Entwickler:innen-Dokumentation

computor – Systemkonzept, Softwarestruktur, Betrieb

computor – netidee-Projekt 8012, TU Graz
Dokumentation der aktuellen Version · Lizenz CC BY 4.0

Überblick **3**
Was ist es 3
Für wen ist es und wem hilft es wodurch 3
Wie funktioniert es 3

Server und Webanwendung (computor-backend) **5**
Computor Documentation 5
Architecture 7
Backend Patterns 9
Development 12
Git Integration 14
Service accounts 18
Figures 24
Public learning on computor.at 30

Test-Framework (computor-testing) **32**
Computor Testing Framework - Developer Guide 32

Betrieb und Installation **38**
Self-Update 38
Computor Backend Installer 41

VS-Code-Erweiterung (computor-vscode) **45**
Architecture Overview 45

KI-Tutorin Lena (computor-agent) **54**
computor-agent 54
Tutor Agent 62

Bewertung von Diagrammen (plot-grader) **70**
plot-grader 70
Acceptance criteria for a plot 71
Computor integration 73

Mitarbeit am Projekt **75**
Contributing to computor-backend 75

Überblick

Was ist es

Die Plattform computor ist eine quelloffene Software für Programmierkurse an Hochschulen. Studierende bearbeiten ihre Aufgaben in Visual Studio Code, entweder lokal oder im Browser, und erhalten Rückmeldung durch automatische Tests sowie durch die KI-Tutorin Lena, die so angeleitet ist, dass sie Hinweise statt fertiger Lösungen gibt. Lehrende verwalten Kurse und Abgaben über eine Webanwendung, die zugleich ein Dashboard mit Kursstatistiken bietet. Diese Dokumentation beschreibt Systemkonzept, Softwarestruktur und Schnittstellen für Entwickler:innen, die computor nach Projektende selbst betreiben, anpassen oder weiterentwickeln wollen.

Für wen ist es und wem hilft es wodurch

Die Dokumentation richtet sich an Entwickler:innen und Betreiber:innen an Hochschulen und grundsätzlich auch an Schulen oder in Unternehmen. Sie zeigt, wie eine eigene Instanz mit Docker, einer verbreiteten Technik zum Paketieren von Serverdiensten, aufgesetzt wird. So kann eine Einrichtung die Software auf eigenen Servern betreiben. Wird auch das Sprachmodell, das Lena nutzt, selbst betrieben, verlassen Daten der Studierenden die eigene Einrichtung nicht. Wer neue Funktionen ergänzen möchte, findet die Architekturregeln, den Entwicklungsablauf und den Weg bis zum Änderungsvorschlag (Pull Request) im jeweiligen Repository.

Wie funktioniert es

Systemkonzept

Die VS-Code-Erweiterung und die Webanwendung greifen über eine gemeinsame Programmierschnittstelle (REST-API) auf den Server zu. Der Server prüft bei jeder Anfrage die Rechte der angemeldeten Person und speichert Kurse, Mitglieder, Testergebnisse und Bewertungen in einer PostgreSQL-Datenbank, Dateien in einem Objektspeicher. Länger dauernde Abläufe, etwa das Anlegen der Git-Repositories für alle Studierenden eines Kurses oder das Ausführen der Tests einer Abgabe, gibt der Server an eigene Arbeitsprozesse ab, die über die Workflow-Software Temporal gesteuert und bei Fehlern wiederholt werden. Die Tests laufen dabei in einer abgeschotteten Umgebung. Optional lassen sich ein eigener Git-Server, eine Anmeldung über das Hochschulkonto (Keycloak) und Arbeitsumgebungen im Browser (Coder) zuschalten.

Lena ist als eigener Dienst angebunden. Er meldet sich mit einem eigenen Dienstkonto am Server an, antwortet, wenn Studierende Lena in einer Kursnachricht erwähnen, und bezieht bei Fragen zu einer Aufgabe die Aufgabenstellung und den Stand der Abgabe ein, soweit diese für Lena freigegeben sind. Das verwendete Sprachmodell ist austauschbar und kann über einen kommerziellen Anbieter oder auf eigener Hardware betrieben werden.

Softwarestruktur

Jede Komponente liegt in einem eigenen Repository der GitHub-Organisation computor-org:

Komponente	Repository	Aufgabe
Server und Webanwendung	computor-backend	Server (Python, FastAPI) mit Datenbank, Rechteverwaltung, Git-Anbindung und automatischen Abläufen; Webanwendung (Next.js) mit Dashboard für Lehrende; Test-Framework `computor-testing` für Python, Octave/MATLAB, R, Julia, C und Fortran
VS-Code-Erweiterung	computor-vscode	Oberfläche für Studierende, Tutor:innen und Lehrende im Editor
KI-Tutorin Lena	computor-agent	Tutor-Dienst, der mithilfe eines KI-Sprachmodells Fragen der Studierenden beantwortet und Abbildungen begutachtet
Bewertung von Diagrammen	plot-grader	Prototyp, der von Studierenden erzeugte Diagramme mit einer Referenzlösung vergleicht
Installation	computor-installer	Skripte zum Aufsetzen einer vollständigen Instanz auf einem Debian- oder Ubuntu-Server

Die folgenden Kapitel stammen aus der Dokumentation in diesen Repositories, die gemeinsam mit dem Code gepflegt wird, und beschreiben die aktuelle Version. Sie sind auf Englisch verfasst, damit sie auch international genutzt werden können. Die Software steht unter der MIT-Lizenz, diese Dokumentation unter CC BY 4.0. Weiterführendes liegt in den Repositories: das Entwicklerhandbuch und die API-Anleitung der VS-Code-Erweiterung, die Anleitungen zu Docker und zu den Umgebungsvariablen sowie die Hinweise zur Mitarbeit an den einzelnen Repositories.

Server und Webanwendung (computor-backend)

Computor Documentation

Quelle: https://github.com/computor-org/computor-backend/blob/main/docs/README.md

Computor is a university programming-course management platform: a monorepo of Python packages (DTOs, HTTP client, CLI, FastAPI backend) plus a Next.js web frontend and a Coder workspace integration.

Doc map

Doc	Read it for
architecture.md	The packages, the layered backend, data flow, and infrastructure services.
development.md	Setup, the daily cycle, adding an entity end-to-end, migrations, code generation, and tests.
backend-patterns.md	The three patterns the backend is built on: EntityInterface/DTOs, RBAC permissions, and Temporal workflows.
git-integration.md	How a course connects to git — in-system Forgejo vs. external GitLab, delivery modes, and configuring it from a deployment file.
service-accounts.md	Machine identities — testing systems, integrations and AI agents: how to add one, the `meta.yaml` ↔slug binding, API tokens and what their scopes do (and don't) restrict.
figures.md	How plots reach a student in a workspace that has no desktop: the figure-folder contract, the matplotlib backend, MATLAB's `+figurewatch`, and the viewer.
consent-gate.md	The GDPR consent gate: policy versions, the `/consent` endpoints, the middleware and how to publish a privacy notice.
public-learning.md	The anonymous `/learn` course reader, public course registration, hosted-workspace capacity and the public Ask Lena form.
tutor-mentions.md	The Lena tutor agent: `@lena` mentions, routing unaddressed messages and unit description drafts.
teacher-guide.md (Deutsch)	For lecturers and tutors: course statistics, a student's attempt timeline and feedback history, typical errors, flags and the tutor grading pages.
../ops/docs/SELF_UPDATE.md	One-click system updates from the admin UI (System → Updates) or `./computor.sh update` — config, update flow, rollback, and the rules that keep changes update-friendly.

New here? Read them in that order. For a specific task, jump straight to the relevant doc.

Quick start

```
# 1. Editable installs (order matters: types → client → backend)
python3.12 -m venv .venv && source .venv/bin/activate
pip install -e computor-types/ -e computor-client/ -e computor-backend/

# 2. Environment: generate .env with fresh secrets (never overwrites an existing .env)
./setup-env.sh

# 3. Docker services (postgres, redis, temporal, minio, traefik, workers)
./computor.sh up dev -d

# 4. Backend on :8000 (runs migrations + seeds admin automatically)
bash api.sh

# 5. Frontend on :3000 (separate terminal)
bash web.sh
```

In **dev**, only the supporting services run in Docker — the API and frontend run locally via `api.sh` / `web.sh`. In **prod** (`./computor.sh up prod -d`) everything runs in Docker. Always drive the stack through `./computor.sh` (`up` / `down` / `status` / `maintenance` / `update` / `test`), never `docker compose` directly. See development.md for the full walkthrough.

Service URLs

Service	URL
Backend API	http://localhost:8000
API docs (Swagger / ReDoc)	http://localhost:8000/docs · `/redoc`
Web frontend	http://localhost:3000
Temporal UI	http://localhost:8088
MinIO console	http://localhost:9001

Conventions

- **Branches**: `feat/*`, `fix/*`, `refactor/*`, `docs/*`. Branch off `main` and open the PR against `main`; see CONTRIBUTING.md.
- **Commits**: PRs are squash-merged, and the PR title becomes the commit subject — a short sentence saying what the change does.
- **Python**: PEP 8, 120-col lines, type hints on all signatures, `snake_case` files.
- **DTOs**: suffix by role — `UserCreate`, `UserGet`, `UserList`, `UserUpdate`, `UserQuery`; interfaces end in `Interface`. See backend-patterns.md.
- **After changing DTOs or models**: regenerate artifacts with `bash generate.sh all` and validate the web build with `npx tsc --noEmit`.

Getting help

- **Interactive API reference**: http://localhost:8000/docs (server running).
- **Ops** (Docker, env vars): `ops/docs/`.

Architecture

Quelle: https://github.com/computor-org/computor-backend/blob/main/docs/architecture.md

Computor is a **monorepo of independent packages** with a single source of truth for data contracts. The Python packages depend on each other in one direction; the web frontend consumes generated TypeScript.

Packages

Package	Purpose	Depends on
computor-types	Pure Pydantic DTOs — the single source of truth for API contracts, `meta.yaml` / `test.yaml` formats, and deployment configs. Zero backend deps.	—
computor-testing	Test-execution framework (`computor-test` CLI). One backend per language (Python, Octave, R, Julia, C, Fortran, Document); MATLAB runs in its own worker. Re-exports its models from `computor-types`.	types
computor-client	Auto-generated async HTTP client (`httpx`). One endpoint client per entity.	types
computor-backend	FastAPI server + business logic + Temporal workflows.	types, client
computor-web	Next.js frontend (yarn). Consumes generated TS types/clients.	(generated)
ops/coder	Terraform templates + scripts for provisioning Coder workspaces (not a Python package).	—

The dependency rule: **types has no dependencies; everything else flows from it.** Never import SQLAlchemy or backend modules into `computor-types`.

Layered backend

Each HTTP request flows top-to-bottom; async work is handed off sideways to Temporal.

```
HTTP request
   │
   ▼  api/          🔵 Thin endpoints — routing, DI, response_model. No business logic.
   ▼  permissions/  🟣 RBAC — Principal/Claims, role checks, permission handlers.
   ▼  business_logic/ 🟢 Fat layer — rules, validation, orchestration. Reusable, testable.
   ▼  repositories/ 🟡 Complex/reused queries beyond basic CRUD.
   ▼  model/        🔴 SQLAlchemy ORM — the DB schema (Alembic migrations derive from it).
   ▼  PostgreSQL

   tasks/     ⚡ Temporal workflows/activities — long-running & external work (git,
testing).
   services/  🔧 Infra helpers — MinIO storage, git service, Redis-backed queues.
```

The core rule: thin endpoints, fat business logic. An endpoint injects `Principal` and `Session` and delegates to a `business_logic/` function that takes explicit parameters, checks permissions early, and returns a DTO (never an ORM model).

```
# api/submissions.py — thin
@router.get("/artifacts/{artifact_id}", response_model=SubmissionArtifactGet)
async def get_artifact_endpoint(
    artifact_id: str,
    permissions: Annotated[Principal, Depends(get_current_principal)],
    db: Session = Depends(get_db),
):
    return get_artifact_with_details(artifact_id, permissions, db)  # all logic lives here
```

Other backend modules: `auth/` (external admin clients, e.g. Keycloak), `plugins/` (pluggable auth providers), `coder/` (workspace API), `git_provider/` (Forgejo and GitLab clients), `generator/` (GitLab group builder), `scripts/` (code generators and maintenance scripts), `websocket/`, `support/`, `issue_reports/`, `exceptions/`, `middleware/`. Startup lives in `server.py`; config in `settings.py`; sessions in `database.py`.

Data flow

Synchronous request: `endpoint → get_current_principal → business_logic → permission check → query/repository → return DTO`.

Asynchronous work:
`endpoint/business_logic → start Temporal workflow (task_queue "computor-tasks") → worker runs activities → external services (git, MinIO, DB) → workflow completes`. The endpoint returns a workflow handle immediately; it does not block. See backend-patterns.md.

Infrastructure services

Service	Role	Notes
PostgreSQL	Primary DB	Audit fields (`created_by/at`, `updated_by/at`) + soft delete (`archived_at`) on every model. Main DB is isolated from the Coder DB.
Redis	Caching	Tag-based view cache; expanding over time.
MinIO	S3-compatible object storage	File uploads/downloads, presigned URLs, example repositories.
Temporal	Workflow orchestration	Reliable async execution with retries; UI at `:8088`.
Traefik	Reverse proxy	Routing + ForwardAuth (workspace access).
Forgejo *(optional)*	In-system git server	Managed per-course git hosting; auto-registers into the GitServer registry at startup. See git-integration.md.
Keycloak *(optional)*	SSO / IdP brokering	External institute logins brokered via Keycloak.
Coder *(optional, `CODER_ENABLED=true`)*	Browser IDE workspaces	Separate Postgres + registry + a dedicated Temporal worker.
Updater *(optional, `UPDATE_ENABLED=true`, prod)*	Self-update sidecar	Executes admin-triggered updates (maintenance page → rebuild → restart, auto-rollback). See ops/docs/SELF_UPDATE.md.

Key design patterns

- **Service accounts** — every non-human caller (testing workers, integrations, AI agents) is a `User` with `is_service=true`, paired 1:1 with a `Service` row and authenticating via an `ApiToken` (`X-API-Token`). The testing worker is the main example. See service-accounts.md.
- **EntityInterface** — one class per entity in `computor-types` defines its Create/Get/ List/ Update/Query DTOs; drives client + TypeScript + OpenAPI generation.
- **Business-logic separation** — thin API, fat `business_logic/` (see above).
- **Repository pattern** — reusable/complex queries in `repositories/`.
- **Auto code generation** — `bash generate.sh {types|clients|python-client|schemas|all}`.
- **Dependency injection** — FastAPI `Depends` for `db`, `permissions`, services.

The first three, plus permissions and Temporal, are detailed in backend-patterns.md.

Backend Patterns

Quelle: https://github.com/computor-org/computor-backend/blob/main/docs/backend-patterns.md

Three patterns carry most of the backend: **EntityInterface** (data contracts + code generation), **the permission system** (RBAC), and **Temporal workflows** (async work).

EntityInterface & DTOs

`EntityInterface` (in `computor-types/…/base.py`) is the single source of truth for an entity's DTOs. One interface per entity references its five CRUD DTO classes:

```
class UserInterface(EntityInterface):
    create = UserCreate
    get    = UserGet
    list   = UserList
    update = UserUpdate
    query  = UserQuery
```

DTO	Used for	Shape
`Create`	POST	Required fields only; no `id`/audit fields.
`Get`	GET one	Extends `BaseEntityGet`; full fields + audit; may nest relations.
`List`	GET many	Extends `BaseEntityList`; lean fields for performance.
`Update`	PUT/PATCH	All fields `Optional`; only provided ones change.
`Query`	filters	All `Optional`, plus pagination (`skip`, `limit`).

`BaseEntityList` carries `created_at`/`updated_at`; `BaseEntityGet` adds `created_by`/`updated_by`. Keep DTOs as pure data — no business logic, no SQLAlchemy imports. Keep field names identical across a family (`email`, not `email_address` in one).

Why it matters — code generation. From these definitions, `bash generate.sh` produces the Python HTTP client (`computor-client`), TypeScript interfaces and TypeScript clients — end-to-end type safety from DB to frontend with no hand-written clients. `get_all_dtos()` discovers every interface for registration.

On the backend, an interface is extended with backend concerns (SQLAlchemy `model`, `endpoint`, `cacheable`, `searchable`), and generic CRUD routers (`CrudRouter`, `LookUpRouter`) turn it into `POST/GET/PUT/DELETE` endpoints automatically. Add custom DTOs (e.g. `enroll`, `bulk_import`) and Pydantic validators for anything beyond plain CRUD.

Permission system (RBAC)

Access control is a **hierarchical RBAC** system checked in the business-logic layer.

- **Principal** — the authenticated user, injected via `Depends(get_current_principal)`. Carries `user_id`, `is_admin`, and `Claims`.
- **Claims** — the user's system roles plus their per-course and per-organization roles (`{course_id: role}`), built from `CourseMember` rows at auth time.
- **Course role hierarchy** — higher roles inherit lower ones:

  ```
  _owner > _maintainer > _lecturer > _tutor > _student
  ```

- **Admins bypass every check** (`if principal.is_admin: return`).

The common check:

```
def get_course(course_id: str, permissions: Principal, db: Session) -> CourseGet:
    course = (
        check_course_permissions(permissions, Course, "_student", db)
        .filter(Course.id == course_id)
        .first()
    )
    if not course:
        raise NotFoundException("Course not found")
    return CourseGet.model_validate(course)
```

Recurring shapes:

- **Course-level** — `check_course_permissions(principal, Entity, minimum_role, db)` returns a query over `Entity` limited to courses where the principal holds at least that role.
- **Resource-owner** — staff (`_tutor`+) see everything; a student only their own rows (membership check on the submission group).
- **Admin-only** — `if not permissions.is_admin: raise ForbiddenException(...)`.
- **Assignment ceiling** — when granting course roles, a member cannot grant a role above their own (admins/org-managers are uncapped).

