NOUMENA DIGITAL

Why Noumena Tech beats other software frameworks

Benchmarking Noumena Tech

Overview

September 2025

Overview

1. **New app development with NPL**
2. IT modernization with NPL

Benchmarking NPL against other software frameworks

Methodology

- Benchmark against other modern software development approaches
- Benchmark generated with following steps:
  - Specify one set of business requirements (generated by LLMs such as ChatGPT)
  - Require pre-prod readiness with deployment to Docker
  - Require non-functional requirements (e.g., authorization, state machine, audit log, …)
- Use **Claude Code** to generate full backend with minimal amount of prompts; minimal corrections only (e.g., to prevent mocking services, ensure functional equivalence, ensuring 100% of NFR scope covered)
- High-level code review by (human) senior software developer
- Compare LOC (excluding test code, comments, etc.)
- Perform AI-based code complexity analysis
- Perform AI-based static vulnerability analysis

*Excluding test code for like-for-like comparison
Source: https://github.com/jk-nd/benchmarking-npl; Claude Code (Sonnet)

Case study: expense approval application

Sample application requiring non-trivial functional authorization and validation steps

- An employee submits a business expense with details like `amount`, `category`, `date`, `vendor`, `currency`, `department`, and `receipts`.
- The expense moves through states: `draft` → `submitted` → `approved` → `paid`, with possible detours to `rejected` or `compliance_hold`.
- Different roles perform actions at different stages:
    a. `Employee`: create, edit, attach receipts, submit, withdraw
    b. `Manager`: approve/reject within data-driven limits (reporting lines, limits, budget)
    c. `Finance`: process payment after validations (tax status, terms, holidays, exchange rates)
    d. `Compliance`: audit or flag suspicious activity (can place on hold)
    e. `VP/CFO`: high-value/exceptions override

*Excluding test code for like-for-like comparison
Source: https://github.com/jk-nd/benchmarking-npl; Claude Code (Sonnet)

NPL beats other modern frameworks

KPIs for new applications

Based on simple expense approval application
(unoptimised, generated by Claude Code)

	NPL	Ruby on Rails	Node.js + Express	Django + DRF
Total LOC*	423	1,514	3,239	2,242
Auth LOC	~50	~400	~800	~330
Files	1	20	21	17
Dependencies	0	12+	20+	15+
Code reduction vs. NPL	–	**3.6x**	**7.7x**	**5.3x**

NPL performance KPIs

- **3.6-7.7x code reduction**
- **Zero-configuration framework**, allowing you to start building immediately from a single file
- **Automatically generates the REST API**, database schema, and documentation, eliminating thousands of lines of manual, error-prone boilerplate code

*Excluding test code for like-for-like comparison
Source: https://github.com/jk-nd/benchmarking-npl; Claude Code (Sonnet)

NPL significantly more secure than other implementations

Results of static vulnerability analyses

Security comparison NPL vs. other frameworks for simple expense approval app
(unoptimised, generated by Claude Code)

Vulnerability Category	NPL	Rails	Node.js	Django	What it is
Broken Access Control (A01)	0	4	4	5	Unauthorized data/action access
Injection (A03)	0	1	2	2	Malicious code execution via untrusted data
Insecure Design (A04)	0	0	1	1	Architectural weakness
Logic Errors	0	2	3	2	Exploitable application bugs
State Manipulation	0	2	4	2	Illegally modifying system state
Mass Assignment	0	2	1	3	Assigning values to unexposed fields

*Excluding test code for like-for-like comparison
Source: https://github.com/jk-nd/benchmarking-npl; Claude Code (Sonnet)

How NPL prevents vulnerabilities

- **Compiler-Enforced Security:** enforces security rules at compile time, eliminating entire vulnerability classes, such as broken access control and injection, before code can be deployed
- **Unified Architecture:** all business logic, authorization, and state management into a single Protocol, preventing security inconsistencies that arise from scattered logic
- **Automatic Generation:** non-functional requirements like APIs and audit trails autogenerated, eliminating human errors and vulnerabilities from manual implementation

