codecentric

Basel One

The Best Prompt Is a Good Visual Model

Spec-Driven Prototyping with Collaborative Modeling and LLMs




helvetia
KARAKUN
Kanton Basel-Stadt
ERNI
INNOFUSE
+++Ba
sel++
++Te+
+++ch
Basel
basel.ch
JAVA
USER
GROUP
CH
BLKB
optravis
company of .msg
cyon
CS2
CSS
IRIX
swiss made
software
ICT
Scouts
Wir fördern Zukunft
SD:>_
[SwissDevJobs.ch]
#BaselOne26
baselone.ch

Johannes Barop

Senior Consultant & Software Architect
codecentric AG • Hamburg

- 15 years of software development
- Frontend, backend and infrastructure
- Experience from start-ups to large corporations and from social networks to insurance



Christian Langmann

Solution Architect
codecentric AG • Hamburg

- 20 years of experience designing, developing and optimizing complex software systems
- Interests range from product discovery with EventStorming to designing distributed architectures



Basel One

Dr. Annegret Junker

Chief Software Architect
codecentric AG • Munich

- 30 years of experience designing and optimizing complex applications
- DDD expert and host of the DDD Meetup in Munich
- Author of several books on DDD

Annegret is not here today. Her material forms the basis of this talk.







Ask an LLM to build an app

Everyone has already tried this

Point an LLM at a domain problem, ask for code, and the result is remarkably consistent.

01

Syntactically plausible

It compiles. It has tests. It looks like something a competent developer wrote on a good day.

02

Domain-ignorant

Invented fields, missing rules, entities that don't exist, entities that should have.

03

Wrong in expensive ways

Not wrong at the surface — wrong three layers down, where diagnosis costs hours.

The temptation is to blame the language model. That is the wrong angle.

LLMs don't fail at domain work because they're not smart.

They fail because they have no access to your domain.





The good news: we already solved this

We build the next three artifacts to align humans.

It turns out they are also excellent machine input.

Domain Storytelling

A pictographic sentence:
actor → activity → work object.

Actors and roles, named explicitly
Work objects — the domain nouns
Sequence, so flows are unambiguous