Entity-specific rules live in **permission handlers** registered at startup (`permission_registry.register(Model, Handler(Model))`). The builtin system roles — `_admin`, `_user_manager`, `_organization_manager`, `_example_manager`, `_git_manager`, `_service_manager`, `_workspace_user`, `_workspace_maintainer` — are seeded into the `role` table by migrations, and their claims are applied idempotently on every start from `permissions/role_setup.py`.

One trap when writing a handler: `check_permissions` calls **only** `build_query`, so a handler that *narrows* instead of raising makes a successful call mean "here is what you may see", not "you are authorized". Callers must then fetch through the returned query rather than re-fetching by id from a repository — see `permissions/handlers_service.py` and its call sites in `business_logic/api_tokens.py`.

Rules of thumb: check permissions in `business_logic/` (not the endpoint), check them **early** (before expensive work), always allow the admin bypass, and raise descriptive `ForbiddenException` messages.

Temporal workflows

Temporal runs long-running and external-service work reliably (automatic retries, durable state, visibility in the UI at `:8088`). Used for git provisioning, student-template/ reference generation, submission testing, and example deployment.

Four pieces:

- **Workflow** (`@workflow.defn`) — orchestration only; deterministic; calls activities.
- **Activity** (`@activity.defn`) — the actual work (DB, git, MinIO). Make it **idempotent** (check-if-exists before create) so retries are safe.
- **Worker** — runs workflows+activities off a task queue (`computor-tasks`, `testing`, `testing-matlab`, `support` with support mail, and — with Coder — `coder-tasks`).
- **Client** — starts workflows from the API/business logic.

```
@workflow.defn
class GenerateStudentTemplateWorkflow:
    @workflow.run
    async def run(self, course_content_id: str) -> dict:
        content = await workflow.execute_activity(
            fetch_course_content_activity, course_content_id,
            start_to_close_timeout=timedelta(minutes=2),
        )
        repo = await workflow.execute_activity(
            create_template_repository_activity, content,
            start_to_close_timeout=timedelta(minutes=10),
        )
        return {"repository_url": repo["repository_url"]}
```

Starting one from the backend returns immediately with a handle:

```
client = await get_temporal_client()
handle = await client.start_workflow(
    SomeWorkflow.run, arg,
    id=f"some-{arg}", task_queue="computor-tasks",
)
return {"workflow_id": handle.id, "status": "started"}
```

Patterns you'll see: **sequential** (activity output → next input), **parallel** (`asyncio.gather` over activities), **retry/fallback** (`RetryPolicy` + compensation on `ActivityError`), and **poll-until-done** for long external operations.

Workers run in Docker as part of the stack (`temporal-worker`, `-testing`, `-matlab`, and `-coder` when enabled); replica counts are env-tunable. Because source is baked into the image, rebuild after editing `tasks/`: `./computor.sh up dev --build -d`.

The course-git workflows (`temporal_hierarchy_management`, `temporal_student_template_v2`) are the backbone of git-integration.md.

Development

Quelle: https://github.com/computor-org/computor-backend/blob/main/docs/development.md

Setup, the daily cycle, and the recurring tasks: adding an entity, migrations, code generation, and tests.

Prerequisites

- **Python 3.12**, **Docker + Docker Compose**, **Git**
- Recommended: VS Code / PyCharm, an API client (HTTPie/Postman), a DB client (DBeaver)

Setup

```
# 1. Virtualenv + editable installs (order matters: types → client → backend)
python3.12 -m venv .venv && source .venv/bin/activate
pip install -e computor-types/ -e computor-client/ -e computor-backend/

# 2. Environment — generate .env with fresh secrets (never overwrites an existing .env)
./setup-env.sh

# 3. Docker services (postgres, redis, temporal + UI, minio, traefik, workers)
./computor.sh up dev -d

# 4. Backend on :8000 — runs Alembic migrations and seeds the admin user on start
bash api.sh

# 5. Frontend on :3000 (separate terminal)
bash web.sh
```

Editable installs mean code changes take effect without reinstalling. **Never** use `docker compose` directly — `./computor.sh` (via `ops/lib/common.sh`) picks the right compose overlays from the `dev|prod` argument and the `.env` flags (`CODER_ENABLED`, `KEYCLOAK_ENABLED`, `GIT_SERVER`, `MATLAB_ENABLED`, `UPDATE_ENABLED`). `api.sh` applies migrations automatically, so `migrations.sh` is only for the rare standalone case.

Verify:

```
curl localhost:8000/docs # Swagger UI is up
```

Daily cycle

```
source .venv/bin/activate
git pull                        # on main
./computor.sh up dev -d         # add --build to rebuild worker/images after code changes
bash api.sh                     # terminal 1
bash web.sh                     # terminal 2
git checkout -b feat/your-thing
# …work…  →  ./computor.sh test  →  git commit  →  push  →  PR into main
./computor.sh down              # stop docker services (auto-detects dev/prod)
```

> Worker code is **baked into the worker images**. After editing anything under `tasks/`, rebuild with `./computor.sh up dev --build -d` or the workers keep running stale code.

Adding an entity, end-to-end

The full loop for a new entity (e.g. `Assignment`). Each step lives in the package named.

1. **DTOs** — `computor-types/…/assignments.py`: an `AssignmentInterface(EntityInterface)` plus `AssignmentCreate/Get/List/Update/Query`. See backend-patterns.md.
2. **Model** — `computor-backend/…/model/assignment.py`: a SQLAlchemy model inheriting the audit base; add the reverse `relationship` on related models.
3. **Migration** — `alembic revision --autogenerate -m "add assignments"`, review it, then `alembic upgrade head`. See Migrations.
4. **Business logic** — `business_logic/assignments.py`: functions taking `(data, permissions, db)` that check permissions and return DTOs.
5. **API** — `api/assignments.py`: thin endpoints delegating to business logic; register the router in `server.py`.
6. **Permissions** — register a handler for the model in `permissions/` if it needs entity-specific rules.
7. **Regenerate** — `bash generate.sh all` (TS types, clients, python-client, schemas).
8. **Test** — add `tests/test_assignments.py`; run `./computor.sh test`.

Migrations

Alembic derives migrations from the SQLAlchemy models. Migration files live in `computor-backend/src/computor_backend/alembic/versions/`; run the `alembic` commands from `computor-backend/src/computor_backend` (where `alembic.ini` lives) with the `.env` variables exported.

```
alembic revision --autogenerate -m "message"   # generate from model changes
alembic revision -m "message"                   # empty (for data migrations)
alembic upgrade head        # apply all pending
alembic downgrade -1        # roll back one
alembic current / history   # inspect state
```

Guidelines:

- **Always review autogenerated migrations** and implement `downgrade()`.
- **Add columns nullable first**, backfill in a separate step, then set `NOT NULL` — avoids breaking a live schema.
- Migration headers vary between `revision = "..."` and `revision: str = "..."`; use `alembic heads` (not grep) to find the correct parent when hand-authoring one.

- To reset a dev DB: `./computor.sh down` intentionally disables `-v`; remove the postgres data dir under `${SYSTEM_DEPLOYMENT_PATH}/postgres` manually, then `./computor.sh up dev -d`
  - `api.sh`.

Code generation

`generate.sh` regenerates web/client artifacts offline from the Pydantic models.

```
bash generate.sh types          # TypeScript interfaces
bash generate.sh clients        # TypeScript API clients
bash generate.sh python-client  # computor-client (needs the backend importable)
bash generate.sh schemas        # JSON schemas for meta.yaml / test.yaml
bash generate.sh all            # everything (default)
```

The checked-in `computor-web/src/generated/` can be **stale**, so a full regen may bundle unrelated drift — for a focused change, edit the generated file surgically and validate with `npx tsc --noEmit`. `computor-web` uses **yarn** (never `npm install`).

Tests

```
./computor.sh test                        # all tests (loads .env for the database)
./computor.sh test --unit                 # unit tests only (no database needed)
./computor.sh test --integration          # integration tests (stack must be up)
./computor.sh test --file test_models     # a single test file
pytest path/to/test_x.py::test_fn         # a single test (from computor-backend/src)
pytest --cov=computor_backend             # coverage
```

Write unit tests against `business_logic/` functions and integration tests against the API via `TestClient`. Database fixtures (`database_url`, `engine`, `session`) live in `tests/conftest.py`; mock principals and sample data (`admin_principal`, `student_principal`, `test_client_factory`, ...) in `tests/fixtures.py`.

Troubleshooting

Symptom	Fix
Port already in use	`lsof -i :8000` → `kill -9 <PID>`
Import errors for `computor_*`	Re-run the editable installs (order: types → client → backend)
Worker runs old code	Rebuild: `./computor.sh up dev --build -d` (source is baked into worker images)
DB connection fails	`docker ps
Migration "table already exists" / out of sync	`alembic current`; `alembic stamp head`; or reset the dev DB (above)

Ops/env details: `ops/docs/`.

Git Integration

Quelle: https://github.com/computor-org/computor-backend/blob/main/docs/git-integration.md

Every course connects to a git backend that hosts its **template** repository (the assignment skeleton students receive), an optional **reference** repository (full solutions, staff-only), and each member's **student repository**. Git is configured **per course** — the choice of server and credentials belongs to the course, not to the organization.

There are two providers and three delivery/hosting modes.

Providers & modes

	In-system Forgejo	External GitLab	Download
`provider`	`forgejo`	`gitlab`	any / none
`base_url`	omit (the managed instance)	required (e.g. `https://gitlab.example.org`)	—
Where creds live	the system server's — nothing on the course	the **course's own** group token, on the binding	—
Course home	one Forgejo org per course	a GitLab group under `parent_group_id`	template archive only
Student access	forked repo + platform-issued clone token	student's own repo via their PAT	archive download, no repo
`delivery`	`git`	`git`	`download`

- **In-system managed Forgejo** — the platform's own Forgejo, run as part of the stack. It auto-registers into the GitServer registry at startup, so a course just points at it with `provider: forgejo` (no URL, no credentials). Best default for new courses.
- **External GitLab** — an existing institutional GitLab. The course brings its own **parent group** (its course group is created underneath) and a **group access token** stored encrypted on the binding. Students bring their **own** GitLab PAT.
- **Download** — no per-student git; assignments are delivered as a template archive. Set `delivery: download`. Useful for lightweight or unbound courses.

Data model

- **GitServer registry** — a global list of git servers (`type`, `base_url`, …), *not* scoped to an organization. Managed Forgejo self-registers; external GitLab instances are registered as **tokenless pointers** (no shared credential — the token lives per course).
- **CourseGitBinding** — one per course. Points at a `GitServer` and carries the course-specific bits: for GitLab, `parent_group_id` and an **encrypted** `token` (the group access token). Template/reference repo references live here too.
- **CourseMemberGitRepository** — the 1:1 mapping of a course member to their student repo. (A separate, older per-submission-group home under `submission_group.properties` still exists for team assignments.)

The binding is immutable once materialized

A course's git binding **locks** as soon as it is materialized — i.e. once the template repo has been created or any student repository exists. Renaming or repointing the template after that orphans every student fork. So: **configure git at course creation**; there is deliberately no casual "edit git" path afterwards. (Before materialization, the binding can still be adjusted.)

Configuring git

Two entry points. Both end in the same `CourseGitBinding`.

1. Manually (web UI)

- **On create** — the course-create page has a git step: pick the server (managed Forgejo or an external GitLab) and, for GitLab, the `parent_group_id` + group token.
- **Admin** — the git-servers admin UI manages the `GitServer` registry.
- Manual creation with **no** git config produces an **unbound** course; git can be configured later (while still unmaterialized).

2. From a deployment file

Deploying a course file — via **web upload** or the **VSCode** extension — sets up the git binding straight from the file. This is the recommended path: the connection travels with the course definition.

The `git:` block on a course is a `CourseGitConfig`:

Field	Applies to	Meaning
`delivery`	all	`git` (fork/clone) or `download` (archive). Default `git`.
`provider`	all	`forgejo` or `gitlab`. `forgejo` with no `base_url` = the in-system managed Forgejo.
`base_url`	GitLab	Server URL. Required for external GitLab; omit for managed Forgejo.
`parent_group_id`	GitLab	Parent group id/path the course group is created under.
`token`	GitLab	Group access token, stored **encrypted** on the binding (taken literally).
`template_repo` / `template_url`	optional	Explicit template ref/URL (auto-derived for managed servers).
`default_branch`	optional	Template default branch (defaults to `main`).
`student_repo_modes`	optional	Allowed hosting modes: subset of `['managed', 'external', 'download']`.

Omit the whole `git:` block to create an **unbound** course.

Portable reference. A server is referenced by `provider` (+ `base_url` for external instances), **not** by a machine-specific `git_server_id` UUID — so the same file deploys across systems. At deploy time the backend resolves `(provider, base_url)` to the registry server, creating a tokenless GitLab pointer if none exists yet.

Token handling. The backend does not expand `${ENV_VAR}` in `token`: it takes the value literally, encrypts it, and stores it on the binding, so a file uploaded via the web or VSCode must carry the real token. See Credentials & security.

Example: in-system Forgejo

```
course:
  name: "Programming 1"
  path: prog1
  git:
    provider: forgejo        # managed instance — no base_url, no token
    delivery: git
```

Example: external GitLab

```
course:
  name: "Programming 1"
  path: prog1
  git:
    provider: gitlab
    base_url: https://gitlab.example.org
    parent_group_id: "12345"          # course group is created under this
    token: glpat-xxxxxxxxxxxx         # group token; stored encrypted
    default_branch: main
```

Example: download-only

```
course:
  name: "Intro Seminar"
  path: intro
  git:
    delivery: download                # template archive, no per-student repos
```

Credentials & security

- **Forgejo** uses the system server's credentials. Students never handle a token: each gets a **per-user clone token** that the platform mints on demand and rotates.
- **GitLab** uses the course's own **group access token** (on the binding) for server-side operations — creating the course group, template/reference repos, and forking student repos. Students authenticate with their **own** GitLab PAT (registered via the register-gitlab flow), which grants them access to their fork.
- **The group token is never exposed over the API.** It is encrypted at rest; its DTOs are **write-only** (you can set it, never read it back); read responses expose only a `has_token: bool`. Every decrypt happens server-side inside a workflow/activity. The template-archive endpoint streams bytes server-side without revealing the token.

Deploy, release & student repos

- **"Release" == deploy to git.** There is no separate visibility flag — releasing an assignment means deploying its assigned example into the course's **template** repo. Trigger `POST /system/courses/{id}/generate-student-template` (`{}` = all pending; `{release: {course_content_ids}}` = specific). The `generate_student_template_v2` workflow is course-git-aware and finds pending content itself.
- The template deploy also populates the **reference** repo (README from the content index, solution files, `additionalFiles`, `mediaFiles`, and exposed `meta.yaml`/`test.yaml`).

- **Student repos are provisioned lazily** — on demand (e.g. the VSCode "Set up repository" action or first access), not eagerly on enrollment. On Forgejo the repo is forked and a clone token issued; on GitLab it is forked using the group token and the student is granted access with their own PAT.

Relevant endpoints

Endpoint	Purpose
`PUT /courses/{id}/git` · `GET /courses/{id}/git`	Set / read the binding (read is token-free — `has_token` only).
`GET /user/courses/{id}/git`	Membership-gated, token-free binding view for a member.
`GET/POST/… /git-servers`	Manage the GitServer registry.
`POST /system/courses/{id}/generate-student-template`	Release: deploy template (+ reference).

See also

- Workflows powering this: backend-patterns.md.
- Deployment file formats (full-hierarchy vs. single-course upload): the `computor-types` deployment configs and `ops/docs/`.

Service accounts

Quelle: https://github.com/computor-org/computor-backend/blob/main/docs/service-accounts.md

Everything that talks to Computor without a human behind it — testing systems, integrations, AI agents — is a **service account**. This page explains the model, how to add one, and the two contracts that trip people up.

The model

```
User (is_service = true)  ──1:1──  Service  ──N:1──  ServiceType
      │                              │
      │                              └── config  (JSONB: language, temporal.task_queue, …)
      └── ApiToken(s)                     slug    (the meta.yaml binding — see below)
```

Table	What it is
`user` with `is_service = true`	The **identity**. Has no password (humans sign in only through Keycloak SSO) and holds **no roles**.
`service`	The **metadata**: `slug`, `config`, `enabled`, `last_seen_at`. One row per service, 1:1 with its user.
`service_type`	The **taxonomy**: an ltree `path` (`testing.temporal`, `agent`) plus a `category` that decides default token scopes.
`api_token`	The **credential**, hung off the *user*, not the service. `ctp_<32 chars>`, stored SHA-256 hashed; only the 12-char prefix is kept in the clear.

A service account authenticates by sending `X-API-Token: ctp_…`. That header takes precedence over `Authorization` (`permissions/auth.py:parse_authorization_header`).

Two contracts worth getting right

1. An example declares what it needs; the slug is an optional pin

An example's `meta.yaml` declares how it must be executed. Prefer **language** — it says what the example actually is, and resolves on any installation that has a runner for it:

```
properties:
  executionBackend:
    language: python        # ← what this example needs (portable)
    version: "3.13"         # ← optional; matched against Service.config.language_version
    settings: { timeout: 60 }
```

Use **slug** only when you genuinely need one specific registered service:

```
properties:
  executionBackend:
    slug: acme.exec.py      # ← must equal Service.slug (a pin)
```

Exactly one of the two is required. A pin is honoured **strictly**: if that service is missing or disabled, resolution returns nothing rather than substituting a different runner — silently running a student's code on a backend the author didn't choose is not a favour. Language binding is what makes examples transferable; a slug hardcodes one deployment's service naming into the content, so sharing those examples between installations requires editing every `meta.yaml`.

Resolution (one rule, shared by upload, assignment and test time — `business_logic/testing_service.py:resolve_service_for_execution_backend`):

1. a pinned `slug` → that enabled service, or nothing;
2. else `language` → a runner whose `config.language` matches, preferring an exact `config.language_version` match, then a runner that declares no version.

That last step is how you run e.g. python 3.11 and 3.13 as separate workers: give each service a `config.language_version`, and examples that don't care still land on whichever runner declares no version.

