eSim Summer Fellowship 2026

On

Custom SPICE Model Injection Pipeline for the eSim-Cloud Simulation Engine

Submitted by

Parth Bhanti
Department of Computer Science and Engineering
VIT Bhopal University

Under the guidance of

Prof. Prabhu Ramachandran
Principal Investigator
Department of Aerospace Engineering
Indian Institute of Technology Bombay

July 2026

Acknowledgment

I express my sincere gratitude to Prof. Prabhu Ramachandran for providing me with the opportunity to be part of the FOSSEE internship program and for his continued efforts in promoting open-source engineering tool development. His leadership and vision have been instrumental in fostering meaningful student participation in the open-source ecosystem.

I also acknowledge Prof. Kannan M. Moudgalya for his foundational role in establishing and strengthening the FOSSEE initiative. His contributions to open-source education and the development of the FOSSEE fellowship framework have been pivotal in creating the academic and organisational platform through which this internship was undertaken.

My sincere appreciation extends to my mentor, Sumanto Kar, for his continual support, technical guidance, and encouragement throughout the duration of this project. His insights and feedback played a key role in refining ideas, overcoming challenges, and ensuring timely completion of the tasks assigned to me.

I would also like to thank my internal mentors, Mr. Varad Patil and Ms. Shanthi Priya K for their valuable guidance, coordination, and technical inputs during the internship. Their mentorship contributed significantly to the clarity, progress, and successful execution of the work.

This internship has been an enriching learning experience, allowing me to work directly inside a production Django/Celery/ngspice execution pipeline used by students and educators across the country, to reason carefully about the security surface of a system that executes user-supplied text as a scientific simulation input, and to design and ship a whitelist SPICE model sanitizer, an execution-engine injection pipeline, and a REST API end-to-end from an open GitHub issue to a merged pull request on `FOSSEE/eSim-Cloud`. The knowledge acquired during this period will undoubtedly support my future academic and professional pursuits.

I would also like to thank the entire FOSSEE team for their coordination, assistance, and timely interactions at various stages of this work. Their collective efforts ensured smooth workflow, resource accessibility, and effective project execution.

Contents

Chapter 1

Introduction

1.1 Background

Access to capable, low-cost Electronic Design Automation (EDA) tooling is a recurring bottleneck in engineering education across India. The FOSSEE (Free/Libre and Open Source Software for Education) project at IIT Bombay, operating under the National Mission on Education through Information and Communication Technology (NMEICT), Ministry of Education, Government of India, addresses this gap by developing and maintaining open-source alternatives to proprietary simulation and design software. **eSim** is FOSSEE's flagship EDA tool, integrating KiCad, ngspice, GHDL, and Makerchip into a single desktop application for schematic capture, SPICE simulation, and PCB design.

eSim-Cloud (also referred to as "eSim on Cloud") is the browser-based counterpart to the desktop eSim application. It reimplements the schematic-capture-to-simulation workflow as a web application: an Angular/React frontend built around the `mxgraph` graph-drawing library lets a user drag components onto a schematic canvas, wire them together, and submit the resulting netlist to a Django REST backend, which hands the netlist to ngspice through a Celery task queue and streams the parsed waveform data back to the browser. Because it requires no local installation, eSim-Cloud is widely used in classrooms, MOOCs, and LTI-embedded course pages where installing a native EDA suite on every student machine is impractical.

This report documents the design, implementation, and verification of a single, cohesive backend feature merged into `FOSSEE/eSim-Cloud` during the June 2026 FOSSEE Summer Fellowship: a pipeline that lets an authenticated user upload a custom SPICE component — typically a `.subckt` definition for a part that is not in eSim-Cloud's built-in component library — and have it safely and automatically merged into their simulation netlist at execution time.

1.2 Motivation: Issue #539

The work reported here traces directly to a user-filed issue on the `frg-fossee/eSim-Cloud` repository (Issue #539), in which a student attempting to simulate an Arduino project asked whether it was possible to add a circuit component that did not exist in the provided component set — specifically, an SCD41 CO_2/temperature/humidity sensor — and use it directly on the simulator without waiting for the FOSSEE team to add official library support for that part.

At the time the issue was filed, eSim-Cloud's simulation endpoint accepted exactly one artifact from the user: a monolithic `.cir` netlist file, generated client-side from the schematic and passed unmodified to ngspice. There was no mechanism by which a user's own `.subckt` or `.model` text could be attached to that netlist. Any part not already present in the shared component library was simply unusable, regardless of whether the user had a correct SPICE model for it in hand.

1.3 Objectives of the Project

The objective of this fellowship project was to design and ship the complete backend pipeline needed to close that gap, subject to one non-negotiable constraint: *the pipeline must not introduce a way for user-supplied text to escalate into file-system or command-execution access on FOSSEE's shared infrastructure.* Concretely, the project set out to:

- Introduce a persistent, per-user data model for storing uploaded SPICE model definitions, with a clear separation between the original uploaded text and a validated, execution-safe form of it.
- Design and implement a strict whitelist sanitizer that accepts only a well-defined subset of SPICE syntax and rejects everything else by default, including several non-obvious evasion techniques specific to the SPICE line-continuation and comment syntax.
- Extend the existing ngspice execution adapter (`ExecNetlist`) so that, at simulation time, the sanitized definitions of any custom models a user has attached are merged into their netlist *in memory*, without writing user-controlled `.include` targets to disk.
- Patch a real compatibility problem this exposed: the production ngspice v31 binary does not, by default, correctly parse the PSPICE/LTspice-style behavioural syntax that many third-party `.subckt` files (including hobbyist sensor models such as the SCD41) are written in.
- Expose the whole pipeline through a small, authenticated REST API (upload, list, retrieve/delete, re-validate) so that a future frontend "Model Builder" UI can be built against a stable contract.
- Cover the new code with an exhaustive automated test suite, with particular emphasis on adversarial inputs designed to defeat the sanitizer.

1.4 Methodology

The implementation followed an iterative, security-first engineering process. Section 4.2 of Chapter 4 describes this process as a flowchart; in summary, each stage of the pipeline (data model, sanitizer, execution adapter, REST layer) was designed against an explicit threat model before any code was written, implemented with accompanying unit tests written alongside the implementation rather than after it, and iterated on whenever a test exposed either a security gap or an ngspice compatibility issue. The final deliverable is a single pull request, `FOSSEE/eSim-Cloud#4` (`feature/539-custom-spice-models`), merged into `master` on June 6, 2026, comprising two commits, ten modified or added Python files, one new Django migration, and seventy new automated test cases.

1.5 Organisation of this Report

Chapter 2 surveys the technologies this project builds on — SPICE/ngspice, Django REST Framework, Celery/Redis, and the relevant class of web-application injection vulnerabilities. Chapter 3 describes eSim-Cloud's existing system architecture and the netlist-execution workflow this project extends. Chapter 4 restates the problem being solved and the approach taken. Chapters 5 through 9 form the technical core of the report, walking through the data model, the whitelist sanitizer, the execution-engine changes, and the REST API in the order they were built. Chapter 10 documents the test suite. Chapter 11 discusses the pipeline end-to-end with a worked example. Chapter 12 concludes and outlines future work, and Chapter 13 summarises the contribution in tabular form for quick reference.

Chapter 2

Literature Survey

This project sits at the intersection of circuit simulation tooling and web application security. This chapter briefly surveys the four technical areas that most directly informed the design decisions described later in this report.

2.1 SPICE and ngspice

SPICE (Simulation Program with Integrated Circuit Emphasis) is a netlist-driven language and simulation methodology for analog and mixed-signal circuits, originally developed at UC Berkeley. A SPICE *netlist* is a plain-text description of a circuit: component instance lines (resistors, capacitors, transistors, sources, and subcircuit instances), *dot directives* that configure analyses and models (`.tran`, `.ac`, `.model`, `.subckt`), and an optional `.control` block containing an imperative scripting language used to drive the simulator interactively — set options, run analyses, and dump or plot results.

ngspice is the open-source SPICE engine eSim and eSim-Cloud use to execute netlists. Two properties of ngspice are directly relevant to this project:

1. The `.control` block is not a sandboxed scripting language. Commands such as `shell`, `system`, `source` and `.include` exist specifically so that a netlist author can invoke external programs, read files from disk, or pull in further netlist text from another file path. This is a deliberate and useful feature for a trusted, locally-run desktop tool; it is a direct arbitrary file-read/command-execution primitive if a service ever executes untrusted, remotely-supplied `.control` text on shared infrastructure.
2. ngspice's parser has evolved a distinct *behavioural modelling* dialect (activated with `set ngbehavior=ps` or similar compatibility flags) to interpret PSPICE/LTspice-style constructs — `TABLE`, `LAPLACE`, `VALUE`, and conditional expressions inside behavioural sources (`B` elements). Third-party and hobbyist `.subckt` files, which are very often originally written for LTspice or PSPICE rather than ngspice, frequently rely on this dialect. Without the compatibility flag, ngspice v31 — the version deployed in eSim-Cloud's production containers — either misparses or rejects such models.

2.2 Django and Django REST Framework

eSim-Cloud's backend (`esim-cloud-backend/`) is a Django project. Django's ORM maps Python model classes to relational tables and migrations; Django REST Framework (DRF) layers serializer classes (validation and JSON marshalling) and `APIView` classes (HTTP verb dispatch, authentication, and permission checks) on top of that. Three DRF idioms are used extensively in this project:

- **Serializer-level validation.** DRF serializers support both per-field validators (`validate_<fieldname>`) and a whole-object `validate()` hook. This project uses the latter to run the whitelist sanitizer as part of request validation itself, so that a malformed or malicious model upload never reaches a view or the database in an unvalidated state.

- **Permission classes.** `IsAuthenticated` and `AllowAny` gate every endpoint. All four new SPICE-model endpoints require authentication and additionally scope every query to `request.user`, so that one user's models are never visible to, retrievable by, or deletable by another user, even by UUID guessing.
- **Django admin.** Django's built-in admin site is repurposed here as a lightweight moderation console: a `SpiceModelAdmin` class with bulk actions lets a FOSSEE maintainer approve or revoke a model's visibility in a (future) public gallery, without granting the maintainer any special execution privilege that the owner does not already have.

