CASE STUDY / AI INTEGRATION

XDATASYS - локальная AI-инфраструктура

К чему я пришёл как AI Integrator / Local LLM Systems Engineer

От локального чат-интерфейса к production-oriented coding-agent среде: локальный inference, реальный доступ к репозиторию, sandbox execution, безопасные мутации, test-fix-retest цикл, измеримый performance tuning и динамическое разделение одной GPU между несколькими моделями.

140.8K стабильный context основной coding-модели	**~30-33 tok/s** реальные coding-turns после tuning	**~275.6 tok/s** cold prefill на 8 117 токенах
97-98% prefix reuse в representative final passes	**1 x GTX 1070** 8 GB VRAM, multi-model GPU sharing	**PASS** PHP + Python autonomous mutation / recovery flows

Ключевой результат. Я собрал не просто локально запущенную LLM, а инженерную среду, в которой модель умеет изучать реальный проект, вызывать инструменты, вносить ограниченные изменения, запускать проверки и завершать работу только после верификации. Основной акцент - интеграция, orchestration, inference engineering, agent tooling, безопасность и воспроизводимые измерения.

МОЯ РОЛЬ

AI Integrator / Local LLM Systems Engineer

Позиционирование отражает фактическую работу точнее, чем «ML Engineer»: здесь не обучение foundation-модели с нуля, а проектирование и эксплуатация полного локального AI runtime - от GPU/CPU placement и llama.cpp до tool-calling, sandbox, mutation policy, verification gates и model routing.

Источник кейса: два последовательных внутренних architecture snapshot - 13.09.2026 и 14.09.2026. Все цифры ниже основаны на фактически зафиксированных runtime-тестах.

01 / ЭВОЛЮЦИЯ СИСТЕМЫ

За один цикл - от рабочего агента к измеренно настроенной платформе

Первая версия уже имела локальный Qwen, native tool-calling, Docker sandbox и автономный test-fix-retest loop. Следующая итерация превратила это в более зрелый operating baseline: новая основная модель, почти 5-кратный рабочий context, агрессивно оптимизированный prefill, безопасный mutation workflow и multi-model GPU router.

Область	13.09.2026	14.09.2026 / v7
Основная coding-модель	Qwen3-Coder-Next Q4_K_M	Qwen3.6-35B-A3B Q4_K_M
Рабочий context	28 672	140 800 (4.91x к прежнему baseline)
KV cache	q8_0 K/V на GPU	q8_0 K/V на GPU сохранён
uBatch	128	512; измеренный optimum
Prefill	профиль ограничен VRAM headroom	~275.6 tok/s на 8 117 токенах
Decode	DFlash n=2 уже использовался	~30-33 tok/s real coding; 32.20 tok/s benchmark
Tool layer	read/write/replace/grep/run_command	Workspace Tool 1.4.16 + safe-edit/recovery policy
Mutation safety	точечные изменения предпочтительны	read guard + 1 retry + hard stop + mandatory verifier/finalize
GPU orchestration	одна coding-модель	llama-router, models-max=1, on-demand Qwen/Impish switching
Operational policy	filesystem as source of truth	Runbook + Model Prompt + pinned-doc/project source routing

Что важно в этой эволюции. Рост произошёл не за счёт «добавить памяти и увеличить цифру context». Каждая настройка была проверена реальным request: 143 872 токена зафиксированы как CUDA OOM boundary, uBatch 544/576 не дали meaningful gain, а DFlash n=4/n=8 оказались хуже n=2.

