Dmitry Aleksandrovich Karataev

Backend Software Developer (C#) · Moscow, Kolomenskaya metro

Contact: +7 (923) 047-13-55 (Telegram preferred) · dkarataev1990@gmail.com · Telegram @Krawler · GitHub · Website

Citizenship: Russia · **Experience in software development:** 8+ years

Format: remote work · full-time / part-time / project-based · **Remote locations:** Canada, Minsk, Norway, USA · business travel not considered

Possible employment arrangements: employment contract, self-employment, contract work

Summary

Backend developer in **C# / .NET** (8+ years): microservices, REST API, SOA, GraphQL, engineering and document-management systems. Full SDLC experience, refactoring and moving monoliths toward service architecture, introducing DI, unit tests, Agile practices (DoR/DoD/AC), documentation, and team knowledge bases.

Alongside my main job I build **infrastructure for AI agents**: a series of open-source **MCP servers** (Model Context Protocol) that give AI assistants access to the compiler, debugger, build system, and multimodal inputs. Author of **agent-first-learn** — a practical guide to productive work with AI.

Beyond coding: process setup, requirements work, technical expertise, non-trivial debugging (memory dumps, thread analysis, Reflection).

AI-assisted engineering: 1.5+ years of daily practice (Cursor IDE, MCP servers of my own design).

Work experience

NOVA Energies LLC (Moscow) — Technical Expert & Support Engineer

September 2023 — present · remote

Energy sector · engineering data management systems; by task shape — **comparable to Harvester and a corporate EDW web portal** (collect, process, and publish data,

EPC digital-twin context). The **team lead** sets priorities and direction; below is **my contribution** and two cases (**Harvester**, **EDW web portal**).

Responsibilities

1. Development and maintenance of services for **collecting, processing, and publishing** engineering data.
2. Development and maintenance of solutions for **publishing engineering data** in the EPC digital-twin context: documents, tags, attributes, relations, and related entities.
3. Architecture review and improvement proposals; feature work and production support.
4. Introducing **dependency injection**, growing **unit testing**; documentation (modules, onboarding, engineering knowledge base).
5. **Agile** practices: DoR, DoD, acceptance criteria; helping build a predictable delivery process.

My contribution — Harvester system (case, STAR)

Situation: a legacy system (in the UI and on **IIS** the product is named **Harvester**): modules written in different styles; evolutionary growth; requirements often conflicted and changed mid-flight. We needed stability and performance on **large volumes** (**1M** objects, relations, and documents with attributes). Switching tasks/modules raised cognitive load; documentation and code comments were missing or stale.

Goal: speed up new features and changes, reduce mental load when picking up others' work, improve transparency for users, and **stability** of the prepare → validate → publish path into the corporate web portal.

Actions:

- Introduced **.editorconfig** (based on **dotnet/runtime**): consistent formatting, less "visual noise" and noise warnings when moving between modules; brought existing code toward the rules.
- Ran meetings with the team and stakeholders to justify the changes.
- In **2 days** proposed and rolled out **dependency injection** with **Microsoft.Extensions.DependencyInjection** (with the team lead).
- Created and maintained a **knowledge base** and **user documentation**: how the system works, support ticket templates, etc.
- Reorganized the shared library (repositories, domain objects) into a **feature-based** structure.
- **Architecture:** shared **core library** on **.NET 9** (**SQL Server**, **Dapper** / **RepoDb** (incl. bulk), **CsvHelper**, **Serilog**, **feature flags**) and three service areas — **file/data preparation service**, **validation service** (incl. tag naming rules and **RDL**), **background publish service** (timers for consistency checks and export, staging, **CSV** export to a web-portal folder per config). On top — **web**: **ASP.NET Core** + **Angular** (SPA for reserved-tag statuses, upload flows, related scenarios). In parallel, data access via **Linq2db** (context in DI, entity names aligned with DB schema) **alongside** existing **Dapper** on some paths. For

compatibility with the **AVEVA** engineering model, a layer of **recreated** API assemblies matches model contracts — without vendor binaries where we could ship our own code.
- **Portal integration:** publish flows respect a "in portal" tag (validation and comments don't add noise if there is no portal card); exports and CSV names converged to a **single naming template**; fixed **concurrent** use of the shared publish pipeline (**locks** in the worker, "ready for test" scenarios) and timer config bugs (interval in **seconds**, not minutes), **CSV encoding/headers** — from incidents and code review; larger export and publish stabilization work happened in dedicated branches.
- **Operations:** PowerShell scripts to refresh **RDL** for **IIS** deployments (separate sites for web and services) alongside support tasks.