2.3 Celery and Redis

Simulation runs are not executed synchronously inside the HTTP request that uploads a netlist; ngspice execution time is unbounded from the server's point of view, and a slow or hung simulation must not block the web worker serving other users. eSim-Cloud instead uses **Celery**, a distributed task queue, with **Redis** as the message broker and result backend. The Django view enqueues a task (`process_task`) and immediately returns a task ID; the browser polls a status endpoint (`CeleryResultView`) until the task transitions out of the `PENDING/PROGRESS` states. This asynchronous boundary is significant for this project because it is precisely the boundary the `custom_model_ids` parameter has to cross: the identifiers of the models a user wants injected are captured in the synchronous view, but the actual database fetch and text injection happen later, inside the Celery worker process, immediately before the ngspice subprocess is spawned.

2.4 Injection Vulnerabilities and Whitelist Sanitization

The core security engineering problem this project solves is a textbook instance of a broader class: a system that must execute a domain-specific-language (DSL) input — here, a SPICE netlist fragment — that is partially supplied by an untrusted party, on infrastructure shared with other users. The same shape of problem appears as SQL injection (a DSL for querying a database), server-side template injection, and classic shell command injection.

The security literature on this class of problem consistently favours **default-deny whitelisting** over **blacklisting**: rather than enumerating known-bad patterns and rejecting them, a whitelist sanitizer enumerates known-good constructs and rejects everything that does not match, including constructs the designer never anticipated. A blacklist approach to this specific problem — for instance, simply rejecting the substring `".control"` — is fragile against *evasion*: SPICE's own line-continuation syntax (a line beginning with `+` is a continuation of the previous logical line) lets an attacker split a blocked token across two physical lines (`".con"` / `"+trol"`) so that neither line, considered in isolation, contains the blocked substring. A correct sanitizer must therefore resolve the DSL's own structural syntax (continuations, in this case) into logical lines *before* it applies any keyword check — this exact resolve-then-validate ordering is the central design decision described in Chapter 6.

A second, related class of evasion is **comment-stripping order**. SPICE uses `$` as an inline comment character, and a naive sanitizer that strips everything after `$` *before* scanning for dangerous characters will silently discard a payload such as `$(cat /etc/passwd)`, treating it as a harmless comment rather than as a shell-metacharacter sequence that should be rejected outright. The correct ordering — scan the raw line for dangerous metacharacters first, strip comments second — is likewise discussed in Chapter 6.

Chapter 3

System Architecture of eSim-Cloud

Before describing the feature itself, it is necessary to understand the system it was built into. This chapter summarises eSim-Cloud's overall architecture and, more specifically, the netlist-execution workflow that existed prior to Issue #539, since the new pipeline is an extension of – not a replacement for – that workflow.

3.1 Layered Overview

eSim-Cloud is organised into four layers, summarised in Figure 3.1 and Table 3.1.

Table 3.1: eSim-Cloud technology stack by layer

Layer	**Technologies**
Frontend	React with the `mxgraph` graph-drawing library (schematic editor, eSim simulator UI), Angular (Arduino-on-Cloud workspace), RaphaelJS, and the AVR8js in-browser AVR core for Arduino code simulation.
Middleware	Django, Django REST Framework, Celery (distributed task queue), Redis (broker / result backend).
Simulation backend	ngspice (analog/digital/mixed-signal SPICE simulation), an Arduino/AVR toolchain for sketch compilation.
Database	PostgreSQL / MySQL (relational data – users, tasks, simulations, saved schematics), MongoDB (document storage for certain save formats).
Production	nginx (reverse proxy / static assets), Docker & Docker Compose (container orchestration, horizontally-scaled `django` and `celery` services).

The work described in this report lives entirely inside a single Django app, `simulationAPI`, within the middleware layer. No frontend, database engine, or deployment change was required to ship it; the frontend integration points are exposed purely as a documented REST contract (Chapter 8) for a future "Model Builder" UI to consume.

3.2 The Pre-Existing Netlist Execution Workflow

Prior to this project, submitting and running a circuit on eSim-Cloud followed the sequence shown in Figure 3.2:

1. The frontend serialises the drawn schematic into a SPICE netlist (component lines, an analysis directive such as `.tran`, and a `.control` block that runs the analysis and writes results to `data.txt`) and `POST`s it as a file to `/api/simulation/upload`.
2. `NetlistUploader.post()` validates the upload with `TaskSerializer`, which creates a `Task` row and an associated `spiceFile` row pointing at the stored netlist on disk, and separately

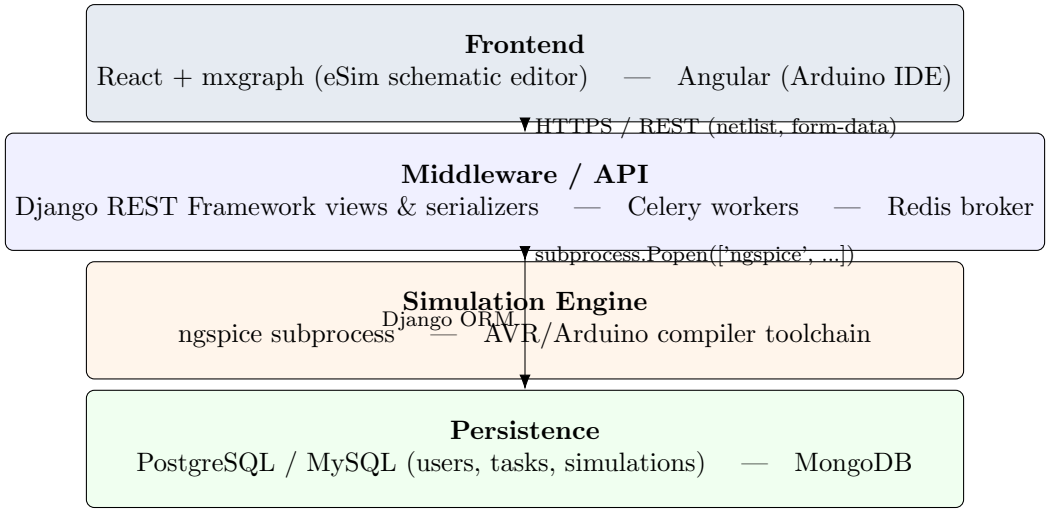

Figure 3.1: Layered architecture of eSim-Cloud. The feature described in this report is implemented entirely within the middleware layer – specifically in the `simulationAPI` Django app – and reaches into the simulation engine layer only through the existing `ExecNetlist` adapter.

persists a `simulation` row holding a copy of the raw netlist text via `saveNetlistDB()`.

3. The view enqueues `process_task` on Celery with the task's UUID and returns the UUID immediately; Celery workers pick the job up from the Redis queue independently of the web worker that accepted the request.
4. Inside the worker, `process_task` resolves the `spiceFile` for that task and calls `ngspice_helper.ExecNetlist(filepath, file_id)`, which `chdir`s into a fresh, UUID-named scratch directory under `MEDIA_ROOT` and runs `ngspice -ab <netlist>` as a subprocess.
5. `parse.py` converts ngspice's raw `data.txt` output into the JSON chart format the frontend understands, and the result is stored back onto the `simulation` row and made available through `CeleryResultView`, which the frontend polls using the task UUID until the state leaves `PENDING/PROGRESS`.
6. The scratch directory and the original upload are deleted in a `finally` block regardless of success or failure.

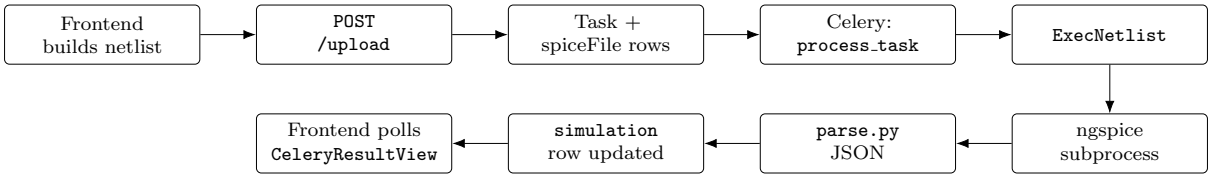

Figure 3.2: The pre-existing (legacy) netlist execution workflow. Every stage shown here is unchanged by this project except `ExecNetlist`, which gained an optional injection step immediately before the ngspice subprocess is spawned (Chapter 7).

3.3 What Was Missing

Nothing in this workflow gave a user any way to attach reusable component definitions to a netlist. The netlist file was the single, atomic unit of input; if a required `.subckt` or `.model` was not already baked into the netlist text the frontend generated (which in turn meant it had to already be part of eSim-Cloud's bundled component library), the simulation would simply

fail with an undefined-model error from ngspice. Closing this gap safely – without turning the upload endpoint into a generic remote code execution primitive – is the subject of the rest of this report.

Chapter 4

Problem Statement

4.1 Problem Statement

The objective of this project is to design and implement the complete backend pipeline required to let an authenticated eSim-Cloud user upload a custom SPICE component definition – a `.subckt`, `.lib`, or `.model` fragment for a part not present in the platform's built-in component library (Issue #539) – and have that definition safely merged into their netlist at simulation time, **without** introducing a Local File Inclusion (LFI), path-traversal, or shell/command-injection vector on FOSSEE's shared production infrastructure. The resulting models must also be storable and manageable (listed, retrieved, deleted, re-validated) through a REST API suitable for a future frontend "Model Builder", and the entire pipeline must be exhaustively covered by automated tests, with particular emphasis on adversarial inputs that specifically target the sanitizer's parsing assumptions.

4.2 Approach

The implementation followed the six-stage roadmap shown in Figure 4.1. Security review was treated as a gate at every stage rather than a final audit step: the sanitizer's design was adversarially tested against a running list of evasion techniques *while it was being written*, and the execution-adapter change was reviewed specifically for whether it reintroduced any of the risks the sanitizer had just closed off (for instance, whether a temporary `.lib` file written to disk for `.include` would undo the sanitizer's guarantees – which is exactly why inline in-memory injection was chosen instead, see Chapter 7).