Groups that hint at boundaries

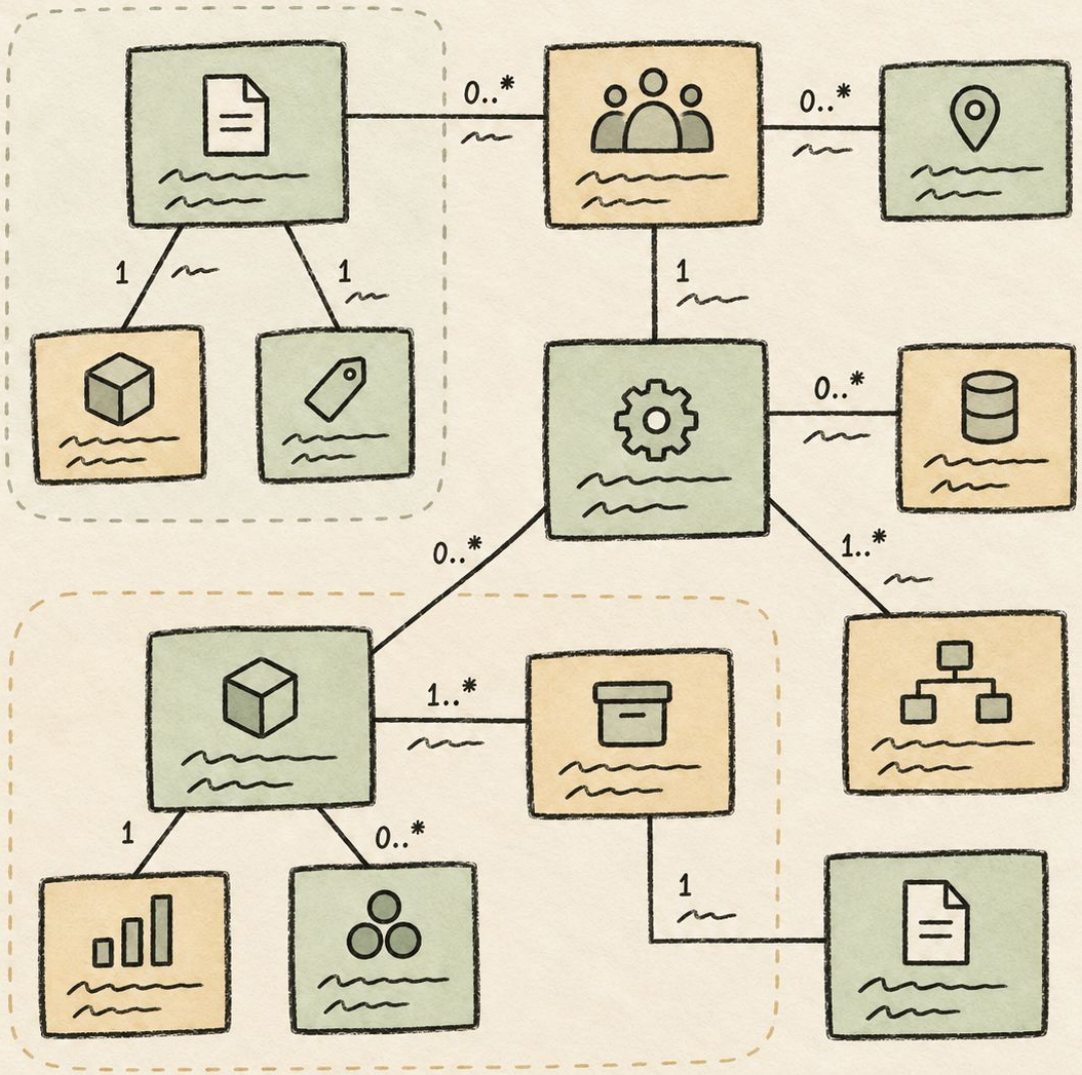


Visual Glossary

The team agrees on one term for each domain concept.

Relationships and cardinalities describe how concepts fit together.

The model gets the agreed vocabulary and the structure of the data.

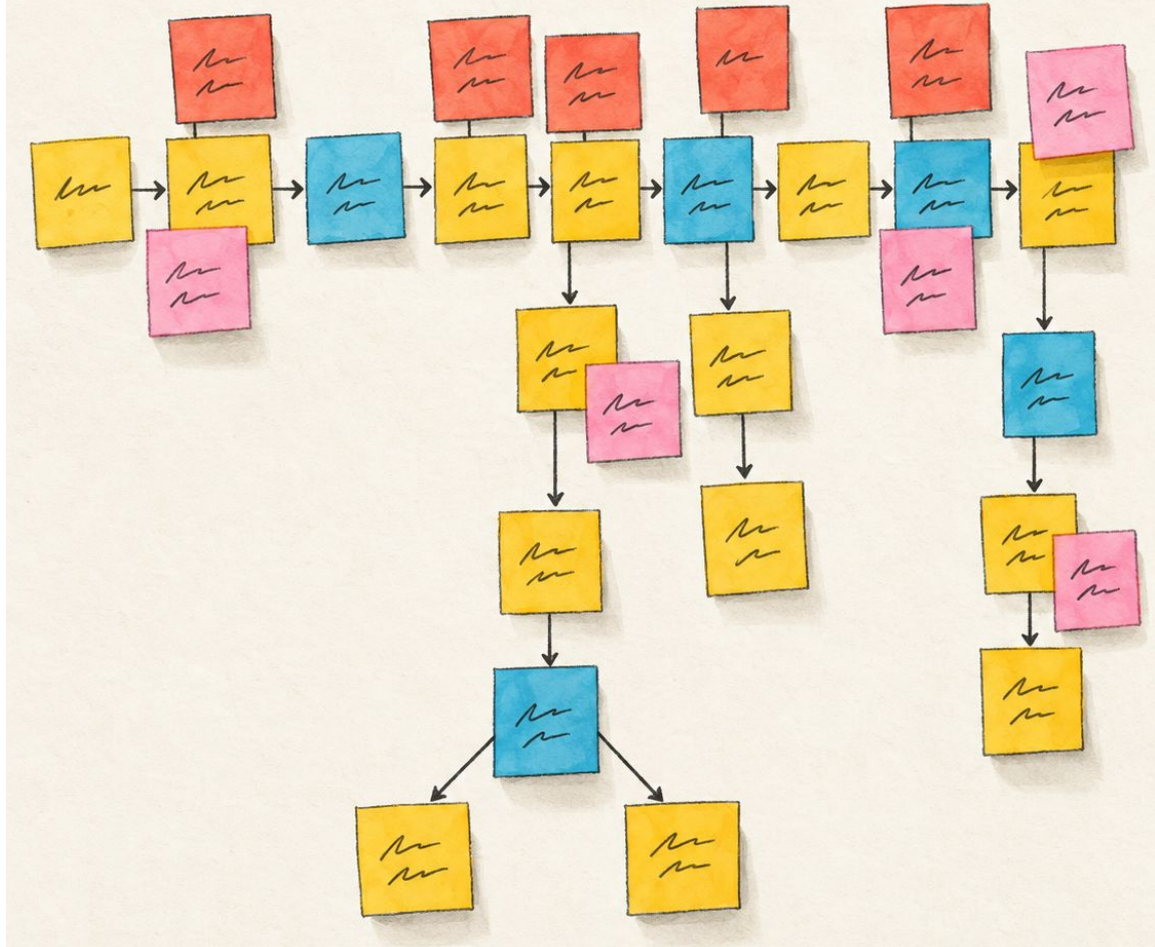


EventStorming

Orange, blue, lilac, yellow —
a grammar, not a mood board.