Result:

- Unified style and predictable structure reduced mental load when taking over tasks or switching modules.
- New features landed faster: easier to add layers; with **feature-based** domain modeling, extensions land where people expect.
- A path to **unit tests** (initially none): easier to substitute implementations via interfaces; separate **test projects** for core layers, publish, validation, web, acceptance, preparation.
- The **prepare → validate → publish → portal** chain is conceptually similar to the **import-substitution web portal** (second case below): same classes of problems (tags, documents, attributes), different stack and code ownership.

My contribution — EDW web portal (case, STAR)

Situation: after several Western vendors left the market, support ended for products in the **AVEVA** line (including **NET Portal**, **AVEVA ISM**). To keep working with engineering data we needed a **replacement** and less dependence on unavailable support and updates.

Goal: per agreed plan — **MVP of a corporate web portal**: view documents, tags, and **relations**, search and file handling (incl. preview) so users could continue **without the original vendor**.

Actions:

- Helped **implement the MVP** on **Blazor WebAssembly**, **GraphQL**, **EF Core**, **.NET 8+** (production on current **.NET 9** on server and client).
- **Architecture in short:** thin **Blazor WASM** client (component library, API via generated **GraphQL** client) and **ASP.NET Core** server with a single **GraphQL** endpoint: projections, filtering, sorting, **cursor pagination** without exploding REST resources per screen. Behind the server — **SQL Server** (portal DB), **DbContext pool**, **NoTracking** for read-heavy paths; separate controllers for files, export, and long operations.
- **Module split:** the online stack (**API + Blazor**) **does not reference** the shared domain library — only network contracts. Shared **EF** model (DB context, entities)

lives in a **separate library** used by **background jobs** (sync, import, file monitoring). The server has **its own** entity copy: deliberate split between "API/UI" and "integrations", with known **schema drift** risk on changes (managed via review and migrations).
- **Integration with corporate engineering data:** three pipelines — **incremental** metadata pull for documents, **full** sync of revision/version chains, then **file attachment** from disk (staging / fallback) so a version gets a **primary file** for portal preview. Helped shape data ordering (document → revision → version → file), **deduplication** in inherited data (views, in-code dedupe), and documentation.
- **Why GraphQL:** one typed contract for the front, per-screen queries, fewer hand-built CRUD endpoints. Downsides are known: heavy queries possible (schema design, pagination, cost limits), higher onboarding than classic REST.
- **Security and perimeter:** beyond AD/domain model, **network isolation** matters — portal in the **corporate perimeter** and over **VPN**; a deliberate addition to app-level measures (no "public internet GraphQL" by default).

Result: users keep running projects, **uploading and updating** engineering information despite **end of support** for prior products and external software constraints; **integration → portal DB → UI** covers EPC-style viewing and update scenarios.

Team outcomes (beyond the cases above; under the team lead)

Reduced vendor lock-in risk and higher modularity; more testing and **CI/CD**; better performance and resource use on some paths after profiling and targeted fixes.

Wissance — Co-founder / Senior Backend Developer (.NET)

February 2022 — September 2023 (1 year 8 months)

- Cross-functional work: analysis, documentation, development, estimation — largely across the **SDLC**.

Electron-Service LLC — software technician

August 2020 — February 2022 (1 year 7 months)

- Stack: **.NET Core**, **Entity Framework Core**, **REST API**, GitLab, TFS.
- Web services and background services; **SOA**; **ASP.NET** controllers.
- Debugging and changes to **MS SQL Server** stored procedures; SQL scripts for testing.
- Project in maintenance and evolution (~6 years old when I joined).

Gazprom Neft — Digital Solutions — Lead .NET Developer

September 2017 — March 2020 (2 years 7 months)

Domain: electronic document management (EDM) on **IBM Document Manager**; the solution covered the **full cycle**: from **scanning primary documents** (accounting

and related) — custom software around **TWAIN** and **production scanners** — through **SAP integration** and control of **posting documents** in the accounting system.