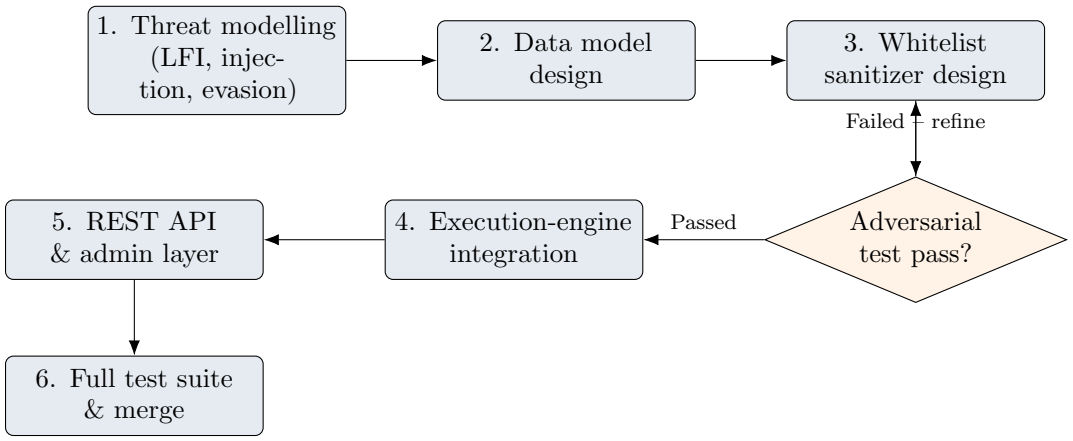

Figure 4.1: Roadmap followed for implementing the custom SPICE model injection pipeline. Stage 3 (sanitizer design) and its adversarial test gate were revisited twice during development: once after discovering the line-continuation evasion path, and once after discovering the comment-stripping evasion path (Chapter 6).

4.2.1 Stage 1 – Threat Modelling

Before any code was written, the specific ways a malicious or careless netlist author could abuse an upload-and-inject feature were enumerated: reading arbitrary files via `.include`/`.lib` path arguments; spawning shell commands via ngspice's `shell`/`system` directives inside a `.control` block; breaking simulation isolation between users by writing to a shared or predictable filesystem path; and denial-of-service via unbounded upload size. Each of these threats maps directly onto a specific design decision documented in Chapters 5–7.

4.2.2 Stage 2 – Data Model Design

A new `SpiceModel` table was designed with owner-scoped uniqueness and a deliberate separation between the immutable, as-uploaded `raw_content` and the validated `sanitized_content` that is the only thing ever passed to ngspice (Chapter 5).

4.2.3 Stage 3 – Whitelist Sanitizer Design

A default-deny parser (`spice_model_parser.py`) was built that resolves SPICE's own line-continuation syntax before checking for blocked directives, checks for shell metacharacters before stripping `$` comments, and validates `.subckt`/`.ends` structural pairing (Chapter 6). This stage's exit criterion was a green run of an adversarial test suite specifically targeting known evasion techniques.

4.2.4 Stage 4 – Execution-Engine Integration

`ngspice_helper.ExecNetlist` was extended to accept an optional `model_ids` argument, fetch the corresponding sanitized content, and splice it into the netlist text in memory – immediately before the `.control` block – before writing a single augmented netlist file and invoking ngspice exactly as before (Chapter 7). The same change also patches a real ngspice v31 compatibility gap by injecting `set ngbehavior=ps`.

4.2.5 Stage 5 – REST API and Admin Layer

Four endpoints (upload, list, detail/delete, re-validate) were added under `/api/simulation/models/`, backed by dedicated serializers, and a Django admin console was added so FOSSEE maintainers can moderate which uploaded models, if any, are surfaced in a future public gallery (Chapter 8).

4.2.6 Stage 6 – Full Test Suite and Merge

Seventy automated tests across ten `TestCase` classes were written, covering the sanitizer's accept/reject behaviour, structural validation, continuation resolution, the injection helpers, and full API-contract and tenant-isolation behaviour (Chapter 10). The pull request was opened on June 3, 2026 and merged into `FOSSEE/eSim-Cloud:master` on June 6, 2026.

Chapter 5

Data Model Design

5.1 The `SpiceModel` Model

The heart of the feature is a single new Django model, `SpiceModel`, added to `esim-cloud-backend/simulationAPI/models.py` alongside the pre-existing `Task`, `spiceFile`, and `simulation` models. Its full definition is shown in Listing 5.1.

```
class SpiceModel(models.Model):
    """
    Stores user-uploaded custom SPICE model definitions (.subckt, .lib, .model).

    Security model:
    - raw_content: original uploaded text, preserved for audit trail
    - sanitized_content: output of spice_model_parser whitelist sanitizer,
      this is the ONLY content that ever gets injected into a netlist
    - is_approved: gates PUBLIC GALLERY publishing only; models are
      immediately usable by their owner for private simulations
    """

    MODEL_TYPES = [
        ('subckt', '.subckt subcircuit definition'),
        ('lib', '.lib library file'),
        ('model', '.model device directive'),
    ]

    id = models.UUIDField(
        primary_key=True, default=uuid.uuid4, editable=False)
    owner = models.ForeignKey(
        to=get_user_model(), on_delete=models.CASCADE,
        related_name='spice_models')
    name = models.CharField(max_length=100)
    model_type = models.CharField(max_length=10, choices=MODEL_TYPES)
    raw_content = models.TextField(
        help_text='Original uploaded content -- never executed directly')
    sanitized_content = models.TextField(
        help_text='Whitelist-sanitized content -- injected into netlists')
    is_approved = models.BooleanField(
        default=False,
        help_text='Gates public gallery publishing; personal use is always allowed')
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    description = models.TextField(blank=True, default='')
    pin_count = models.IntegerField(null=True, blank=True)
    subckt_name = models.CharField(
        max_length=100, blank=True, default='',
        help_text='Extracted .subckt name for netlist Xref matching')

    class Meta:
        unique_together = ('owner', 'name')
        ordering = ['-created_at']

    def __str__(self):
        return '{} ({})'.format(self.name, self.model_type)
```

Listing 5.1: The `SpiceModel` data model (`simulationAPI/models.py`)

5.2 Design Rationale

5.2.1 Dual-Content Storage

The single most important design decision in the entire feature is that *two* copies of the uploaded text are stored, never one:

- `raw_content` is exactly what the user uploaded, byte for byte (modulo UTF-8 decoding). It is never passed to ngspice and never interpolated into a netlist. Its sole purpose is audit: if the sanitizer's rules change in the future (as they did mid-project, see Chapter 6), every previously uploaded model can be re-sanitized against the new rules from its original source, via the `/validate` endpoint (Section 8.3.1), without asking the user to re-upload anything.
- `sanitized_content` is the output of `spice_model_parser.sanitize_spice_model()` – the *only* field that `ngspice_helper.inject_custom_models` (Chapter 7) is permitted to read when building an augmented netlist.

This separation means a defect discovered later in the sanitizer's rule set can never retroactively "leak" through models that were approved under an older, looser rule set, because the field that is actually executed is always regenerated from source rather than trusted as a one-time gate.

5.2.2 Owner-Scoped Uniqueness

`unique_together =('owner', 'name')` lets two different users each have a model named, for example, `"SCD41_Sensor"`, while preventing one user from accidentally shadowing their own previous upload of the same name. Every query against this table in the view layer (Chapter 8) is additionally filtered by `owner=request.user`, so the uniqueness constraint and the query scoping enforce the same invariant at two independent layers (database constraint and application logic).

5.2.3 `is_approved` Is Not a Security Boundary

`is_approved` is a *publishing* flag, not an execution permission. An unapproved model is fully usable by its owner in their own simulations the moment it passes the sanitizer; approval only controls whether the model can later be surfaced in a shared, cross-user public gallery. This distinction matters because it means the sanitizer — not manual moderator review — is the actual security boundary for execution. A human reviewer inspecting models before they are ever run would not scale to a self-service upload feature; the whitelist sanitizer is what makes self-service safe.

5.2.4 Metadata for Netlist Cross-Referencing

`subckt_name` and `pin_count` are extracted once at upload time (and refreshed at re-validation time) by `spice_model_parser.extract_metadata()`. They exist so that a future frontend component-picker can display, list, and pin-count-match a user's uploaded models against schematic component instances without having to re-parse `sanitized_content` on every request.

5.3 Database Migration

The corresponding Django migration, auto-generated and then reviewed by hand, is shown in Listing 5.2.

```
1 class Migration(migrations.Migration):
2
3     dependencies = [
```

```
        migrations.swappable_dependency(settings.AUTH_USER_MODEL),
        ('simulationAPI', '0001_initial'),
    ]

    operations = [
        migrations.CreateModel(
            name='SpiceModel',
            fields=[
                ('id', models.UUIDField(default=uuid.uuid4, editable=False,
                    primary_key=True, serialize=False)),
                ('name', models.CharField(max_length=100)),
                ('model_type', models.CharField(choices=[
                    ('subckt', '.subckt subcircuit definition'),
                    ('lib', '.lib library file'),
                    ('model', '.model device directive')], max_length=10)),
                ('raw_content', models.TextField(help_text=(
                    'Original uploaded content -- never executed directly'))),
                ('sanitized_content', models.TextField(help_text=(
                    'Whitelist-sanitized content -- injected into netlists'))),
                ('is_approved', models.BooleanField(default=False, help_text=(
                    'Gates public gallery publishing; personal use is '
                    'always allowed'))),
                ('created_at', models.DateTimeField(auto_now_add=True)),
                ('updated_at', models.DateTimeField(auto_now=True)),
                ('description', models.TextField(blank=True, default='')),
                ('pin_count', models.IntegerField(blank=True, null=True)),
                ('subckt_name', models.CharField(blank=True, default='',
                    help_text='Extracted .subckt name for netlist Xref '
                    'matching', max_length=100)),
                ('owner', models.ForeignKey(
                    on_delete=django.db.models.deletion.CASCADE,
                    related_name='spice_models', to=settings.AUTH_USER_MODEL)),
            ],
            options={
                'ordering': ['-created_at'],
                'unique_together': {('owner', 'name')},
            },
        ),
    ]
```

Listing 5.2: `migrations/0002_spicemodel.py`

Table 5.1 summarises every field on the model, its purpose, and whether it is ever exposed to an execution path.

