Source: cppcc docs/Adaptive_Programming_p2t_Use_Case_Draft_0.1.md @ 7c04345 — curated 2026-08-10; edits land in the source first.

Adaptive Programming — the p2t use case

Designing around an intractable problem, and the record of doing it

Draft 0.1 — 2026-08-09

The companion guide, `Adaptive Programming — a user's guide`, uses b3ubot as its worked example and describes the method in general. This document is about one project: p2t, which translated a working Python web service into TypeScript and then had the result graded by a test suite written by strangers.

It has two halves, and they answer two different questions.

§2 is the design — how the problem was reshaped into one that could be attacked at all. §3 is the steps — the record of doing it, including the parts that went differently than planned.

---

1. The claim being tested

p2t was not built to ship a TypeScript service. It was built to test a prior claim — that substrate-grounded construction works well for *data-model-shaped* code (CRUD over explicit entities, relationships and validation rules) and poorly for *script-shaped* code (imperative pipelines, metaprogramming, side effects orchestrated through the shell).

A web API is the first kind. So the experiment picked one, translated it, and submitted the result to an oracle nobody involved controlled.

That framing matters for reading everything below: **p2t is an experiment with a stated hypothesis, not a product with a deadline.** Where the record shows slices expanding tenfold, that is the experiment reporting what it found.

---

2. Design to address the NP-hard nature of code translation

2.1 Why the direct approach does not scale

A conventional transpiler maps source syntax to target syntax. It works when the languages are close and degrades badly when they are not, because the interesting content of a program is not in its syntax — it is in the relationships between its parts.

Treated as a search, whole-program translation is brutal. The translator must choose, for every fragment, among many locally-valid renderings, and those choices are *coupled*: how you render an entity constrains how you render every query touching it, every route returning it, every validation rule guarding it. Choosing a globally consistent assignment across a program's fragments is the shape of problem that does not admit an efficient general solution — the combinatorics are the point, not an implementation detail.

Two escapes exist from a problem of that shape. Approximate, or change the problem. p2t changes the problem.

2.2 The reframe: reverse-engineer, then forward-engineer

The pivot is stated in the arc's own specification:

> Rather than translating Python syntax directly into JavaScript syntax (which is what conventional transpilers attempt), p2t first **reverse-engineers the upstream Python implementation back to a design-level representation** — a UML-style description of entities, relationships, request/response contracts, and control flow — and then **forward-engineers that design into TypeScript**.

The observation underneath it is almost banal: **this is what a competent engineer does when porting a service between languages.** Read the source, write up the design, write the target. Nobody ports a system by walking its syntax tree.

What that buys is decomposition. Once the program is expressed as a design rather than as text, it can be partitioned — and if the partition is chosen so that pieces do not interact, the coupled global search collapses into many small independent ones.

That partition is the whole design problem, and it is what `notes/` contains.

2.3 The design layer

Nine documents, 2 393 lines, hand-authored, none of it generated:

document	lines	what it fixes
realworld-uml/ — written in .B, describes the upstream service		
entities.md	323	ER diagram, per-entity fields, types, validation, DB tables
routes.md	212	all 19 routes, auth class, error envelope
contracts.md	338	per-route request / response DTO shapes
data-flow.md	206	how a request moves through the layers
auth-vertical.md	252	the auth path end to end, as a walkthrough
layered-architecture.md	281	per-file layer map
README.md	102	what the tree is for
notes/ — the partition and the bring-up		
fragment-mapping.md	556	THE PARTITION — every source fragment → its family
setup.md	123	local bring-up

The first seven were written in sub-slice `.B` and describe *the upstream Python service as it is* — pinned to an exact commit, so the description has a subject that cannot move underneath it. They are observation, not invention.

`fragment-mapping.md` is different in kind, and it is where the design argument lives.

2.4 The partition: fragment families

`.C` walked every layer of the upstream service and assigned each source fragment to a **fragment family** — a category of thing whose members share a structure. The taxonomy was then **frozen**, and everything downstream is organised by it: one grammar per family, one extractor per family, one emitter per family.

ID	family	count	what it captures

```
├────┼──────────────────────────┼──────┼──────────────────────────────┤
│ F1 │ DataModelEntity          │   28 │ entities and wire facets     │
│ F2 │ DataModelRelationship    │    6 │ junction tables, links       │
│ F3 │ RouteContract            │   19 │ one per endpoint             │
│ F4 │ RouteHandler             │   19 │ paired with its contract     │
│ F5 │ RepositoryQuery          │  12+ │ per persistence method       │
│ F7 │ AuthPolicy               │   13 │ security, JWT, deps/auth     │
│ F8 │ ValidationRule           │  11+ │ per rule                     │
│ F10│ ErrorHandling            │   5+ │ handlers, types, registry    │
│ F11│ Configuration            │    8 │ settings and factory         │
│ F12│ ApplicationBootstrap     │   13 │ wiring and startup           │
└────┴──────────────────────────┴──────┴──────────────────────────────┘
```

Roughly 110 surface-level fragments; composites contribute more than one family assignment each, which is why the column sums higher.

