Coldcode Si Labs Library WORKING PAPER 2026-01

Constraint-Native Business Model Design

Quota-bounded autonomy, proof-first acquisition and verification debt in a zero-cash, AI-built venture

Courage Garikai Funded by Coldcode Si Labs

Coldcode Si Labs Library Working Paper 2026-01 · First edition · 2026

Disclosure. The case venture, TightBucket, was commissioned by the same owner who commissions the work of Coldcode Si Labs. The author had access to the venture's source code, documentation and operating telemetry. The case is therefore not an independent evaluation. At the time of writing TightBucket had not launched commercially: it had taken no payments and has no customer outcomes to report. Nothing in this paper should be read as evidence of commercial success. The frameworks proposed here are design propositions offered for testing. The venture was built with AI coding agents, and AI tools assisted in drafting this paper; the author is responsible for its content.

Abstract

Generative AI coding tools and generous cloud free tiers have pushed the cash cost of building a working software business towards zero. When building stops being the scarce input, the design problem for a new venture changes. This paper asks what a business model looks like when it is designed *around* hard external constraints rather than despite them, and what new scarcities appear when a small team can build almost anything quickly.

The paper reports a design-science case study of TightBucket, a website "revenue-leak" auditing service built in three days in October 2026 by its owner working with AI coding agents. The owner set explicit constraints. The budget was zero. Checkout had to run entirely on a third-party merchant platform. Email had to go through a free mailbox. Four customer audiences had to be served under one brand. The system had to operate autonomously, with a human only for important decisions. And it had to stay at or below 20% of every free cloud quota while running at full capacity.

Drawing on the venture's code, its documentation, its commit history, a log of 43 defects found in two review passes, and three days of usage telemetry compared against the cloud provider's own analytics, the paper makes four contributions.

1. **Constraint-native business model design.** It conceptualises this as a distinct design mode in which constraints are first-class design inputs, recorded in a *constraint ledger*, rather than environmental conditions to be overcome.
2. **Quota-bounded autonomy.** It specifies this as a governance mechanism for autonomous operations, built from four steps: measure, cross-check, pace and degrade. It reports that the venture's self-meter agreed with provider analytics on AI use and, after the first day, on reads, but under-counted writes by a factor of 2.7 to 11. That gap justifies the rule "govern by the higher of the two numbers".

CONSTRAINT-NATIVE BUSINESS MODEL DESIGN

Working Paper 2026-01

Copyright © 2026 Courage Garikai.

Published by Coldcode Si Labs Library, Harare, Zimbabwe.

First edition, 2026.

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means withou
written permission of the publisher, except in the case of brief quotations in a review or certain other noncommercial uses permit
copyright law. The frameworks, worksheets and checklists in this publication are meant to be used, copied and adapted by the rea

This is a work of analysis and practical guidance. It is not financial, legal, investment or professional advice, and no outcome is pr
results depend on the reader's own market, effort and judgement. Platform behaviour and figures were checked at the time of wri
change.

COLDCODE SI LABS LIBRARY · Harare, Zimbabwe

3. **The autonomy boundary.** It derives this from two properties of an action, its reversibility and its externality to third parties, and explains why the venture automated first-touch email but kept social-network outreach and every reply in human hands.
4. **Verification debt.** It introduces this as the liability created when AI produces systems faster than people can check them. In the case, 40 of the 43 defects were silent rather than visible, about half would have affected people outside the firm, and a fifth were residues of decisions the owner had changed.

The paper closes with seven testable propositions and a Constraint-Native Business Model Canvas for practitioners.

Keywords: business model design; constraints; bootstrapping; freemium; design science; AI-assisted software development; autonomy; platform dependence; serverless computing; verification debt.

---

1. Introduction

For most of the history of the software start-up, the cost of building was the first obstacle a founder faced. Code had to be written by scarce engineers, servers had to be rented before the first customer arrived, and a payment stack had to be integrated, secured and maintained. Business model research developed under those conditions. Its classic questions concerned how a firm creates, delivers and captures value [1][2]. Its practical tools, such as the Business Model Canvas [3], were designed to help teams decide what to build before committing expensive resources to building it.

Two developments have since loosened that first constraint considerably. The first is the arrival of AI systems that write working software. Controlled studies have found large speed gains on bounded programming tasks [4], and field and experimental studies of generative AI in knowledge work report productivity effects that are substantial but uneven [5][6][7]. The second is the maturing of serverless cloud platforms that bundle compute, storage, model inference and scheduling into free allowances. Those allowances are large enough to run a real product at low volume. The serverless model was anticipated as a simplification of cloud programming in which developers stop provisioning servers and pay, if at all, per unit of use [8]. Its free tiers turn that into a zero-cash proposition for a venture that stays small enough.