NPL leads to significant code complexity reduction

Code complexity analysis

Complexity Measure	NPL	Rails	Node.js	Django	NPL Advantage
Cyclomatic Complexity (avg)	1.2	3.8	4.2	3.5	~3x lower
Decision Points	~15	~85	~120	~95	5.7-8x reduction
Coordination Points	0	~25	~40	~30	Eliminated
Manual Authorization Checks	0	~30	~45	~35	Eliminated
Business Validation Rules	Unified	Scattered (5 files)	Scattered (8 files)	Scattered (6 files)	Single source

- **Fewer branches:** NPL encodes authorization/state as single permission guards, avoiding the multi-file conditional explosion common in policies/ middleware/ models/ controllers
- **Less coordination/duplication:** One protocol definition removes cross-layer synchronization and repeated validation code found across controllers, services, and serializers.
- **Minimal boilerplate:** API, state transitions, and audit are generated, eliminating hundreds of lines of "plumbing" code.
- **Lower cognitive load:** A single source of truth with compile-time guarantees replaces multi-layer reasoning and reduces mental context switching.

*Excluding test code for like-for-like comparison
Source: https://github.com/jk-nd/benchmarking-npl; Claude Code (Sonnet)

NPL eliminated large parts of architectural overhead

Architectural overhead reduction

Infrastructure Component	NPL	Rails, Node.js, Django	NPL Advantage
Controllers/Serializers	Auto-generated	Manual implementation	~500-800 LOC eliminated (20-35%)**
OpenAPI Specs	Auto-generated	Manual documentation	~200-300 LOC eliminated (10-15%)**
TypeScript Interfaces	Auto-generated	Manual type definitions	~150-200 LOC eliminated (6-10%)**
GraphQL Schemas	Auto-generated	Manual schema definition	~100-200 LOC eliminated (5-10%)**

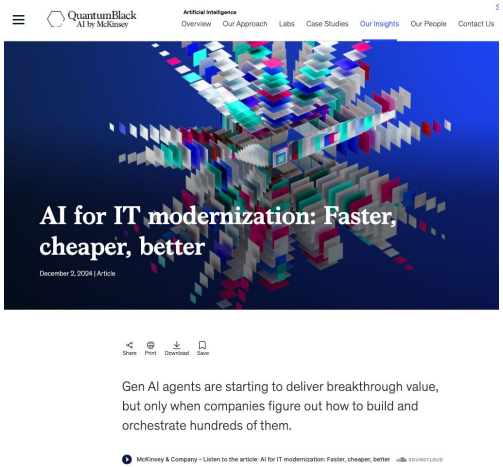
- **Largest savings are in "plumbing" layers**: Auto-generated controllers/serializers alone remove ~20–35% of average LOC, dwarfing any single optimization in traditional stacks
- **Compounding reductions across multiple concerns**: Eliminating OpenAPI, migrations, types, schemas, and error-handling together cuts another ~30–45%—shrinking coordination points, tests, and review surface
- **Fewer lines, fewer bugs, faster changes**: Lower LOC directly reduces maintenance overhead and authorization drift risk, while keeping docs and APIs automatically in sync

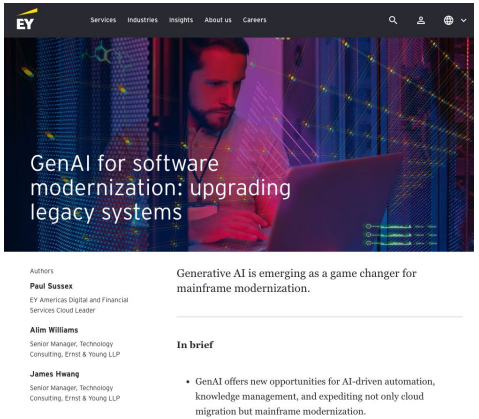
*Excluding test code for like-for-like comparison
** Basis is average LOC of c. 2,330 across Rails, Node.js and Django
Source: https://github.com/jk-nd/benchmarking-npl; Claude Code (Sonnet)

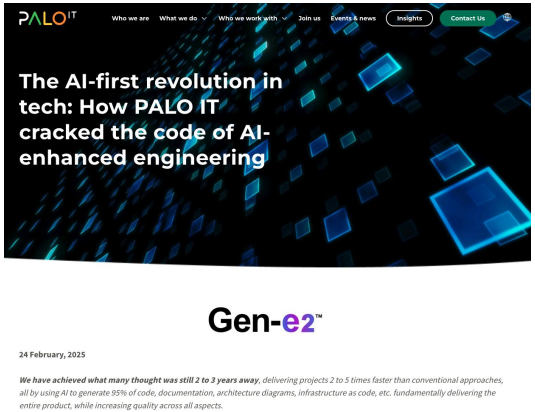
Overview

1. New app development with NPL
2. **IT modernization with NPL**

AI is rewriting the game, but still early days







"40-50% acceleration in tech modernization timelines and a 40% reduction in costs derived from technology debt"