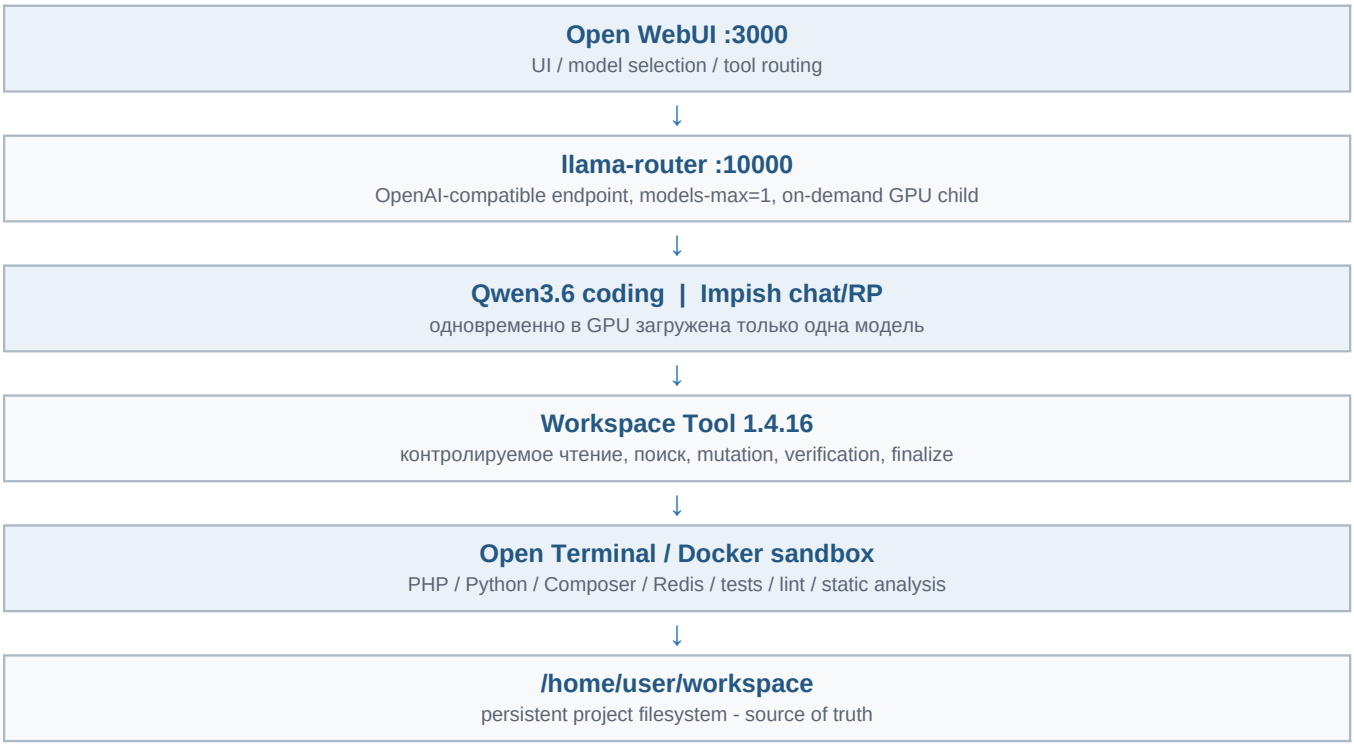
Вывод для портфолио

- Построил итерационный engineering process: benchmark -> изменение одного параметра -> реальный agent request -> измерение -> фиксация production baseline.
- Сохранил качество KV-cache на q8 вместо более агрессивного q4-компромисса и добился большого рабочего context на 8 GB GPU за счёт гибридного CPU/GPU placement.
- Отделил лабораторные microtests от реалистичных coding-turn metrics, чтобы не выдавать аномальные цифры за production performance.

02 / АРХИТЕКТУРА И ИНТЕГРАЦИЯ

Полный локальный AI runtime, а не отдельная модель

Я связал инфраструктурный, inference, orchestration и execution слои в единый контур. Модель не получает прямой shell-доступ к Proxmox host: код и команды выполняются в изолированном Docker sandbox внутри LXC, а проекты хранятся в persistent workspace.



Интеграционные решения

- Proxmox VE + LXC + nested Docker: локальная AI-среда отделена от основного хоста и может выполнять реальный код в контролируемой зоне.
- llama.cpp: production inference без облачного API, с гибридным CPU/GPU offload, q8 KV, DFlash speculative decoding и server-side prompt cache.
- Единый router endpoint: Open WebUI выбирает preset-модель обычным selector, а GPU автоматически передаётся нужному child-процессу.
- Отдельная CPU-only task model: фоновые операции не занимают VRAM основной GTX 1070.

Инженерный смысл. Я решал ограничение реального домашнего/edge-железа: как дать тяжёлой coding-модели большой context и приемлемую скорость на GTX 1070 8 GB, не отдавая модели host shell и не превращая окружение в набор несвязанных сервисов.

03 / INFERENCE ENGINEERING

Производительность получена измерениями, а не случайным preset

Конфигурация Qwen3.6 подбиралась сериями sweep-тестов. Для каждого спорного параметра фиксировались скорость, VRAM и реальный runtime outcome. В production оставлялись только изменения с воспроизводимым выигрышем.