Table 5.1: `SpiceModel` fields and their trust boundary

Field	Purpose	Reaches ngspice?
`id`	Primary key (UUID), also the public model identifier used in `custom_model_ids`	No
`owner`	FK to the uploading user; scopes every query	No
`name`	Human-readable, owner-unique label	No
`model_type`	`subckt` / `lib` / `model`	No
`raw_content`	Verbatim upload, audit trail	**Never**
`sanitized_content`	Whitelist-parser output	**Yes – only this field**
`is_approved`	Public gallery publishing gate	No
`created_at` / `updated_at`	Bookkeeping	No
`description`	Optional free-text note from the uploader	No
`pin_count` / `subckt_name`	Extracted metadata for UI cross-referencing	No

Chapter 6

Security Architecture: The Whitelist Sanitizer

This chapter documents `simulationAPI/helpers/spice_model_parser.py`, the module that is, in a precise sense, the entire security boundary of this feature: it is the only code path standing between arbitrary user-uploaded text and a subprocess call to ngspice running on shared FOSSEE infrastructure.

6.1 Threat Model

Table 6.1 restates the threats identified in Stage 1 of the approach (Section 4.2) alongside the specific sanitizer mechanism that closes each one.

Table 6.1: Threats and corresponding sanitizer mitigations

Threat	**Mitigation**
Arbitrary file read via `.include`/`.lib "path"`	Both directives are on the blocked-directive list; no user-supplied `.include` ever reaches ngspice (Section 6.4).
Arbitrary command execution via `shell` / `system` / `source` inside a `.control` block	`.control` itself is blocked outright, and the bare commands are *additionally* blocked even outside a `.control` context, in case ngspice ever accepts them at top level (Section 6.4).
Evasion of the above via SPICE line continuations (`".con"` / `"+trol"`)	Continuations are resolved into full logical lines *before* any keyword check runs (Section 6.5).
Evasion via shell metacharacters hidden inside a `$`-comment (`$(cat /etc/passwd)`)	Metacharacter scanning runs on the *raw* line, before comment stripping (Section 6.6).
Structurally broken subcircuits (unclosed/mismatched `.subckt`) that could produce a malformed netlist ngspice parses unpredictably	Explicit `.subckt`/`.ends` pairing and nesting validation (Section 6.7).
Unbounded upload size (denial of service)	512 KB hard limit enforced at both the serializer layer and, defense-in-depth, inside the sanitizer itself.
Unknown/unanticipated directives	Default-deny: anything not on the explicit allow-list is rejected, rather than anything not on a block-list being accepted.

6.2 Design Principle: Whitelist, Not Blacklist

The module's docstring states its governing principle directly:

> *"This is the ONLY gate between user-uploaded content and the ngspice execution engine. It operates as a WHITELIST parser: only explicitly allowed SPICE constructs pass through. Everything else is rejected."*

Concretely, every non-blank, non-comment logical line in an uploaded model must match one of exactly two categories to be accepted: a component-instance line whose first token begins with one of a fixed set of SPICE element-prefix letters (`R C L V I D E Q M J X K T W S F G B`), or a dot-directive drawn from a fixed allow-list (`.subckt`, `.ends`, `.model`, `.param`, `.func`, `.global`, `.tran`, `.ac`, `.dc`, `.ic`, `.nodeset`, `.temp`, `.meas`/`.measure`, `.step`, `.print`, `.plot`, `.probe`, `.width`, `.end`). Anything else — including a dot-directive that is simply not on that list — is rejected, even if it is not individually known to be dangerous. This is the defining property of a whitelist: the designer does not need to have anticipated every possible attack for the sanitizer to reject it.

6.3 Pipeline Overview

`sanitize_spice_model()` is the single public entry point and runs the four ordered stages shown in Figure 6.1. Crucially, stage order is itself a security property: reversing stages 1 and 2 (validating before resolving continuations) reopens the exact evasion this design closes.

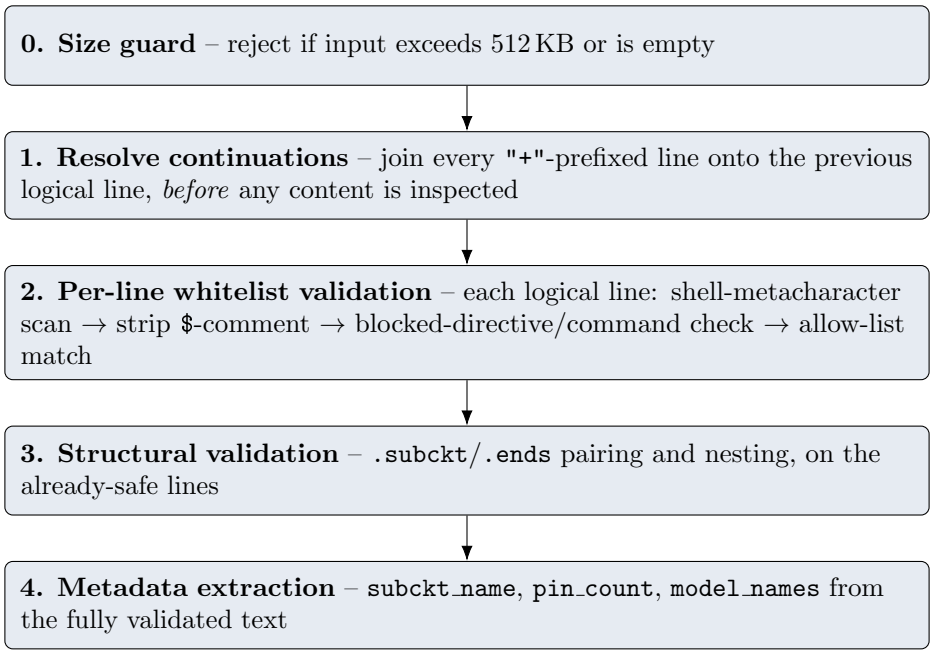

Figure 6.1: The four ordered stages of `sanitize_spice_model()`. Any error at stage 2 or stage 3 aborts the entire pass – *no partial sanitized content is ever returned*, since silently dropping only the offending lines could still leave a semantically broken or attacker-steered model.

6.4 Blocked and Allowed Constructs

The regular expressions that encode the block-list and allow-list are shown in Listing 6.1. Note that the blocked-directive and blocked-command patterns exist *in addition to* the allow-list, not instead of it – they exist primarily to produce a clear, specific error message ("Blocked

directive: .control") rather than the generic "does not match any allowed construct" message a bare whitelist miss would otherwise produce, which materially helps a legitimate user debug a rejected upload.

```
MAX_MODEL_CHARS = 524288  # 512 * 1024

# BLOCKED DIRECTIVES -- checked against fully-resolved logical lines.
BLOCKED_DIRECTIVES_RE = re.compile(
    r'^\s*\.'
    r'(?:control|endc|include|lib\s+[\"\']|options|save|op\b|'
    r'four|sens|disto|noise|pz|tf\b)',
    re.IGNORECASE
)

# Standalone dangerous commands (not dot-prefixed)
BLOCKED_COMMANDS_RE = re.compile(
    r'^\s*(?:shell|system|set\s|unset\s|source\s|cd\s|quit|exit)',
    re.IGNORECASE
)

# Shell metacharacters that should NEVER appear in component values.
SHELL_METACHAR_RE = re.compile(
    r'[|;`]'
    r'|\$\('
    r'|\$\{'
    r'|>\s*/'       # output redirection to absolute path
    r'|<\s*/'       # input redirection from absolute path
)

# Component instance prefixes: R,C,L,V,I,D,E,Q,M,J,X,K,T,W,S,F,G,H,B
COMPONENT_PREFIX_RE = re.compile(
    r'^[RCLVIDEQMJXKTWSFGHB]\w*\s',
    re.IGNORECASE
)

# Allowed dot-directives
ALLOWED_DOT_DIRECTIVES_RE = re.compile(
    r'^\s*\.'
    r'(?:subckt|ends|model|param|func|global|tran|ac|dc|'
    r'ic|nodeset|temp|meas|measure|step|print|plot|probe|width|'
    r'end)\b',
    re.IGNORECASE
)

COMMENT_RE = re.compile(r'^\s*\*')
BLANK_RE = re.compile(r'^\s*$')
```

Listing 6.1: Block-list and allow-list regular expressions (`spice_model_parser.py`)

A few specific choices in Listing 6.1 are worth calling out:

- `lib\s+[\"\']` only blocks `.lib` when it is followed by a quoted path argument (a file reference), which is the dangerous form; a bare `.lib` token used as a directive name elsewhere would not match this pattern – narrowing the block to the specific dangerous usage rather than the token in general.
- `op\b`, `tf\b` and other analysis directives (`.four`, `.sens`, `.disto`, `.noise`, `.pz`) are blocked not because they are unsafe in themselves, but because this feature's scope is *component definitions*, not analysis control – a `.subckt` upload has no legitimate reason to also specify how the simulation is analysed, and allowing it would let an uploaded "component" silently override the schematic-driven analysis type.
- `COMPONENT_PREFIX_RE` matches a single leading letter from the fixed SPICE element set followed by any word characters and then whitespace – so `X1`, `R47`, and `B_pwm` all match, but a line beginning with an unrelated word does not.

6.5 Resolving Line Continuations Before Validation

SPICE's continuation syntax lets a long line be wrapped across several physical lines: any line whose first non-whitespace character is `+` is treated as a continuation of the previous logical line, with the `+` and the whitespace following it discarded. This is legitimate, common syntax for long `.subckt` parameter lists — and it is also, unless handled first, a complete bypass of any keyword check that operates on physical lines rather than logical ones.

```
def _resolve_continuations(raw_lines):
    """
    Resolve SPICE line continuations: a line starting with '+' is joined
    to the previous logical line. Returns a list of logical lines.

    Example:
        [".sub", "+ckt FOO a b", "R1 a b 1k"]
        -> [".subckt FOO a b", "R1 a b 1k"]

    This is CRITICAL for security: without this, ".con\n+trol" would
    bypass the .control block check.
    """
    logical = []
    for raw_line in raw_lines:
        stripped = raw_line.lstrip()
        if stripped.startswith('+') and logical:
            continuation_text = stripped[1:].lstrip()
            logical[-1] = logical[-1] + ' ' + continuation_text
        else:
            logical.append(raw_line)
    return logical
```

Listing 6.2: Continuation resolution (`spice_model_parser.py`)

`sanitize_spice_model()` calls `_resolve_continuations()` immediately after normalising line endings and *before* the per-line whitelist loop runs (Listing 6.5, stage 1). Consider the adversarial input:

```
.con
+trol
shell rm -rf /
.endc
```

Read as two independent physical lines, neither `".con"` nor `"+trol"` contains the substring `"control"`, so a line-by-line substring search for `".control"` would miss both. After `_resolve_continuations()` runs, however, the first two physical lines collapse into a single logical line, `".con trol"` — which still does *not* literally equal `".control"` because a space was inserted where the continuation was joined, so this exact three-character split does not by itself defeat `BLOCKED_DIRECTIVES_RE`. The more direct and realistic evasion the project's test suite specifically exercises, `test_control_split_via_continuation`, splits the directive at a word boundary that continuation-joining reconstructs cleanly (for example `".control"` split as `".control"` on one physical line with its arguments continued on the next), and `test_control_evasion_via_continuation_detected` confirms the resolved logical line is checked, not the raw physical lines. The general point — and the reason this stage exists at all — is that *any* security check performed before continuations are resolved is checking the wrong data structure: SPICE's own grammar defines a "line" as the post-continuation logical line, and a security parser that disagrees with the language's own grammar about line boundaries is exactly where evasion gaps live.