The binding is resolved at upload (`api/examples.py:_resolve_testing_service_id`), at assignment (`lecturer_deployment:_link_testing_service`) and again at test time (`business_logic/testing_service.py:resolve_testing_service`) — all three delegate to the single rule above.

`CourseContent.testing_service_id`, `ExampleVersion.testing_service_id` and `Result.testing_service_id` are **caches** of that resolution, self-healing. So an example can be uploaded and assigned before its service exists — testing starts working the moment a matching runner appears.

The slug is an **identifier you choose**. It does not select the test runner and carries no meaning to the code — `config.language` does that. Any name works, which is exactly why binding content to it is a liability: the name is arbitrary and local to one deployment.

2. `config.language` selects the runner

```
config:
  language: python                  # selects the runner
  temporal:
    task_queue: testing             # must equal the worker's --queues= value
```

Valid languages: `python`, `octave`, `r`, `julia`, `c`, `cpp`, `fortran`, `document`, `matlab`. The first eight run through the `computor-test <language> run` CLI; `matlab` talks to a MATLAB engine over Pyro5.

Creating a `testing.*` service without a language is rejected at creation. If one somehow reaches dispatch without it, the run fails loudly naming the service and the valid values — it never guesses.

Adding a service account

Two routes to the same rows. Pick by how reproducible it needs to be.

Web UI — `/admin/services`

Admin or `_service_manager`. Create the service, then mint its token on the detail page; the value is shown **once**. Best for ad-hoc and delegated work.

Bootstrap YAML — the reproducible one

Any `services:` block under `data/deployments/*.yaml` is applied idempotently on **every API start** (`business_logic/bootstrap.py:ensure_bootstrap_services`): created once, then a no-op.

```
services:
  - slug: itpcp.exec.py
    service_type_path: testing.temporal
    language: python                    # folded into config.language at apply time
    user:
      email: testing-worker@computor.local
      given_name: Testing
      family_name: Worker
    api_token:
      token: ${TESTING_WORKER_TOKEN}    # expanded from the environment
      name: Testing Worker Token
      expires_days: 365
    config:
      temporal:
        task_queue: testing
```

This is the only route that can pin a **predefined** token value, which is what keeps the API and the worker container from drifting: `api_token.token` and the container's `API_TOKEN` read the same environment variable.

Tokens are deliberately **never auto-rotated** here. Changing `TESTING_WORKER_TOKEN` does not re-issue anything — that would silently break a running worker. Rotate on purpose: mint the new token, update the container's `API_TOKEN`, restart it, then revoke the old one.

The token value is also validated with the same checker authentication uses (`ctp_` + exactly 32 url-safe base64 characters), so a near-miss is rejected at seeding rather than on every request.

Expiry is a cutoff, not a nudge. `expires_days: 365` means that a year after first boot the worker stops authenticating and every test fails. Startup warns for the last 30 days, and if the token has already lapsed while the deployment file still pins that same value, it extends the expiry (same secret, so nothing needs reconfiguring). Omit `expires_days` for a token that never expires.

Wiring a testing worker

```
docker-compose        Service (DB)                 worker container
  API_TOKEN=ctp_…  →  api_token.token_hash    →    GET /service-accounts/me
  --queues=testing    config.temporal.task_queue    → config.language → runtime setup
```

Three things must agree — and two of them are checked for you:

1. The container's `API_TOKEN` must be an active token on the service's user.
2. `config.temporal.task_queue` must equal the container's `--queues=` value. **Checked:** `POST /tests` refuses a service whose queue has no listening worker, naming the queue, instead of leaving the run stuck. You can also just omit `task_queue` — see below.
3. `config.language` must name a runtime the image actually has. **Checked:** the worker refuses to start if its image cannot provide the language its service advertises, rather than claiming tests it will certainly fail.

One worker per language

The intended topology is one worker image per language, so the queue is derived from the language: a service with `language: octave` and no `temporal.task_queue` routes to **`testing-octave`**, and you start that worker with `--queues=testing-octave`. An explicit `task_queue` always wins, so existing deployments (and bespoke topologies like `testing-python-gpu`) are unaffected.

To add a language:

```
scripts/add-testing-language.sh octave
```

That mints a conforming token into `.env`, writes `data/deployments/testing-worker-octave.yaml` (bootstrap globs that directory, so no existing file is edited), and prints the compose block to paste. Restart the API to seed the service, then `./computor.sh up`.

Scaling *one* language across several containers does not need any of this — use `replicas` on the same queue.

`last_seen_at` on the service is a heartbeat (`PUT /service-accounts/{id}/heartbeat`). The unified testing worker does not call it, so "never" there is no evidence of a problem.

Token scopes are additive — they never restrict

This is the single most misread part of the system.

`PrincipalBuilder.build` (`permissions/auth.py`) appends every scope on a token to the principal as a `("permissions", scope)` claim. Scopes **grant**. There is no mechanism by which a scope removes anything.

Consequences:

- A **service** user holds no roles at all, so its token's scopes are its entire authority. That is what makes scopes meaningful — and why leaving the scope list empty at creation is normal: the backend fills in the defaults for the service type's category (`DEFAULT_SERVICE_SCOPES` in `business_logic/api_tokens.py`; `testing` gets 15, including `result:create` and `example:download`).
- A token on a **human** account carries that person's full role set regardless of what scopes are listed. A "read-only" personal token does not exist. Every `workspace-auto-login:*` token in the system has `scopes: []` and full user power.

That asymmetry is why only admins may mint a token for another human, and why a `_service_manager` is confined to service accounts (below).

...but there is a ceiling on what you may grant

Because scopes only ever add authority, "who may mint" is not by itself enough: a `_service_manager` holds nothing but `service:*` and `api_token:*`, and without a limit could mint a service token carrying `result:update` or `user:create` and then authenticate as that service — forging grades in any course.

`assert_may_grant_scopes` (`business_logic/api_tokens.py`) closes that:

Caller	May grant
`_admin`	any scope
anyone else (e.g. `_service_manager`)	only the scopes in `DEFAULT_SERVICE_SCOPES` for the target service type's **category**

So provisioning a testing worker still works — `testing` category ⇒ its 15 scopes — while a service manager cannot invent authority the type was never meant to have. The ceiling applies at every write: `POST /api-tokens`, `POST /api-tokens/admin/create`, and `PATCH /api-tokens/admin/{id}` (re-scoping is minting by another name). Because the same table is also the allow-list, keep the defaults tight — `worker` deliberately excludes `course:create`/`course:update`.

Governance

	`_admin`	`_service_manager`
Create / edit / archive services	✅	✅
See service-owned tokens	✅	✅
Mint / revoke tokens on **service** accounts	✅	✅

	`_admin`	`_service_manager`
See or revoke another **human's** tokens	✅	❌ (404)
Mint a token for another human	✅	❌ (403)
Create service **types**	✅	❌
Enrol a service in a course	✅	only where they also hold a course role

`_service_manager` exists so machine identities can be delegated without handing out admin. `ApiTokenPermissionHandler` narrows its token queries to service-owned rows; `assert_may_mint_token_for` blocks the mint path. Both are needed: without the narrowing a service manager could revoke an admin's tokens (denial of service), and without the mint guard they could issue themselves admin authority.

AI agents

The `agent` ServiceType is seeded and usable as **plumbing**: create an agent service, mint it a token, and enrol it in a course from the service detail page. Enrolment gives it that course role's permissions through normal course-membership claims, independent of its token scopes. Because the `agent` type has `requires_workspace = false`, `_should_skip_service_account` skips the post-create hooks — no git repository is provisioned.

The backend contains **no agent runtime** and makes no LLM calls. The course tutor Lena runs as a separate `computor-agent` process under a configured tutor account (see tutor-mentions.md). `DEFAULT_SERVICE_SCOPES` has no `agent` entry, so agent tokens default to `[]` and an agent's authority comes from its course membership.

Fields that look meaningful but are not

- **`ServiceType.schema`** — a JSON Schema for `Service.config`, shipped to workers but **never validated against**. Treat it as documentation.
- **`ServiceType.plugin_module`** — a filter column only; nothing imports it.
- **`ApiToken.last_used_at` / `usage_count`** — written only on a Redis cache *miss* (`permissions/auth.py`, 120 s TTL), so they undercount badly. A liveness hint, not an audit log.
- **Archiving a service** is a soft delete and `service.user_id` is `RESTRICT`, so the user survives and **the slug stays taken**. Archiving is not a way to free a slug for reuse.

Where things live

Concern	File
Models	`computor-backend/…/model/service.py`
Business logic	`…/business_logic/service_accounts.py`, `…/business_logic/api_tokens.py`
Boot seeding	`…/business_logic/bootstrap.py`, `data/deployments/*.yaml`
Endpoints	`…/api/services.py` (`/service-accounts`), `…/api/api_tokens.py` (`/api-tokens`), `…/api/service_type.py` (`/service-types`)
Authorization	`…/permissions/handlers_service.py`, `…/permissions/role_setup.py`
Runner dispatch	`…/testing/backends.py`

Concern	File
Slug resolution	`…/business_logic/testing_service.py`
Web UI	`computor-web/app/admin/services/`

Figures

Quelle: https://github.com/computor-org/computor-backend/blob/main/docs/figures.md

Workspaces are containers without a desktop. Nothing in them can open a plot window: the MathWorks VS Code extension renders MATLAB figures into a separate desktop window that does not exist here, and Python is no different the moment a student runs `python task.py` in a terminal. A student who writes `plot(x, y)` would see nothing at all.

So plots are published as **files**. One folder, one naming convention, and a viewer in the Computor VS Code extension. A static PNG is enough — the goal is that a student sees their plot, not that they get a real figure window.

Because it is only files, anything that can write a PNG (Octave, C++, a shell script) gets the same viewer, and it works no matter how the code was started: terminal, Run button or debugger.

The contract

The folder is named by `COMPUTOR_FIGURES_DIR`, defaulting to `/tmp/computor-figures`. Both workspace images set it.

Per figure, two files:

File	Contents
`fig-NNNNNN.png`	The rendered figure. `NNNNNN` is the figure number, zero-padded to six digits.
`fig-NNNNNN.json`	`{"number": 1, "title": "Sales", "source": "matplotlib"}`

`number` repeats the number in the file name, `title` is what the viewer labels the figure with, and `source` names the producer (`matplotlib`, `matlab`, …).

The rules:

- **A new PNG is a new figure.** The viewer shows it and reveals itself, without taking keyboard focus.
- **An overwritten PNG is an update.** The viewer reloads that figure in place. Re-running a script therefore refreshes its plots instead of piling up new ones.
- **A deleted PNG closes the figure.** This is the only way to close one, and it works in both directions: `close(fig)` in MATLAB and `plt.close()` in Python delete the files, and the viewer's close button deletes the files, which the producer notices and closes its own figure to match.

Two rules for anyone writing to the folder:

1. **Write atomically.** Fill a temporary file in the same folder and rename it over the target. The viewer reacts to files appearing, so a PNG written in place would be read half-finished. Name the temporary file so it cannot be mistaken for a figure — the producers here use `.fig-NNNNNN.tmp.png`, and readers match `fig-NNNNNN.png` exactly.
2. **Write the sidecar first.** The viewer keys off the PNG; writing the JSON first means the metadata is already there when the image shows up. A figure whose sidecar is missing is still shown, just as "Figure N".

The number is whatever the producing session calls the figure — MATLAB's `figure(3)` is `fig-000003.png` — so one folder belongs to one producer. That is how the images are built: the `vscode` template has Python and no MATLAB, the `matlab-vscode` template the other way round.

Files outlive the session that wrote them. A producer never clears the folder on startup, because the VS Code extension restarts its background MATLAB and that must not wipe the plot a student is looking at. Figures from an earlier session are shown, but do not pop the viewer open.

Who writes

Python — `computor_figures`

`ops/coder/templates/vscode/figures/computor_figures.py` is a matplotlib backend derived from Agg. The `vscode` image installs it into site-packages and sets `MPLBACKEND=module://computor_figures`, so the student configures nothing:

- `plt.show()` publishes every open figure and returns. It never blocks — there is no event loop to block on, and the script's plots stay on screen after it exits.
- `plt.close(fig)` removes that figure from the folder.
- `plt.ion()` sessions republish on every change, so a REPL stays live.
- Interpreter shutdown does **not** withdraw anything. matplotlib closes every figure at exit, which would otherwise delete a script's plots the instant it finished.

Without `COMPUTOR_FIGURES_DIR` the backend behaves exactly like stock Agg and writes nothing, so the same image can run gradings without littering the file system.

Notebooks keep their inline plots. Without this, `MPLBACKEND` would apply to ipykernel too and a notebook cell would write a file and render nothing. The workspace's startup script writes `~/.ipython/profile_default/ipython_config.py` with `c.IPKernelApp.matplotlib = "inline"`. That trait is scoped to the kernel application, so a notebook cell renders its plot under the cell while plain `python` and terminal IPython keep publishing to the figure folder.

It has to be written at start rather than baked into the image: IPython reads config from the profile directory under `IPYTHONDIR` and from nowhere else — there is no `/etc/ipython` search path. The startup script only writes the file when it is absent, so a student who edits it keeps their version.

MATLAB — `+figurewatch`

`ops/coder/templates/matlab-vscode/figures/+figurewatch` watches the session's figures on a timer and mirrors them into the folder. `matlabrc-figurewatch.m`, appended by the Dockerfile to `$MATLABROOT/toolbox/local/matlabrc.m`, starts it for every MATLAB session in the image — including the background MATLAB that the MathWorks extension launches to run and debug code, which is where a student's plots actually come from.

It is `matlabrc.m` and not `startup.m`. `startup.m` is the conventional place, but a file there publishes nothing: MATLAB runs the *first* `startup` on the search path, the userpath `~/Documents/MATLAB` comes before `$MATLABROOT/toolbox/local`, and the MathWorks base image ships a `startup.m` of its own into exactly that folder — a stub that copies proxy settings out of the environment. The shared home volume is seeded from that image, so every workspace has it, and a Computor `startup.m` would be shadowed in every session. `matlabrc` is the hook in the same directory that nothing claims, because MathWorks tells you not to edit it; it also runs before `startup.m`, so a student who writes one of their own keeps their figures.

Two consequences worth remembering. The hook lives entirely in the image, so workspaces pick it up on a rebuild without touching the home volume. And `matlabrc.m` is a vendor file: the append is only sound because R2025b's is a plain script with no trailing local functions, which is worth rechecking whenever the MATLAB version moves.

No display is involved. MATLAB exports figures perfectly well from a session that has none, which is how this is verified. The startup script attaches no `Xvfb`: every MATLAB in the container would inherit a `DISPLAY` and block on it whenever it does not answer. Software OpenGL is the only renderer here, so the bootstrap also silences the `MATLAB:hg:AutoSoftwareOpenGL` warning MATLAB otherwise prints the first time a student draws anything — the display-driver advice it links to cannot be acted on inside a container.

`COMPUTOR_FIGUREWATCH=0` on the workspace agent turns publishing off, for the workspace where it is the problem rather than the fix. Nothing else reads it.

Change detection is a fingerprint, not an event. Each tick hashes everything about a figure that shows in its picture — data, text, colours, limits, scales, tick modes, the figure's own size — and exports when the hash moves. The obvious mechanism, a `MarkedClean` listener per axes, was measured wrong in both directions on a session with no display: it leaves plots stale and runs the container out of CPU:

	R2024b	R2025b (the image)
`set(h, 'YData', …)`, `hold on; plot(…)`	missed	missed
`title`, `xlim`, `legend`	caught	missed
an idle session	quiet	three events in three seconds, so every figure re-exported twice a second: 7.5 s of CPU per 20 s of nothing

What the fingerprint leaves out is the geometry MATLAB recomputes when it renders — an axes `Position`, `PlotBoxAspectRatio` and their neighbours, which move on their own about a second after an export and would keep the export loop running by themselves. Their `…Mode` properties are hashed instead, so `axis equal` and `axis manual` still register. The figure's own `Position` is hashed: that one is the size of the exported image and the student's to set.

Costs, measured in the workspace image: 10–20 ms to fingerprint a figure, against 0.3 s to export one warm and 1.5 s cold. The timer runs every two seconds, which is both the floor on how often a figure can be exported and the worst-case delay before a plot shows up. Three figures over nine axes sitting idle cost 0.7 s of CPU per 20 s and rewrite nothing.

Two other things the watcher will not do. A figure with nothing in it is not published — `figure` and `plot(x, y)` are two statements and the first one lets a tick through, so a blank picture would otherwise beat the real one to the viewer. And a figure closed in MATLAB takes its files with it, which is the same protocol the viewer's close button uses from the other side.

One folder belongs to one producer, and the MathWorks extension keeps a background MATLAB running alongside any MATLAB in a terminal. Each publishes only the figures it owns, so they coexist — but if both sessions have a figure numbered 3, they will write over each other's `fig-000003.png`.

R — `computor_figures.R`

`ops/coder/figures/r/computor_figures.R` makes the default device an offscreen PNG with its display list kept, and publishes from `addTaskCallback` — after every top-level expression, while the device is still alive. Source it from `Rprofile.site` and a plain `plot(x, y)` publishes, `dev.off()` closes.

Most top-level expressions change nothing, so it compares the freshly rendered image with the published one and only rewrites when they differ; otherwise the viewer would reload on every keystroke. `computor_figure_title("…")` names a figure — R carries no title on the device and the `main=` of a plot cannot be read back, so without it a figure is "Figure N".

Only devices it opened are published: a student's own `png("out.png")` is their business. Deleting the PNG closes the device, as everywhere else.

Octave — `computor_figures.m`

`ops/coder/figures/octave/computor_figures.m` publishes on `computor_figures("sync")`, and an interactive session calls that by itself through `add_input_event_hook`.

Octave is the awkward one, and the reason is worth writing down so nobody re-derives it: **there is no hook that runs while a script's figures still hold their content.** `atexit` and `onCleanup` both run *after* the graphics are torn down (a figure's own `deletefcn` fires too late as well — `print` there fails with "no axes object in figure to print"), plotting never routes through the m-file `drawnow` so shadowing it does nothing, `newplot` wipes any listener put on an axes, and Octave has no `timer`. All of that was measured, not assumed.