Responsibilities

1. Development and maintenance of EDM modules: **WCF** services, integrations, and over time **extracting microservices** from monoliths.
2. Refactoring toward **SOLID**; multithreading (scanning, uploads, background flows).
3. Use of **.NET Reflection** for non-trivial debugging and architectural workarounds (including scanner driver interaction and **UI** threads).

Architecture (context)

- Server-side ECM on **Java**; **IKVM.NET** bridged JVM ↔ CLR for platform classes from **.NET**.
- Two storage modes: **high-level client scenario** (easier but needs **ECM client running in the browser**) and **low-level API to content** (no browser, harder, more control).
- The platform had its own form and **plugin** model for user actions; we used it heavily.
- The system was **legacy** (years of evolution before I joined); documentation existed but was **sparse** and fully **manual**; version control was **TFS**, work through **Visual Studio** / **VSTS**.

Stack: .NET Framework 2.0–4.8, **MongoDB**, **Quartz.NET**. **MongoDB** and **Quartz** powered a **document processing pipeline**: jobs in collections, **background services** moved them between collections along the **business process**.

Case: assigning a document package "curator" (STAR)

Situation: each document package had a **responsible curator** for processing deadlines; we needed to **compute** that person from the **org chart**: for a line employee — **their direct manager**; if missing — **walk up** the hierarchy.

Goal: correct curator from the **reporting tree** without duplicating logic for each new client UI.

Actions:

- Implemented an **ECM plugin**: walked corporate hierarchy **XML** with **System.Xml**, classic **tree traversal**.
- When a **WinForms** desktop client appeared with **System.Reactive** and document-package flows, the same logic had to be embedded. With the **team lead** we **reused** the existing plugin: it was essentially a **console app** with a clear contract — **call it correctly** from the new UI.

Result: the team **saved time** reimplementing rules; curator logic stayed in one place.

Case: custom package viewer and unit tests (STAR)

Situation: the **platform UX** increasingly **limited** scenarios; we needed document packages to hold not only **files** (including **PDF**) but **nested packages** — the **high-level ECM API** made that awkward or impossible.

Goal: move off **high-level API** and build a **custom desktop package viewer** on **low-level storage API** (same layer as above), with acceptable **UX** for the new composition rules.

Actions:

- Designed and helped implement the client: package tree and nesting on **low-level API**, without stock UI limits.
- **Unit testing:** initially **no automation**, checks were **manual**. Experiments with **Microsoft Fakes** produced stubs that saw **COM interfaces** from the vendor library; that enabled an **abstraction layer** and a **large test suite** that **cut regressions** and manual effort.
- The same abstraction later enabled **MVVM**, pairing well with **System.Reactive** in the desktop client.

Result: manageable UI architecture and testability instead of "manual only"; foundation to grow the client without COM/API details in the view layer.

Case: hang, memory dump, and Reflection (STAR)

Situation: scanning used **TWAIN.NET** (wrapper over **TWAIN**) from a **background thread**. The app **hung** "nowhere": silent logs, **no exceptions**, hard to reproduce locally; **highest-priority incident**; investigation lasted **several days**.

Goal: find the **deadlock** and remove it without months of guessing.

Actions: suggested a **memory dump** of the hung process and **thread stack** analysis. The dump showed: through a dependency chain the app **subscribed to Windows user settings change notifications**; per **WinAPI** docs those must run on the **UI thread**, but the chain led handling **off-thread** — on the event everything **froze**. Found the subscription via **.NET Reflection** and **unsubscribed in a few lines**.

Result: hang fixed. **Repro** was subtle: enable **desktop wallpaper slideshow** — on **image change** the OS sends the notification and the bug triggered reliably.

Outcomes (beyond cases)

Refactoring monoliths toward **microservices** where it made sense; **ECM (Java) ↔ .NET** integration, background pipelines on **MongoDB** / **Quartz** and accounting (**SAP**); **desktop package viewer** with **MVVM**, **Rx.NET**, **unit tests** via **Microsoft Fakes** and abstraction over **COM** storage API; **memory-dump debugging**, **multithreading**, **WinAPI**/UI thread, and **Reflection** to find hidden subscriptions in third-party code.