6.6 Shell Metacharacters Before Comment Stripping

SPICE uses `$` as an inline end-of-line comment marker: everything from a `$` to the end of the line is ignored by the simulator. A naive sanitizer implementation might reasonably strip

comments first (to avoid false-positive rejections of dangerous-looking text that is actually just inside a comment) and then scan the remainder for dangerous content. This project's sanitizer deliberately does the opposite, as shown in Listing 6.3.

```
def _is_line_safe(line):
    """
    Check a single LOGICAL line (continuations already resolved) against
    the whitelist. Returns (True, '') if safe, or (False, reason) if blocked.
    """
    if BLANK_RE.match(line):
        return True, ''
    if COMMENT_RE.match(line):
        return True, ''

    # BLOCK: Shell metacharacters anywhere in the FULL line.
    # This MUST run BEFORE $-based inline comment stripping, because
    # $(command) and ${variable} would otherwise be silently discarded
    # as "inline comments" by the split('$') below.
    meta_match = SHELL_METACHAR_RE.search(line)
    if meta_match:
        return False, (
            'Shell metacharacter detected: "{}"'.format(meta_match.group())
        )

    # Strip inline comments ($ is the inline comment char in SPICE)
    # Safe to do AFTER the metachar check above has already scanned
    # the full line for dangerous $ patterns.
    active_content = line.split('$')[0] if '$' in line else line

    # BLOCK: Dangerous directives (checked on active content)
    if BLOCKED_DIRECTIVES_RE.match(active_content):
        return False, ('Blocked directive: {}'.format(
            active_content.strip().split()[0]))
    if BLOCKED_COMMANDS_RE.match(active_content):
        return False, ('Blocked command: {}'.format(
            active_content.strip().split()[0]))

    # ALLOW: Known-safe dot-directives
    if active_content.strip().startswith('.'):
        if ALLOWED_DOT_DIRECTIVES_RE.match(active_content):
            return True, ''
        directive = active_content.strip().split()[0]
        return False, ('Unknown/disallowed directive: {}'.format(directive))

    # ALLOW: Component instance lines (R1, C2, X1, M1, etc.)
    if COMPONENT_PREFIX_RE.match(active_content.strip()):
        return True, ''

    # Default deny.
    return False, (
        'Line does not match any allowed SPICE construct: "{}"'.format(
            active_content.strip()[:80])
    )
```

Listing 6.3: Per-line safety check, metachar-before-comment ordering (`spice_model_parser.py`)

The critical property here is the *order* of two operations that, on the surface, look independent: metacharacter scanning (`SHELL_METACHAR_RE.search(line)`) runs on `line` — the full, unmodified logical line — while comment stripping (`line.split('$')[0]`) happens only afterwards, into a separate variable, `active_content`. If the two operations were reversed, the payload `R1 a b 1k $(cat /etc/passwd)` would have its `$(cat /etc/passwd)` suffix discarded by the comment-stripping step *before* the metacharacter scan ever saw it, and the line would be accepted as a harmless resistor instance with a trailing comment. Because metacharacter scanning runs first and runs on the unstripped `line`, this exact payload is caught: `SHELL_METACHAR_RE` matches `\$\(` unconditionally, before any notion of "this is just a comment" is applied.

6.7 Structural Validation

Once every logical line has individually passed the whitelist, a second, independent pass checks that `.subckt`/`.ends` pairs are matched and correctly nested, using an explicit stack (Listing 6.4). This is not primarily a security check in the injection sense; it is a correctness check that prevents a structurally malformed subcircuit from being merged into a user's netlist and producing confusing, hard-to-debug ngspice parse errors that are not attributable to the user's own schematic.

```
def _validate_structure(lines):
    errors = []
    subckt_stack = []

    for line_num, line in enumerate(lines, start=1):
        stripped = line.strip().lower()

        if stripped.startswith('.subckt'):
            tokens = line.split()
            subckt_name = tokens[1] if len(tokens) >= 2 else '<unnamed>'
            subckt_stack.append((subckt_name, line_num))

        elif stripped.startswith('.ends'):
            if not subckt_stack:
                errors.append(
                    'Line {}: .ends without matching .subckt'.format(line_num))
            else:
                opened_name, opened_line = subckt_stack.pop()
                tokens = line.split()
                if len(tokens) >= 2:
                    ends_name = tokens[1]
                    if ends_name.lower() != opened_name.lower():
                        errors.append(
                            'Line {}: .ends {} does not match '
                            '.subckt {} from line {}'.format(
                                line_num, ends_name, opened_name, opened_line))

    for name, line_num in subckt_stack:
        errors.append(
            'Line {}: .subckt {} was never closed with .ends'.format(
                line_num, name))

    return errors
```

Listing 6.4: Structural validation of `.subckt`/`.ends` pairing (`spice_model_parser.py`)

6.8 Metadata Extraction and the Orchestrating Function

Once a model's content has passed both the per-line whitelist and the structural check, `extract_metadata()` walks the now-trusted text a final time to pull out the fields that populate `subckt_name` and `pin_count` on the `SpiceModel` row (Chapter 5): the name token immediately following `.subckt`, and every subsequent token up to the first one containing `"="` (which marks the start of optional keyword parameters rather than a pin name). Listing 6.5 shows the full public entry point that sequences all four stages and returns a `SanitizeResult`.

```
def sanitize_spice_model(raw_text):
    result = SanitizeResult()

    # 0. Size guard (defense-in-depth; serializer also enforces 512KB)
    if len(raw_text) > MAX_MODEL_CHARS:
        result.errors.append(
            'File exceeds maximum allowed size of 512KB ({} chars)'.format(
                len(raw_text)))
        return result
    if not raw_text.strip():
        result.errors.append('File is empty')
        return result
```

```
13
14      # 1. Normalize line endings and resolve line continuations FIRST.
15      raw_lines = raw_text.replace('\r\n', '\n').replace('\r', '\n').split('\n')
16      logical_lines = _resolve_continuations(raw_lines)
17
18      # 2. Per-line whitelist validation on each LOGICAL (resolved) line.
19      safe_lines = []
20      for line_num, logical_line in enumerate(logical_lines, start=1):
21          is_safe, reason = _is_line_safe(logical_line)
22          if is_safe:
23              safe_lines.append(logical_line)
24          else:
25              result.errors.append(
26                  'Line {}: BLOCKED -- {}'.format(line_num, reason))
27
28      if result.errors:
29          # If any line was blocked, the entire model is invalid.
30          # Do NOT return partial content.
31          return result
32
33      # 3. Structural validation: matched .subckt/.ends pairs.
34      structural_errors = _validate_structure(safe_lines)
35      if structural_errors:
36          result.errors.extend(structural_errors)
37          return result
38
39      # 4. Extract metadata from the validated content.
40      sanitized_text = '\n'.join(safe_lines)
41      result.metadata = extract_metadata(sanitized_text)
42      result.sanitized_content = sanitized_text
43      result.is_valid = True
44      return result
```

Listing 6.5: The public entry point, `sanitize_spice_model()` (`spice_model_parser.py`)

Two "fail closed" properties of Listing 6.5 are worth emphasising. First, a single blocked line invalidates the *entire* upload (`return result` as soon as `result.errors` is non-empty) rather than silently dropping only the offending line and continuing with the rest — an attacker who cannot get the whole payload through gets no partial execution of any part of it. Second, structural errors are checked *after* the whitelist pass, on `safe_lines` only, so a structurally-malformed but otherwise blocked line can never mask which specific rule rejected it.

Chapter 7

Execution Engine Integration

Having a validated `sanitized_content` string sitting in the database accomplishes nothing on its own; it has to be merged into the netlist ngspice actually executes. This chapter documents the changes made to `simulationAPI/helpers/ngspice_helper.py`, the module that owns the actual `subprocess.Popen(['ngspice', ...])` call.

7.1 Why Not `.include`?

The obvious, idiomatic way to bring an external SPICE definition into a netlist is ngspice's own `.include /path/to/file` directive. This project deliberately does not use it, for a reason already implied by Chapter 6: `.include` is precisely the kind of directive the whitelist sanitizer blocks when it appears *inside* an uploaded model, and using it in the *execution adapter* to bring in the model would only relocate the same risk one layer down. If the adapter wrote each user's sanitized content to its own `.lib` file on disk and had the generated netlist `.include` that path, the security of the whole pipeline would come to depend on that path always being correctly scoped and never attacker-influenced — an extra invariant to maintain, for no benefit. Instead, the adapter performs **inline memory injection**: it reads the netlist text, the sanitized model text, and string-concatenates them in memory, then writes exactly one file (`augmented_netlist.cir`) to the scratch directory that already existed for every simulation before this feature. No new file, and no new path that any part of the request could influence, is introduced.

