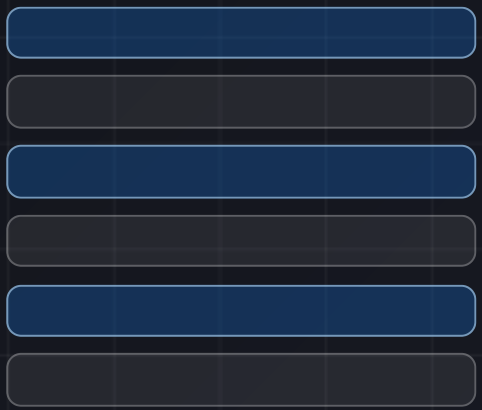
zAppDev

REFERENCE BOOK

The Intended Design in the agentic world

Why the thing to own, review and govern is the Intended Design of the application, and not the code that agents write.



Draft 0.97 | October 2026 | For technical leads, architects and decision-makers

© 2026 zAppDev. All rights reserved.

CONTENTS

What is inside

01	**Summary**	the argument on one page
02	**The problem in the agentic world**	code is cheap, coherence is not
03	**What an Intended Design is**	elements, relationships, kinds
04	**The Intended Design or the code base**	what AI should work on
05	**The anatomy of an Intended Design**	shape, references, storage, validation, build
06	**The agent's workflow**	read, change, validate, build
07	**Governing agents with an Intended Design**	review, gates, audit, access
08	**Ownership and exit**	what you keep
09	**When code is the better choice**	the limits of an Intended Design
10	**Evidence and open questions**	what is measured, what is not

Reading guide: decision-makers will get most from the summary and chapters 1, 3, 7 and 9. Architects and developers will want 2, 4, 5, 6 and 8.

> **Draft.** This is version 0.97. Chapter 9 will gain measured comparisons of cost and speed, and is marked where it is incomplete. The ownership chapter reflects the intended licence terms and will be confirmed against the final licence text.

© 2026 zAppDev. All rights reserved.

SUMMARY

In the agentic world, the Intended Design is the asset

AI coding tools can now produce code faster than any team can review it. In zAppDev, the engine produces the code from an Intended Design that is checked and verified first. What stays scarce is a trustworthy description of what the application is.

> **The Intended Design is that description: one connected, checked statement of the application's data, screens, services, logic, look and security. People and agents change the Intended Design. The engine produces the code from it. The Intended Design, not the code, is what you review, govern and own.**

The argument in five points

1. **Agents infer; Intended Designs state.** Given a code base, an agent assembles context by searching and guessing. Given an Intended Design, it asks and is told, because the relationships are written down.
2. **Mistakes should be caught before code exists.** An Intended Design can be validated as a whole. An invalid change stops at validation, with an error that names the element, and no code is generated from it.
3. **Knowledge should run, not be re-explained.** The architecture and the experience of how to write the code live in the engine, in a program called an implementation strategy, that gives the same result every time, instead of being decided again for every file.
4. **Review should happen at the level of intent.** A change to an Intended Design is small, named and kept in history. A change to thousands of generated files is neither.
5. **You should be able to leave.** The Intended Designs and the generated code can be downloaded. The code is ordinary, readable source code.

What this book does not claim

It does not claim that an Intended Design is cheaper or faster than working on a code base. That is a hypothesis, and chapter 9 sets out how to test it and what has been measured so far. It does not claim that an Intended Design is right for every program. Chapter 8 says where plain code is better.

© 2026 zAppDev. All rights reserved.

1 · THE PROBLEM

In the agentic world, code is cheap and coherence is not

AI coding tools have made code cheap to produce. They have not made it cheap to keep correct, consistent and understood.

What changed

A developer who used to write a hundred lines a day can now ask an agent for a thousand. The bottleneck has moved from writing code to everything around it: knowing what the application should do, knowing what a change will affect, checking that the result is right, and keeping hundreds of screens and services consistent with one another.

Where agents struggle on a large code base

PROBLEM	HOW IT SHOWS UP
Context is assembled, not given	The agent searches files and infers how they relate. It sees part of the system and guesses the rest.
Relationships are implicit	That a form field is bound to an attribute, or a permission guards an action, is spread across files and conventions. Nothing states it.
Errors surface late	A wrong change may compile. It fails at runtime, in a test, or in production.
Review does not scale	A person cannot read the diff of a generated change that spans dozens of files, so review turns into a glance.
Architecture is decided again	Each request is an opportunity to write the same thing a little differently. Consistency depends on the prompt, the day and the context window.
Drift	The code changes faster than any document that describes it, so the design lives only in the code.

A small example

Rename the field **Price** to **UnitPrice** on a product, in an application with forms, a service and a report. In a code base the work is: search for the word, decide which hits are the field and which are other things with the same name, edit every place, build, and find out what was missed, including names inside strings. An agent does this quickly, and it can also miss one place and leave no sign.