So `octave task.m` gets its sync from outside the script, via the `octave-computor` shim: install it as `octave` ahead of the real binary on `PATH` and point `COMPUTOR_OCTAVE_REAL` at the real one. The student still configures nothing. (`--eval` and a script file are mutually exclusive in Octave, so the shim runs the script through `run()`.)

Change detection is a fingerprint of what a student can see — object count, axis limits and scales, every text object's string, and a summary of each line's data. The strings and the data summary both have to be in there: an `xlabel("time")` writes into a text object that already existed, and `set(h, "ydata", …)` changes neither the object count nor the handles.

C++ — `computor_figures.hpp`

C++ has no plotting library of its own, so `ops/coder/figures/cpp/computor_figures.hpp` is the publishing half only. Render however you like — a gnuplot pipe, matplotplusplus, your own rasteriser — then hand the PNG over:

```
#include <computor_figures.hpp>

std::system("gnuplot -e \"set terminal png; set output 'plot.png'; plot sin(x)\"");
computor::figures::publish("sin(x)", "plot.png");
```

Header-only, C++17, no dependencies. `publish` takes either a buffer or a path, with or without an explicit figure number; `is_open` lets a long-running program notice the viewer closed a figure; `withdraw` closes one.

Wiring a producer into an image

A template's Docker build context is its own directory, so a Dockerfile can only `COPY` from within it — which is why the Python and MATLAB producers live under `ops/coder/templates/<template>/figures/`. The R, Octave and C++ producers live under `ops/coder/figures/<language>/` instead, because no image ships those runtimes yet. To use one, copy its directory into the template that needs it and, in the Dockerfile:

- **R** — `Rscript -e 'source("…/computor_figures.R")'` from `Rprofile.site`
- **Octave** — put `computor_figures.m` on the Octave path, install `octave-computor` as `octave`, and set `COMPUTOR_OCTAVE_REAL`
- **C++** — drop the header in `/usr/local/include`

and set `COMPUTOR_FIGURES_DIR` as the existing templates do.

Anything else

Write the two files by the rules above and the viewer will show them. Pick a `source` string of your own.

Who reads

The Computor VS Code extension:

- `src/services/FigureFolderWatcher.ts` — watches the folder and reports what it holds.
- `src/ui/panels/FiguresPanel.ts` — the "Figures" panel: a thumbnail strip over the selected figure in full, opened beside the editor. Closing a figure deletes its PNG, which is what closes it at the producer.

The viewer only runs where there is something to watch: `COMPUTOR_FIGURES_DIR` set, or the default folder already present. On a lecturer's own machine it stays out of the way, command palette included.

- `src/ui/panels/ImagePreviewPanel.ts` — the image editor, for a PNG opened from the file tree rather than published to the folder. The workspace templates make it the default for image files by writing `workbench.editorAssociations` into the workspace settings; it is contributed at `option` priority so it takes over there and nowhere else.

Images travel to the webview as `data:` URIs rather than through `asWebviewUri`, for the same reason `renderWebviewPage` inlines CSS and JS: those URIs are served through a service worker that Firefox blocks under code-server, which is what left webviews blank in issue #267.

What that service worker actually does, measured in issue #282. A webview's content is not the iframe VS Code creates — it is a *second* iframe inside that one. Measured against a code-server container in Firefox, that grandchild is not served by the service worker: `navigator.serviceWorker.controller` is set there and lies, so even a same-origin request from that frame is answered by the network, while the identical request one frame up is answered by the worker. Chrome and current WebKit route both. When it happens, *every* `asWebviewUri` resource in that webview fails at once — which is what made VS Code's own image editor show "An error occurred while loading the image" beside "The image is stored with Git LFS", neither true, and both visible only because the stylesheet keeping them hidden had failed to load too. Older Safari fails the same way for a reason of its own (issue #274).

It is not consistent, and the remaining variable is not known. The built-in editor has been seen working in Firefox against a dev-mode Coder workspace on the same day it failed elsewhere, and a warm browser profile does not explain it — three consecutive visits with the service worker already installed broke identically. The untested difference between the two setups is the path the editor is served on: a bare container on `/`, against Coder's Traefik route with `--abs-proxy-base-path`, which is also what changes the worker's scope. Worth knowing before trusting a single "works for me" either way.

None of this reaches anything inlined, which is why the figures above kept working the whole time while opening the same PNG as a file did not — and why the fix is to stop fetching rather than to chase the worker.

Troubleshooting

A plot never appears. Check that the folder exists and is being written to:

```
ls -l "${COMPUTOR_FIGURES_DIR:-/tmp/computor-figures}"
```

Files there but nothing in the panel means the viewer side; no files means the producer side.

Python writes nothing. `python3 -c "import matplotlib; print(matplotlib.get_backend())"` must print `module://computor_figures`, and `COMPUTOR_FIGURES_DIR` must be set in that shell.

MATLAB writes nothing. `timerfind('Name', 'ComputorFigureWatch')` shows whether the watcher is running in that session; `figurewatch.start()` starts it by hand and reports why if it cannot. If the timer is missing in every session, the bootstrap is not running: `type matlabrc` should end with it, and `which -all startup` is worth a look for a file that has taken its place. A figure that stays out of the folder while others arrive is a figure with nothing in it — `get(gcf, 'Children')`. `DISPLAY` is expected to be empty; MATLAB does not need one to export.

The editor has no Computor sidebar and no Figures panel at all. Check the status bar for *Restricted Mode*. Workspace trust is disabled in the settings the startup script writes, but a `settings.json` from an older workspace is the student's to own and is only added to — if `security.workspace.trust.enabled` is missing there it will have been added on the next start, and *Trust* in the Manage menu fixes the session in the meantime.

Public learning on computor.at

Quelle: https://github.com/computor-org/computor-backend/blob/main/docs/public-learning.md

`/learn` is an anonymous, first-party static course reader. It reads generated JSON and raster images under `/learning/`; it makes no API or GitHub requests. The reader uses no API endpoints and no runtime database or storage access. The web app renders the course hierarchy, descriptions, math and DE/EN variants. GitHub is optional provenance and reuse information.

Every frontend `yarn build` regenerates the bundle from the public mirror pinned in `computor-web/learning-source.json`. The build downloads that exact commit and checks the archive SHA-256 before extraction. Updating course content means reviewing the new public revision, updating both pins and rebuilding. A failed download or checksum mismatch stops the build; deployed content stays available.

The generator selects only the three reviewed Python manifests, public `content/index*.md`/`README*.md` descriptions and referenced PNG/JPEG/GIF/WebP images whose signatures match. It rejects symlinks, traversal and oversized files, and omits hidden/archived/unreleased branches. It never copies solutions, tests, submissions, credentials, metadata authors or arbitrary example files. Only publish source content approved for anonymous reading; the bundle is a snapshot and does not follow later database visibility changes until rebuilt.

Raw HTML, external/inline images and SVG are not rendered. KaTeX disables trusted commands and bounds expansion. Static assets carry nosniff and a sandboxed CSP. An image that is not in the bundle is shown as its alt text rather than fetched.

The reader can deploy by replacing only the frontend image. No database migration, API/worker restart or workspace-template rebuild is required.

Run the publication checks with `yarn test:public-learning` and the browser checks with `yarn test:e2e e2e/public-learning.spec.ts --workers=2`. After deployment, repeat the same browser checks with `E2E_BASE_URL=https://computor.at` to verify the actual public site without starting a local server. These checks read public pages and check rejected write methods only on the static catalog.

The Next standalone server denies unsupported static-file methods with `Allow: GET, HEAD`, but its default 405-page rendering throws `NoFallbackError` and returns 500. Development returns 405. The production checks accept this known rejection only with the read-only Allow header and unchanged file bytes; successful writes always fail verification. Reproducer: POST/PUT/PATCH/DELETE `/learning/catalog.json` in the network-isolated image. The framework response-rendering repair belongs upstream; the application carries no workaround for it. Recheck this focused case on the next supported Next upgrade and restore the exact 405 expectation when fixed.

A course opens on an explicit welcome/start content item when one exists. Without such an item, the reader shows the course description and a clear link to the first released exercise. Learners can then sign in to join the course or work in a hosted Coder workspace, local VS Code, or GitHub Codespaces.

For a public deployment with Coder enabled, set `CODER_MAX_RUNNING_WORKSPACES=30`. The API refuses hosted creation/start when this positive limit or the verified workspace cgroup is missing. It counts all running and starting containers, including staff workspaces, separately from the existing distinct-user limit. Concurrent launches use a Postgres transaction lock to compare a complete Coder fleet inventory with durable reservations before calling Coder. Reservations have no guessed timeout, because a queued build can outlive one. A new active or terminal Coder build reconciles its reservation. If a launch failed before Coder created a build, an operator must compare `public_workspace_reservation` with the Coder fleet, verify no matching build remains queued, and remove only that stale row. For example, after confirming that `alice/vscode` has no pending build, delete only that row with `DELETE FROM public_workspace_reservation WHERE owner_name='alice' AND workspace_name='vscode';`. Database and Coder failures refuse new launches, leaving local VS Code and Codespaces available.

A learner who joins a public course lands in its `Selbsteinschreibung` course group, which the backend creates on the first join. The course page then shows a "Hilfe & Kontakt" block (Lena instead of tutors, the support form, the guides) and lets the learner leave the course. Leaving keeps the membership with the role `_inactive`, which grants nothing, so submissions and results stay, and joining again restores it. Learners who joined before the group existed are moved into it once with `python -m computor_backend.scripts.backfill_self_enrollment_group --apply` (without `--apply` it only reports what it would move).

Verified, signed-in learners can join public courses and submit work. Once `PUBLIC_LUNA_ENABLED=true` and the restricted worker are deployed, an assignment page also shows a short Ask Lena form. A learner sends a question and selected code or result text; the backend adds only the visible assignment description. It does not fetch repositories, reference solutions, hidden tests, or other learners' work. Existing course messages remain a separate teaching channel.

The backend admits at most 10 requests per account in a rolling hour, 30 in a rolling day, one outstanding per account, 20 waiting globally, and four active. Requests expire after 15 minutes; a completed answer remains available to its author for 24 hours. Queue operations are atomic in Redis and fail closed if Redis is unavailable. The worker pulls jobs over HTTPS, sends them through the restricted model-gateway credential to the configured language model, and posts one answer. It has no general Computor API token, inbound listener, tool bridge, or code execution capability. Before enabling it, verify that the model host writes no content logs. Lena availability is off by default and the UI hides the form until enabled.

The extension keeps its existing publisher and ID and uses supported server capabilities so one Marketplace version works with every supported server release.

Test-Framework (computor-testing)

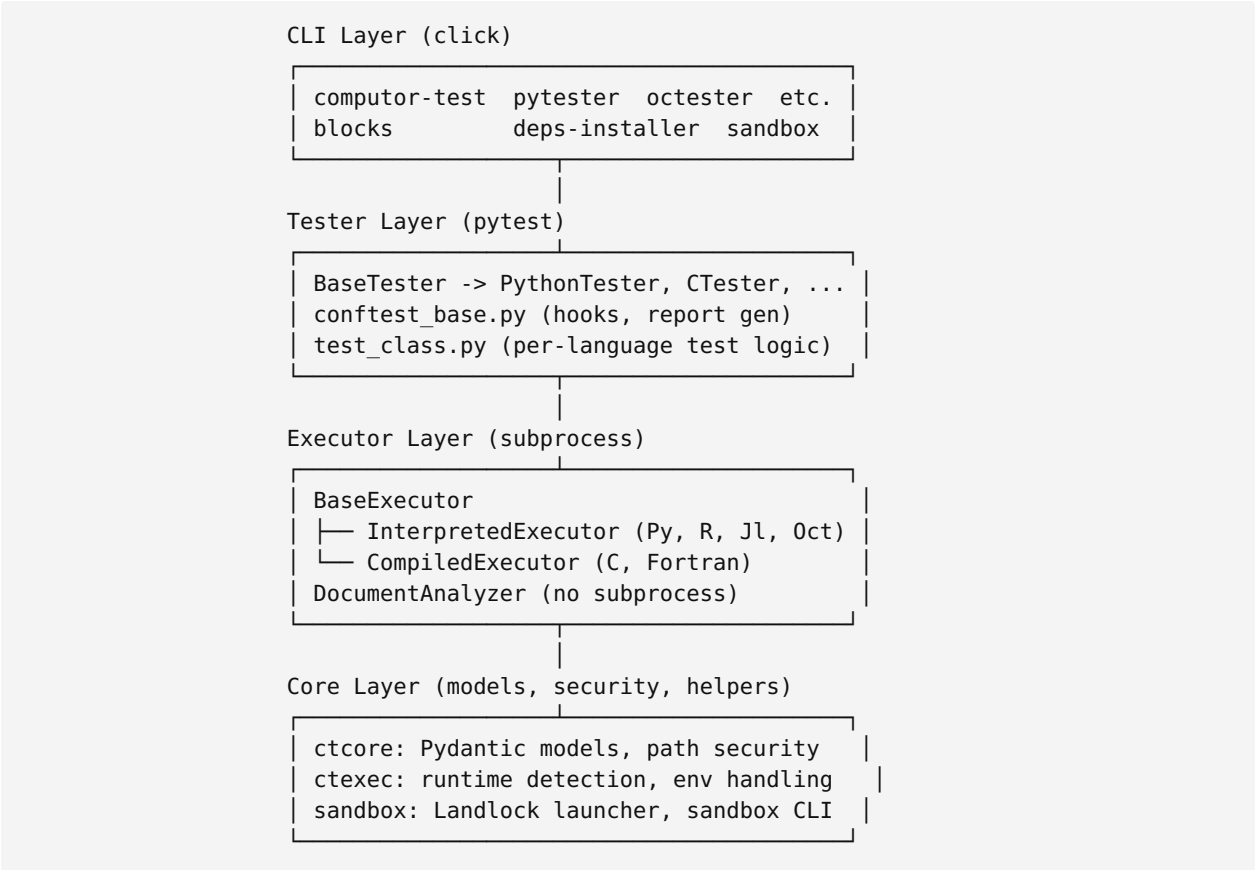
Computor Testing Framework - Developer Guide

Quelle: https://github.com/computor-org/computor-backend/blob/main/computor-testing/docs/DEVELOPER_GUIDE.md

Project Overview

This is a pytest-based testing framework for evaluating student code submissions across multiple programming languages. It is used by the backend of the Computor platform at TU Graz.

Architecture Layers



Module Map

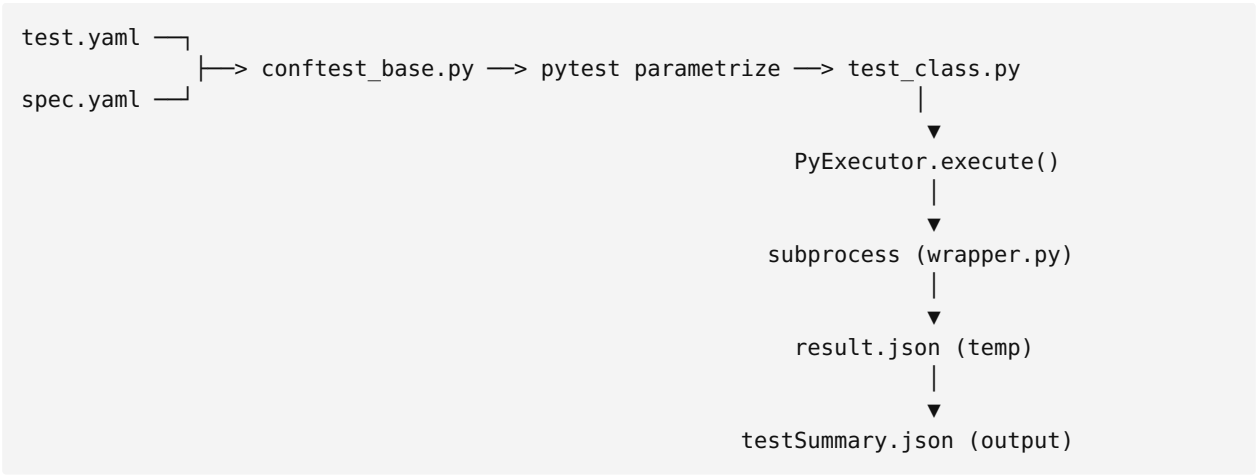
Module	Purpose	Key Files
`ctcore/`	Shared Pydantic models, helpers, security	`models.py`, `helpers.py`, `security.py`, `stdio.py`
`ctexec/`	Base executor classes, runtime detection	`base.py`, `interpreted.py`, `compiled.py`, `environment.py`, `runtime.py`

Module	Purpose	Key Files
`testers/`	Language-specific testers, CLI, pytest integration	`base.py`, `cli.py`, `runners.py`, `executors/`, `tests/`
`sandbox/`	Kernel sandbox launcher for student processes, `sandbox` CLI backends (local, Docker)	`launch.py`, `config.py`, `backends.py`, `executor.py`, `security.py`
`blocks/`	Test block definitions for VSCode extension	`models.py`, `cli.py`
`dependencies/`	Package dependency management	`models.py`, `installer.py`

How Tests Execute (End-to-End)

1. **User runs:** `computor-test python run -t student/ -T test.yaml`
2. **CLI** (`testers/cli.py`) creates a `PythonTester` and calls `.run()`
3. **BaseTester.run()** writes a temp specification.yaml and invokes `pytest.main()`
4. **conftest.py** hooks load `test.yaml` and `specification.yaml` into pytest stash
5. **test_class.py::test_entrypoint()** is parametrized with `(main_idx, sub_idx)` pairs
6. For each test case:
   - Executor runs student code in subprocess -> extracts variables via JSON
   - (Optionally) Executor runs reference code the same way
   - Test logic compares actual vs expected values
   - Result is stored in the report structure
7. **Session finish** hook writes `testSummary.json` to the output directory

Data Flow



Adding a New Language

Step 1: Create an Executor

Create `testers/executors/yourlang.py`:

```
from ctexec import InterpretedExecutor  # or CompiledExecutor

class YourLangExecutor(InterpretedExecutor):
    language = "yourlang"

    def _get_interpreter_command(self):
        return ["yourlang"]

    def _get_wrapper_extension(self):
        return ".yl"

    def _build_wrapper_script(self, script_path, variables_to_extract,
                              setup_code, teardown_code, input_data, result_path):
        # Build wrapper that:
        # 1. Runs the student script
        # 2. Extracts requested variables
        # 3. Writes JSON to result_path
        ...
```

Step 2: Register the Executor

In `testers/executors/__init__.py`:

```
from .yourlang import YourLangExecutor
EXECUTORS["yourlang"] = YourLangExecutor
```

Step 3: Add Runtime Info

In `ctexec/runtime.py`, add to `RUNTIMES`:

```
"yourlang": RuntimeInfo(
    name="YourLang",
    language="yourlang",
    binary="yourlang",
    runtime_type=RuntimeType.INTERPRETER,
),
```

Step 4: Create a Tester Runner

In `testers/runners.py`:

```
@register_tester
class YourLangTester(BaseTester):
    language = "yourlang"
```

Step 5: Create Test Infrastructure

Create:

- `testers/tests/yourlang/__init__.py`
- `testers/tests/yourlang/conftest.py` (use `create_conftest_hooks`)
- `testers/tests/yourlang/test_class.py` (implement `test_entrypoint`)

Step 6: Register CLI

In `testers/__init__.py`, add to `LANGUAGES` dict and update `list_testers()`. In `testers/cli.py`, add a language-specific CLI entry point if needed. In `pyproject.toml`, add console script entry.