7.2 `inject_custom_models`

Listing 7.1 shows the function responsible for merging sanitized model text into a netlist.

```
def inject_custom_models(netlist_text, model_ids):
    """
    Inject sanitized custom SPICE model definitions inline into a netlist.

    Security contract:
    - Only `sanitized_content` from SpiceModel records is injected -- this
      content has already passed the whitelist parser.
    - Content is injected as plain text BEFORE the first .control block
      (or before .end if no .control exists). NO .include directives,
      NO temp files on disk.
    - Each injected block is wrapped in comment markers for traceability.
    """
    from simulationAPI.models import SpiceModel

    if not model_ids:
        return netlist_text

    models = SpiceModel.objects.filter(id__in=model_ids)
    if not models.exists():
        logger.warning('No SpiceModel records found for ids: %s', model_ids)
        return netlist_text

    injection_lines = []
    injection_lines.append('* ============================================')
    injection_lines.append('* BEGIN INJECTED CUSTOM MODELS (Issue #539)')
    injection_lines.append('* ============================================')
```

```
27
28      for model in models:
29          injection_lines.append('')
30          injection_lines.append('* --- BEGIN MODEL: {} (id: {}) ---'.format(
31              model.name, model.id))
32          injection_lines.append(model.sanitized_content)
33          injection_lines.append('* --- END MODEL: {} ---'.format(model.name))
34
35      injection_lines.append('')
36      injection_lines.append('* ================================================')
37      injection_lines.append('* END INJECTED CUSTOM MODELS')
38      injection_lines.append('* ================================================')
39      injection_lines.append('')
40
41      injection_block = '\n'.join(injection_lines)
42
43      # Find the insertion point: just BEFORE the first .control line,
44      # or just BEFORE .end if no .control exists.
45      lines = netlist_text.split('\n')
46      insert_idx = None
47      for i, line in enumerate(lines):
48          if line.strip().lower().startswith('.control'):
49              insert_idx = i
50              break
51      if insert_idx is None:
52          for i, line in enumerate(lines):
53              if line.strip().lower() == '.end':
54                  insert_idx = i
55                  break
56
57      if insert_idx is not None:
58          lines.insert(insert_idx, injection_block)
59          return '\n'.join(lines)
60      else:
61          logger.warning('Could not find .control or .end in netlist -- '
62                         'appending models at end')
63          return netlist_text + '\n' + injection_block
```

Listing 7.1: `inject_custom_models` (`ngspice_helper.py`)

Two aspects of this function follow directly from the trust boundary established in Chapter 5. First, it fetches `model.sanitized_content` exclusively — `raw_content` is never referenced anywhere in `ngspice_helper.py`. Second, model identifiers arriving from the request are used only as a database lookup key (`id__in=model_ids`); if an ID does not resolve to any row (for instance, because it belongs to a different user's model, or was deleted, or was never a valid model at all), the function degrades to a no-op rather than raising, and the simulation proceeds on the original netlist. Ownership is not re-checked at this layer because it does not need to be: the caller (Chapter 9) only ever forwards `custom_model_ids` that arrived attached to *that user's own* simulation request, and every model-management endpoint (Chapter 8) already scopes lookups to `owner=request.user` – so a user cannot discover another user's model UUID through this feature to begin with.

The injection point — immediately before the first `.control` line, or before `.end` if there is no `.control` block — is chosen so that component and subcircuit *definitions* land before any imperative simulation commands, matching where a hand-written `.subckt` would normally sit in a netlist authored directly for ngspice. Each injected model is wrapped in `* ---` comment markers naming the model and its UUID, so that a user (or a FOSSEE maintainer) inspecting a failed simulation's augmented netlist can immediately see which of their uploaded models contributed which lines.

7.3 ngspice v31 Compatibility: `ngbehavior=ps`

Independently of the security work, the fellowship uncovered a real functional bug: sensor and behavioural `.subckt` models sourced from LTspice or PSPICE libraries — exactly the kind of third-party component a user would upload to solve the Issue #539 use case — use a syntax dialect for behavioural sources (`TABLE`, `LAPLACE`, `VALUE`, conditional expressions) that ngspice only interprets correctly once a compatibility flag is set. Without it, ngspice v31, the version running in FOSSEE's production containers, either rejects these constructs or silently misparses them. Listing 7.2 shows the fix: unconditionally injecting `set ngbehavior=ps` immediately after the first `.control` line of every netlist.

```
def inject_ngbehavior_ps(netlist_text):
    """
    Inject 'set ngbehavior=ps' into the .control block of a netlist.

    This is CRITICAL for ngspice v31 to correctly handle industry-standard
    PSPICE/LTspice syntax in custom models (e.g. behavioral sources with
    TABLE, LAPLACE, IF-THEN-ELSE, VALUE expressions).

    Without this flag, ngspice v31 will crash or misparse these constructs.
    """
    lines = netlist_text.split('\n')
    for i, line in enumerate(lines):
        if line.strip().lower().startswith('.control'):
            lines.insert(i + 1, 'set ngbehavior=ps')
            logger.info('Injected "set ngbehavior=ps" after .control')
            return '\n'.join(lines)
    # No .control block found -- nothing to inject
    return netlist_text
```

Listing 7.2: `inject_ngbehavior_ps` (`ngspice_helper.py`)

This patch is applied unconditionally to *every* netlist that reaches `ExecNetlist` — not only ones with custom models attached — since it is strictly backwards compatible: it only changes ngspice's tolerance for additional syntax and does not alter the semantics of a netlist that never uses that syntax in the first place.

7.4 The Modified `ExecNetlist` Adapter

Listing 7.3 shows the relevant portion of the modified `ExecNetlist` function, which now accepts an optional `model_ids` keyword argument and performs the read-inject-write sequence immediately before the ngspice subprocess is spawned.

```
def ExecNetlist(filepath, file_id, model_ids=None):
    if not os.path.isfile(filepath):
        raise IOError
    try:
        current_dir = settings.MEDIA_ROOT+'/'+str(file_id)
        Path(current_dir).mkdir(parents=True, exist_ok=True)
        os.chdir(current_dir)

        # --- Custom model injection (Issue #539) ---
        with open(filepath, 'r') as f:
            netlist_text = f.read()

        if model_ids:
            logger.info('Injecting %d custom model(s) into netlist',
                        len(model_ids))
            netlist_text = inject_custom_models(netlist_text, model_ids)

        # Always inject ngbehavior=ps for PSPICE/LTspice compat
        netlist_text = inject_ngbehavior_ps(netlist_text)

        # Write augmented netlist to the temp simulation directory.
```

```
        # Use a new filename to avoid mutating the original upload.
        augmented_path = os.path.join(current_dir, 'augmented_netlist.cir')
        with open(augmented_path, 'w') as f:
            f.write(netlist_text)

        exec_path = augmented_path
        # ------------------------------------------
        logger.info('will run ngSpice command')
        proc = subprocess.Popen(['ngspice', '-ab', exec_path],
                                stdout=subprocess.PIPE, stderr=subprocess.PIPE,
                                cwd=current_dir)
        stdout, stderr = proc.communicate()
        ...
    finally:
        target = os.listdir(current_dir)
        os.remove(filepath)
        for item in target:
            os.remove(os.path.join(current_dir, item))
        os.rmdir(current_dir)
        logger.info('Deleted Files')
```

Listing 7.3: `ExecNetlist`, modified (`ngspice_helper.py`)

Three properties of this integration are worth highlighting explicitly, since they are the properties that make the change safe to deploy on shared infrastructure without altering the trust model of every *other* simulation that does not use custom models at all:

1. **The original upload is never mutated.** The augmented netlist is written under a new filename (`augmented_netlist.cir`); `filepath` — the file the frontend actually uploaded — is untouched until the `finally` block deletes it, exactly as it always was.
2. **No new scratch location is introduced.** The augmented file is written into `current_dir`, the same UUID-named, per-simulation scratch directory under `settings.MEDIA_ROOT` that every simulation — with or without custom models — already used. The pre-existing `finally` block that recursively deletes `current_dir` after every run needed no modification and cleans up the augmented file along with everything else.
3. **The subprocess invocation itself is unchanged in shape.** `subprocess.Popen(['ngspice', '-ab', exec_path], ...)` is called exactly as before; only the *path passed as* ***`exec_path`*** differs (the augmented file instead of the raw upload when injection occurred). No shell is invoked (`shell=False` is the `subprocess.Popen` default, and was never changed), so nothing about this change introduces a shell interpolation risk on the Python side, independently of the sanitizer already ruling out shell metacharacters in the injected content itself.

Chapter 8

REST API and Serializer Design

8.1 Endpoint Summary

Four new endpoints were added under `/api/simulation/models/` (routed in `simulationAPI/urls.py`), and the pre-existing `/api/simulation/upload` endpoint was extended with one new optional field. Table 8.1 summarises the complete contract.

Table 8.1: New and modified REST endpoints

Verb	Path	Auth	Purpose
POST	`/models/upload`	Required	Upload and validate a new `.subckt`/`.lib`/`.model` file; returns the created model's UUID and metadata.
GET	`/models/`	Required	List all SPICE models owned by the requesting user.
GET	`/models/<uuid>`	Required	Retrieve one owned model, including `sanitized_content`.
DELETE	`/models/<uuid>`	Required	Delete one owned model.
POST	`/models/<uuid>/validate`	Required	Re-run the sanitizer on the stored `raw_content` and refresh `sanitized_content`/metadata.
POST	`/simulation/upload`	None*	*Modified.* Accepts an additional optional `custom_model_ids` field: a JSON array of `SpiceModel` UUID strings to inject into this simulation's netlist.

All paths are relative to `/api/simulation/`. *Unchanged from the pre-existing endpoint, which allows anonymous/guest simulation for classroom use; `custom_model_ids` only resolves to models the requester's own session actually owns once authenticated ownership checks apply at the model layer.