> **What is needed is a source of truth that is exact, connected, checkable and reviewable, and that agents and people can both work on.**

Source code is exact, and it is checkable by compilers, but it is neither connected in the sense that matters here nor reviewable at the level of intent. Documents are reviewable and not exact. The Intended Design is an attempt to be all four.

© 2026 zAppDev. All rights reserved.

2 · THE CONCEPT

What an Intended Design is

The Intended Design is the application as people think about it: purposes, business concepts, rules and screens, and the links between them. Written down, it is the complete, connected description of one application. The engine checks it, and everything else, including the running application and its code, is produced from it.

Elements and relationships

The Intended Design is made of **elements**: a class, an attribute, a form, a control, an operation, a role, a theme variable. Elements are **related** to one another. A form control is bound to a field of the form's view model. A button names the form and the action it runs. An association between two classes states how many of each there are, which side can reach the other, and what happens on delete. A permission restricts an action. A service operation uses a class.

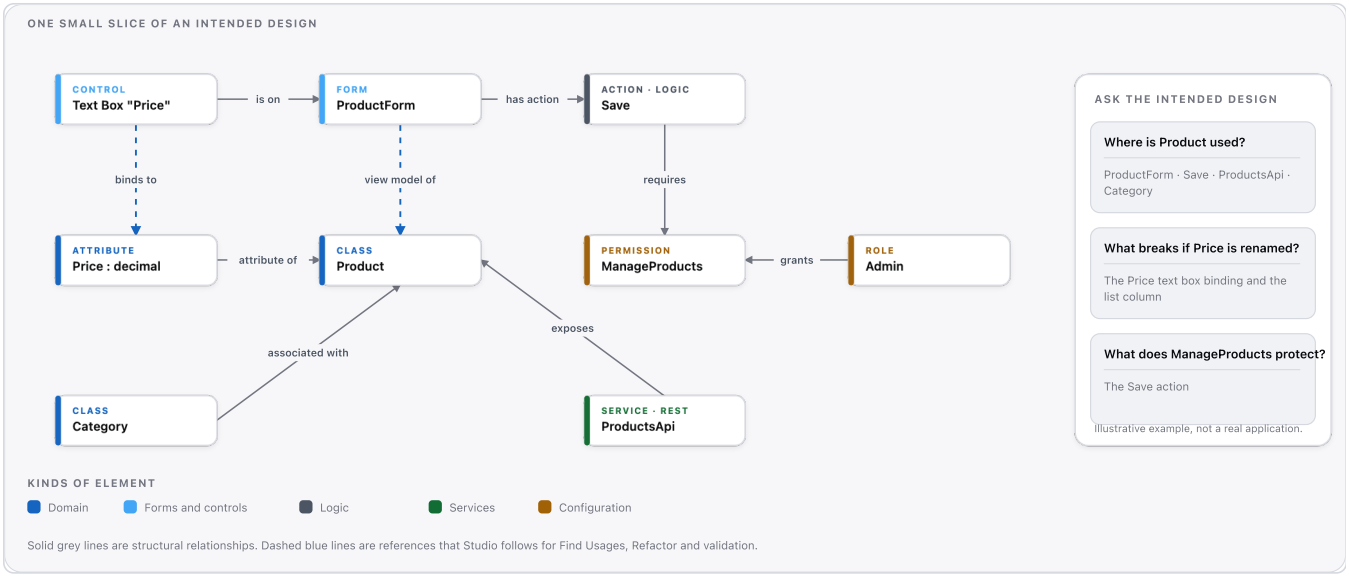


Figure 1. A small, illustrative slice of an Intended Design. Solid lines are structural relationships. Dashed lines are references that Studio follows for Find Usages, Refactor and validation.

Because the Intended Design is connected, it can answer questions that a folder of files cannot: where is this class used, what would break if this attribute changed, which actions does this permission protect.

What is in it

The Intended Design is organised by kind. The Project Explorer of an application lists the kinds, and each has its own editor in Studio.

KIND	WHAT IT DESCRIBES
Configuration	Settings, roles, permissions, authentication, localisation, and the target
Resources	Binary resources such as images and files
Domain	Business objects: classes, attributes, enumerations, associations and class operations
Forms	Screens, master pages and partial views, with their view model, controller actions and logic
API adapters	The external services the application calls
Services	The services the application exposes, as REST or MCP
Automation flows	Scheduled and event-driven work
Data mappings	Moving data between structures
Reports	Printable and exportable reports
Themes	Design variables, components, styles and icons
Unit tests	Tests of the logic

© 2026 zAppDev. All rights reserved.

Structure and behaviour

There are two sides. The **structure** is what you fill in with the editors: classes, forms, controls, services and settings. The **behaviour** is logic written in a typed language for operations of classes, actions of forms, operations of services and flows. Both sides belong to one Intended Design and are validated together, so an expression in the language that uses a class that does not exist is found before anything is built.