Эксперимент	Наблюдение	Принятое решение
DFlash	26.11 tok/s без DFlash; 32.20 при n=2; n=4 и n=8 деградируют	DFlash Q8_0, n-max=2
MoE placement	n-cpu-moe=36: ~33.33 tok/s; 34 потребляет больше VRAM без speed gain	n-cpu-moe=36
uBatch	128: 111.05 tok/s prefill; 512: ~274.6-276.2; 544/576 почти не быстрее	uBatch=512, batch=2048
Context boundary	131072 PASS; 140800 PASS; 143872 request -> CUDA OOM	140800 production ceiling
Prefix cache	representative final passes: ~97-98% cached prefix	сохранять cache; не делать лишних model swaps

+23%	2.48x	4.91x
decode: 26.11 -> 32.20 tok/s в DFlash sweep	prefill throughput: uBatch 128 -> 512	рабочий context: 28 672 -> 140 800 между snapshots

Текущий production inference profile

Параметр	Baseline
Model	Qwen3.6-35B-A3B Q4_K_M, ~20.4 GB GGUF
Context / parallel	140 800 / 1
Threads	8 / batch threads 8
Batch / uBatch	2048 / 512
GPU / CPU MoE	n-gpu-layers=all; n-cpu-moe=36
KV	K=q8_0, V=q8_0
DFlash	Q8_0, n-max=2, draft GPU offload=all
Reasoning	ON, budget=4096, preserve OFF

Практическая зрелость. Важным результатом является не максимальная цифра, а найденная граница устойчивости: 140 800 токенов остаются рабочим baseline, тогда как 143 872 были отклонены после реального CUDA OOM. Это защищает систему от «конфигурация загрузилась = значит работает».

04 / AGENT TOOLING И БЕЗОПАСНЫЕ МУТАЦИИ

От function calling к контролируемому software-engineering agent loop

Самая важная часть окружения - не сама LLM, а правила, инструменты и границы, которые заставляют её работать с проектом как инженерный агент. Workspace Tool был доведён до версии 1.4.16 с отдельной стратегией для малых/больших файлов и формализованным recovery flow.

Механизм	Что он предотвращает / обеспечивает
Small-file strategy	Полный файл читается один раз на неизменённый SHA; повторный full read блокируется.
Large-file strategy	Outline-first -> grep/symbol/references -> узкие диапазоны вместо загрузки всего файла.
Structural edit	Повреждённый control-flow восстанавливается целиком в границах coherent function/class/closure.
Mutation budget	После candidate_rejected допускается один evidence-based retry; вторая ошибка -> mutation_hard_stop.
No mass rewrite	Whole-file rewriting существующих файлов запрещён; write_file используется только для новых файлов.
Verification gate	После accepted edit обязательны language/project verifier(s) и finalize_project_changes.

Рабочий fast path

1. read_file(full small file) / targeted reads for large file
2. identify coherent damaged symbol
3. replace_file_range(whole enclosing function, fresh SHA)
4. verify_python_project / verify_php_project
5. finalize_project_changes(project_root)
6. completion_gate = PASSED

Почему это важно для портфолио. Я не ограничился подключением tools. Я формализовал поведение агента вокруг чтения, изменения и проверки кода: guardrails снижают циклическое перечитывание, случайные full-file rewrites и бесконечные попытки «починить починку». Это уже слой agent engineering, а не просто UI-интеграция модели.

Подтверждённые сценарии

Сценарий	Статус
Native tool call / JSON arguments	PASS
DFlash + native tool call serialization	PASS
Read-only repository analysis	PASS
PHP mutation -> lint/static/tests -> finalize	PASS
Python broken-file recovery -> compile/Ruff -> finalize	PASS
Mutation hard-stop after exhausted budget	PASS
Workspace Tool 1.4.16 regression suite	PASS

05 / MULTI-MODEL И OPERATING POLICY

Одна GPU - несколько задач без разрушения coding baseline

На следующем этапе я добавил llama.cpp router и перевёл GPU-модели в on-demand режим. При models-max=1 в VRAM одновременно находится только один child: coding Qwen3.6 или вторичная chat/RP модель. CPU-only task model остаётся отдельным процессом.