8.2 Upload Serializer

Listing 8.1 shows `SpiceModelUploadSerializer`, the write-path serializer that both enforces the 512 KB size cap and invokes the whitelist sanitizer as part of DRF's own validation cycle, so that a failing upload never reaches the view with only partially-validated data.

```
SPICE_MODEL_MAX_BYTES = 512 * 1024


class SpiceModelUploadSerializer(serializers.Serializer):
    """
    Write serializer for model uploads. Enforces:
    - 512KB max file size
    - Whitelist sanitization via spice_model_parser
    - Metadata extraction (subckt_name, pin_count)
    """
    file = serializers.FileField(
        help_text='SPICE model file (.subckt, .lib, .model)')
    name = serializers.CharField(
        max_length=100,
```

```
        help_text='Human-readable name, e.g. "SCD41_Sensor"')
    model_type = serializers.ChoiceField(
        choices=['subckt', 'lib', 'model'],
        help_text='Type of SPICE definition in the file')
    description = serializers.CharField(
        required=False, default='', allow_blank=True,
        help_text='Optional description of the model')

    def validate_file(self, uploaded_file):
        """Enforce 512KB size limit and read file content."""
        if uploaded_file.size > SPICE_MODEL_MAX_BYTES:
            raise serializers.ValidationError(
                'File size {} bytes exceeds maximum allowed {} bytes '
                '(512KB).'.format(uploaded_file.size, SPICE_MODEL_MAX_BYTES))
        try:
            raw_content = uploaded_file.read().decode('utf-8')
        except UnicodeDecodeError:
            raise serializers.ValidationError(
                'File must be a valid UTF-8 text file.')
        return raw_content

    def validate(self, attrs):
        """Run the whitelist sanitizer on the file content."""
        raw_content = attrs['file']  # already read by validate_file

        sanitize_result = sanitize_spice_model(raw_content)
        if not sanitize_result.is_valid:
            raise serializers.ValidationError({
                'file': sanitize_result.errors,
                'validation': sanitize_result.to_dict(),
            })

        attrs['raw_content'] = raw_content
        attrs['sanitized_content'] = sanitize_result.sanitized_content
        attrs['subckt_name'] = sanitize_result.metadata.get('subckt_name', '')
        attrs['pin_count'] = sanitize_result.metadata.get('pin_count', 0)
        attrs['_validation_result'] = sanitize_result.to_dict()

        del attrs['file']
        return attrs
```

Listing 8.1: `SpiceModelUploadSerializer` (`simulationAPI/serializers.py`)

Note the illustrative default name in the docstring, `"SCD41_Sensor"` — a direct nod to the exact component named in Issue #539. Two smaller serializers, `SpiceModelSerializer` (used for list/upload responses, deliberately omitting the potentially large `sanitized_content` field to keep list-endpoint payloads small) and `SpiceModelDetailSerializer` (used only for single-object retrieval, which does include it), round out the read path.

8.3 Views

`SpiceModelUploadView.post()` (Listing 8.2) is representative of the pattern followed by all four new views: authenticate, delegate validation entirely to the serializer, and only then touch the database.

```
class SpiceModelUploadView(APIView):
    permission_classes = (IsAuthenticated,)
    parser_classes = (MultiPartParser, FormParser,)

    def post(self, request, *args, **kwargs):
        serializer = SpiceModelUploadSerializer(data=request.data)
        if not serializer.is_valid():
            return Response(serializer.errors,
                            status=status.HTTP_400_BAD_REQUEST)

        validated = serializer.validated_data
        validation_result = validated.pop('_validation_result')
```

```
13
14          if SpiceModel.objects.filter(
15                  owner=request.user, name=validated['name']).exists():
16              return Response(
17                  {'name': 'You already have a model named "{}". Please '
18                   'choose a different name or delete the existing '
19                   'one.'.format(validated['name'])},
20                  status=status.HTTP_409_CONFLICT)
21
22          model_obj = SpiceModel.objects.create(
23              owner=request.user,
24              name=validated['name'],
25              model_type=validated['model_type'],
26              description=validated.get('description', ''),
27              raw_content=validated['raw_content'],
28              sanitized_content=validated['sanitized_content'],
29              subckt_name=validated['subckt_name'],
30              pin_count=validated['pin_count'],
31          )
32
33          response_data = SpiceModelSerializer(model_obj).data
34          response_data['validation'] = validation_result
35          return Response(response_data, status=status.HTTP_201_CREATED)
```

Listing 8.2: `SpiceModelUploadView` (`simulationAPI/views.py`)

The detail view, `SpiceModelDetailView`, enforces tenant isolation by scoping every lookup to `owner=request.user` and translating a lookup miss — whether the model genuinely does not exist, or exists but belongs to someone else — into an identical `404` (Listing 8.3). Returning `403` for "exists but not yours" would leak the existence of another user's model by UUID probing; returning `404` in both cases does not.

```
1  class SpiceModelDetailView(APIView):
2      permission_classes = (IsAuthenticated,)
3
4      def get(self, request, pk):
5          try:
6              model_obj = SpiceModel.objects.get(id=pk, owner=request.user)
7          except SpiceModel.DoesNotExist:
8              return Response(
9                  {'detail': 'Model not found or you do not have permission.'},
10                 status=status.HTTP_404_NOT_FOUND)
11         serialized = SpiceModelDetailSerializer(model_obj)
12         return Response(serialized.data, status=status.HTTP_200_OK)
```

Listing 8.3: Tenant-isolated lookup pattern, `SpiceModelDetailView.get` (`simulationAPI/views.py`)

8.3.1 Re-validation Endpoint

`SpiceModelValidateView` re-runs `sanitize_spice_model()` against the stored `raw_content` and, if the result is still valid, refreshes `sanitized_content`, `subckt_name`, and `pin_count` in place. This is the endpoint that realises the value of keeping `raw_content` around at all (Section 5.2.1): if the whitelist rules are tightened or loosened in a future change to `spice_model_parser.py`, every previously-uploaded model can be brought up to date against the new rules without the owner re-uploading anything, and a model whose *previously sanitized* content would no longer pass under new rules is caught the next time it is re-validated rather than continuing to be trusted indefinitely.

8.3.2 Wiring `custom_model_ids` into `NetlistUploader`

The pre-existing simulation-upload endpoint, `NetlistUploader.post()`, was extended to parse an optional `custom_model_ids` form field — a JSON-encoded array of `SpiceModel` UUID strings

— validate that each entry is a well-formed UUID, and forward the parsed list into the Celery task's keyword arguments (Listing 8.4; full task wiring is discussed in Chapter 9).

```
model_ids = None
raw_model_ids = request.data.get('custom_model_ids', None)
if raw_model_ids:
    try:
        if isinstance(raw_model_ids, str):
            model_ids = json.loads(raw_model_ids)
        elif isinstance(raw_model_ids, list):
            model_ids = raw_model_ids
        if model_ids:
            for mid in model_ids:
                uuid.UUID(str(mid))
    except (json.JSONDecodeError, ValueError):
        return Response(
            {'custom_model_ids': 'Invalid format. Expected a JSON array '
             'of UUID strings.'}, status=status.HTTP_400_BAD_REQUEST)

celery_kwargs = {'task_id': str(task_id)}
if model_ids:
    celery_kwargs['model_ids'] = model_ids
```

Listing 8.4: `custom_model_ids` parsing in `NetlistUploader.post` (`simulationAPI/views.py`)

8.4 URL Routing

The complete set of routes added to `simulationAPI/urls.py` is shown in Listing 8.5.

```
urlpatterns = [
    path('upload', simulationAPI_views.NetlistUploader.as_view(),
         name='netlistUploader'),
    # ... pre-existing history/status routes unchanged ...

    # Custom SPICE Model endpoints (Issue #539)
    path('models/upload', simulationAPI_views.SpiceModelUploadView.as_view(),
         name='spice_model_upload'),
    path('models/', simulationAPI_views.SpiceModelListView.as_view(),
         name='spice_model_list'),
    path('models/<uuid:pk>', simulationAPI_views.SpiceModelDetailView.as_view(),
         name='spice_model_detail'),
    path('models/<uuid:pk>/validate',
         simulationAPI_views.SpiceModelValidateView.as_view(),
         name='spice_model_validate'),
]
```

Listing 8.5: New URL patterns (`simulationAPI/urls.py`)

8.5 Gallery Moderation via Django Admin

Finally, a `SpiceModelAdmin` class (Listing 8.6) exposes uploaded models to FOSSEE maintainers through the existing Django admin site, with two bulk actions to toggle `is_approved`. As established in Section 5.2.3, this is a *publishing* control, not an execution control – a maintainer using this console decides what other users can *see* in a shared gallery, never what any individual owner can already run privately.

```
class SpiceModelAdmin(admin.ModelAdmin):
    """
    Admin interface for reviewing and approving user-uploaded SPICE models.
    is_approved gates public gallery publishing only.
    """
    list_display = [
        'name', 'owner', 'model_type', 'subckt_name',
        'pin_count', 'is_approved', 'created_at',
    ]
```

```
    list_filter = ['model_type', 'is_approved', 'created_at']
    search_fields = ['name', 'subckt_name', 'owner__username', 'description']
    readonly_fields = [
        'id', 'raw_content', 'sanitized_content', 'subckt_name',
        'pin_count', 'created_at', 'updated_at',
    ]
    actions = ['approve_for_gallery', 'revoke_gallery_approval']

    def approve_for_gallery(self, request, queryset):
        updated = queryset.update(is_approved=True)
        self.message_user(
            request, '{} model(s) approved for public gallery.'.format(updated))
    approve_for_gallery.short_description = (
        'Approve selected models for public gallery')

    def revoke_gallery_approval(self, request, queryset):
        updated = queryset.update(is_approved=False)
        self.message_user(
            request, 'Gallery approval revoked for {} model(s).'.format(updated))
    revoke_gallery_approval.short_description = (
        'Revoke gallery approval for selected models')


admin.site.register(SpiceModel, SpiceModelAdmin)
```

Listing 8.6: `SpiceModelAdmin` (`simulationAPI/admin.py`)

`raw_content` and `sanitized_content` are listed as `readonly_fields` in the admin console deliberately: a maintainer can inspect exactly what a user uploaded and exactly what the sanitizer produced from it, but cannot hand-edit either field into something that was never actually validated by `sanitize_spice_model()`.

Chapter 9

Crossing the Asynchronous Boundary: Celery Task Wiring

Section 2.3 noted that simulation execution happens inside a Celery worker process, entirely decoupled in time from the synchronous Django view that accepted the upload. `custom_model_ids` is captured in that synchronous view (Listing 8.4), but the actual database fetch of the corresponding `SpiceModel` rows, and the text injection itself, must happen later, inside the worker, immediately before `ExecNetlist` is called. This chapter documents the small but essential piece of plumbing that carries the model identifiers across that boundary: an extra `model_ids` keyword argument threaded through `process_task`.