What it is not

- **Not a diagram.** A diagram describes a system. The Intended Design is what the system is produced from.
- **Not documentation.** It does not drift, because there is nothing separate to keep in step.
- **Not generated from code.** The direction runs from the Intended Design to the code.

Two kinds of reader

People work on the Intended Design in Studio, with an editor for each kind. Agents work on it through the application's own MCP server, with tools for reading, changing, validating and building. Both act on the same Intended Design and pass through the same validation.

© 2026 zAppDev. All rights reserved.

3 · THE COMPARISON

The Intended Design or the code base

AI coding tools and the zAppDev engine both produce code. The question is what each works from, and where a wrong answer is caught.

	AI ON A CODE BASE	AI ON THE INTENDED DESIGN
What it works on	Large text, with relationships implied	A typed structure, with relationships stated
How it finds context	Searches and infers, so it can guess wrong	Asks the Intended Design, for example where an element is used
When a wrong change is caught	Often at build time or at runtime	At validation, with an error that names the element
Consistency across many screens	Decided again each time	Set once in the engine and applied to every screen
What a developer reviews	A large diff of generated code	A change to the Intended Design
What a fix looks like	A patch to generated files	A change to the Intended Design, which regenerates the code

Both end in ordinary source code that you can read.

Six things the Intended Design adds

1. **Less to read for a given question.** The Intended Design carries no implementation detail, and an agent reads the element it needs, such as one class or one form, through targeted tools, instead of searching files. The details of one class came back in under 1 KB, against 24 KB of generated code for it. For a form the tool output was 22 KB against 168 KB of generated code. Chapter 9 reports these and also what we measured about size, which is more modest than the word "compact" suggests.
2. **Stated relationships.** Bindings, associations, permissions and references are explicit, so the agent does not infer them.
3. **Failure early.** Validation checks the whole Intended Design before a build.
4. **Repeatable output.** The same Intended Design gives the same code, because the architecture is in a program and not in a prompt.
5. **Reviewable changes.** A change to the Intended Design is small, named and kept in Git history.
6. **Replaceable output.** Generated files are rebuilt on every build. A mistake in them is corrected in the Intended Design, and the correction applies everywhere it matters.

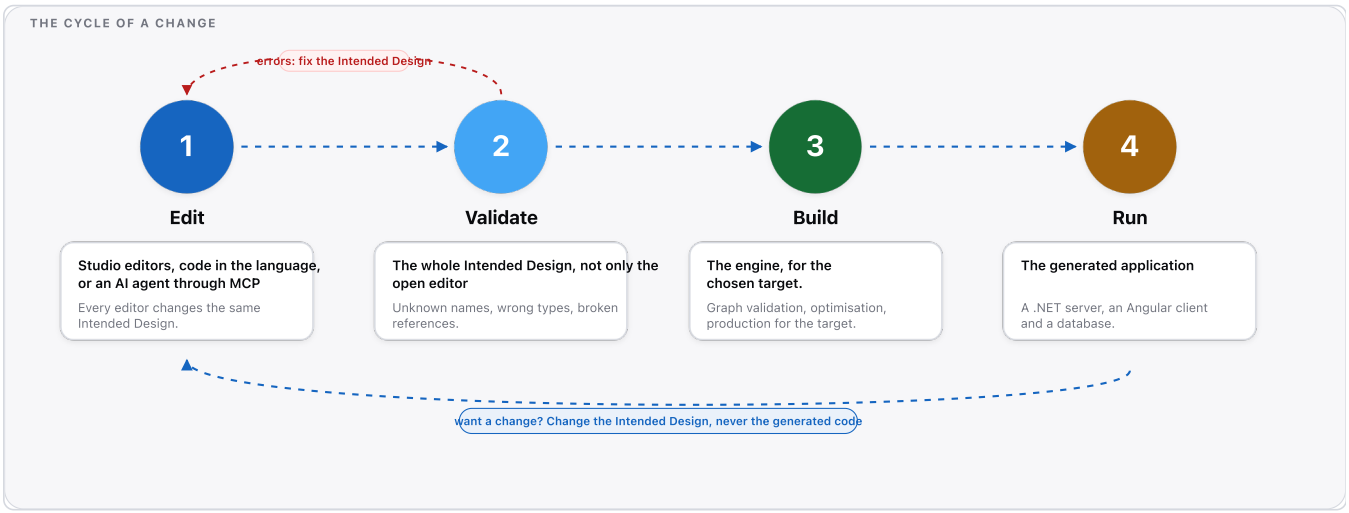


Figure 2. A change starts in the Intended Design. Errors return to editing. A change to the running application is made in the Intended Design, never in the generated code.

Spend the intelligence once

An AI that writes each file decides, again, how to structure it. The engine, with an **implementation strategy** for the target, decides once. An implementation strategy is a program that encodes the patterns, the architecture, the libraries and the security practice for one target. It runs with no inference or further reasoning and gives the same result every time, for every file of every application built with it. It is kept current as a service, so adopting an update means building again. The result of a rebuild can differ from the previous one, so it is reviewed and tested before it is deployed.