Commands: what a user may ask for
Events: the facts that follow

Policies: the rules nobody wrote down
Aggregates: where invariants live



Each artifact maps onto what the model needs

The translation is close to mechanical — which is exactly why it works.

ARTIFACT	WHAT THE WORKSHOP PRODUCED	WHAT THE MODEL CAN USE IT AS
Domain Story	Actors, work objects, ordered steps	Roles, entities, user flows, screen sequence
Visual Glossary	One agreed term per concept	Type names, field names, no synonyms in the code
EventStorming	Commands, events, policies, aggregates	API operations, state transitions, business rules
OpenAPI spec	Schemas, operations, contexts	A machine-readable domain model

A well-derived OpenAPI spec is, essentially, a domain model a machine can read.

A three-step pipeline, four artifacts

Each step narrows the language. Each narrowing shows up in the generated output.

01

Domain Story + Visual Glossary

Who acts, on what, in what order

→

02

EventStorming

commands • events • policies

What may happen, and what must hold

→

03

OpenAPI

split by bounded context

The contract, scoped per context

→

Prototype

generated frontend

Clickable dummy, no invented fields

AI supports all four steps of the workflow.

CookWithUs

A recipe-sharing platform: cooks publish recipes, home cooks plan meals, the platform moderates and recommends.

Modeled and built three times, with the artifacts getting richer each round.



Basel One

Three modeling iterations

01

Domain Story
+ Visual Glossary

02

+ EventStorming

03

+ OpenAPI

Split by bounded context

Iteration 1 — Domain Story and Visual Glossary

The minimum viable domain artifact. And a surprisingly good one.

What the model was given

A numbered domain story: Cook publishes Recipe, Home Cook saves Recipe to Meal Plan …

A visual glossary: Recipe, Ingredient, Meal Plan, Cook, Home Cook

One agreed term per concept — no synonyms

What came back

The right entities, with the team's own names on them

Screens that follow the story's sequence

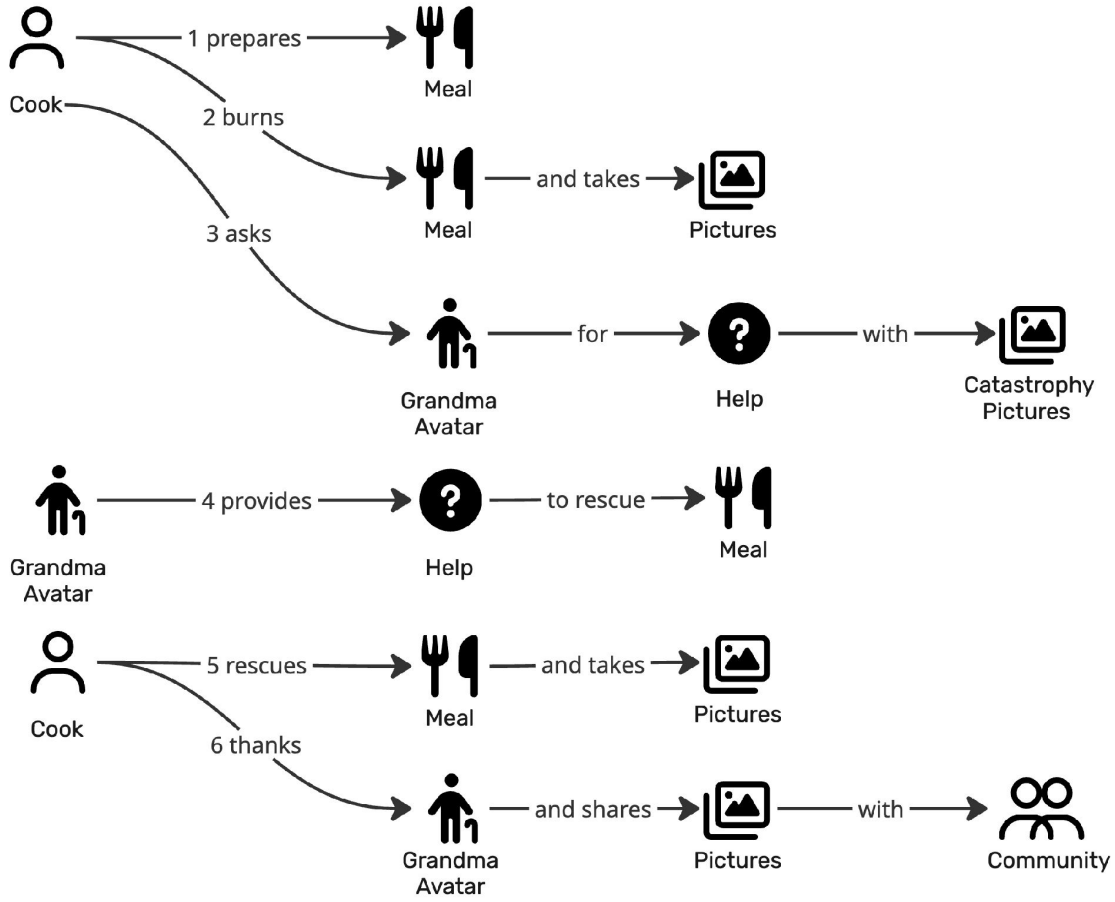
Roles kept apart — Cook is not Home Cook