Key Design Decisions

Property Inheritance

Test properties cascade: `properties` -> `test collection` -> `individual test`. A `None` value at a lower level inherits from the parent.

All tester configurations set `has_property_inheritance = True`; the inherited fields are listed in `BaseTesterConfig.sub_fields` and `main_fields` (`testers/tests/conftest_base.py`). `allowedOccuranceRange` and `occuranceType` are deliberately not inherited.

Subprocess Isolation

Student code ALWAYS runs in a subprocess, never via in-process `exec()`. This is a critical security boundary. The wrapper script:

1. Changes to the working directory
2. Redirects stdout/stderr
3. Executes the student code
4. Extracts variables and writes JSON to a temp file
5. The parent process reads the JSON

Environment Security

The `ctexec/environment.py` module provides a minimal safe environment for subprocess execution. Key principles:

- Start with minimal PATH, HOME=/tmp (R keeps the real HOME), UTF-8 locale
- Block all known secret patterns (API keys, passwords, tokens)
- Pass through only language-specific variables (PYTHONPATH, R_LIBS_USER, etc.)

Report Format

The output `testSummary.json` follows the `ComputorReport` Pydantic model:

- Top level: timestamp, type, duration, status, result, summary
- `tests[]`: One entry per test collection (ComputorReportMain)
  - `tests[]`: One entry per individual test (ComputorReportSub)
  - Each has: name, status, result (PASSED/FAILED/SKIPPED), resultMessage and resultMessageI18n

---

Blocks System

The `blocks/` module defines test types, qualifications, and fields for each language as Pydantic models. These are designed to be exported to JSON Schema and TypeScript for use in the VSCode extension.

How It Works

```
blocks/models.py (Python definitions)
        │
        ├──> blocks schema   -> JSON Schema (for validation)
        ├──> blocks typescript -> TypeScript interfaces (for VSCode)
        ├──> blocks data     -> Raw JSON (for runtime use)
        └──> blocks templates -> Pre-built YAML snippets
```

Extending Blocks

To add test types or qualifications for an existing language, edit the relevant `get_*_blocks()` function in `blocks/models.py`. To add a new language, create a new `get_yourlang_blocks()` function and register it in `get_all_blocks()`.

Block definitions exist for all seven languages.

Common Patterns

Reading test.yaml in test code

```
_report = item.config.stash[report_key]
testsuite = _report["testsuite"]
specification = _report["specification"]
main_test = testsuite.properties.tests[main_idx]
sub_test = main_test.tests[sub_idx]
```

Executing student code

```
from testers.executors import get_executor

executor_class = get_executor("python")
executor = executor_class(working_dir="/path/to/student/code", timeout=30)
result = executor.execute("script.py", variables_to_extract=["x", "y"])
# result.namespace = {"x": 42, "y": [1, 2, 3]}
```

Path validation

```
from ctcore.security import validate_path_in_root, safe_join

# Ensure path doesn't escape root
safe_path = validate_path_in_root("student/../../../etc/passwd", root_dir)
# Raises PathValidationError

# Safe path joining
path = safe_join(root_dir, "student", "solution.py")
```

Testing the Framework Itself

```
# Install in development mode (from the repository root)
pip install -e computor-types/ -e computor-testing/

# Run the framework's own tests
cd computor-testing && python -m pytest tests/

# Check all runtimes
computor-test check --all

# Run a test locally
computor-test python run -t ./localTests/correctSolution -T test.yaml -v 2

# Test sandbox backends
sandbox check -v
sandbox test
```

File Naming Conventions

File	Purpose
`test.yaml`	Test definitions (required)
`meta.yaml`	Exercise metadata (required for upload to the platform)
`specification.yaml`	Execution configuration (optional, usually auto-generated)
`testSummary.json`	Test results output
`dependencies.yaml`	Package dependencies for the exercise

Known Issues & Gotchas

1. **`allowedOccuranceRange`** and **`occuranceType`** are misspelled; the models also accept `allowedOccurrenceRange` and `occurrenceType`
2. **Document test types** `avgwordlength` and `avgsentencelength` are handled in `testers/tests/document/test_class.py` but are not in `TypeEnum`, so a test.yaml using them fails validation
3. **Two execution systems** exist (ctexec and sandbox) with overlapping functionality
4. **Two meta.yaml models** exist in `computor-types`: `CodeAbilityMeta` (`slug`, read by the platform) and the proposed `ComputorMeta` (`identifier`), which the bundled `examples/` use

Betrieb und Installation

Self-Update

Quelle: https://github.com/computor-org/computor-backend/blob/main/ops/docs/SELF_UPDATE.md

One-click system updates from the admin UI (**System → Updates**), or from the CLI (`./computor.sh update`). The system compares its running git commit with the tip of a configured repository branch; an admin can then trigger an update that serves a maintenance page, checks out and builds the new version, restarts the stack, and automatically rolls back code if the new version fails its health check. A Keycloak version change needs explicit recovery instead: Keycloak database migrations cannot be undone by reverting its image.

Configuration (.env)

```
UPDATE_ENABLED=true                # opt-in; adds the updater sidecar in prod
SYSTEM_REPO_URL=https://github.com/computor-org/computor-backend.git
SYSTEM_REPO_BRANCH=main            # branch this deployment tracks
SYSTEM_REPO_TOKEN=                 # only for private repos (see Security)
# UPDATE_GIT_FORCE=true            # allow `git reset --hard` on a dirty tree
```

The availability *check* works everywhere (also in dev). The update *executor* is production-only: in dev the API and frontend run on the host, so there is nothing the updater could restart — the UI shows the check but the button stays disabled.

How an update runs

Triggering (`POST /system/update`, admin-only) only queues a request in Redis; the **updater sidecar** (a small always-on container with the docker socket and the repo bind-mounted) picks it up and launches a detached **runner** container *outside* the compose project — so stopping/recreating stack services can never kill the process driving the update. The runner executes `ops/lib/update.sh`:

1. **preflight** — refuse if recovery is pending, or on a dirty working tree (unless `UPDATE_GIT_FORCE`).
2. **fetch + checkout** — `git fetch $SYSTEM_REPO_URL $SYSTEM_REPO_BRANCH`, checkout `FETCH_HEAD`. Then `docker compose config -q` against the *new* tree: a broken compose config (e.g. a new required variable missing from your `.env`) aborts here, **before any downtime**, and reverts the checkout.
3. **build** — all images are built while the old version is still serving.
4. **maintenance** — Redis maintenance flag + Traefik catch-all to the static maintenance page; every service stops except `traefik`, `static-server`, `redis`, `socket-proxy`, and the `updater` itself.
5. **Keycloak guard** — refuse a downgrade. For an upgrade, verify Keycloak has stopped, take a host-persistent database dump, and persist a recovery marker before the new version can migrate the database.

6. **start + health check** — `compose up -d` (migrations run on API boot), then the API and frontend are polled for up to 5 minutes each. A Keycloak upgrade also checks Keycloak readiness and the resulting schema version.
7. **success** — recovery state is cleared, maintenance ends, state `success`. **Failure → automatic rollback**: checkout of the previous commit, rebuild (cache-fast), restart, health check → state `rolled_back`. If even the rollback fails, the maintenance page **stays up**, state is `failed`, and the error in the UI / `./computor.sh update status` tells you how to recover (typically `./computor.sh maintenance exit prod` after fixing). After a Keycloak upgrade, code rollback is refused and **explicit recovery** is required instead.

Progress is written to Redis (`update:state`, `update:log`), so the admin page can narrate the run even while the API itself is down (the page keeps polling and resumes automatically).

Scheduled updates

`POST /system/update/schedule` (admin-only, also on the admin page) stores a one-shot schedule in Redis; `DELETE /system/update/schedule` cancels it. The **sidecar** fires it: each 30 s loop iteration it compares `date -u +%s` against the stored epoch and, when due, performs exactly the manual trigger's enqueue (lock → `update:state` → `update:queue`). The API pre-computes `scheduled_at_epoch` so the sidecar never parses ISO datetimes.

Redis keys:

- `update:schedule` — hash: `scheduled_at` (ISO8601), `scheduled_at_epoch`, `scheduled_by`, `scheduled_by_name`, `created_at`. No TTL; persists until fired or cancelled. Re-posting replaces it.
- `update:schedule:claimed` — transient: the sidecar claims a due schedule via atomic `RENAME`, so a concurrent cancel and a firing sidecar can never both win.
- `update:schedule:result` — hash (TTL 7 days): `outcome` (`fired` | `missed` | `skipped_lock`), `scheduled_at`, `resolved_at`, `detail` — shown on the admin page so an absent admin sees what happened.

Semantics:

- **Already up to date at fire time** — fires anyway; the run short-circuits with "Already up to date" before entering maintenance (zero downtime).
- **Missed window** — if the system was down at the scheduled time, the update still fires up to 60 min late (`SCHEDULE_GRACE_SECONDS` in `docker/updater/watch.sh`); beyond that the schedule resolves as `missed` — never a surprise update long after the window the admin picked.
- **Lock held at fire time** — resolves as `skipped_lock`, no retry loop.
- **Countdown reminders** — the API's reminder loop watches `update:schedule` alongside `maintenance:schedule` and broadcasts the same `maintenance:reminder` WebSocket events (30/20/10/5/4/3/2/1 min before).
- `POST /system/update/reset` clears runs, **not** schedules.
- Scheduling requires a live sidecar heartbeat (like the manual trigger), so a schedule that could never fire is rejected upfront.
- Clock assumption: the epoch is computed by the API and compared against the sidecar host's clock — same kernel clock in the single-host compose deployment. Multi-host would need NTP (out of scope).

Rules for update-friendly changes

- **Migrations must be backward-compatible for one version.** Alembic upgrades run automatically on boot, but a rollback restores *code only, never the schema* — `alembic downgrade` is deliberately not automated. Ship additive migrations first; destructive ones a release later.
- **New compose variables need `:-` defaults.** The running `.env` is never regenerated by an update. A `${NEW_VAR:?}` in a compose file would fail the preflight (safe, but blocks the update) — use `${NEW_VAR:-default}` plus a template entry and a release note instead.
- **Keep the updater contract stable.** The sidecar image only knows `source ops/lib/common.sh + ops/lib/update.sh; cmd_update_exec prod` — an old watcher must be able to drive a new repo's scripts. Both files are sourced fully into memory before the run, so the mid-run checkout is safe.

CLI

```
./computor.sh update check    # compare local HEAD vs. remote tip (read-only)
./computor.sh update status   # Redis state, lock, sidecar heartbeat, recent log
./computor.sh update run prod # full update from the host (no sidecar needed)
./computor.sh update recover prod restore  # restore pre-upgrade Keycloak DB/code
./computor.sh update recover prod keep-new # validate and keep a fixed new version
```

`POST /system/update/reset` (or clearing `update:lock` in Redis) recovers from a run that died without cleaning up; the lock also expires on its own after 2 hours.

Keycloak backups and recovery

`computor.sh up prod` prepares `${SYSTEM_DEPLOYMENT_PATH}/updater/` and its `.host-persistent` sentinel. The detached runner bind-mounts that directory, the Traefik dynamic configuration directory, and the maintenance page directory at their host paths. A missing mount or unwritable state directory prevents the Keycloak upgrade; backups and maintenance routes survive removal of the runner.

Pre-upgrade backups are gzip-compressed, database-only `pg_dump --create` SQL under `updater/backups/keycloak/`, created with `umask 077`; inherited deployment ACLs can also grant authorized operators access. They include Keycloak credentials and personal data: retain them according to the deployment's backup policy. They are not an off-host backup and do not cover the application database, workspace data, or PostgreSQL cluster roles. The existing `keycloak` database role is retained during recovery. Dumps are not automatically deleted.

The authoritative gate is `updater/recovery-required`; `update:recovery` in Redis mirrors it. It is written **before** starting the new Keycloak, so killing the runner during migration also leaves recovery required. While it exists, another update (even at the same commit) and `maintenance exit` refuse, and `up` preserves the maintenance route. Clearing the Redis update lock or losing Redis state does not remove this gate.

After inspecting the update log, choose one recovery command:

- **`restore`** verifies the dump and stopped Keycloak, checks out the recorded pre-upgrade commit, drops and restores only the Keycloak database, and verifies the old schema version. Every SQL error aborts recovery. It then rebuilds and starts the old services. This discards Keycloak changes made after the dump.
- **`keep-new`** keeps the current checkout, which must use the recorded new Keycloak version. Commit any fix-forward changes first; recovery requires a clean tracked working tree. It rebuilds and starts that checkout.

Both require Keycloak readiness, the expected schema version, and healthy API and frontend before clearing the gate and lifting maintenance. Any failure keeps maintenance. Once a restore has started, `keep-new` refuses: retry `restore` after fixing its cause, because a partial SQL restore must never be accepted merely because HTTP health checks succeed. Do not remove the marker manually. This restores Keycloak only; application migrations must still follow the backward-compatibility rule above.

Updating from an executor without Keycloak recovery

The running executor is sourced before checkout, so an update cannot install its own backup and recovery protection. If the deployed executor, watcher image, mounts and host directories predate this protection, first update to the current version **while keeping the current Keycloak version**, or use the manual Keycloak runbook: maintenance, verified stop and dump, checkout, then `computor.sh up prod`.

Security notes

- The updater mounts the **raw docker socket** (root-equivalent on the host); builds and compose orchestration cannot go through the read-only prod socket-proxy. Contained by: no published ports, no Traefik route, non-root user (socket via `group_add`), and the only actionable input is a Redis queue behind the admin-only API — the deployed branch always comes from `.env`, never from the queue payload.
- `SYSTEM_REPO_TOKEN` is passed to git through an inline credential helper reading it from the environment: it never appears in URLs, argv, logs, or API responses (stderr is scrubbed before it is stored).

Computor Backend Installer

Quelle: https://github.com/computor-org/computor-installer/blob/main/README.md

This repository contains a small set of scripts to automate the setup of a production-ready development backend stack on a fresh Debian/Ubuntu server.

It can install:

- Computor Backend, including its in-system Forgejo git server, Coder workspaces and Keycloak
- a standalone GitLab
- a standalone Coder
- Nginx and TLS via Let's Encrypt

Docker must already be installed; `setup.sh` stops if it is missing.

🚀 Quick start

Public reading and user-owned runtimes (local or GitHub Codespaces) can be offered independently of hosted workspace capacity; see the backend public-learning guide. Before opening registration or server grading, verify on the deployed runtime that Coder workspaces and grading workers isolate untrusted code (tenants, credentials, network, resource and time limits).

Use `setup.sh` to install the full stack in one step.

```
# Download the installer
curl -fsSL https://raw.githubusercontent.com/computor-org/computor-installer/main/setup.sh
-o setup.sh
chmod +x setup.sh

# Example: install all services with SSL enabled
# Important: wrap passwords with special characters in single quotes
sudo ./setup.sh -d example.com -m admin@example.com -p 'My#Password123' -b -g -c
```

This will configure the services for:

- Computor Backend: `example.com` or the configured domain, with Forgejo under `/forgejo`, Coder under `/coder` and Keycloak under `/auth`
- GitLab: `git.example.com`
- Coder: `coder.example.com` (separate from the Coder inside the backend stack)

If you want to skip SSL and use HTTP only, add `-n`:

```
sudo ./setup.sh -d example.com -m admin@example.com -b -g -c -n
```

Options (`setup.sh`)

Flag	Description	Required	Default
`-d`	Main server domain	Yes	-
`-m`	Email for SSL and admin accounts	Yes	-
`-p`	Global password for GitLab, Coder and backend	Required when installing a service	No default
`-b`	Install Computor Backend (`domain.tld`) on port `80` `80`	No	Disabled
`-g`	Install GitLab (`git.domain.tld`) on port `9080`	No	Disabled
`-c`	Install Coder (`coder.domain.tld`) on port `7080`	No	Disabled
`-n`	Skip SSL / Let's Encrypt and use HTTP only	No	`false`

🛠 Standalone scripts

1. Computor Backend setup (`backend-setup.sh`)

Clones the backend repository at a pinned commit, creates the `.env` configuration with the backend's `setup-env.sh`, and starts the stack with `./computor.sh up prod --build -d`.

```
./backend-setup.sh -u example.com -m admin@example.com -s 'Password' -w
```

- Debian 13 compatibility fixes
- Automatic patching for Python 3.10 references to generic `python3`
- Optional Nginx setup via `-w`
- Generates the required JWT and auth secrets automatically

2. Coder setup (`coder-setup.sh`)

Installs Coder with Docker and creates an admin user automatically.

```
./coder-setup.sh -u coder.example.com -m admin@example.com -s 'Password' -w
```

- Creates a default admin account automatically
- Maps Docker group permissions so workspaces can run containerized jobs
- Optional Nginx setup via `-w`

3. GitLab setup (`gitlab-setup.sh`)

Creates a GitLab instance with Docker Compose.

```
./gitlab-setup.sh -u git.example.com -s 'Password' -p 9080 -w
```

- Uses port `8080` by default; `setup.sh` passes `-p 9080`
- Sets the initial root password automatically
- Optional Nginx setup via `-w`

4. SSL certificate setup (`certify.sh`)

Requests a Let's Encrypt certificate for a domain and enables automatic HTTP-to-HTTPS redirect.

```
./certify.sh -d example.com -m admin@example.com
```

🏗 Architecture and ports

Service	Example subdomain	Internal port
Backend	`example.com`	`8080`
GitLab	`git.example.com`	`9080`

Service	Example subdomain	Internal port
Coder	`coder.example.com`	`7080`

- Nginx configurations support IPv4 and IPv6 natively
- `-n` disables SSL and keeps the setup on port 80 only
- `setup.sh` prints a final status summary after installation

⚠️ Requirements

- DNS records for the required subdomains must point to the server IP
- Valid domain ownership for Let's Encrypt certificate requests
- Ubuntu 22.04/24.04 or Debian 12/13
- Sudo/root access
- At least 4 GB RAM recommended (8 GB recommended for all services)

Contributing

See CONTRIBUTING.md.