This is the division of labour: AI is used where judgement and language are needed, to understand a request and change the Intended Design. The engine is used where repetition and exactness are needed, to turn a valid Intended Design into code.

© 2026 zAppDev. All rights reserved.

4 · THE TECHNICAL VIEW

The anatomy of an Intended Design

How the Intended Design is organised, and how it behaves when you edit, validate, store, share and build it.

The shape of every Intended Design

zAppDev knows a fixed set of **element types** and the **named relationships** that are allowed between them. This set is the **shape** of every Intended Design, and every Intended Design is a graph of elements that follows it. Validation compares an Intended Design with this shape. The editors are views onto this graph: when you select a control in the Form Editor you select an element, and the Properties panel shows its attributes and relationships.

References

Elements refer to one another and not to copies of one another. A control's **Model Binding** names a member of the form's view model. A button's **On Click** names a form and an action, shown as *ProductForm / Add*. An association connects two classes. A theme component refers to theme variables. Because references are kept in the Intended Design, Studio can work with them: its context menu offers **Refactor** and **Find Usages** for an element, and a part can be moved to a module.

Storage: one unit per part

Each part is saved on its own, and this is also the unit of sharing. **Download** offers the groups Configuration, Binary Resources, Forms, Domain, Automation Flows, API Adapters, Services, Data Mappings, Themes, Snippets, UnitTests and Reports. **Upload** puts parts into an application, and fails on a name that already exists unless you choose to overwrite. The files that Studio produces are an internal exchange format: do not edit them by hand, upload and validate them in Studio instead.

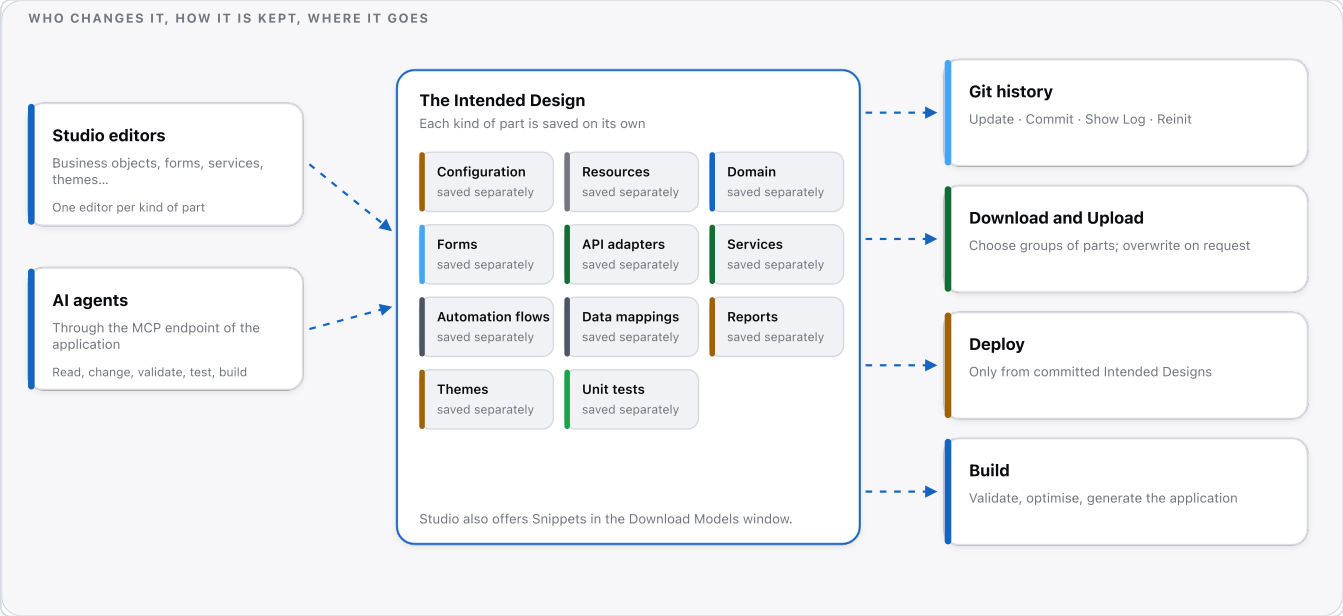


Figure 3. People and agents change the Intended Design. It is kept in units, and moves to Git history, to download and upload, to deployment and to the build.

What a download contains

A download is a folder per kind of part, with one file per part. This is the download of a real application in Studio 10.7.0, a small team application written by an agent, with ten forms of its own and the forms of the Web Application template.