A prototype you can click through in under an hour

Nouns and flows: solved. Rules: nowhere in sight.

Iteration 1 – Domain Story and Visual Glossary

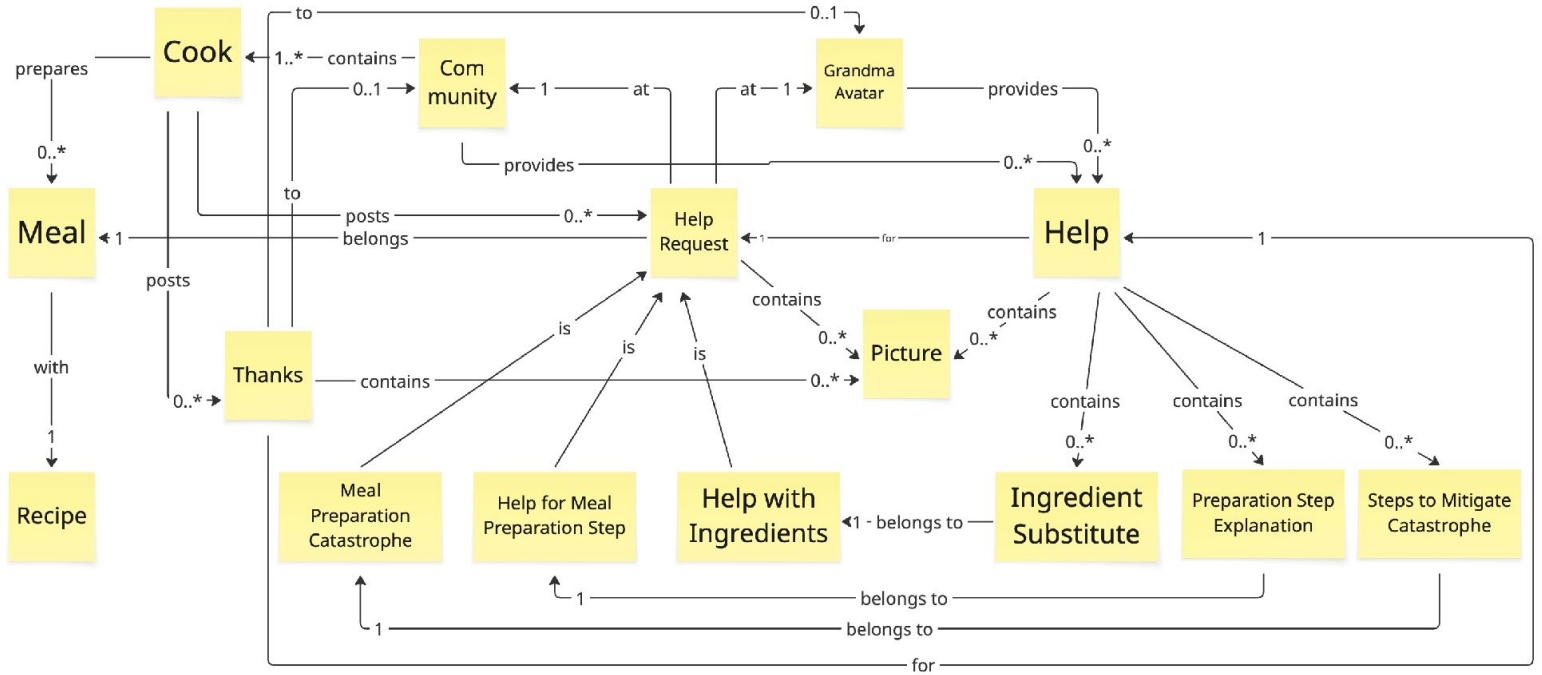
The minimum viable domain artifact. And a surprisingly good one.



Excerpt of book “DDD meets AI”

Iteration 1 — Domain Story and Visual Glossary

The minimum viable domain artifact. And a surprisingly good one.



Excerpt of book “DDD meets AI”

…and exactly where it falls apart

Every gap in the artifact becomes a guess in the output. The guesses are confident.