License

Code: MIT (see LICENSE). Documentation (Markdown guides): CC BY 4.0 (see LICENSE-DOCS).

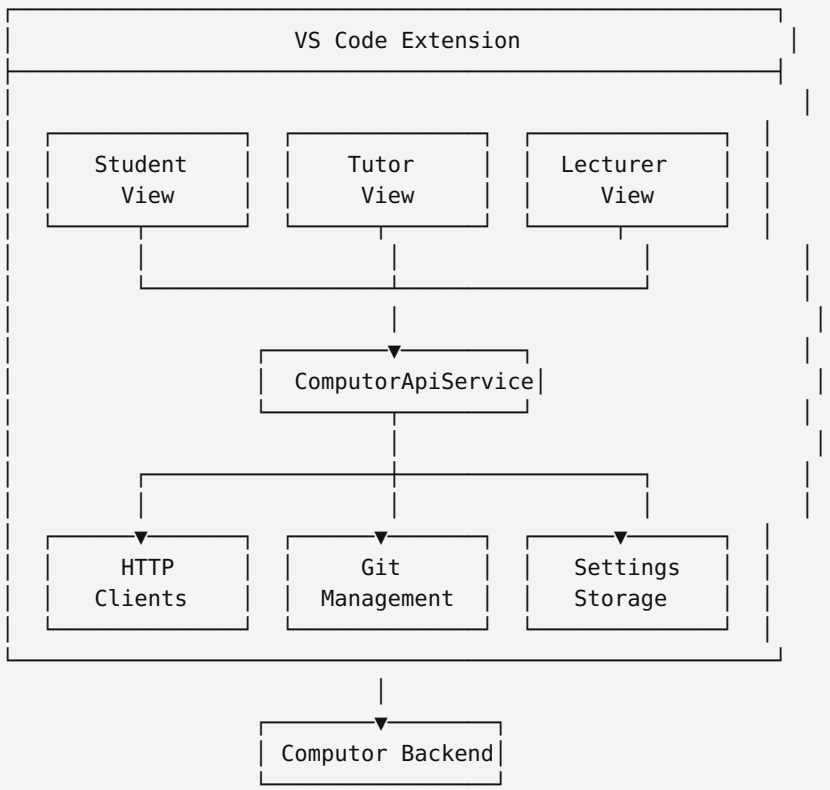
VS-Code-Erweiterung (computor-vscode)

Architecture Overview

Quelle: https://github.com/computor-org/computor-vscode/blob/main/docs/architecture.md

This document describes the architecture of the Computor VS Code Extension.

High-Level Architecture



Project Structure

```
src/
├── authentication/          # Keycloak SSO browser sign-in
│   └── SsoLoginService.ts
│
├── commands/                # VS Code command implementations
│   ├── DocumentsCommands.ts
│   ├── LecturerCommands.ts
│   ├── LecturerCourseGroupCommands.ts
│   ├── LecturerExampleCommands.ts
│   ├── StudentCommands.ts
│   ├── StudentFileCommands.ts
│   ├── StudentOfflineCommands.ts
│   ├── TutorCommands.ts
│   ├── UserManagerCommands.ts
│   └── manageRepositoryTokens.ts
│
```

```
├── exceptions/              # Backend error catalog and error classes (see USAGE.md)
│
├── git/                     # Git integration layer
│   ├── CTGit.ts             # Custom git operations
│   ├── GitWrapper.ts        # simple-git wrapper
│   ├── GitErrorHandler.ts
│   ├── gitCloneHelpers.ts
│   └── simpleGitFactory.ts
│
├── http/                    # HTTP client implementations
│   ├── cache/               # HTTP caching strategies
│   ├── ApiKeyHttpClient.ts
│   ├── BearerTokenHttpClient.ts
│   └── HttpClient.ts        # Base client
│
├── services/                # Business logic services
│   ├── BackendConnectionService.ts
│   ├── CacheService.ts
│   ├── ComputorApiService.ts     # Main API service
│   ├── CourseSelectionService.ts
│   ├── ErrorRecoveryService.ts
│   ├── ExtensionUpdateService.ts
│   ├── GitEnvironmentService.ts
│   ├── GitService.ts
│   ├── LecturerRepositoryManager.ts
│   ├── PerformanceMonitoringService.ts
│   ├── RepositoryTokenManager.ts
│   ├── RequestBatchingService.ts
│   ├── StudentRepositoryManager.ts
│   ├── TestResultService.ts
│   ├── TutorSelectionService.ts
│   ├── VirtualScrollingService.ts
│   └── WebSocketService.ts
│
├── settings/                # Settings and storage
│   ├── ComputorSettingsManager.ts
│   ├── JsonSettingsStorage.ts
│   ├── SecureStorage.ts
│   ├── SettingsStorage.ts
│   └── VscodeSecureStorage.ts
│
├── types/                   # TypeScript type definitions
│   └── generated/           # Generated types from backend
│       ├── auth.ts
│       ├── common.ts
│       ├── courses.ts
│       ├── examples.ts
│       ├── messages.ts
│       ├── organizations.ts
│       ├── roles.ts
│       ├── sso.ts
│       ├── tasks.ts
│       ├── users.ts
│       └── websocket.ts
│
├── ui/                      # User interface components
│   ├── panels/              # Webview panels
│   ├── tree/                # Tree view providers
│   │   ├── chat/
│   │   ├── lecturer/
│   │   ├── lecturer-documents/
│   │   ├── student/
```

```
│   │   ├── tutor/
│   │   └── user-manager/
│   ├── webviews/            # Webview providers
│   └── StatusBarService.ts
│
├── utils/                   # Utility functions
│   ├── deploymentHelpers.ts
│   ├── exec.ts
│   ├── gitUrlHelpers.ts
│   ├── IconGenerator.ts
│   ├── repositoryBackup.ts
│   ├── repositoryNaming.ts
│   └── workspaceStructure.ts
│
├── activation.ts            # Workspace marker detection
└── extension.ts             # Extension entry point
```

Core Components

Extension Entry Point

File: extension.ts

The main extension activation flow:

1. Activates when the workspace contains a `.computor` or `.computor-public.json` marker, or when one of its commands or views is used
2. Checks for existing authentication
3. Prompts for login if needed
4. Initializes role-based views
5. Sets up commands and providers

Key class: `UnifiedController` - manages the extension lifecycle and view initialization.

Authentication System

Directories: authentication/, http/

Two ways to authenticate, both ending in a token sent on every request:

- **Keycloak SSO (default)** — a browser-based OIDC login. The extension starts a temporary loopback HTTP server, opens `{backend}/auth/keycloak/login` in the system browser, and captures the opaque **session token** the backend redirects back (SsoLoginService.ts). The token is used as `Authorization: Bearer <token>` via `BearerTokenHttpClient`.
- **API token (escape hatch)** — for automation / CI / headless workspaces. A Computor API token (`ctp_…`) is sent as `X-API-Token` via `ApiKeyHttpClient`, supplied interactively (`Computor: Sign in with API Token`) or through the `COMPUTOR_AUTH_TOKEN` environment variable.

The session token (opaque, sliding ~1h server TTL) is refreshed proactively via `POST /auth/refresh` while VS Code is active; an expired session is recovered by re-running the browser SSO flow. A refresh starts when less than 10% of the token's lifetime remains, concurrent requests

share one refresh, and after an unrecoverable 401 further refreshes are suppressed until the user signs in again. Tokens are stored in VS Code's secret storage. The extension only ever talks to the backend, never directly to Keycloak.

HTTP Layer

Directory: http/

Architecture:

- Abstract `HttpClient` interface
- Implementations: `BearerTokenHttpClient` (SSO session token, with automatic refresh) and `ApiKeyHttpClient` (API token)
- Built-in caching with `InMemoryCache` and `NoOpCache` strategies
- Centralized error handling through `HttpError`

Features:

- Request/response caching
- Automatic retry logic
- Request batching
- Performance monitoring

ComputorApiService

File: services/ComputorApiService.ts

Central service for all backend API interactions. Provides type-safe methods for:

- Organizations and course families
- Courses and course content
- Examples and repositories
- Users and profiles
- Messages and comments
- Submissions and grading
- Test results

Uses generated types from types/generated/ for type safety.

Git Integration

Directory: git/

Key Components:

- `GitService`: Commit, push and status operations used by the commands
- `GitWrapper`: Wrapper around simple-git library
- `CTGit`: Custom git operations specific to Computor
- `GitEnvironmentService`: Git environment validation

Features:

- Repository cloning with authentication

- Repository token management
- Commit and push operations
- Conflict detection and resolution
- Repository backup and recovery

UI Architecture

Directory: ui/

Tree Views:

- `LecturerTreeDataProvider`: Course management tree
- `LecturerExampleTreeProvider`: Example repository tree
- `StudentCourseContentTreeProvider`: Student course content tree
- `TutorStudentTreeProvider`: Tutor grading tree
- `TestResultsTreeDataProvider`: Test results tree

Webview Providers:

- Course details and management
- User profile editor
- Messages and comments
- Submission details
- Course member import (CSV/TSV, XLSX/XLS, XML and JSON files, previewed against the course and imported one member at a time)
- Test results panel

Base Classes:

- `BaseTreeDataProvider`: Common tree view functionality
- `BaseWebviewProvider`: Common webview functionality

Settings Management

Directory: settings/

ComputorSettingsManager manages:

- Backend URL configuration
- Auto-login
- GitLab instance URLs
- Workspace-specific settings via `.computor` file

Settings are stored in:

- `~/.computor/config.json` (non-sensitive)
- VS Code secret storage (credentials)
- Workspace `.computor` file (backend URL, course ID)

Services Layer

Directory: services/

Key Services:

- **BackendConnectionService**: Backend health checks
- **CacheService**: Multi-tier caching for API responses
- **CourseSelectionService**: Course selection state management
- **ErrorRecoveryService**: Automatic error recovery and retry
- **ExtensionUpdateService**: Extension version checking
- **RepositoryTokenManager**: GitLab PAT management per origin
- **LecturerRepositoryManager**: Lecturer repository operations
- **RequestBatchingService**: Request batching and deduplication
- **StudentRepositoryManager**: Student repository lifecycle
- **TestResultService**: Test execution and result parsing
- **WorkspaceStructureManager** (`utils/workspaceStructure.ts`): Workspace structure management

Data Flow

Student Assignment Workflow

```
1. Student browses courses
   └─> StudentCourseContentTreeProvider
       └─> ComputorApiService.getStudentCourseContents()
           └─> HTTP Client → Backend

2. Course repository is set up
   └─> StudentRepositoryManager.autoSetupRepositories()
       └─> git clone with the repository token

3. Student tests assignment
   └─> computor.student.testAssignment
       └─> GitService.stageAll() + commitChanges(), push
       └─> TestResultService.submitTestByArtifactAndAwaitResults()
           └─> Tests run on the server → Test Results panel

4. Student submits assignment
   └─> computor.student.submitAssignment
       └─> ComputorApiService.createStudentSubmission()
```

Tutor Grading Workflow

```
1. Tutor selects course, group and member
   └─> Courses & Members tree (tutor-filter-tree-provider)
       └─> TutorSelectionService

2. Tutor opens a student's assignment
   └─> TutorStudentTreeProvider
       └─> ComputorApiService.listSubmissionArtifacts()
           └─> Download of submission and reference

3. Tutor grades submission
   └─> computor.tutor.assignment.grading
       └─> Input boxes for grade and status
           └─> ComputorApiService.submitTutorGrade()
```

Lecturer Course Management

```
1. Lecturer creates course content
   └─> LecturerCommands (Create Unit / Create Assignment)
       └─> ComputorApiService.createCourseContent()

2. Lecturer uploads example
   └─> LecturerExampleCommands.uploadExample()
       └─> Collects the example's files
           └─> ComputorApiService.uploadExample()

3. Lecturer releases content
   └─> LecturerCommands.releaseCourseContent()
       └─> ComputorApiService.validateCourseForRelease()
       └─> ComputorApiService.generateStudentTemplate()
```

Authentication Flow

```
1. Extension activates
2. On a detected .computor workspace, try to authenticate silently, in order:
   a. COMPUTOR_AUTH_TOKEN env var → ApiKeyHttpClient (X-API-Token)
   b. stored API token → ApiKeyHttpClient
   c. stored SSO session token → BearerTokenHttpClient (validated against /user)
3. If nothing valid is stored, prompt the user to sign in:
   - "Computor: Login" runs the Keycloak SSO browser flow:
     a. Start a loopback server on 127.0.0.1:<random-port>
     b. Open {backend}/auth/keycloak/login?redirect_uri=<loopback> in the browser
     c. User authenticates; backend 302s the session token back to the loopback
     d. Build BearerTokenHttpClient with the captured token
   - "Computor: Sign in with API Token" uses an entered ctp_ token instead
4. Store the token in secret storage
5. Initialize ComputorApiService
6. Fetch user views/roles
7. Initialize appropriate UI views
8. Create/refresh the .computor marker file in workspace
9. Session token auto-refreshes via POST /auth/refresh; on hard 401, re-run SSO
```

Performance Optimizations

Caching

- **HTTP Response Caching**: Reduces redundant API calls
- **Multi-tier Cache**: Memory cache with TTL
- **Cache Invalidation**: Automatic on mutations

Request Batching

- Groups multiple similar requests
- Reduces network overhead
- Configurable batch windows

Virtual Scrolling

- Efficient rendering of large lists
- Lazy loading of tree items

- Pagination support

Error Recovery

- Automatic retry with exponential backoff
- Circuit breaker for dead sessions
- Graceful degradation

Security Considerations

Credential Storage

- Credentials stored in VS Code secret storage (OS keychain)
- Never stored in plain text
- Repository tokens managed separately per origin

Git Operations

- Token injection for HTTPS URLs
- Automatic token refresh
- Repository backups before destructive operations

API Communication

- Session token expiration handling

Extension Points

VS Code Contributions

Commands: Commands for all user actions, with titles localized in `package.nls.json` (English) and `package.nls.de.json` (German) **Views**:

- Activity bar containers for Student, Tutor, Lecturer, User Manager and Chat
- Multiple tree views per role
- Test results panel

Providers:

- Status bar items

Menus: Context menus for all tree items and views

Testing

Test Structure:

- Unit tests with Mocha
- Extension host tests in a real VS Code (`test/extension-host/`)
- Integration tests with Jest
- Manual integration tests for Git and GitLab

Test Files:

- `.mocharc.unit.json`: Unit test configuration
- `jest.integration.config.js`: Integration test configuration
- `test/integration/manual/`: Manual test scripts

Build Process

Webpack Configuration:

- TypeScript compilation
- Bundling for distribution
- Source maps for debugging

NPM Scripts:

- `npm run compile`: Type-check + webpack
- `npm run watch`: Development mode
- `npm run package`: Production build
- `npm run test:unit`: Run the unit tests
- `npm run test`: Compile, lint and run all Mocha tests

Dependencies

Key Dependencies

- **simple-git**: Git operations
- **node-fetch**: HTTP requests
- **jszip**: ZIP file handling
- **js-yaml**: YAML parsing
- **xlsx**, **fast-xml-parser**: Course member import files
- **semver**: Version comparison

Dev Dependencies

- **typescript**: TypeScript compiler
- **webpack**: Module bundler
- **eslint**: Code linting
- **mocha/chai**: Testing framework

KI-Tutorin Lena (computor-agent)

computor-agent

Quelle: https://github.com/computor-org/computor-agent/blob/main/README.md

AI agents for the Computor system.

Features

- **LLM Abstraction Layer**: Unified interface for multiple LLM providers
- **Multiple Providers**: Support for OpenAI, LM Studio, Ollama, and custom providers
- **Configurable Model**: The language model is chosen in `config.yaml` (`llm.provider`, `llm.model`, `llm.base_url`). It can run on your own hardware through any OpenAI-compatible server such as Ollama, LM Studio, vLLM or llama.cpp, or in the cloud through the OpenAI API (see docs/local-llm-setup.md)
- **Streaming Support**: Real-time response streaming
- **Figure Review**: Vision-LLM review of plots/images in student submissions, used in messaging and grading (see `docs/tutor-agent.md`)
- **Highly Configurable**: Extensive configuration options via Pydantic models
- **Testing Support**: Dummy provider for testing without API calls
- **CLI Tool**: Interactive chat and single-query commands

Running the tutor agent (Docker)

The production deployment needs only Docker, a config file, and the start script:

```
./start.sh                  # first run creates config.yaml, then edit it
./start.sh --workers 8      # start the agent with 8 workers
./start.sh status           # container state + health snapshot
./start.sh logs             # follow logs
./start.sh down             # stop
```

The worker count is how many messages are processed concurrently. See docker/README.md for the full deployment guide (environment overrides, health/restart semantics, troubleshooting).