McKinsey (2025)

"GenAI is a game changer for mainframe modernization," showing 35-60% efficiency gains

EY (2025)

"Legacy systems modernization and re-platforming is seeing a 3-5 times acceleration"

Palo IT (2025)

But point AI at the wrong stack, and the pain will come back

Why AI-coding requires guardrails

LLMs recreates boilerplate and business logic spaghetti

- LLMs happily create lots of boilerplate
- Business logic continues to be scattered across multiple files
- Completeness and validation as challenging as before

Security remains stochastic

- Authorization remains scattered across annotations, filters, and environment config
- Enforcement relies on developer discipline and tests
- Broken asset control defects remain invisible to LLMs

Non-functionals are just an afterthought

- External effects get mixed into business code; failures cause inconsistent endpoints
- Atomicity and audit aren't guaranteed
- Logging/observability are bolted on via libraries with diverging behavior across services.

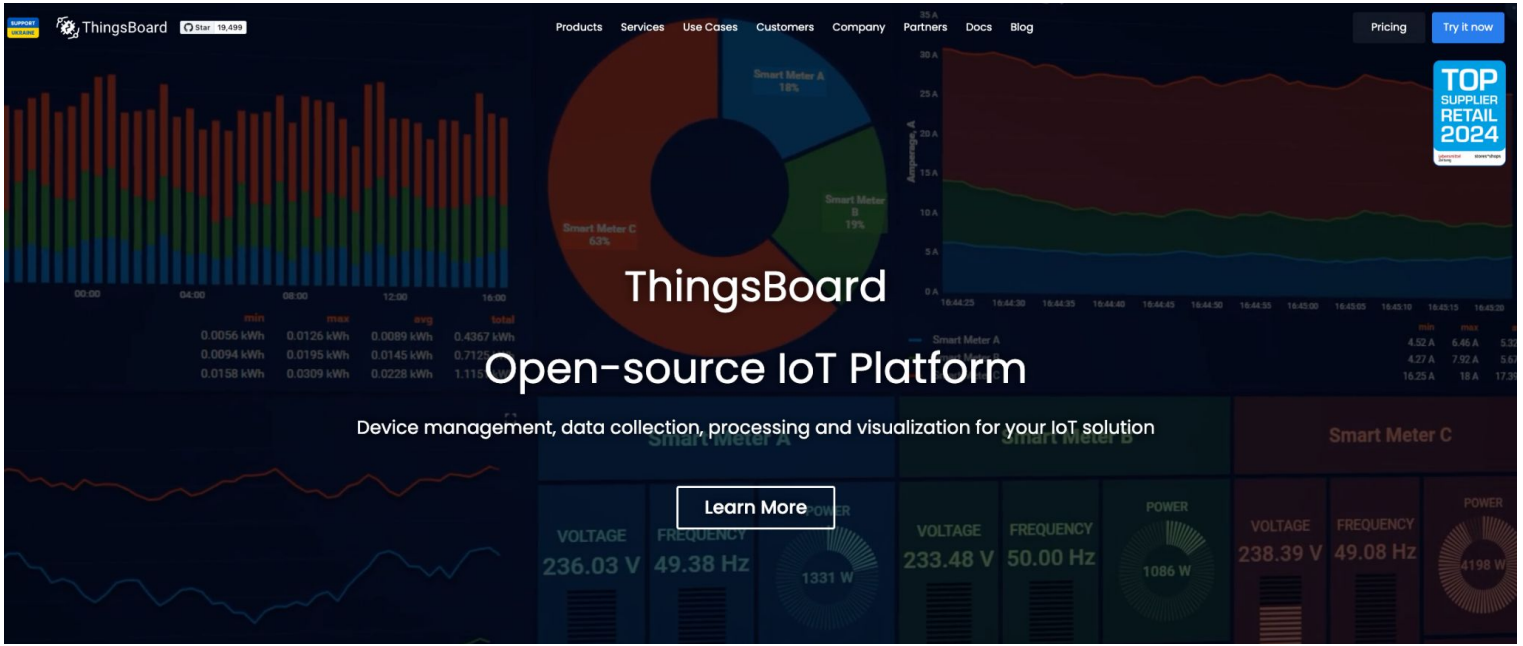
Source: Noumena Digital (2025) NPL Technical Whitepaper.