Invented state

The AI model gave Recipe a status field with values nobody in the domain uses.

Missing policies

Nothing stopped a Meal Plan from referencing an unpublished Recipe.

Flattened lifecycle

Draft, published and retired collapsed into one editable object.

Guessed ownership

Moderation landed inside the Recipe entity, because the story never said otherwise.

None of these are model failures. Each one is a question the workshop had not yet answered.



Iteration 2 — sharpening the language

The same product, modeled one layer deeper. Three things changed in the artifact.

01

The Ubiquitous Language got sharper

”Plan meal" split into steps and ingredient substitution. Two commands, two events, two operations.

02

Hidden policies came to the surface

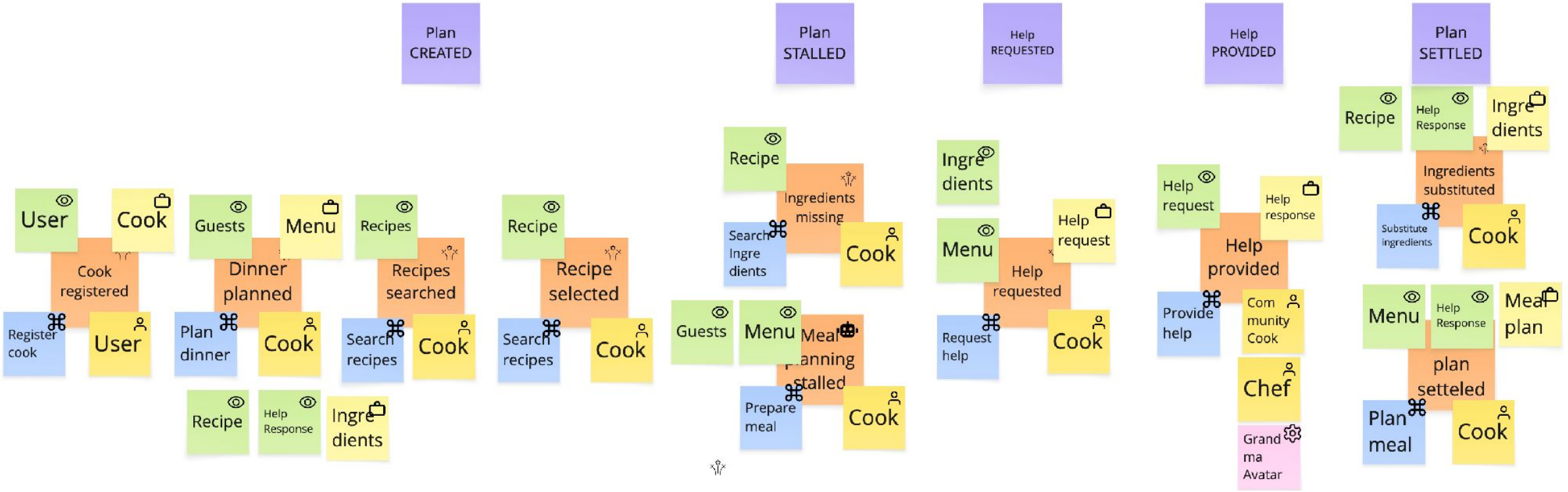
Every lilac sticky was a rule the team knew and had never written: when a plan is stalled, help can be requested

03

Boundaries stopped being implicit

Aggregates told us where invariants live, and where one context ends and the next begins.