Three properties of this table are doing the real work.

It is finite and small. Ten families cover a complete web service. The intractable global choice becomes ten local, reusable ones — and each is made *once*, in a grammar, not per fragment.

It is derived, not imagined. Every row of the assignment table names a real fragment at a real file and line in the pinned upstream source. The taxonomy was validated against the code before it was frozen, which is why `.C`'s histogram differs from `.B`'s provisional one: `.B` counted per file, `.C` per fragment. Both are recorded, with the discrepancy explained rather than quietly reconciled.

It was allowed to shrink. Two proposed families were dropped during validation, each with its reasoning kept:

- **F6 ServiceOrchestration** — dropped.
- **F9 ImportDependency** — dropped, because imports are *edges of the dependency graph, not nodes of it*. Modelling them as fragments would have made the count unbounded (~∞ in the table) for no gain.

That second one is the good kind of design decision: a category that sounds obviously necessary, examined against the actual code, and found to be a different sort of thing entirely. The numbering gap — there is no `F6` or `F9` in the shipped grammars — is the visible scar of a decision that was made rather than assumed.

2.5 Why this design froze

`notes/` has been touched by two commits. It has not changed since.

That is not neglect. **p2t's design was reading, not deciding.** What it describes — an existing service at a pinned commit, and a public API specification — was fully determined before the project began. An accurate description of something that does not change has no reason to change.

Compare b3ubot, whose design document grew 2.6× across 27 commits because every question it answered had to be *decided*, and decisions get revisited.

Same method, same slot in the method, opposite dynamics — and the contrast gives the general rule: **a design document stops moving exactly when it runs out of decisions to record.** Freezing is a virtue when the subject is fixed and a warning sign when it is not.

---

3. The steps

3.1 One arc, not a queue

b3ubot's plan is a queue of numbered units of work, picked up as they unblock. p2t's is a single arc — `step.3000` — cut into lettered sub-slices, because its work is a pipeline: each slice consumes what the previous one produced.

```
.A  bring up the upstream service, record the baseline
 │
 ▼
.B  reverse-engineer the design         →  notes/realworld-uml/
 │
 ▼
.C  assign fragments to families, FREEZE →  notes/fragment-mapping.md
 │
 ▼
.D  author one grammar per family
 │
 ▼
.E  decompose Python into the fragment DAG
 │
 ▼
.F  emit TypeScript, one emitter per family
 │
```

```
   ▼
.G  serve it, run the external suite
```

The dependency is visible in the filename. A backlog would have hidden it.

3.2 What was planned, and what happened

The arc file's roadmap listed ten sub-slices, `.A` through `.J`, each estimated at "half a day to a day". Here is the estimate against the record:

slice	scope	planned	actual specs
.A	upstream + baseline	1	1
.B	reverse-engineer the design	1	1
.C	taxonomy, frozen	1	1
.D	grammars, one per family	1	13
.E	decomposer	1	10
.F	emitters	1	19
.G	TS runtime + external suite	1	3
.H	article entity	1	0
.I	comments, tags, relationships	1	0
.J	Rust target (stretch)	1	0

Read that table honestly and it says three things.

The design slices landed as estimated. `.A`, `.B`, `.C` — one each. The work of *understanding* the problem was predicted accurately.

The construction slices ran 10–19×. `.D`, `.E`, `.F` — the parts that build machinery — expanded by more than an order of magnitude. Forty-two of the forty-eight specifications sit in those three letters.

Three planned slices never happened as slices. `.H`, `.I` and `.J` have no step files. The article, comment and tag work was absorbed into the `.F` series instead of being sequenced separately, and the Rust stretch goal was not attempted. The arc file had already licensed this: *"This roadmap is indicative, not contractually committed: sub-slice boundaries may shift as the experiment surfaces real difficulties."*

That clause is worth copying. A plan that cannot be re-cut mid-flight gets falsified into uselessness by the first surprise, and then gets quietly ignored. A plan that says in advance how it may move stays honest to the end.

3.3 The record itself

artifact	count	what it is
step.3000.txt	1	the arc: hypothesis, test case, roadmap, external references
step.3000.<slice>.txt	48	one specification per slice
step.3000.<slice>.diff	48	narrative retrospective
step.3000.<slice>.adiff	48	the actual diff that landed
total	145	

Of the 48 specifications, 46 are ordinal-numbered sub-slices — `.A` is the first, `.G3` the forty-sixth. The other two are out of band, and they are interesting.