FOLDER	CONTENTS
BusinessObjects	One XML file: classes, enumerations, associations and class operations
Webforms	One file per form, 33 in all. Each is a JSON document with actions, event listeners, resources, view model, data sets, conditional formatting, validations and the tree of controls
Configuration	The application's name, authentication type, roles, permissions and settings
Theme	One XML file with variables, components and style rules, and the largest file of the download
UnitTests, Snippets, Resources	Unit test files (JSON), reusable control snippets (JSON), and binary resources

© 2026 zAppDev. All rights reserved.

The download was 46 files and about 5.5 MB, or 631 KB zipped. The format is internal and can change between versions, so use the files to move parts between applications and not as an interface to edit by hand.

History

The history of the Intended Design is kept in Git. Studio shows the branch, and offers Update, Commit, Show Log and Reinit. A deployment is made from committed Intended Designs, and it refuses to run while there are uncommitted changes.

Validation

Validation runs before a build and on demand, over the whole Intended Design, not only the open editor. It finds structural problems: unknown names, wrong types and broken references. For each problem it lists the description, the location, the type of element and the part it belongs to. When validation fails, a build stops and reports that the solution metamodel could not be obtained, and asks you to validate to see the list of errors.

Validation does not find a wrong business rule. That is what unit tests and running the application are for.

The build

A build takes the validated Intended Design through three steps: graph validation, optimisation, and production for the target chosen in the application's configuration. The generated files are overwritten on every build.

A real example: the solution generated for the team application had about 900 source files (130 C#, 698 TypeScript, 84 HTML), around 5.9 MB and 150,000 lines. It contained a hash for each generated file, so that edits to generated files can be detected, and a **code map** that maps each operation in the Intended Design to the generated symbol and lines that implement it. With the code map, a line in a stack trace leads back to the operation of the Intended Design it came from.

What the Intended Design does not contain

The Intended Design does not contain the generated code, which is rebuilt each time, or the data of the running application, which lives in its database. It contains what is needed to recreate both the code and the database structure.

© 2026 zAppDev. All rights reserved.

5 · THE AGENT

The agent's workflow

An agent works on an Intended Design in the same order as a careful developer: read, change, check, build.

Connecting

Every zAppDev application has its own MCP server. An agent that speaks the Model Context Protocol connects to it with an **access token**. A token is scoped to one application, can have an expiry date, and can be revoked at any time. Studio shows ready-made configuration for Claude Desktop, Claude Code, Cursor, VS Code and Codex CLI, and a generic option for any other MCP client.

The four steps

STEP	WHAT THE AGENT DOES	WHAT STOPS A MISTAKE
1 · Read	Lists the business objects, forms, services and settings, and reads an existing example to follow	The agent works from what exists, not from a guess
2 · Change	Adds a class or an attribute, creates or edits a form, adds a service operation	Each change is made to an element of a known type
3 · Validate	Validates the whole Intended Design	An invalid change is reported with the element it concerns, and the agent corrects it
4 · Test and build	Runs unit tests and builds the application	Nothing is generated until the Intended Design is valid

The tools cover reading business objects, forms, services, API adapters, reports, themes, unit tests, roles, permissions and settings, creating and changing them, validating the Intended Design, running unit tests, building, and reading the Git difference of the Intended Design.

Your own logic as tools

A service can be of type MCP. Each operation of an MCP service becomes a tool that registers automatically in the application's MCP server, so an agent can call the application's own business logic, and not only edit the Intended Design.

A real application written by an agent

The download described in chapter 4 is an application that an AI agent wrote, as Intended Design files, from a description, and that was then imported into Studio, validated, built and tested there. It is a small team application: a board of projects and tasks, a team calendar, a timesheet, time off, a meeting planner and a weekly report.

10 + 7
classes and enumerations in one business object, with 9 associations and 27 class operations

10 forms
of its own, on top of the 23 forms of the Web Application template

5 of 5
unit tests passed in Studio, in about four seconds, and the build finished successfully

The tests are written in the same language as the application's logic. One of them seeds sample data twice and checks that the second call added nothing:

```
function void TestSeedCreatesBoardData() {
    Domain.Project.SeedSampleData();
    Domain.Project.SeedSampleData();
    UnitTests.Assert.AreEqual(5, Domain.Person.GetAll().Length, "people count (second call must not duplicate)");
    UnitTests.Assert.AreEqual(4, Domain.Project.GetAll().Length, "project count");
}
```

This shows that an agent can produce a working, tested application as an Intended Design. It was not produced through the MCP server, but as files that a person imported, so it demonstrates the Intended Design, validation, build and tests, and not the MCP workflow itself.

A recorded session

The session below was run against the application of the case above through its MCP server, with an agent making a small change and then undoing it. Every output is the real one returned by the server. The change added a **Budget** attribute to the class **Project**, and an operation that uses it.

STEP	WHAT THE AGENT DID	WHAT CAME BACK
1 · Read	Asked for the application, its business objects, the classes of one of them, and the details of the class *Project*	Four business objects. For *Project*: seven attributes (*Id, Name, Status, StartDate, DueDate, Notes, CreatedOn*) and two operations

© 2026 zAppDev. All rights reserved.

