (PM, architect, developer roles; brainstorm to plan to architect to build)
- Archon: a workflow engine/harness that makes AI coding deterministic and repeatable (YAML workflows, isolated runs, validation gates)
- Where they fit: they orchestrate much of what the rest of this course covers by hand
- Build-your-own vs adopt-a-framework

Setup Your Environment for Success

- Security: keeping API keys and account information private
- Generative AI: tuning to your voice and/or brand
- Tweaking your front end UI/UX output (Pexels, ShadCN, Pencil.dev)
- Diagramming output (Excalidraw, Mermaid)
- Branding: global skill
- Documentation: Docusaurus

Architecture Concepts 101

Frame

- L1: Why AI-assisted apps become slop (the demo-to-production gap)
- L2: Two tiers, and why picking the wrong one is the real mistake

Tier 1: SMB

- L3: The SMB profile (1k–5k users, light quick IO, clock-in/CRM examples)
- L4: Request/response mental model
- L5: Client vs server (boundary is a choice)
- L6: REST APIs (verbs, resources, status codes, statelessness)
- L7: JSON as the common format
- L8: Deployment shapes (local to self-hosted to single backend)
- L9: Sync vs async work
- L10: Auth and multi-tenancy (vocabulary only)
- L11: Why this is right-sized, not slop

Tier 2: Enterprise

- L12: The trigger (global/SaaS, sustained load)
- L13: Why request/response-to-one-backend breaks
- L14: Load balancing and scale sets
- L15: Session state breaking (statelessness pays off)
- L16: Caching
- L17: Load distribution vs one backend
- L18: What the cloud handles vs what you design

LLMs

- What an LLM actually is (token predictor, not a database or search engine)
- Context window (why it forgets, why big inputs cost more)
- Tokens and cost (how billing works, why it matters at scale)
- Local vs hosted models (privacy, cost, capability tradeoffs)
- Model families and picking one (frontier vs small/open)
- Temperature and determinism (why the same prompt varies)
- Hallucination (why it happens, why grounding matters)
- Where LLMs fit in an app's architecture (the slow async call)

Application Types

- Web applications (the primary build)
- Desktop applications: Electron (Mac, PC, Linux native)
  - Two other frameworks to explore on your own: Tauri and Neutralino
- Mobile (mention only): pointer to AI-assisted coding with Flutter

Databases

- JSON file as poor-man's NoSQL (under ~100 rows)
- SQLite
- RDBMS (full)
- RDBMS vs Mongo/NoSQL: when and why (compare, not build)

Security

- JSON Web Tokens / authentication
- Multi-tenancy vs instance-based
- Encryption in transit (TLS/HTTPS) vs at rest
- Authentication vs authorization (RBAC)
- Secrets management
- Input validation and injection (SQL injection, XSS)
- Audit logging
- Data residency and sovereignty (GDPR, Germany as strictest)
- Regimes by industry: HIPAA, GDPR, PCI-DSS, SOC 2
- Data retention and right to deletion
- PII/PHI handling and data minimization
- Tenant isolation as compliance requirement
- Prompt injection
- Never sending PHI/PII/privileged data to third-party models
- Local/self-hosted models when data can't leave
- Output as untrusted

QA, Testing & Performance

- Why testing is the line between working software and slop
- QA testing (unit, integration, end-to-end: what each catches)
- Load and stress testing (proving the Tier 2 architecture holds)
- Performance testing vs functional testing (fast enough vs correct)
- Skills and tools for all of this

Infrastructure, Deployment & CI/CD (DevOps)

- What DevOps means here (path from your machine to production)
- Environments (local to staging to production)
- Containers (Docker: how self-hosted apps ship)
- CI/CD pipelines (automated build, test, deploy)
- Where it runs (cloud vs self-hosted)
- Secrets and config in deployment
- Monitoring and logging in production

Documentation

- The app is only as good as its documentation
- Planning docs (PRD, specs)
- Technical/architecture docs
- User documentation
- Runbooks
- API documentation

- AI angle: docs as LLM context, and docs the LLM generates (Docusaurus)

The Build

- Support-ticket helpdesk app
- Web app first, then convert to desktop (Electron wrapper, "shorten the wire")
- Detailed phase-by-phase plan in the companion Build Outline document

[SEE NEXT PAGE TO SEE DETAILS ABOUT THE BUILD]

David Soden

The Build

Support-Ticket / Helpdesk App

Companion build outline to the AI-Assisted Coding syllabus

The build is where the course theory turns into running software. Each phase is chosen to exercise specific concepts from the syllabus, noted inline. The LLM feature is bolted on after a plain working app rather than baked in, so students feel the app work before the slow async call enters and the LLM concepts land as a felt upgrade.

Delivery note: the PRD in Phase 5 is written first in real practice. It is grouped under documentation here to keep that section whole, but in sequence it comes before Phase 1.

Phase 1: The Web App (Core CRUD)

- Data model: tickets, users, comments, status (pays off Databases)
- Start on SQLite, then the same schema on a full RDBMS (pays off the SQLite to RDBMS progression)
- REST API: create / read / update / close a ticket, add comments (pays off REST: verbs, resources, status codes)
- JSON as the request/response format (pays off JSON)
- Front end: ticket list, ticket detail, new-ticket form (pays off UI/UX setup: ShadCN, Pencil)

Phase 2: Auth, Roles, Tenancy

- Login with JWTs (pays off JWT / statelessness)
- Three roles: end user files, agent resolves, admin configures (pays off authentication vs authorization / RBAC)
- Multi-tenancy: one instance serving several client companies, tenant-isolated data (pays off multi-tenancy, the SMB-vs-enterprise thread)
- Secrets kept out of code (pays off secrets management)

Phase 3: Desktop Conversion

- Wrap the working web app in Electron (pays off Application Types)
- The "shorten the wire" moment: same code, client and server collapse toward one process (pays off client/server boundary)
- Build native for Mac, PC, Linux

Phase 4: Add the LLM Feature

- Auto-summarize a long ticket thread, or draft a suggested agent reply (pays off LLMs section)
- Handle it as a slow async call without freezing the UI (pays off sync vs async)
- Never send another tenant's data or PII to the model (pays off LLM security surface)

Phase 5: Test, Document, Ship

- Unit and integration tests on the API (pays off QA/Testing)
- A load test against the ticket endpoints (pays off load/stress testing, Tier 2 theory made real)
- Docker container plus a basic CI/CD pipeline (pays off DevOps)
- Write the docs: a short PRD up front, plus a runbook and user doc at the end (pays off Documentation)

Enterprise-tier scaling (load balancers, scale sets) is pointed at by the Phase 5 load test rather than built by students, matching how Tier 2 is framed in the syllabus: know to design for it, do not hand-build it.