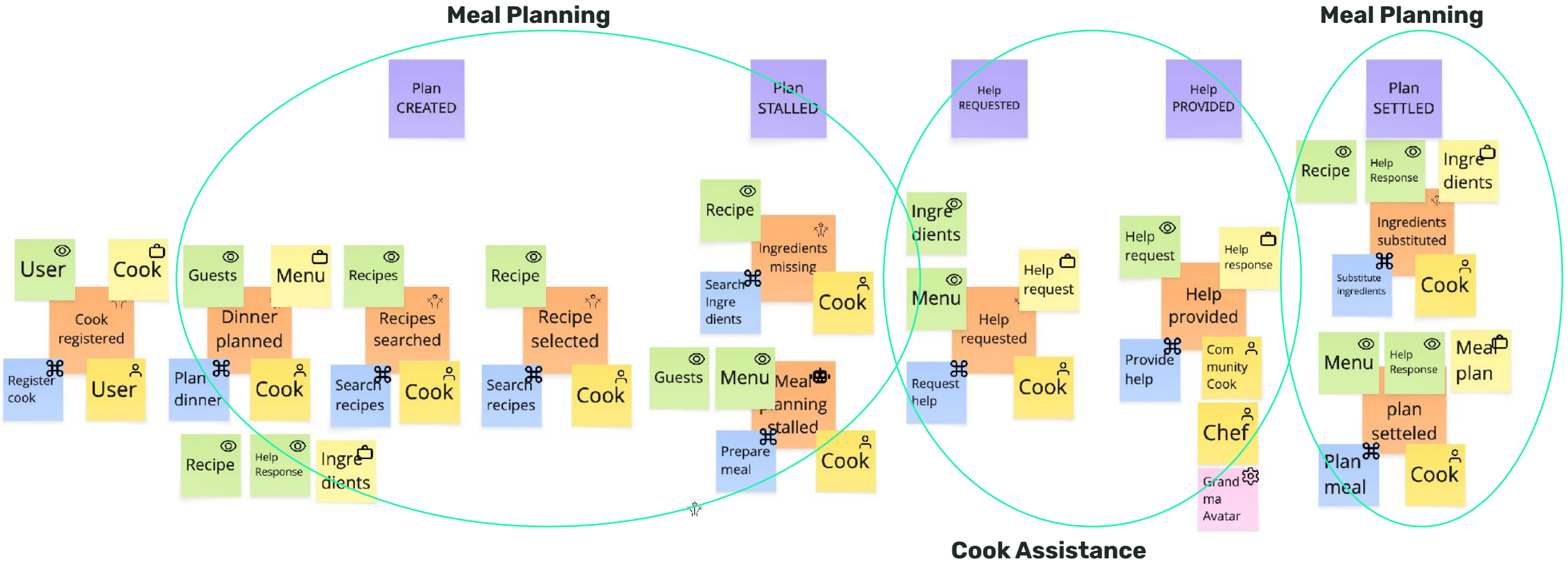
Iteration 2 – sharpening the language



Commands Events Actors Aggregates Read models States External system

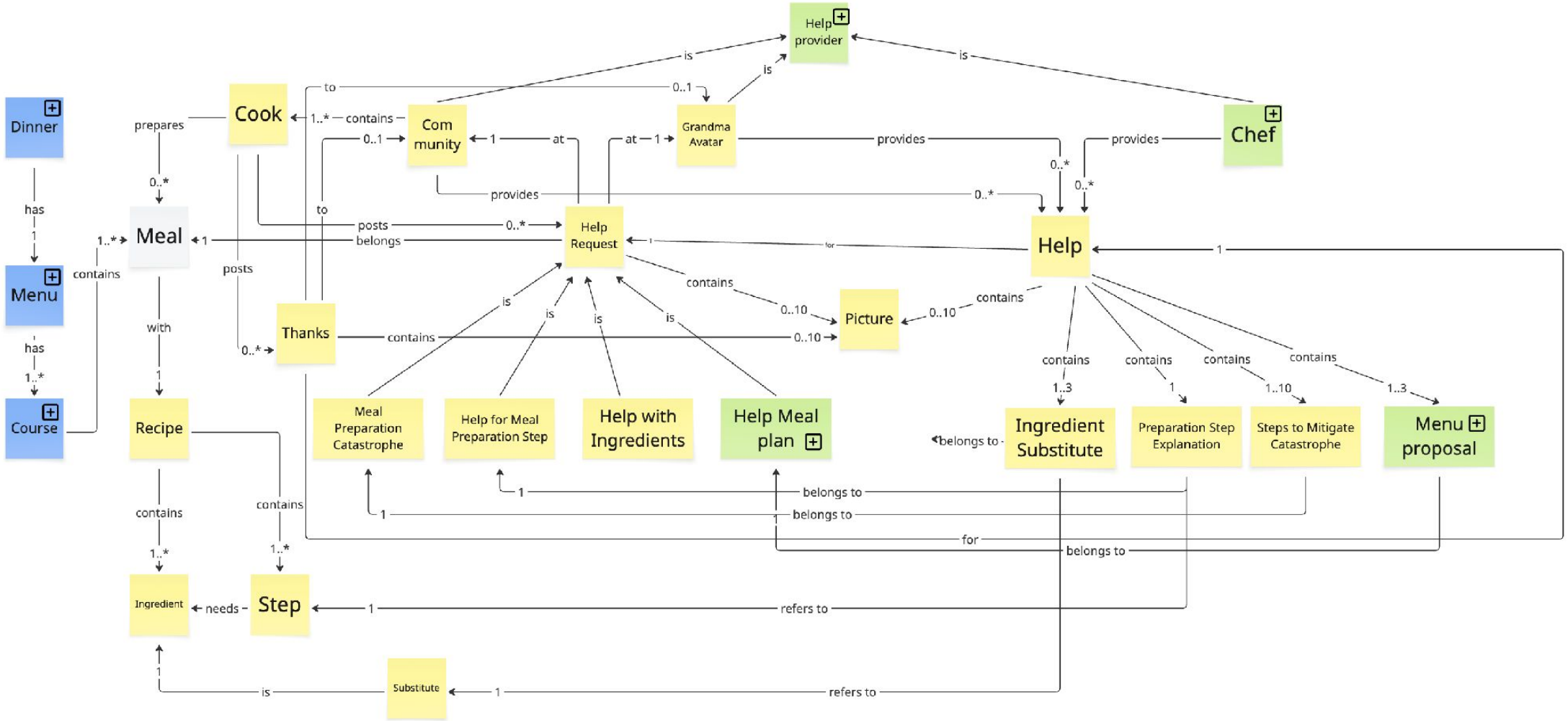
Excerpt of book “DDD meets AI”