Why AI-driven modernization is best done with NOUMENA technology

Large-language models are great at pattern matching but terrible at enforcing rules. NPL flips that: it gives AI the guardrails

1. **AI gets guardrails.** NPL syntax provides compile-time guardrails that prevent AI invalid authorization logic and process states
2. **Reverse-engineer to workflow graphs.** NPL semantics align 1-to-1 with AI-discovered actions & data flows
3. **Prompt-to-code generation.** AI directly generates productive, integrable stateful business services from business requirements
4. **Streamlined testing.** Focusing on business outcomes rather than implementation details enables AI to create simpler, more durable tests
5. **Strangler-pattern slicing.** Auto-generated adapters let AI wire new NPL slices beside legacy modules in hours
6. **Self-documenting business logic.** NPL serves as living specifications that auto-generate APIs, tests, and documentation

Thingsboard is one of the leading IoT platforms with over 150k lines of Java/Spring code



Source: https://github.com/thingsboard/thingsboard

Why NOUMENA beats traditional software tech when modernizing

KPIs based on modernization case study

NPL impact	Description	Compared to Java/Spring
Write far less code	• Core logic LOC	↓ **~60-70%**
Reduce complexity at the source	• Cyclomatic complexity • Decision points • Coordination overhead across multi-layered architecture	↓ **~60-70%** ↓ **~30-40%** **Eliminated**
Security is compiled-in	• Manual authorization checks • Business validation logic • Error-handling boilerplate eliminated by the NPL runtime	**Eliminated** ↓ **~85–100%** **Eliminated**
APIs/read models autogenerate	• OpenAPI specs auto-generated • Controllers/serializers • GraphQL schemas • TypeScript (or any other client) interfaces	**Code gen from NPL** **Eliminated** **LLM gen from NPL** **Code gen from NPL**
Tests & performance validated	• Test coverage: on protocol state transitions + permission • Test scenarios • Real service integration testing • Performance testing	**100%** **LLM generated** **LLM generated** **LLM generated**
Cognitive load eliminated	• Validation coordination across layers • Authorization logic to remember (embedded in permission syntax) • Error handling patterns to maintain • Single source of truth requiring layer synchronization	**Eliminated** **Eliminated** **Eliminated** **Eliminated**

Source: https://github.com/thingsboard/thingsboard; Claude-4-sonnet

Approach realized

- Strangler-style rollout with hybrid backend
- Real-time sync between NPL and legacy
- Transparent frontend interceptors routing to NPL or legacy

How NOUMENA compares to other software technologies for modernization

Technology	Architecture Pattern	LOC Range / 1000 FP	Notes
Java/Spring Boot	Layered	45,000-65,000	Enterprise standard, verbose syntax
.NET Core/Framework	MVC/API Pattern	42,000-58,000	Microsoft ecosystem, moderate verbosity
Node.js/Express	REST API/Microservices	40,000-55,000	Flexible, less boilerplate, heavy use of libraries
Python/Django	MVT (Model-View-Template)	25,000-35,000	Clean syntax, batteries included
Ruby on Rails	Convention over Configuration	22,000-32,000	High productivity, "magic" conventions
NPL	**Protocol-Driven Development**	**15,000-20,000**	**Rails/Django-level productivity PLUS:**

Key capabilities packaged together uniquely in the NPL approach

- Compile-time security validation
- State machine validation
- Built-in authorization
- Automatic API generation (REST + GraphQL)
- Type-safe contracts with compile-time verification
- Living specification
- Strangler pattern support through protocol connectors
- Generation of test coverage from protocols
- Built-in business process orchestration

Raiffeisen Austria x NOUMENA Stable Coin

Initiative

NOUMENA DIGITAL

NOUMENA DIGITAL AG
Bahnhofstrasse 22
CH-6300 Zug
info@noumenadigital.com