STEP	WHAT THE AGENT DID	WHAT CAME BACK
2 · Change	Added *Budget*, a persisted decimal	Accepted. The class now shows *Budget* with precision 18 and scale 2, the defaults
3 · Validate	Validated the business object	An empty list: no errors
4 · Fail	Added an operation that uses *p.Budgit*, a misspelling of *Budget*	The tool accepted the code, because its own check is only syntactic. The validation of the whole Intended Design then returned the error below
5 · Correct	Replaced the operation with *p.Budget* and validated again	An empty list
6 · Test and build	Ran all unit tests, then built	5 tests passed, 0 failed. The build generated 737 files in 14.7 seconds (compile 3.1 s, database synchronisation 13.3 s)
7 · Undo	Removed the operation and the attribute, validated, rebuilt and ran the tests	The class is as it was. The second build generated 148 files in 12.6 seconds. 5 tests passed again


```
[{"severity":"Error",
  "description":"'Project' does not contain a definition for 'Budgit'",
  "navigationInfo":{"model":"StarterBO","modelType":"BusinessObject","symbolType":"Operation"}}]
```


Three things stand out. The wrong name was caught by validation, before any code was generated, and the error named the class, the part and the kind of element. The agent needed no search to find what it was changing, because it asked the Intended Design. And the second build regenerated 148 files, not 737, so a small change to the Intended Design gave a small regeneration.

Two limits are worth stating. The tool that changes an operation checks syntax only, so a wrong name is found by validation, one step later. And the tool that returns the Git difference of a single part failed with a server error in this session. The workspace-level difference worked: it listed each part that had changed and whether it was new or modified.

© 2026 zAppDev. All rights reserved.

6 · GOVERNANCE

Governing agents with an Intended Design

When agents contribute to an application, the questions are who may change what, how a change is checked, how it is reviewed, and how it is traced. An Intended Design gives each question a place to be answered.

QUESTION	WHAT THE INTENDED DESIGN OFFERS	WHAT TO DECIDE
Who may connect an agent?	Access tokens scoped to one application, with an optional expiry, revocable at any time	One token per agent, an expiry on each, and a rule for who may create tokens. Treat a token like a password: whoever holds it can change that application's Intended Design
How is a change checked?	Validation of the whole Intended Design before any build, with errors that name the element	Make a valid Intended Design and passing unit tests the condition for a build
How is it reviewed?	A change to the Intended Design, kept in Git history, and readable as a difference between versions	Review the Intended Design change first, and read the generated code only where something looks wrong
How is it traced?	Git history of the Intended Design, and application logs	Commit agent changes separately and describe them, so the log shows who or what changed what
How is it released?	Deployment environments, and deployment from committed Intended Designs only	Which environment an agent may reach, and who commits and deploys
How is a defect fixed?	Generated files are rebuilt on every build	Fix the cause in the Intended Design, not in the generated files, so the fix is not lost

Roles in a team that uses agents

- **Design owner.** Decides the structure: the domain, the views, the roles and permissions.
- **Agent operator.** Gives agents their work and their tokens, and removes both when the work is done.
- **Reviewer.** Reads the change to the Intended Design, checks validation and tests, and approves the commit.
- **Release owner.** Deploys from committed Intended Designs to an environment.

One person can hold several roles in a small team. The point is that each question has an owner.

Security of the applications themselves

Roles, permissions and authentication are elements of the Intended Design like any other. A permission is attached to the action it protects, and a menu item is hidden from users who lack the permissions of the action it opens. When an agent adds a form or an action, the Intended Design shows which permissions it requires, so a reviewer can check them.

> **Note.** This chapter describes what the platform offers and a way to use it. It is not a compliance framework, and a team that works under a regulation should map its own controls to these features.

© 2026 zAppDev. All rights reserved.

7 · OWNERSHIP

Ownership and exit

An Intended Design is only an asset if you own it and can take it away.

> **Status.** This chapter states the intended position. It will be confirmed against the final text of the licence agreement before version 1.0.

What you own

Your Intended Designs
Everything you create in zAppDev: business objects, forms, services, themes, logic and settings.

Your source code
The code that zAppDev generates from your Intended Designs.

Your data
The data you use to test your applications and the data of your running applications.

What zAppDev owns

The platform: the software, the engine, its targets and the service. The licence lets you use them. It does not transfer them.

Access is separate from ownership

You own the Intended Designs and the code. The Intended Designs and the generated solution can be downloaded at any time.

What leaving looks like

- You download the Intended Designs and the generated solution.
- The generated code is ordinary source code that you can read, keep in your own Git repository and run on your own servers. It is open and free to run anywhere.
- You then maintain that code yourself. You give up regeneration: a change to the Intended Design no longer produces new code unless you continue to use the engine.
- The Intended Designs stay yours. If you come back, they can be uploaded again.