Education

Omsk State University named after F. M. Dostoevsky, Omsk.

Faculty of Mathematics — until 2019

Applied mathematics and computer science · incomplete higher education

Faculty of Physics — until 2017

Applied mathematics and physics · incomplete higher education

Faculty of Computer Science — until 2013

Computers, complexes, systems and networks · incomplete higher education

Skills

Languages and platforms: C# (8+ years), .NET Framework 2.0–4.8, .NET Core, .NET 9, ASP.NET Core, ASP.NET MVC, Blazor WebAssembly, Entity Framework / EF Core, ADO.NET, LINQ, SQL

Architecture and practices: microservices, SOA, REST API, GraphQL (HotChocolate), WCF, SOLID, design patterns, DDD (elements), feature-based architecture, dependency injection, unit / integration testing, Agile (DoR / DoD / AC), BPMN

Data and integrations: SQL Server, MongoDB, Dapper, RepoDb, Linq2db, RabbitMQ, CSV / XML / XSD pipelines, SAP integrations

AI and agent tooling: MCP server development (Model Context Protocol), Roslyn API, Debug Adapter Protocol (DAP), Whisper integration, prompt engineering, structured context management, AI safety guardrails

Debugging and systems: memory dump analysis, thread debugging, .NET Reflection, COM interop, TWAIN, background processing (Quartz.NET), reactive programming (Rx.NET / System.Reactive), MVVM

Tools: Git, GitHub, GitLab, Docker (basics), CI/CD, PowerShell, Cursor IDE, VS Code, Visual Studio

Languages: Russian — native; **English — B2** (upper intermediate)

AI-assisted engineering environment

I have built a working environment where AI is used as an **accelerator for engineering work**, while outcomes stay **verifiable and reproducible**. The focus is not "code generation for its own sake" but **process and infrastructure**: context, knowledge, guardrails, quality checks, and reuse of solutions.

What was built

- **Knowledge base and "environment memory":** one place for decisions, conventions, postmortems, patterns, examples, checklists, and known pitfalls. This cuts repeated questions and speeds up returning to context after context switches or breaks.

- **Context standards:** rules for what counts as a fact, how to record decisions and rationale (decision log), how to dose context for stable output without noise.
- **Guardrails (safety and quality):** anonymization and no sensitive data; explicit "hypothesis / verified" split; mandatory validation via build/tests/diagnostics and manual checks where needed.
- **SDLC integration:** AI inside the normal development loop — task wording, PR descriptions, test plans, technical notes, build/test failure analysis, regressions.
- **Reuse:** wording templates, document structures, and reusable building blocks so new tasks are faster and consistent.

Conclusions from joint practice

- AI is most effective for **routine and structure** (draft docs, triage, templates, repetitive edits), not for "magic" fixes to complex business logic without context.
- Result quality is driven not by the model but by **verification discipline**: tests/build/diagnostics + clear done criteria give predictability.
- Long term, what wins is not "one lucky prompt" but the **environment**: knowledge base + standards + guardrails → less cognitive load, fewer repeat mistakes, faster delivery.

What this demonstrates

Knowledge management, engineering discipline, systems thinking, development-process automation, requirements/DoD formalization, quality and safety when using AI.

KB memory: L0–L3 layers and domains

A short version of the **layered** model and load contract — the same idea as the KB onboarding overview (no full load), **without** naming files or paths.

Mermaid diagrams render in **HTML** and in **PDF printed from a browser**; **DOCX** usually does not execute JS — the bullet list mirrors the diagrams.

- **L0 — always on:** integrity, epistemics, core when barriers fail, principled clarity; **not** tied to the current task.
- **L1 — operational memory by scope slices: status → playbook → matrix → kb**; compact artifacts first, heavy `kb-*` only on explicit request.
- **L2 — archive and evidence:** revisions, rule batches, evidence docs — when facts are missing or history is needed (no "load everything at session start").
- **L3 — semantics:** routing by **context facets**; moving between **worlds** (domains) only through **explicit boundaries**.
- **Domains and glue:** Git, PR review, HCI, Developer Experience, IT, Knowledge Engineering, psychology, aviation, reading, integrity under pressure, etc. — tied together by a **single index** and the same load contract.