When the cost of building falls, the binding constraints on a venture move elsewhere. Herbert Simon's observation that "a wealth of information creates a poverty of attention" [9] applies with new force to markets flooded with cheaply built software: the buyer's attention becomes scarcer as supply grows. Other constraints that were always present become decisive. These include the daily sending limits and reputation rules of email providers, the terms under which social platforms allow automation, the policies of payment platforms, the per-day quotas of free infrastructure, and the time and judgement of the one person who is legally and reputationally responsible for everything the system does. None of these is a cost that can be paid down with more engineering. They are limits within which the business model must fit.

This paper starts from that shift and asks two research questions:

> **RQ1.** How can a business model be designed around hard external constraints, so that the constraints become inputs to the design rather than obstacles to it?

> **RQ2.** When AI makes building cheap and fast, which new scarcities and liabilities emerge, and how can a venture govern them?

We address the questions through a design-science case study [10][11] of a single venture, TightBucket, observed in fine detail during its construction and pre-launch operation. TightBucket audits websites for "revenue leaks", meaning problems with speed, pricing clarity, forms, social proof and email deliverability that cost a business customers. It scores each site out of 100 and ranks the fixes by impact. It serves four audiences: e-commerce merchants, software-as-a-service (SaaS) founders, sales directors and agencies. It offers free instant audits and ten free tools, a ladder of paid plans, a done-for-you "Fix Sprint" service, and integrations for three third-party platforms. It runs on a single serverless worker and database on a free cloud plan.

What makes the case useful is not its scale, which is still small, but its *specification*. The owner wrote down the constraints in advance and held the build to them:

- **Budget:** zero cash.
- **Payments:** checkout handled entirely by a third-party merchant platform.
- **Email:** a free personal mailbox, with no scripting add-ons.
- **Audiences:** four, each with its own product and free tools, under one roof.
- **Autonomy:** autonomous operation, with a human reserved for important decisions.
- **Social networks:** no automation of social-network outreach.
- **Audience at launch:** none to start from.
- **Growth model:** customers should come looking for the business, like "bees to the flower", rather than only being chased.
- **Quotas:** at most 20% of each free cloud quota and of the free monthly allowance of continuous-integration minutes, "so that I never need to worry about quotas", while running at full capacity.

A specification this explicit allows each design decision to be traced back to the constraint that produced it. The venture's source repository, its operating documents, a review log of every defect found after the first release, and its usage telemetry together give an unusually complete record of how a constraint-bound design came together.

The paper makes four contributions.

First, it proposes *constraint-native business model design* as a distinct mode of design. Prior work treats constraints in three main ways:

- as resource scarcity that entrepreneurs overcome through bricolage or bootstrapping [12][13];
- as the means-driven starting point of effectuation [14];
- as a stimulus to creativity whose effects depend on its type and degree [15].

The constraint-native view goes further, and treats external constraints as *specifications* to be recorded, quantified and given budgets in a constraint ledger before any value proposition is chosen. The case shows how the ledger shaped the activity system [16], the pricing architecture, the distribution design and the operating model.

Second, it specifies *quota-bounded autonomy*, a governance mechanism that lets an autonomous system run "flat out" without the risk of exhausting a hard external limit. The mechanism has four steps:

1. **Measure** every unit of resource use where it is incurred.
2. **Cross-check** that meter against the provider's own analytics.
3. **Pace** background work evenly across the quota window.
4. **Degrade gracefully**, rather than fail, when a budget is reached.

The case data show why the cross-check is necessary. Over three days the venture's own meter matched the provider exactly on AI consumption and, after the first day, to within 2% on database reads. But it recorded between 2.7 and 11 times fewer database writes than the provider counted. The difference came from activity outside the application's instrumentation: schema migrations, deployments and administrative queries.

Third, it develops the concept of an *autonomy boundary*: the line between what the system may do alone and what it may only prepare for a human. The boundary was drawn using two properties of each action, how reversible it is and how much it affects third parties. This explains choices that otherwise look inconsistent. The system sends first-touch emails automatically but never posts to a professional network on the owner's behalf. It audits any public website without asking, but never publishes a negative audit of a named company. It prepares every reply but sends none.

Fourth, it introduces *verification debt*: the liability created when systems are produced faster than they can be checked. After its first complete release the venture went through two line-by-line reviews that documented 43 defects. Among them:

- a deployment check that blocked all releases;
- a login lockout that any stranger could trigger against the owner;
- a scheduled email that would have reached real people signed with a literal placeholder name.

Classified by type, the findings suggest that the dominant risk of AI-built ventures lies less in code that fails to run than in code that runs and does the wrong thing, plausibly and quietly. This is consistent with security research on AI-assisted programming [17][18].

The remainder of the paper proceeds as follows.

- **Section 2** reviews the relevant literatures on business model design, resource-constrained entrepreneurship, free and freemium pricing, signalling, platforms, automation and AI-assisted work.
- **Section 3** describes the design-science method and the data.
- **Section 4** describes the case.
- **Section 5** reports the findings as a set of design principles, each traced to evidence.
- **Section 6** proposes frameworks and testable propositions.
- **Section 7** discusses contributions, limitations and further research.
- **Section 8** concludes.