This is the honest shape of the exit. The code is not locked, and the productivity of the engine is the thing you lose when you leave it.

Why this matters for agents

If an agent's work lives only in a vendor's runtime, the organisation does not own what the agent built. When the Intended Designs and the code are yours, agent output is a normal asset: it can be audited, moved, backed up and handed over.

© 2026 zAppDev. All rights reserved.

8 · THE LIMITS

When code is the better choice

An Intended Design is not the right tool for everything. A book that said otherwise would not be worth reading.

SITUATION	WHY CODE IS EASIER	IN ZAPPDEV
A new algorithm	It has no place in an Intended Design of data, screens and services	Write it in the language, or in an external library that the Intended Design calls
An unusual stack	No target exists for it	A target can be written for another stack. Contact the zAppDev team about writing one
Performance-critical code	Every instruction matters and you want direct control	Keep that component outside the Intended Design and call it
A very small, short-lived program	The overhead of an Intended Design is not repaid	Write a script
A team that knows its stack deeply	AI and people are both fluent in common languages	The Intended Design and the docs give agents the vocabulary, but the gap exists

An Intended Design has a shape

An Intended Design describes structured, repeated, relational applications well: data, screens, services, security and look. That is also its limit. It cannot express what it has no element for, which is the price of being checkable. zAppDev keeps a way out in two places: logic is written in the language, and external and custom libraries can be called from the Intended Design.

Combine them

The practical answer for a full-stack application is a combination. The Intended Design carries what is structured and repeated. Code carries what is new. The boundary should be explicit: a piece of custom code is called from the Intended Design through a defined operation, so it stays visible to the Intended Design, to validation and to anyone asking what depends on it.

> **Use the Intended Design for the structure of the application and code for what is new. Do not use code to repeat what an Intended Design could state once.**

© 2026 zAppDev. All rights reserved.

9 · EVIDENCE

What is measured, and what is not

The argument of this book is partly reasoning and partly observation. This chapter separates them.

CLAIM	STATUS
Validation reports errors with the element and the part, and stops a build	OBSERVED IN STUDIO 10.7.0 AND OVER MCP
Relationships are stated and can be queried (Find Usages, Refactor)	SEEN IN STUDIO
An agent can change an Intended Design through MCP, have a wrong change caught at validation, correct it, and rebuild	OBSERVED: ONE SESSION
An agent can write a working application as an Intended Design, and its tests pass	OBSERVED: ONE APPLICATION, 5 OF 5 TESTS
Generated files are overwritten on every build	DOCUMENTED BEHAVIOUR
The Intended Design is smaller than the generated code	MEASURED ON TWO APPLICATIONS (below): smaller, but only modestly by bytes
Validation finds every broken use after a change (rename test: 8 of 8)	OBSERVED: ONE TEST
A search over the Intended Design finds all references	NOT IN THIS TEST: 25 OF 34
An agent on the Intended Design makes fewer mistakes than on a code base	HYPOTHESIS
Working on the Intended Design costs less than working on a code base	HYPOTHESIS
Review at the level of the Intended Design is faster than code review	HYPOTHESIS
Repeated runs of an agent on the Intended Design agree more often	HYPOTHESIS

Two measurements: size

We compared an Intended Design with the source code generated from the same build, twice. Third-party libraries, binary resources and build outputs were left out. The comparison counts files on disk. It is not a count of tokens.

APPLICATION	INTENDED DESIGN	GENERATED SOURCE	INTENDED DESIGN AS A SHARE
zERP_Plus: large line-of-business application, 605 form files and 172 business object files, an earlier platform version	about 105 MB and 0.5 million lines	about 179 MB and 3.9 million lines	about 60% of the bytes, about 1/8 of the lines
Team application: small, written by an agent, 10 forms of its own on top of the 23 of the template, Studio 10.7.0	about 5.1 MB and 53,000 lines	about 5.9 MB and 151,000 lines	about 87% of the bytes, about 1/3 of the lines

Two things follow. First, the Intended Design is smaller than the generated code, but by bytes it is not dramatically smaller, and in the small application it is nearly the same size, because the theme and the template forms make up much of it (the theme alone is 2.7 MB). Second, line counts flatter the Intended Design, because a form is stored as one very long line of JSON, so bytes are the fairer measure.

So the claim that an Intended Design is "small" holds only modestly. The stronger advantage is not size. It is that the relationships are stated, so an agent can read the one element it needs and ask where it is used, instead of reading and searching files. The size of the part an agent has to read for a given task, which a targeted tool returns, is what should be measured next.

A benchmark: three tasks, both routes

We ran three tasks on the team application, once through the Intended Design (the MCP tools) and once on the generated code base. The code base could not be built here: the generated solution needs a newer .NET than the machine has. So the comparison counts what can be counted

© 2026 zAppDev. All rights reserved.

honestly on both sides: the calls made, the amount of text the agent has to read, and whether the answer is complete, checked against the Intended Design itself. It is not a comparison of tokens billed or of time to a working build on the code side.