Open-source projects (MCP)

Public GitHub repositories — **Model Context Protocol** servers for Cursor and compatible environments: they give the AI assistant access to tools (build, debug, C# semantics, etc.), not just file text.

Card schema (Purpose / Motivation / Vision and agent hints): open-projects/README.md.

- **RoslynMcp** — MCP over Roslyn: diagnostics, quick fixes, go-to-definition, find usages, rename, solution structure. **Why:** so the assistant relies on C# semantics and suggests edits consistent with the compiler.
  - **Purpose:** give the model the same code-analysis layer as the IDE (Roslyn), not only file text.
  - **Motivation:** fewer edits that break the build and less API guessing without compiler checks.
  - **Vision:** main C# workflow in Cursor where MCP is connected: diagnostics → code actions → apply, symbol navigation.
  - **Scope:** solution/project, open documents, Roslyn operations (within server capabilities).
  - **Non-goals:** replacing a full IDE, running apps, debugging (other MCPs).
  - **For the agent:** on errors/warnings use `roslyn_get_diagnostics` and code actions, not only `dotnet build`; tool positions are 1-based; do not treat text grep as a substitute for semantics.
- **dotnet-debug-mcp** — .NET debugging via DAP (netcoredbg): breakpoints, launch under debugger, stack, variables, stepping, continue/stop. **Why:** reproducible debugging from chat without manually copying stack traces.
  - **Purpose:** one DAP debugging session to a target .NET process or launch under the debugger.
  - **Motivation:** agent and human see the same facts (stack, variables) when IDE integration is set up; otherwise at least a predictable loop from chat.
  - **Vision:** the same "single layer" of state as the IDE UI (breakpoints, stops); in CascadeIDE see `Financial/software/open/cascade-ide/docs/debug-human-agent-parity-v1.md`.
  - **Scope:** netcoredbg, breakpoints from JSON workspace, attach/launch per host agreement.
  - **Non-goals:** debugger UI inside another app (unless designed that way); replacing the system debugger in all scenarios.
  - **For the agent:** before rebuilding target DLL end the session (`debug_stop`), else PDB is locked; attach vs launch; do not kill netcoredbg externally. Parameters and limits — repo README.
- **dotnet-build-test-mcp** — `dotnet build` / `dotnet test` with a queue and structured error/test output. **Why:** so the AI sees the essence of build and test failures without log walls.

  - **Purpose:** standardized build and test invocation with a queue (single-flight) and compact output for the model.
  - **Motivation:** the agent gets error lists and failed tests without manual console parsing.
  - **Vision:** reliable queue on heavy repos; async jobs and chunked log reads when needed.
  - **Scope:** `dotnet build`, `dotnet test`, path to solution/folder.
  - **Non-goals:** C# semantics and refactorings (Roslyn MCP); stepping debug (dotnet-debug-mcp).
  - **For the agent:** pass `solution_path`; large `raw_output` only when explicitly needed; do not duplicate compiler checks where Roslyn already applies.

- **webcam-mcp** — webcam and audio capture, analysis, and transcription (Whisper). **Why:** multimodal scenarios (voice/video) in the assistant pipeline.
  - **Purpose:** local media capture and derived artifacts (frames, wav, mp4 if needed) in the workspace.
  - **Motivation:** "show screen/self", voice input, follow-up analysis without manual file upload to chat.
  - **Vision:** predictable save paths, optional pairing with a separate analysis MCP for OCR/reports.
  - **Scope:** devices by index, duration/FPS parameters, formats per repo README.
  - **Non-goals:** cloud processing by default; replacing OS privacy settings.
  - **For the agent:** almost always needs `workspace_path`; Whisper model from env or parameter; check README for split capture/analysis if the repo is split.

Licenses and details are in each repository's README.

Additional information

Medical context (transparent for employers): Group I disability (cerebral palsy; mobility with elbow crutches). **Does not affect** the ability to perform developer duties — I state this upfront to avoid surprises at offer stage.

Beyond development I can help with:

- automating documentation creation;
- shaping processes that are painless for participants;
- sensible Agile adoption, working through the Agile manifesto, and cross-functional collaboration.