Development installation

```
# From the repository root
pip install -e .

# With development dependencies
pip install -e ".[dev]"

# Run the agent from the venv (instead of Docker)
computor-agent tutor messaging -c config.yaml --workers 4
```

See CONTRIBUTING.md for how to propose changes.

Quick Start

As a Library

```
from computor_agent import create_provider, LLMConfig, ProviderType

# Quick setup with defaults (LM Studio)
provider = create_provider(
    model="gpt-oss-120b",
    base_url="http://localhost:1234/v1",
)

# Complete response
response = await provider.complete("What is Python?")
print(response.content)

# Streaming response
async for chunk in provider.stream("Explain async/await"):
    print(chunk.content, end="", flush=True)

# Don't forget to close
await provider.close()
```

With Full Configuration

```
from computor_agent import LLMConfig, ProviderType, get_provider

config = LLMConfig(
    provider=ProviderType.OLLAMA,
    model="devstral-small",
    base_url="http://localhost:11434/v1",
    max_tokens=2000,
    system_prompt="You are a helpful coding tutor.",
)

async with get_provider(config) as provider:
    response = await provider.complete("How do I write a for loop?", temperature=0.7)
    print(response.content)
```

CLI Usage

```
# Interactive chat (default: LM Studio with gpt-oss-120b)
computor-agent chat

# Chat with specific provider/model
computor-agent chat -p ollama -m devstral-small

# Single question
computor-agent ask "What is Python?"

# With streaming
computor-agent ask "Explain recursion" --stream

# List available models
computor-agent models

# List providers
computor-agent providers
```

Supported Providers

Provider	Type	Default URL
LM Studio	Local	`http://localhost:1234/v1`
Ollama	Local	`http://localhost:11434/v1`
OpenAI	Cloud	`https://api.openai.com/v1`
Dummy	Testing	N/A

All providers use the OpenAI-compatible API format, so any other OpenAI-compatible server (for example vLLM or llama.cpp) works by setting its `base_url`.

Setting Up Ollama (Linux)

Ollama is a lightweight tool for running LLMs locally. Here's how to get started:

1. Install Ollama

```
# One-line install script
curl -fsSL https://ollama.com/install.sh | sh
```

This installs Ollama and sets it up as a systemd service that starts automatically.

2. Verify Installation

```
# Check if Ollama is running
ollama --version

# Check the service status
systemctl status ollama
```

3. Pull a Model

Choose a model based on your hardware. Smaller models run faster and need less RAM:

```
# Small models (4-8GB RAM) - Fast, good for testing
ollama pull qwen2.5-coder:1.5b      # 1.5B params, ~1GB, coding focused
ollama pull llama3.2:1b              # 1B params, ~700MB, general purpose
ollama pull phi3:mini                # 3.8B params, ~2GB, good quality

# Medium models (8-16GB RAM) - Better quality
ollama pull qwen2.5-coder:7b         # 7B params, ~4GB, excellent for code
ollama pull llama3.2:3b              # 3B params, ~2GB, balanced
ollama pull mistral:7b               # 7B params, ~4GB, versatile

# Large models (16-32GB+ RAM) - Best quality
ollama pull llama3.1:8b              # 8B params, ~5GB
ollama pull codellama:13b            # 13B params, ~8GB, code specialist
```

4. Test the Model

```
# Interactive chat
ollama run qwen2.5-coder:1.5b

# Or via API
curl http://localhost:11434/api/generate -d '{
  "model": "qwen2.5-coder:1.5b",
  "prompt": "Write a hello world in Python",
  "stream": false
}'
```

5. Use with Computor Agent

```
# Interactive chat with Ollama
computor-agent chat -p ollama -m qwen2.5-coder:1.5b

# Single question
computor-agent ask "Explain Python lists" -p ollama -m qwen2.5-coder:1.5b
```

Or in your configuration file (`config.yaml`):

```
llm:
  provider: ollama
  model: qwen2.5-coder:1.5b
  base_url: http://localhost:11434/v1
```

Ollama Commands Reference

```
# List downloaded models
ollama list

# Show model info
ollama show qwen2.5-coder:1.5b

# Remove a model
ollama rm qwen2.5-coder:1.5b

# Pull/update a model
ollama pull qwen2.5-coder:1.5b

# Run interactively
ollama run qwen2.5-coder:1.5b

# Start/stop the service
sudo systemctl start ollama
sudo systemctl stop ollama

# View logs
journalctl -u ollama -f
```

Recommended Models for Tutoring

The minimum recommended model for the Tutor AI Agent is `mistral:7b` (~4.4GB). Larger models generally produce better pedagogical guidance but require more resources.

Model	Size	RAM Needed	Notes
`mistral:7b`	~4.4GB	8GB	Minimum recommended; decent tutoring quality
`qwen3.5:35b-a3b`	~23GB	16GB	MoE (3B active); good quality/speed trade-off
`qwen3.5:122b-a10b`	~81GB	48GB	MoE (10B active); strong tutoring quality

Tutor AI Agent

Restricted public Lena worker

The public worker is a separate outbound-only process. Start it with `python -m computor_agent.public_luna_worker` after the matching backend and model-gateway route are deployed. It requires `PUBLIC_LUNA_BACKEND_URL` (HTTPS), `PUBLIC_LUNA_MODEL_URL` (private-IP HTTP(S) base URL), a 32-character or longer `PUBLIC_LUNA_WORKER_KEY` shared only with the backend, and a distinct `PUBLIC_LUNA_MODEL_KEY` for the restricted model-gateway credential. Do not give this process a general Computor API token or run the broad tutor scheduler as the public worker. The backend flag `PUBLIC_LUNA_ENABLED` remains false until the route, egress policy, logs, and synthetic response are verified.

The native macOS deployment supplies a dedicated non-login account, per-UID firewall and sandbox. Use `https://computor.at/api` as the external backend base URL.

The process polls at most four jobs concurrently, keeps request text in memory, and logs no prompt or answer content. Its backend lease is reclaimed after a worker failure. Keep its network egress restricted to the Computor HTTPS API and the private model-gateway address. Existing TU teaching workers remain separate.

The Tutor AI Agent is an autonomous agent that monitors student messages and submissions, responding automatically using an LLM. It has two modes: **messaging** (help conversations) and **grading** (submission review).

Commands

```
# Messaging agent - responds to student questions
computor-agent tutor messaging
computor-agent tutor messaging -c config.yaml -v
computor-agent tutor messaging --dry-run

# Grading agent - grades student submissions
computor-agent tutor grading --dev --reference ./assignment --student ./submission

# Development mode - interactive testing without API calls
computor-agent tutor messaging --dev
computor-agent tutor messaging --dev --assignment ./my-assignment
```

CLI Reference

`computor-agent tutor messaging`

Responds when a learner @mentions the tutor in a course message, and to follow-up replies in threads the tutor already takes part in.

Option	Description
`-c, --config PATH`	Config file (default: `config.yaml`)
`-v, --verbose`	Enable verbose logging
`--dry-run`	Log actions without sending responses
`--dev`	Development mode (interactive shell, no API calls)
`--prompts-dir PATH`	Custom prompts directory
`--assignment PATH`	[Dev mode] Assignment directory with `meta.yaml`
`--scenario PATH`	[Dev mode] Scenario directory with `scenario.yaml`
`-p, --prompt TEXT`	[Dev mode] Process a single message and exit
`-l, --log-file PATH`	Log to a rotating file
`--log-llm-content`	Log LLM prompts and replies at DEBUG level (off by default)
`--api-port PORT`	Enable the HTTP dashboard on this port
`--api-host HOST`	Dashboard bind address (default: `127.0.0.1`)
`--llm-probe-interval SECONDS`	Seconds between LLM liveness probes; `0` disables them (default: 30)
`-w, --workers N`	Messages processed concurrently (1-50)

`computor-agent tutor grading`

Reviews and grades student submissions.

Option	Description
`-c, --config PATH`	Config file (default: `config.yaml`)
`-v, --verbose`	Enable verbose logging
`--dev`	Development mode (local files only)
`--prompts-dir PATH`	Custom prompts directory
`--reference PATH`	[Dev mode] Reference solution directory
`--student PATH`	[Dev mode] Student submission directory
`-l, --language`	Language for assignment (e.g., `en`, `de`)

Configuration

All settings are in a single `config.yaml` file:

```
# Backend API connection
backend:
  url: https://api.computor.example.com
  api_token: ctp_your_api_token_here

# LLM provider settings
llm:
  provider: ollama
  model: qwen2.5-coder:7b
  base_url: http://localhost:11434/v1

# Tutor agent behavior
tutor:
  personality:
    name: "Lena"
    tone: "friendly_professional"
```

How It Works

Messaging Mode:

1. Student @mentions the tutor in a course message
2. Agent gathers the thread, assignment and submission context
3. Agent generates the response with one LLM call
4. Student can reply in the thread without a new mention for follow-ups

Grading Mode (Dev):

1. Point to reference solution and student submission directories
2. Agent compares code against requirements in `meta.yaml`
3. Agent generates grade and feedback

See docs/tutor-agent.md for detailed documentation.

Mentioning Lena

The tutor is displayed as Lena. Learners reach her through a structured course-message mention of the tutor account; typed or pasted `@lena`, `@luna` and `@ai` all resolve to that same account, with or without choosing autocomplete. Set the backend `TUTOR_AGENT_USER_ID` to the user ID of the account this worker authenticates as. The worker consumes `mentions_me` and recognizes follow-ups by user ID.

The author label and mention dropdown show the backend account's name. `tutor.personality.name` only sets how the tutor introduces itself in its replies, so keep both set to Lena.

Configuration

Environment Variables

```
# API key (for OpenAI or authenticated endpoints)
export OPENAI_API_KEY=sk-...
# or
export LLM_API_KEY=your-key
```

LLMConfig Options

```
LLMConfig(
    # Provider settings
    provider=ProviderType.LMSTUDIO,  # lmstudio, ollama, openai, dummy
    model="gpt-oss-120b",            # Model identifier
    base_url="http://localhost:1234/v1",
    api_key=None,                    # Optional API key

    # Generation parameters (temperature is passed per call,
    # e.g. provider.complete(prompt, temperature=0.7))
    max_tokens=None,                 # Max tokens to generate
    top_p=None,                      # Nucleus sampling
    top_k=None,                      # Top-k sampling
    frequency_penalty=None,          # Repetition penalty
    presence_penalty=None,           # Topic penalty
    stop_sequences=None,             # Stop strings
    seed=None,                       # For reproducibility

    # Request settings
    timeout=120.0,                   # Request timeout (seconds)
    max_retries=3,                   # Retry attempts

    # System prompt
    system_prompt=None,              # Default system prompt
)
```

Testing with DummyProvider

```
from computor_agent import DummyProvider, DummyProviderConfig, LLMConfig, ProviderType

config = LLMConfig(provider=ProviderType.DUMMY)
dummy_config = DummyProviderConfig(
    response_text="This is a test response",
    stream_chunks=["Hello ", "World!"],
    delay_seconds=0.1,  # Simulate latency
)

provider = DummyProvider(config, dummy_config)

# Test complete
response = await provider.complete("Any prompt")
assert response.content == "This is a test response"

# Test streaming
chunks = []
async for chunk in provider.stream("Any prompt"):
    chunks.append(chunk.content)
assert "".join(chunks) == "Hello World!"

# Test error handling
provider.set_should_fail(True, "Simulated error")
# Now all calls will raise LLMError
```

Running Tests

```
# Install dev dependencies
pip install -e ".[dev]"

# Run tests
pytest

# With coverage
pytest --cov=computor_agent
```

License

Code: MIT (see LICENSE). Documentation (`docs/` and Markdown guides): CC BY 4.0 (see LICENSE-DOCS).

Tutor Agent

Quelle: https://github.com/computor-org/computor-agent/blob/main/docs/tutor-agent.md

The Tutor AI Agent responds to student questions and grades submissions using an LLM.

Commands

Messaging Agent

Responds when a student **@mentions the agent** in a course message, and to follow-up replies in a thread the agent is already part of. Course-wide conversations use their own thread as context, while exercise conversations also include the permitted assignment and submission context. Explicit mentions of other recipients take precedence over automatic follow-ups.

```
# Production mode
computor-agent tutor messaging
computor-agent tutor messaging -c config.yaml --verbose
computor-agent tutor messaging --dry-run

# Development mode (interactive, no API calls)
computor-agent tutor messaging --dev
computor-agent tutor messaging --dev --assignment ./my-assignment
computor-agent tutor messaging --dev --prompts-dir ./custom-prompts
```

Options:

Option	Short	Description
`--config`	`-c`	Config file path (default: `config.yaml`)
`--verbose`	`-v`	Enable verbose logging
`--dry-run`		Log actions without sending responses
`--dev`		Development mode (interactive shell)
`--prompts-dir`		Custom prompts directory
`--assignment`		[Dev] Assignment directory with `meta.yaml`
`--scenario`		[Dev] Scenario directory with `scenario.yaml`
`--prompt`	`-p`	[Dev] Process a single message and exit
`--log-file`	`-l`	Log to a rotating file
`--log-llm-content`		Log LLM prompts and replies at DEBUG level
`--api-port`		Enable the HTTP dashboard on this port
`--api-host`		Dashboard bind address (default: `127.0.0.1`)
`--llm-probe-interval`		Seconds between LLM liveness probes; `0` disables
`--workers`	`-w`	Messages processed concurrently (1-50)

Grading Agent

Reviews and grades student submissions.

```
# Development mode (required for now)
computor-agent tutor grading --dev --reference ./assignment --student ./submission
computor-agent tutor grading --dev --reference ./assignment --student ./submission -l en

# Production mode (not yet implemented)
computor-agent tutor grading
```

Options:

Option	Short	Description
`--config`	`-c`	Config file path (default: `config.yaml`)
`--verbose`	`-v`	Enable verbose logging
`--dev`		Development mode (local files only)
`--prompts-dir`		Custom prompts directory
`--reference`		[Dev] Reference solution directory
`--student`		[Dev] Student submission directory
`--language`	`-l`	Language for assignment (e.g., `en`, `de`)

Development Mode

Development mode allows testing without making API calls.

Messaging Dev Mode

```
computor-agent tutor messaging --dev
```

Interactive commands:

Command	Description
`/new`	Start a new conversation
`/reply <id>`	Reply to a specific message
`/show`	Display conversation history
`/assignment`	Show loaded assignment details
`/reload`	Reload all prompt files
`/clear`	Clear all messages
`/exit`	Exit development mode
(text)	Send a message

Example session:

```
You: How do I write a for loop in Python?
Processing message...
Intent: question_howto

AI Response:
In Python, you can write a for loop like this:
for item in [1, 2, 3]:
    print(item)
```

Grading Dev Mode

```
computor-agent tutor grading --dev --reference ./assignment --student ./submission
```

Directory structure:

```
assignment/           # Reference solution
├── meta.yaml         # Assignment metadata and requirements
├── solution.py       # Reference implementation
└── ...

submission/           # Student submission
├── solution.py       # Student's code
└── ...
```

Custom Prompts

You can customize the agent's prompts by providing a prompts directory:

```
computor-agent tutor messaging --prompts-dir ./my-prompts
computor-agent tutor messaging --dev --prompts-dir ./my-prompts
```

The default location is `~/.computor/prompts`.

Directory structure:

```
prompts/
├── personality/           # Tone and style, prepended to the tutor prompt
│   ├── friendly_professional.md
│   ├── strict.md
│   ├── casual.md
│   └── encouraging.md
├── strategy/
│   └── tutor.md           # the live messaging prompt
├── grading/
│   └── grading.md         # rubric for `tutor grading`
├── security/              # Security prompts
│   ├── detection.md
│   └── confirmation.md
├── figure_review/
│   └── review.md
└── unit_description/
    └── unit_description.md   # drafts of unit descriptions for lecturers
```

Which file actually runs?

- **Live replies** use exactly two files: `personality/<tone>.md` and `strategy/tutor.md`.
- **Grading** uses `grading/grading.md` when present. Supplying it selects single-step grading, because that is the path a custom rubric applies to; without it, grading runs its more accurate multi-step analysis.

Features:

- Missing files are auto-created with defaults (including `strategy/tutor.md`)
- Existing files are never overwritten
- Hot reload in dev mode (changes apply immediately)
- With no prompts directory at all, the built-in prompts are used

---

Configuration

config.yaml

```
# Backend API
backend:
  url: https://api.computor.example.com
  api_token: ctp_your_token_here

# LLM provider
llm:
  provider: ollama          # ollama, lmstudio, openai
  model: qwen2.5-coder:7b
  base_url: http://localhost:11434/v1

# Secondary vision-capable LLM, used only for figure review (optional)
# vision_llm:
#   provider: ollama
#   model: llava:13b
#   base_url: http://localhost:11434/v1

# Tutor behavior
tutor:
  personality:
    name: "Lena"
    tone: "friendly_professional"

  triggers:
    # Activation is @mention-based and on by default; set enabled: false to
    # switch the agent off. Also react to submitted artifacts:
    check_submissions: true

  # Figure review (see section below)
  figure_review:
    enabled: false

# Scheduler is a TOP-LEVEL key, not part of `tutor:`. Putting it under `tutor:`
# is rejected outright (unknown keys are errors).
scheduler:
  # Interval of the periodic REST catch-up scan that backstops the WebSocket
```

```
  # event stream. Not a polling transport - see "Transport" below.
  poll_interval_seconds: 60
  cooldown_seconds: 60
  max_concurrent_processing: 5
```

Figure Review

When enabled, the agent automatically detects figures (plots/pictures: `.png`, `.jpg`, `.jpeg`, `.gif`, `.webp`, `.bmp`) in a student submission and reviews each one with a vision-capable LLM — one call per figure. The findings (labels/units, legend, readability, data plausibility, assignment fit) are injected into both the tutor messaging prompt and the grading prompts, where they can lower the grade and appear in the feedback.

```
vision_llm:                 # secondary model, used only for figure review
  provider: ollama
  model: llava:13b
  base_url: http://localhost:11434/v1

tutor:
  figure_review:
    enabled: true
    use_agent_llm: false    # true = reuse the main llm instead of vision_llm
    max_figures: 10         # figures reviewed per submission (1-50)
    max_image_bytes: 5242880
    image_extensions: [".png", ".jpg", ".jpeg", ".gif", ".webp", ".bmp"]
    max_response_tokens: 800
    temperature: 0.2
```

Notes:

- **Model choice is your responsibility.** With `use_agent_llm: true` the main `llm` model is used and must itself be vision-capable (e.g. `llava`, `qwen2.5-vl`, `gpt-4o`). A text-only model will fail per figure.
- **Fail fast:** enabling `figure_review` without a `vision_llm` section and without `use_agent_llm: true` is a startup error.
- **Graceful degradation:** if a vision call fails at runtime, the figure is reported as "could not be reviewed" and messaging/grading continue.
- **Environment overrides:** `COMPUTOR_VISION_LLM_PROVIDER`, `COMPUTOR_VISION_LLM_MODEL`, `COMPUTOR_VISION_LLM_BASE_URL`, `COMPUTOR_VISION_LLM_API_KEY`.
- **Custom prompt:** the review prompt can be overridden via `<prompts_dir>/figure_review/review.md` (hot-reloaded in dev mode).
- **Dev modes:** both `tutor messaging --dev` (scenario `submission/` directory) and `tutor grading --dev` (student submission directory) pick up figures from disk and run the same review pipeline.
- **Security note:** text rendered inside student images is untrusted; the review prompt instructs the model to ignore embedded instructions, but review output is still LLM-generated from student-controlled input.