TASK	THROUGH THE INTENDED DESIGN	ON THE GENERATED CODE
1 · Impact. Where is *Project.DueDate* used?	1 search call, about 9 KB: 25 matches, each with part, class, operation and line. **Found 25 of the 34 occurrences (74%).** All the code was found. The attribute definitions and the 7 control bindings in three forms were not. Completing it means inspecting forms one by one (a view model is about 3 KB, a layout about 22 KB)	1 search, 63 matching lines in 21 files, about 9 KB. Case-insensitive: 119 lines in 26 files, about 17 KB. Complete for the generated files, but spread over the C# domain, view models, controllers and the Angular client, and not attributed to an Intended Design element
2 · Rename. Rename the class *UiPref*	Rename: 1 call, which **does not update references** (the tool says so). Validating 3 parts then reported **all 8 operations and actions** that used the old name, with their part and kind, plus 5 derived errors. Renaming it back and validating gave no errors	46 lines in 11 files, 3 of them file names, in 3 spellings (*UiPref*, *UiPrefExtensions*, *uiPref*). All of them are generated files that the next build overwrites, so the change has to be made in the Intended Design anyway
3 · Add a field end to end. Add *Budget* to *Project* and show it on the form	8 calls: read the layout (about 22 KB), add the attribute, add the view-model property, add the control, validate two parts, build, run tests. Validation clean. **Build: 163 files in 16.5 seconds. 5 of 5 tests passed.** No generated file was touched by hand	Not run. By analogy with *DueDate*, a field shown on a form and a list appears in about 21 generated files and 63 lines. Those edits would be lost at the next build

What the benchmark shows

- **Targeted reads are far smaller than the generated code.** The details of the class *Project* came back in under 1 KB, and the generated code for it is three files totalling 24 KB. The form *ProjectForm* is 14 generated files totalling 168 KB, an Intended Design file of 88 KB, and a layout of 22 KB from the tool.
- **Validation is a good impact tool.** After the rename, one validation per part found every broken use, including those inside form actions. That is more reliable than the search in task 1, which skipped non-code references.
- **The tools have gaps, and an agent has to know them.** The search covers code and not control bindings. Renaming a class does not update its references. An attribute that was added to a form's view model cannot be removed through the tools: removing the attribute from the class left a stale entry in the form, which validation reported with a helpful hint, and which has to be removed in Studio.
- **The Intended Design route is not cheaper on its face.** Task 1 cost about the same amount of text on both routes, and task 3 needed eight calls. The tool definitions also take space: the definition of the single tool that changes form controls is about 20 KB. The benefit in these tasks was correctness and feedback, not fewer calls.
- **Builds are incremental.** The same change cost 737 files on the first build and 148 to 163 on later ones. The build after adding the control and the build after removing it both generated 163 files.

What it does not show

These are three tasks on one small application, run once, by the same agent that wrote the application, so they are an illustration and not a controlled study. Nothing here measures billed tokens, human review time, or the agent's mistakes on the code side. Whether three runs of an agent agree was not tested. The code-side half of task 3 is an estimate. The error found at validation (a misspelled attribute) has no code-side equivalent here, because the code could not be built.

> **To add.** A run of the same tasks on the code side with a build, billed token counts for both routes, and repeated runs. Until then, the claims about cost, speed and fewer mistakes stay hypotheses, and the final licence wording in chapter 7 is also to be confirmed.

© 2026 zAppDev. All rights reserved.

GLOSSARY

Terms used in this book

Intended Design
The complete, connected description of one application: the structure drawn in Studio and the behaviour written in the language. Everything else is produced from it.

Element
A single node in the Intended Design, such as a class, a form, an operation or a role.

Relationship
A named link between two elements, such as a binding, an association or a reference.

Part
One saved piece of an Intended Design, such as a form or a business object diagram, kept as its own file.

Kind of part
A part of the Intended Design, named by what it describes: business objects, forms, services, and so on. Each kind has an editor.

Shape
The types of element and the named relationships allowed between them. The shape every Intended Design follows.

Validation
The check of the whole Intended Design before a build and on demand. It finds structural problems, not wrong business rules.

The engine
Checks and optimises the Intended Design, then produces the application for the chosen target.

Target
One stack the engine produces for, through an implementation strategy: a program that encodes the architecture, patterns and practice for one target and runs the same way every time.

The language
The typed language for the logic of an application.

MCP
Model Context Protocol. Lets an AI agent use an application through standard tools. Every zAppDev application has its own MCP server.

Agent
An AI that carries out a task by reading and changing things through tools.

© 2026 zAppDev. All rights reserved. This document is provided for information and may be shared in its original, unmodified form. Other product and company names are the property of their respective owners.

See the Intended Design in Studio

Create a free account and open a template application to look at its Intended Design. The documentation page on the Intended Design goes deeper into how it is organised.