Iteration 2 – sharpening the language



Iteration 2 — sharpening the language

Excerpt of book “DDD meets AI”



Same prompt shape, richer input

The prototype from iteration 2, compared line by line with iteration 1.

	BEFORE — DOMAIN STORY ONLY	AFTER — + EVENTSTORMING
Help lifecycle	invented status / lifecycle	**OPEN, HELP_PROVIDED, CLOSED**
Dinner	-	**added ressource**
Recipe	Ingredients + Steps independent fields	**relationships between ingredients, steps and substitutes**
Endpoints	invented endpoints communities, cooks, grandma-avatars	**removed from bounded context**

No prompt engineering happened between these two columns. Only modeling.

Iteration 3 — the spec is the domain model

Derive OpenAPI from the EventStorming board, not from the screens you imagine.

Command	→	**Operation** PublishRecipe → POST /recipes/{id}/publication
Event	→	**State transition** RecipePublished → status transitions, webhooks
Policy	→	**Constraint & error** 409 when a draft recipe is added to a plan
Aggregate	→	**Schema & scope** Recipe owns its ingredients; Meal Plan references it

Nothing in the spec is invented. Every schema traces back to a sticky on the board.

Split the spec by bounded context

One large spec invites the model to blur boundaries. Four small ones don't let it.

Recipe Authoring

Draft, publish, retire

Meal Planning

Plan, schedule, shop

Cook Assistance

Help in planning, catatrophes

Sharing

Share the experience

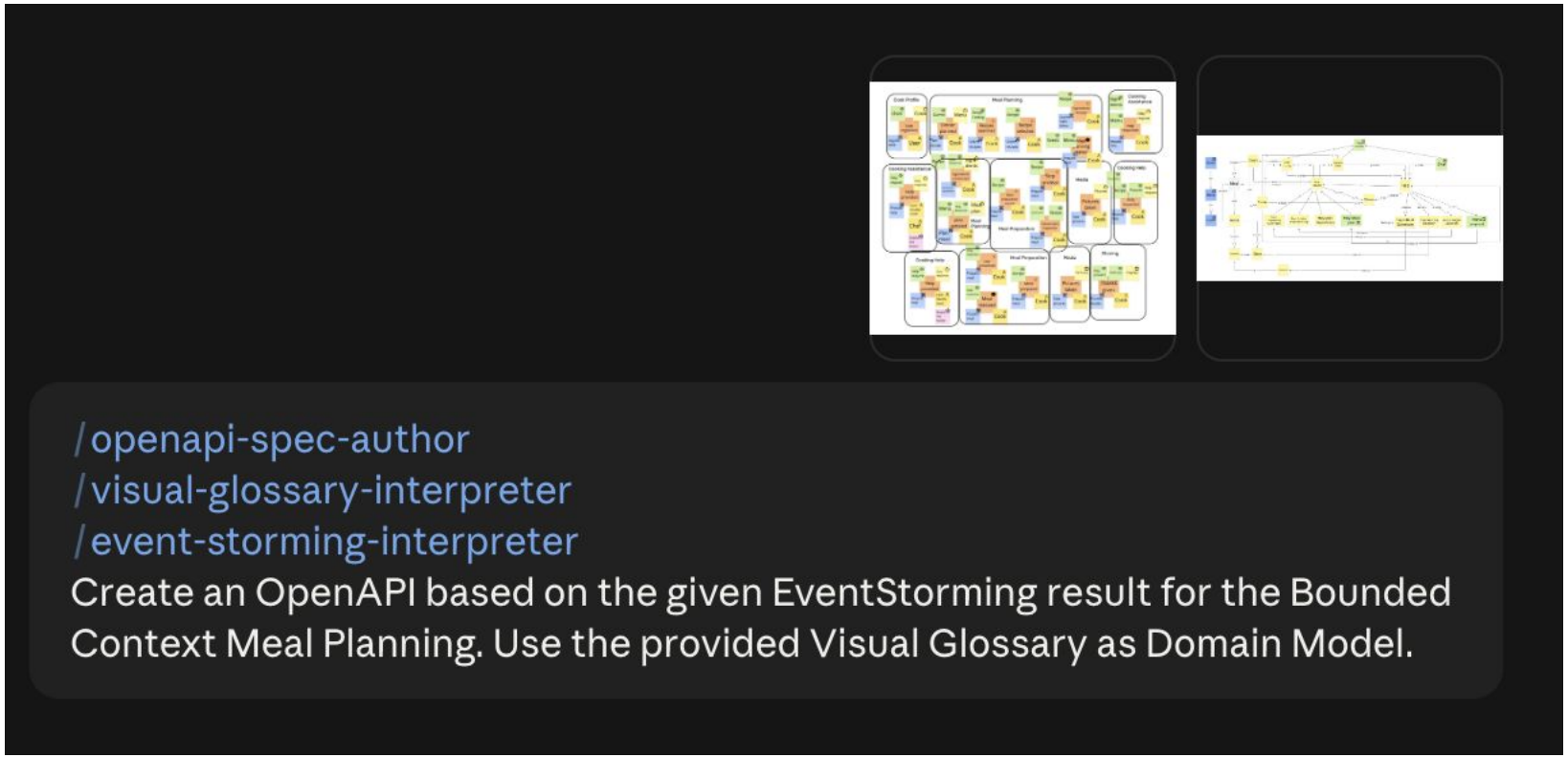
Each context fits in the model's working attention — no truncation, no drift