Unit Description Drafts

A lecturer can ask Lena for a draft description of a unit, written from the assignments in it. The extension and the web UI send the request to the backend, which queues a job with the unit's title and the student-visible text of each assignment (`content/index*.md`). No submissions, messages, grades or names are part of a job.

The messaging agent claims these jobs alongside its normal work, as the tutor agent user: `POST /course-contents/description-drafts/worker/claim`. For each requested language it makes one call to the main `llm` with the prompt in `unit_description/unit_description.md` (built-in: `UNIT_DESCRIPTION_PROMPT`) and posts the drafts back. A draft longer than the backend's 4096-character limit is asked for again once, shorter; if it is still too long the job fails with `too_long` rather than being cut off. An answer cut off by the token budget before it got long (a reasoning model thinking too long) is retried once with twice the budget, then fails with `model_error`. The whole job has `job_timeout_seconds` (540) so it ends before the backend's 600-second lease; past that it fails with `model_error` instead of being handed out again. Nothing is written to the course: the lecturer reviews the draft and saves it.

```
tutor:
  unit_descriptions:
    enabled: true            # default
    poll_interval_seconds: 3
    max_response_tokens: 2048
    temperature: 0.3
    job_timeout_seconds: 540
```

The backend offers the feature only while a worker has claimed within the last 90 seconds or is still working on a claimed job, so lecturers do not see it when no agent is running. Against a backend without the routes, the worker checks again every ten minutes. When the backend answers a completion with 409 (the lecturer cancelled, or the lease ran out) the job is dropped; when it refuses the drafts (400/422) the worker fails the job with `invalid_output` so the lecturer is not left waiting.

How It Works

Messaging Flow

1. **Trigger Detection**: the student @mentions the agent, or replies in a thread the agent has already answered in
2. **Context Gathering**: recent conversation turns, student info, assignment description, the latest submission, test results
3. **Security Check**: Scan for prompt injection (optional)
4. **Response Generation**: one LLM call against a single unified prompt
5. **Send Response**: reply in the thread

The agent identifies its own messages by author id, so no response tag is needed or written.

Grading Flow

1. **Load Reference**: Read assignment requirements from `meta.yaml`

2. **Load Submission**: Read student's code
3. **LLM Analysis**: Compare against requirements
4. **Generate Feedback**: Determine grade and status

Grading status:

Status	Value	Meaning
`CORRECTED`	1	Requirements met
`CORRECTION_NECESSARY`	2	Requirements not met
`IMPROVEMENT_POSSIBLE`	3	Met, but could be better

Transport

The messaging agent is **WebSocket-only**. It needs an API token to connect and exits with an error without one — there is no HTTP-polling fallback.

```
backend:
  api_token: ctp_your_token_here
```

A periodic REST **catch-up scan** backstops the event stream, so a message that arrives while the socket is down is still picked up once it reconnects. That is a safety net, not a transport:

```
scheduler:
  poll_interval_seconds: 60   # top-level key, NOT under `tutor:`
```

Troubleshooting

LLM Connection Error

```
# Check if LLM is running
curl http://localhost:11434/v1/models

# For Ollama
ollama serve
```

Authentication Failed

```
# Check config has valid credentials
cat config.yaml | grep -A5 backend
```

No Messages Detected

- Verify the message actually @mentions the agent's user (a plain message in a thread the agent has never replied to is not a trigger)
- Confirm `triggers.enabled` is not set to `false`
- Check `--verbose` output for the WebSocket connection and the catch-up scan
- Ensure backend URL is correct

Bewertung von Diagrammen (plot-grader)

plot-grader

Quelle: https://github.com/computor-org/plot-grader/blob/main/README.md

Grades matplotlib plot submissions against a reference solution. This is the plot-evaluation prototype for Computor (netidee project 8012). docs/criteria.md defines when a plot is acceptable and how the tolerant judge applies that, and docs/integration.md describes how the judge relates to the Computor test harness.

Data is not included. This repository contains code only: no assignments, reference solutions or student submissions. The evaluation scripts in `scripts/` and `tools/` need your own assignment and submission data; the library and the test suite do not.

Install

```
uv sync               # base
uv sync --extra dev   # ruff, mypy, pytest-cov
uv sync --extra mlx   # Qwen3-VL Tier 4 via mlx-vlm (Apple Silicon only)
```

How it works

`PlotGrader` (`grader.py`) runs the student script and the reference, then applies the tiers in order. It stops at the first tier whose confidence reaches `confidence_threshold` (default 0.90). The older catester tests compared exact properties such as `figure(1).axes[0].lines[0]._linestyle`. They fail correct plots that differ in line order, colour notation or OOP vs pyplot style. The tiers normalise these differences first and compare images only when the extracted data cannot decide.

1. `tier1.py`: checks on the extracted plot data (values, colours, line styles, markers, labels), independent of plotting order.
2. `tier2.py`: looser structural matching (sets of colours and styles, curve count, fuzzy labels).
3. `tier3.py`: image similarity with OpenCV (SSIM, histograms, edges).
4. Tier 4, a vision-language model, used only when one is configured. `judges.py` calls models over HTTP: chat completions with a JSON verdict, or JEV `/v1/systemone` with yes/no probabilities. Endpoints are in `configs/vlm_endpoints.json`, which ships with placeholder hosts and model names to replace with your own OpenAI-compatible endpoints. `tier4_real.py` runs Qwen3-VL-4B locally through `mlx-vlm` (`use_real_ai=True`).

Run

```
uv run python scripts/run_validation.py          # synthetic variants; --ai, --test-mode
uv run python scripts/run_ai_eval.py             # Tier-1 failures from real submissions,
writes data/ai_eval/
uv run python scripts/run_vlm_eval.py --reference REF   # models x chat/JEV x views, writes
data/vlm_eval/
```

`run_validation.py` and `run_ai_eval.py` take `--judge 27b|4b|flash-next --mode chat|jev --view pair|pair_tolerant|student` to use an HTTP model as Tier 4.

These scripts expect your own assignment repository in `../assignments/` (`XX_UnitXX/YY_name/` with `*_master.py`, `test.yaml` and `localTests/_reference/`) or a reference given with `--reference`. All output goes to `data/`, which is gitignored.

```
from pathlib import Path
from plot_grader.grader import PlotGrader

result = PlotGrader().grade_submission(
    submission_path=Path("student_plot.py"),
    reference_path=Path("reference_plot.py"),
)
print(result.status, result.tier, result.score, result.confidence)
```

Tests

```
uv sync --extra dev
uv run pytest tests webjudge/tests
```

The tests use only trivial synthetic plots and a stub model.

Rules for contributors

Never commit reference solutions or student data: students could find the solutions here. `.gitignore` covers `data/`, `assignments/`, `submissions/` and the solution patterns; check `git status` before committing. `tools/` contains the scripts that fetch submissions from a GitLab group into `data/` (see tools/README.md). See CONTRIBUTING.md for the workflow.

License

Code: MIT (see LICENSE). Documentation (`docs/` and Markdown guides): CC BY 4.0 (see LICENSE-DOCS).

Acceptance criteria for a plot

Quelle: https://github.com/computor-org/plot-grader/blob/main/docs/criteria.md

The vision judge exists to be more tolerant than the deterministic tests: a plot that shows the same content as the reference passes, even if a mechanical detail differs that makes a data or property test fail (line order, colour name, title wording). Pixel-identical plots are checked cheaply by image comparison; the judge decides whether the rest is "similar enough". These criteria define "similar enough". They are used for the visual labels of the evaluation set and in the tolerant prompt.

A plot **passes** if a reader gets the same information from it as from the reference. It **fails** if any of these is violated:

- **M1 Same curves.** Every curve, point set or bar set of the reference is present. No extra data series (an extra grid line or reference line that the task asks for is not extra).

- **M2 Same data.** Each curve has the same shape and values as far as they can be read off the plot: position, slope, amplitude, period, number of points in a scatter. A visibly steeper, shifted or rescaled curve fails.
- **M3 Same axes.** Same axis type (linear or log) and an axis range that shows the same relevant region of the data.
- **M4 Axis labels.** Where the reference labels an axis, the student's axis is labelled with the same quantity (and unit, if the reference gives one).
- **M5 Curves identifiable.** Where the reference tells curves apart by a legend or labels, the student's curves are distinguishable and identified with the same meaning; the legend matches what is drawn.
- **M6 Same panels.** Every subplot of the reference exists with its content; the arrangement may differ.

These differences are **tolerated** and never fail a plot on their own:

- **T1 Style.** Colours, line styles, markers and line widths, as long as M5 holds.
- **T2 Mechanics.** Drawing order, sampling density (a smooth curve either way), figure size, resolution, fonts.
- **T3 Decoration.** Legend position, grid, tick spacing, small differences in axis padding.
- **T4 Wording.** Different wording, language, capitalisation or notation of title, axis labels and legend entries with the same meaning ("Zeit t [s]" and "time t (s)").
- **T5 Title.** A missing or differently worded title, if axis labels and legend identify the content.
- **T6 Random data.** A different draw of random data with the same statistics (spread, trend, convergence), unless the task fixes the seed.

When the task text explicitly requires a style (for example "plot the data as red circles"), that requirement counts like M1–M6. The visual labels of the evaluation set use the reference image only, so style requirements in the task text are not checked there.

How the judge applies the criteria

`plot_grader.tolerant_judge` passes a plot that is pixel-identical to the reference without a model call; on the evaluation set almost every pixel-identical plot was acceptable. Otherwise both of these must hold:

- The vision model, asked with the tolerant prompt (`pair_tolerant`, chat mode, images scaled to 75 %), says PASS. The prompt, not the model, sets the tolerance: the same model with a strict prompt failed most of the acceptable plots that the deterministic tests also fail. Chat mode is used because JEV was about as accurate but markedly slower.
- `tolerant_check` (`src/plot_grader/tolerant.py`) finds no content problem in the extracted plot data. It covers what the tolerant model alone lets through, mainly missing axis labels and rescaled data: at least as many panels as the reference (M6), axes the reference labels stay labelled in any wording (M4), every reference curve matched in any order and sampling within 5 % of its y range (M1/M2), and at least 80 % of the reference x data range visible (M3). Complex-valued data is compared by its real part, as matplotlib plots it. When the reference draws random numbers without a fixed seed (`unseeded_random`, criterion T6), the curve values are left to the model (`random_data=True`).

An unusable model reply gives ERROR unless the data checks already found a problem. The prompt and the check thresholds were tuned on the synthetic evaluation set, not on human labels of real submissions.

Computor integration

Quelle: https://github.com/computor-org/plot-grader/blob/main/docs/integration.md

How the plot judge (`plot_grader.tolerant_judge`) relates to the Computor test harness. Paths refer to the `computor-backend` repository on `main`.

How graphics tests run today

- The worker runs `computor-test python run -T test.yaml -s specification.yaml` without a shell and with a 300 s default timeout (`computor-backend/src/computor_backend/testing/backends.py`). It removes `API_TOKEN`, `TESTING_WORKER_TOKEN` and `COMPUTOR_*` from the environment, except the sandbox switches and the `COMPUTOR_JOB_*` resource limits, and leaves other variables alone.
- `computor-test` runs pytest on `testers/tests/python/test_class.py`. The harness seeds `random` and `numpy` with 1 and runs a `type: graphics` collection like every other test in the bounded, sandboxed job (`testers/executors/python.py`). Each subtest `name` is evaluated inside the job as `plt.<name>` and returned to the harness as data; the figure objects never reach the harness process.
- Every subtest is compared value by value (`test_entrypoint` in `test_class.py`, then `compare_variable_by_qualification` in `testers/tests/test_base.py`). The image is never compared. When `storeGraphicsArtifacts` is on, the job saves the open figures as PNG and the harness copies them, size-capped and without symlinks, to the artifact directory.
- Each subtest ends up in `outputDirectory/testSummary.json` with `result` and `resultMessage`, where `resultMessage` is the pytest failure or skip text. The worker stores that JSON as the result (`computor-backend/src/computor_backend/tasks/temporal_student_testing.py`), and the student sees the message. The worker zips `artifacts/` and uploads it through the API.
- Sandbox: student code runs under a Landlock filesystem allow-list and, where the host permits it, in a loopback-only network namespace (`computor-testing/ctexec/base.py`). The harness process itself keeps the worker container's network.

Hook: `qualification: plotJudge` (not on `main`)

`computor-testing` on `main` has no plot-judge hook. The proposed opt-in is one extra subtest in the graphics collection:

```
- type: graphics
  name: figure
  entryPoint: plot.py
  tests:
    - name: figure(1).axes[0].get_xlabel()   # existing tests stay as they are
    - name: figure(1)
      qualification: plotJudge
```

- Schema: one enum value, `QualificationEnum.plotJudge`, in `computor-types/src/computor_types/testing.py`, plus the same value in the generated `computor-web` test-yaml schema and types. The models use `extra="forbid"`, so no new keys are needed.

- Harness: a `plotJudge` subtest of a graphics collection sends the student's and the reference's figure as PNG, together with the extracted axes, labels, limits and line data, to `$PLOT_JUDGE_URL`. Because graphics run in the sandboxed job, the PNGs and plot data have to come from the job output, not from figure objects in the harness process.
  - A judge **FAIL** fails the subtest, and the model's findings and hint become `resultMessage`.
  - A **PASS** passes it.
  - No URL, a timeout (`$PLOT_JUDGE_TIMEOUT`, default 120 s), a network error or no verdict all SKIP the subtest. The student is never failed because the judge is missing.
- Service: `python -m plot_grader.judge_server --bind HOST:PORT` (this repo) answers `POST /judge` with `judge_plot`. It allows at most `--parallel` (default 8) model calls at once.

What it needs at runtime

1. **A reachable judge service.** `judge_server` must run on a host that can reach the vision model endpoint configured in `configs/vlm_endpoints.json` (and read its API key, if one is set), and the testing worker must reach that host. Set `PLOT_JUDGE_URL=http://<host>:<port>/judge` on the `temporal-worker-testing` service. The variable name avoids the `COMPUTOR_*` filter.
2. **Time.** One judge call takes a median of 25 s at 8 parallel requests, and more when the queue is full. The harness has 300 s in total (`backends.py`). Use `PLOT_JUDGE_TIMEOUT=90` and one `plotJudge` subtest per collection.
3. **Load.** The endpoint is shared and should see at most 8 parallel calls. Every submission test run triggers one call. Worker concurrency and `--parallel` of the service bound this.
4. **Reference image.** Only the judge service receives the reference figure. Unrelated to the hook, when artifacts are stored, the worker already uploads the reference's figure PNGs next to the student's images. Check whether students can download these (`computor-backend/src/computor_backend/api/results.py`).

Limits

- **Trust.** The plot data and figures come from the job that runs the student code, so student code can fake them. This is true for all graphics tests.
- **Figures.** The hook judges `figure(n)` objects one by one. Stacking all figures, as `render.py` does, is not done.
- **Language.** The model's differences and hint are in English. The bilingual text in `scripts/judge_course.py` covers the fixed parts only.

Mitarbeit am Projekt

Contributing to computor-backend

Quelle: https://github.com/computor-org/computor-backend/blob/main/CONTRIBUTING.md

This repository holds the whole platform: the Python packages (`computor-types`, `computor-client`, `computor-backend`), the grading framework (`computor-testing`), the web frontend (`computor-web`) and the operations tooling (`ops/`).

Set up

Follow docs/development.md for the backend and web frontend. For the grading framework see computor-testing/docs/DEVELOPER_GUIDE.md; for Docker, environment variables and deployment see ops/README.md and ops/docs/.

Install the pre-commit hook once (secret scan, import guard, client-drift check, web styling ratchet), see scripts/git-hooks/README.md:

```
bash scripts/git-hooks/install-hooks.sh
```

Make a change

1. Branch from `main` (`feat/…`, `fix/…`, `refactor/…`, `docs/…`).
2. After changing DTOs or models, regenerate with `bash generate.sh all`.
3. Run the checks for what you touched:

   ```
   ./computor.sh test --unit                      # backend unit tests
   ./computor.sh test --integration               # backend integration tests (dev stack
   up)

   (cd computor-web && yarn lint && yarn test:public-learning && yarn build)
   (cd computor-web && yarn test:e2e)             # Playwright, backend mocked
   (cd computor-testing && python -m pytest tests) # grading framework
   ```

4. Open a pull request against `main`. Its title becomes the commit subject, so write it as a short sentence saying what the change does.

Merging

Pull requests are merged through the merge queue as one squashed commit. The required CI check (`ci-gate`) runs in the queue, not on the pull request, so a pull request shows green before the queue has tested it.

Licence

By contributing you agree that your code is licensed under the MIT licence (LICENSE) and your documentation under CC BY 4.0 (LICENSE-DOCS).