The retrospective is doubled, deliberately. `.diff` is written prose: what was assumed, what turned out false, what the gates caught. `.adiff` is the mechanical diff of what actually landed. Keeping both is a guard against a failure mode specific to this way of working — a narrative retrospective is written by the same process that did the work, and a comfortable story is cheap. The mechanical diff is not persuadable. When the two disagree, the diff wins.

3.4 When a step reaches into another repository

Two files break the naming pattern: `step.3000.D3.cppcc.txt` and `step.3000.D6.cppcc.txt`. Both are described in their own headers as a **substrate-fix step** — specifications written against the compiler rather than against p2t.

This is what happens when executing a step reveals that the tool, not the work, is wrong. The options are to work around it silently, or to stop and write a specification for fixing the tool. p2t did the second, twice, and left the evidence in its own step directory under a name that says where the change actually went.

A third, `step.3000.D3.shapeB.txt`, is a follow-on cleanup — a slice spawned because the previous one left something in a state its author was not willing to build on.

Neither pattern is in the method's diagram. Both are what the method looks like when it meets a real toolchain.

3.5 What the record shows

Forty-eight specifications produced a service that answers nineteen routes across five entities with authentication, relational data and pagination — and the deliverable is checkable by anyone, because the checker is not ours.

The measured result, stated exactly:

- The **upstream Python** implementation scored **280 of 280** assertions on the vendored collection when the baseline was recorded at `.A`, on a pinned commit.
- The bar set for the generated TypeScript was **at least 280 assertions passing**.
- A current run of the generated TypeScript reports **293 of 293**.

That is not the translation outscoring the original, and it is worth being precise about why, because the reason turned out to be more interesting than the caution.

On 2026-08-10 both numbers were re-measured on the same day, with the same collection and the same flags: upstream Python scores 280 of 280, the generated TypeScript 293 of 293. Same conditions — so the gap is not a versioning artefact. Every one of the thirteen extra assertions belongs to a single request:

```
request                       Python    TypeScript
Feed                               5            18
TOTAL assertions                 280           293
FAILED assertions                  0             0
```

The collection's `Feed` test **branches on the response**, and `Feed` runs before the follow steps — so the viewer follows nobody and the feed must be empty. The Python service returns an empty feed and the script makes one assertion: *"articlesCount is 0 when feed is empty"*. The generated service returns the viewer's own article, so the script takes the non-empty branch and runs fourteen article-shape assertions instead. All of them pass.

The single assertion that would have caught the difference is the one that does not run.

That is the finding worth carrying away: **a `passed ≥ baseline` bar cannot see a divergence that adds passing assertions — it rewards it.** The score went up because the two services behaved differently. An acceptance bar is a floor, and a floor tells you nobody fell through it; it says nothing about whether two implementations agree. Comparing the *assertion profile* — which requests asserted how many times — is what shows that, and it is now a script rather than an observation.

The defensible claim is therefore the one the bar was written for: *the generated service meets an external standard that the original also meets*. That is a real result against a checker this project did not author. It is not behavioural equivalence, and the suite alone cannot tell you which of the two you have.

Separately, each pipeline stage regenerates its own output and diffs it byte-for-byte against what shipped. A green run therefore means the pipeline is *deterministic*, not merely that it produced something — which matters far more for generated code than for hand-written code, because nobody is going to notice instability by reading it.

---

4. What the two halves say together

§2 is a design that made an intractable problem tractable by partitioning it. §3 is the record of building on that partition, including a tenfold estimate miss and two occasions where the compiler had to be fixed first.

Neither half is impressive alone. A clean taxonomy with no implementation is a diagram; forty-eight specifications with no organising idea is a lot of typing. The claim worth making is that **the design is what made the steps small enough to specify** — ten families, each one grammar, each one emitter, each verifiable on its own — and that the steps are what turned the design from an assertion into something an outside test suite would sign off on.

And the honest limits, stated plainly:

- The emitted TypeScript is not offered as code you would enjoy maintaining. It passes a behavioural specification.
- This is not a general Python-to-TypeScript translator. The pipeline was built for one service's shape. What generalises is the method and the checking, not the emitter.
- The claim under test was about *data-model-shaped* code, and a web API is the friendliest possible instance of it. p2t is evidence for that class and says nothing about script-shaped

code — which is exactly what the original claim predicted, and exactly why the test case was chosen.

Further reading

- **Adaptive Programming — a user's guide** — the method in general, with b3ubot as its worked example.
- **Cross-Language Translation** — the full account of p2t.
- **Two Domains** — the position paper whose claim this experiment was built to test.
- **The `p2t` use case** — the pipeline, its ten grammars, and the external suite. `scripts/convert_and_test.sh` reproduces the result on your own machine.