Terms cannot leak across borders, because the neighbouring vocabulary is not in the prompt

The frontend was generated per context and composed — no field was invented

Prompting and design are the same

Share of required behaviour present in the generated prototype, per iteration.



VisualGlossarySkill/SKILL.md

EventStormingSkill/SKILL.md

OpenApiSkill/SKILL.md

Prompting and design are the same

Share of required behaviour present in the generated prototype, per iteration.

Command (blue)	Event (orange)	Operation
Plan dinner	Dinner planned	`POST /dinners`
Search recipes	Recipes searched	`GET /recipes`
Search recipes*	Recipe selected	`PUT …/meals/{mealId}/recipe`
Search Ingredients	Ingredients missing	`GET /dinners/{id}/ingredients?availability=missing`
Prepare meal*	Meal planning stalled 🤖	`PUT /dinners/{id}/plan-status` (`stalled`)
Substitute ingredients	Ingredients substituted	`POST /dinners/{id}/ingredient-substitutions`
Plan meal	plan settled	`PUT /dinners/{id}/plan-status` (`settled`)

```yaml
openapi: 3.1.0
info:
  title: Meal Planning API
  description: |
    API of the Meal Planning bounded context of the cooking companion platform.
    A cook plans a dinner, builds its menu from courses and meals, searches the recipe catalog and selects a recipe per meal, checks which ingredients are missing, substitutes ingredients, and finally settles the meal plan.
    It is consumed by the cook-facing application and by the Cooking Assistance context, which reads a stalled plan and writes back menu proposals and ingredient substitutes.
  version: 1.0.0
  contact:
    name: Meal Planning Team
  x-api-id: 2539311a-1e44-40be-a2e4-7355f181c7b3
  x-audience: company-internal

servers:
  - url: 'https://apis.cooking-companion.org/meal-planning'

security:
  - oAuth2:
      - meal-planning:read
      - meal-planning:write
      - meal-planning:admin
```

Shared material



Each modeling step sharpens the language.
A sharper language produces better output.

That is the entire mechanism. There is no prompt trick underneath it.

Workshop decisions travel into the code

Workshop decisions were always consequential. Now the consequence is visible within the hour.

Every naming decision

becomes a type name, a field name, a route

Choosing "Home Cook" over "User" changes the generated model, not just the conversation.

Every policy sticky

becomes a guard, a validation, an error case

The rule you almost skipped is the 409 the prototype would otherwise have missed.

Every scoping choice

becomes a module boundary in the output

Where you drew the context line is where the generated code splits.

The Ubiquitous Language is not only a communication tool. It is the specification.



This is not a talk about replacing domain experts

The pipeline has a strict order of operations, and humans hold the front of it.

What the model cannot do

Decide what the business means by a word

Notice the rule that only the operations lead knows
Choose where a boundary should fall

Tell you the artifact is incomplete

What it does very well

Turn an agreed model into running software, fast
Expose gaps by filling them with confident nonsense
Shorten the loop between a workshop and a demo
Make modeling decisions falsifiable the same day

Human collaboration comes first — not as an ethical footnote, but because the pipeline does not work without it.

Take these three things with you

01

A pipeline you can run on Monday

Domain Story → EventStorming → OpenAPI, split by bounded context → prototype.

02

A test for your artifacts

Hand them to a model. Wherever it guesses, your model is under-specified — and now you know where.

03

A new lens on workshops

Your modeling artifacts are also your first prompt. Facilitate them accordingly.

The Ubiquitous Language is what makes LLM-assisted development reliable.

How was the talk?

Thanks for joining us!

Johannes & Christian