Компонент	Роль	Фактический результат
Qwen3.6-35B-A3B	основной coding / reasoning	~8055-8083 MiB VRAM; ~30-33 tok/s real coding
Impish Bloodmoon 12B	изолированный chat/RP/creative	~7673 MiB VRAM; ~24.5 tok/s
Qwen3.5-0.8B	background task model	CPU-only; не занимает GPU
llama-router	LRU eviction + child autoload	Qwen -> Impish -> Qwen switching подтверждён

Я также зафиксировал операционные правила

- Не переключать GPU-модели без необходимости внутри длинной coding-сессии: выгрузка child уничтожает его KV/prefix cache и повышает latency следующего cold prefill.
- Project/code/runtime state брать из фактического Workspace Tool filesystem, а не из памяти модели.
- Version-specific поведение проверять через pinned documentation source; отсутствие источника явно отмечать.
- Stable operating rules держать в Agent Runbook, а не раздувать каждый пользовательский prompt.
- ZIP/project uploads сначала импортировать и распаковывать в sandbox, затем анализировать конкретную файловую систему.

Системное мышление. В результате AI-окружение имеет не только модели и tools, но и source-routing policy: для каждого типа факта определён авторитетный источник. Это уменьшает hallucination risk и делает работу агента воспроизводимой.

Что это показывает как компетенцию

- LLM runtime orchestration и resource scheduling на ограниченном железе.
- Безопасная интеграция AI с реальным source code и command execution.
- Performance profiling: context, prefill, decode, VRAM, cache, speculative decoding.
- Agent reliability: source-of-truth, bounded mutation, verification, recovery и completion gates.
- Production-oriented эксплуатация: измеримые baseline, OOM boundary, regression tests и explicit operating rules.

06 / ФОРМУЛИРОВКА ДЛЯ ПОРТФОЛИО

Как описывать эту работу на сайте

Название проекта

Local AI Coding-Agent Infrastructure Production-oriented локальная LLM/agent среда для PHP/Python разработки с собственным tool workflow, sandbox execution, safe mutations, test-fix-retest и multi-model GPU routing.

Короткая версия

Спроектировал и интегрировал локальную AI coding-agent инфраструктуру на Proxmox/LXC/Docker и llama.cpp. Оптимизировал Qwen3.6-35B-A3B под GTX 1070 8 GB до стабильного context 140.8K, ~275.6 tok/s cold prefill и ~30-33 tok/s в реальных coding-turns. Доработал Workspace Tool с безопасным mutation/recovery workflow, обязательной верификацией PHP/Python и hard-stop после повторно отклонённой мутации. Добавил multi-model router, позволяющий нескольким локальным моделям разделять одну GPU on-demand.

Расширенная версия

Построил локальный AI engineering environment, где LLM работает не как чат, а как ограниченный coding-agent: изучает фактическую структуру репозитория, читает релевантные файлы, выполняет команды в Docker sandbox, вносит минимальные изменения и повторяет lint/static analysis/tests до фактического PASS. Отдельно провёл inference tuning Qwen3.6-35B-A3B на 8 GB VRAM: DFlash, MoE CPU/GPU placement, batch/uBatch, q8 KV, prefix cache и context boundary. Рабочий context был доведён до 140 800 токенов; у стабильного preset измерены ~275.6 tok/s cold prefill и ~30-33 tok/s на реальных coding-turns. Для надёжности агентского editing flow ввёл read guards, coherent structural recovery, mutation budget, hard stop и mandatory finalize/verification. Позже добавил multi-model llama.cpp router с on-demand GPU switching и отдельной CPU task-моделью.

Факты / proof points для карточек

140.8K local coding context	**35B MoE** Qwen3.6 primary model	**8 GB** GTX 1070 VRAM
~275.6 cold prefill tok/s	**~30-33** real coding tok/s	**97-98%** prefix cache reuse

Теги / skills

Local LLM · AI Integration · Agentic Coding · llama.cpp · Open WebUI · Proxmox · LXC · Docker · CUDA · GPU/CPU Offload · MoE · Speculative Decoding · Context Engineering · Prompt/Prefix Cache · Tool Calling · Sandbox Execution · PHP 8.4 · Python 3.12 · PHPUnit · PHPStan · pytest · Ruff · mypy · AI Guardrails · Evaluation & Benchmarking

Граница корректного позиционирования. Не стоит писать, что модель была обучена или что создан собственный inference engine. Сильная и честная формулировка: спроектировал, интегрировал, оптимизировал и провалидировал локальный LLM/agent runtime и собственный agent tooling вокруг open-source inference stack.

Итог

Этот кейс уже подтверждает компетенцию не просто в «использовании AI», а в системной интеграции LLM в software-development workflow: инфраструктура, inference, инструменты, безопасность, измерения, recovery и эксплуатационные правила сведены в один локальный production-oriented контур.