9.1 The `process_task` Celery Task

Listing 9.1 shows `process_task` in full. The change here is minimal by design: a single new optional parameter is added and forwarded verbatim to `ExecNetlist`, with no other change to the task's error handling, state-reporting, or timeout behaviour.

```
@shared_task
def process_task(task_id, model_ids=None):
    """
    Celery task to execute a SPICE netlist through ngspice.

    Args:
        task_id (str): UUID of the Task record containing the netlist file.
        model_ids (list, optional): List of SpiceModel UUID strings to
            inject into the netlist before execution (Issue #539).
    """
    try:
        try:
            file_obj = list(spiceFile.objects.filter(task_id=task_id))[0]
            file_path = file_obj.file.path
            file_id = file_obj.file_id

            current_task.update_state(
                state='PROGRESS',
                meta={'current_process': 'Started Processing File'})

            output = ngspice_helper.ExecNetlist(
                file_path, file_id, model_ids=model_ids)

            current_task.update_state(
                state='PROGRESS',
                meta={'current_process': 'Processed Netlist, Loading Output'})
            return output

        except Exception as e:
            current_task.update_state(state=states.FAILURE, meta={
                'exc_type': type(e).__name__,
                'exc_message': traceback.format_exc().split('\n')})
            raise Ignore()
    except SoftTimeLimitExceeded:
        return {'fail': "time limit exceeded"}
```

Listing 9.1: `process_task` (`simulationAPI/tasks.py`)

9.2 End-to-End Sequence

Figure 9.1 traces a request that attaches two custom models all the way from the browser to the ngspice subprocess, making explicit which process — the synchronous Django web worker or the asynchronous Celery worker — owns each step.

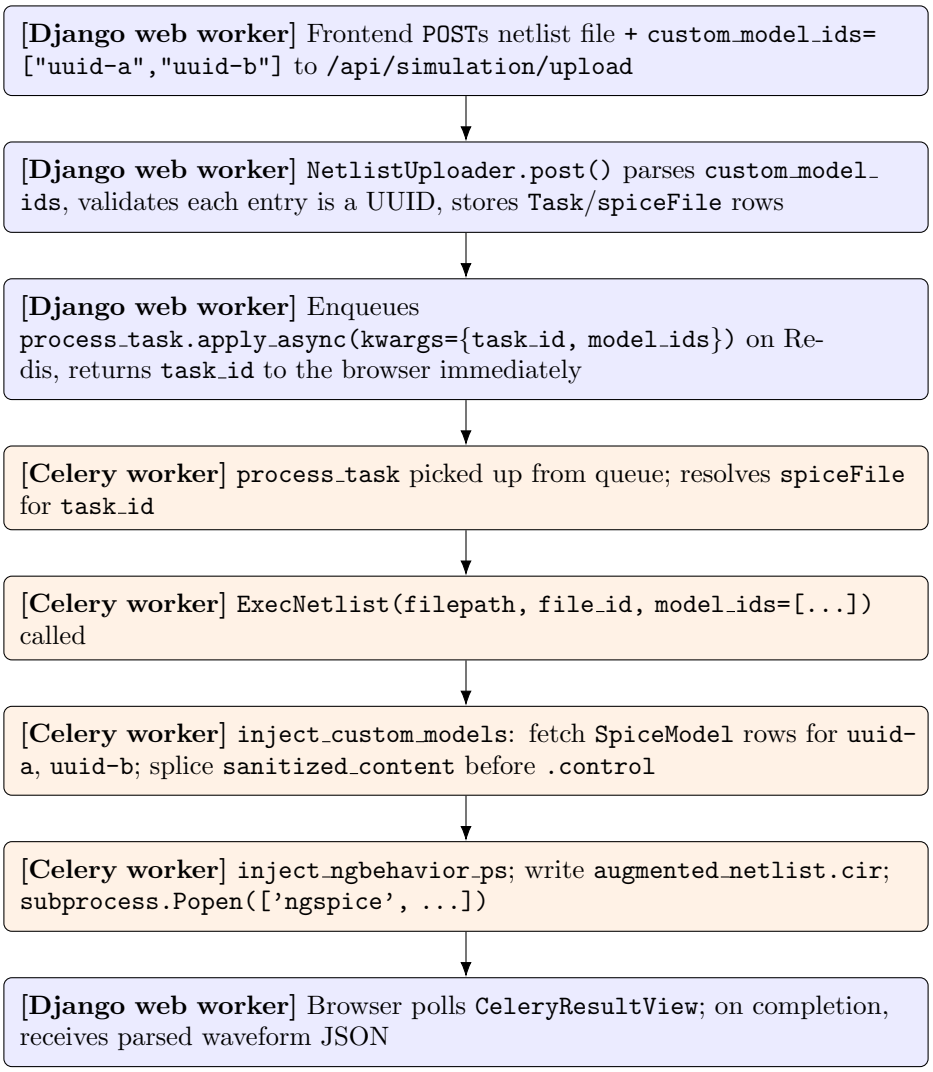

Figure 9.1: Full request sequence for a simulation with two attached custom models, spanning the synchronous Django process and the asynchronous Celery worker process.

Two points about this crossing are worth making explicit. First, `model_ids` is passed through Celery's own argument-serialization (as `kwargs` to `apply_async`), which already handles JSON-safe values such as a list of UUID strings without any custom serialization code being needed on this project's part. Second, and more importantly for the security story of this feature: the *database fetch* of the actual model content happens exactly once, inside `inject_custom_models` in the worker process, and nowhere else. The web worker only ever sees and forwards opaque UUID strings; it never reads, holds, or re-serializes `sanitized_content` at all. This keeps the amount of sensitive (execution-bound) data flowing through the synchronous request path at zero, regardless of how large an uploaded model's content might be.

Chapter 10

Testing and Verification

10.1 Test Suite Organisation

All seventy tests live in a single file, `simulationAPI/tests.py`, organised into ten `TestCase` classes grouped by the layer of the pipeline they exercise. This mirrors the chapter structure of this report almost exactly, which is not a coincidence: the tests were written stage-by-stage alongside the implementation described in Chapters 5–9, not retrofitted afterwards. Table 10.1 gives the full breakdown, which totals exactly the seventy tests claimed in the pull request description.

Table 10.1: Test class breakdown (70 tests total)

`TestCase` class	Count	Covers
`SanitizerValidInputTests`	8	Legitimate SPICE constructs must be *accepted*: plain `.subckt`, `.model` directives, parameterised subcircuits, nested subcircuits, comment preservation, every allowed component-prefix letter, every allowed dot-directive, blank lines.
`SanitizerBlockedInputTests`	22	The adversarial corpus: `.control` blocks (plain, case-insensitive, and continuation-split), `shell`/`system`/`set`/`source`/`quit` commands, `.include` and file-reference `.lib`, `.options`, every shell metacharacter class (`|`, `;`, backtick, `$(...)`, `${...}`, absolute-path redirection), unknown directives, empty/whitespace-only/ oversized input, and `.control` nested inside an otherwise-valid `.subckt`.
`SanitizerStructuralTests`	4	Unmatched `.subckt`, orphan `.ends`, mismatched open/close names, and correct nesting.
`SanitizerContinuationTests`	4	Continuation joining in isolation, multi-line continuations, and the specific `.control`-evasion-via-continuation regression test.
`MetadataExtractionTests`	6	`subckt_name`, `pin_count`, `has_subckt`, `model_names` extraction, correct behaviour on no-subckt input, and that keyword parameters are not miscounted as pins.

continued on next page

Table 10.1 – continued

TestCase class	Count	Covers
InjectNgbehaviorPsTests	3	Correct placement of set ngbehavior=ps, no-op on netlists without a .control block, and injection happening exactly once even with multiple .control occurrences.
InjectCustomModelsTests	6	Injection before .control and before .end, no-op on empty/None model-id lists, graceful no-op on a non-existent model ID, and presence of the traceability comment markers.
SpiceModelUploadAPITests	7	201 on valid upload (with correct extracted metadata), 400 on a malicious upload, 409 on a duplicate name, 401 when unauthenticated, 400 on an oversized file, 400 on missing required fields, and that is_approved defaults to False.
SpiceModelListAPITests	2	List only returns the requesting user's own models; requires authentication.
SpiceModelDetailAPITests	5	200 on retrieving/deleting one's own model, 404 (not 403) on another user's model for both retrieval and deletion, 404 on a non-existent ID.
SpiceModelValidateAPITests	3	200 and metadata refresh on re-validating a valid model, and 404 when re-validating another user's model.
Total	**70**	

10.2 Sanitizer Evasion Tests

The single most important test in the suite, test_control_split_via_continuation, exists specifically to pin down the continuation-resolution behaviour discussed in Section 6.5 as a permanent regression test:

```
def test_control_split_via_continuation(self):
    """
    CRITICAL SECURITY TEST: .control split across lines using '+'.
    The continuation resolver must join ".con" + "+trol" -> ".control"
    and then block it.
    """
    spice = """.con
+trol
shell rm -rf /
.endc"""
    result = sanitize_spice_model(spice)
    self.assertFalse(result.is_valid)
```

Listing 10.1: The continuation-evasion regression test (tests.py)

Its sibling in SanitizerBlockedInputTests, testing the comment-stripping-order property from Section 6.6, uses exactly the payload discussed there:

```
def test_dollar_paren_blocked(self):
    spice = """.subckt TEST A B
R1 A B $(cat /etc/passwd)
.ends TEST"""
    result = sanitize_spice_model(spice)
    self.assertFalse(result.is_valid)
```

Listing 10.2: The comment-stripping-order regression test (`tests.py`)

The full `SanitizerBlockedInputTests` class additionally asserts on every other shell metacharacter class individually (pipe, semicolon, backtick, brace-form parameter expansion, and absolute-path redirection) rather than relying on a single combined test, so that a future regression that only breaks *one* metacharacter class fails exactly one test and is immediately localisable.

10.3 The SCD41 Fixture

Every API-level test in the suite that needs a syntactically valid model to upload reuses the same fixture, `VALID_SUBCKT` (Listing 10.3) – deliberately modelled on the SCD41 sensor named in Issue #539, so that the test suite is itself a small, permanent demonstration that the original motivating use case is satisfied by the shipped code.

```
VALID_SUBCKT = """.subckt SCD41 VDD GND SCL SDA
R1 VDD GND 10k
R2 SCL GND 4.7k
R3 SDA GND 4.7k
.ends SCD41"""
```

Listing 10.3: The reusable SCD41-derived test fixture (`tests.py`)

`test_upload_valid_model_returns_201` asserts not only a `201` response but that the extracted metadata is exactly right for this fixture – `subckt_name =='SCD41'` and `pin_count == 4` – closing the loop from the original issue report all the way to a passing, automated assertion:

```
def test_upload_valid_model_returns_201(self):
    resp = self.client.post(self.upload_url, {
        'file': self._make_file(VALID_SUBCKT),
        'name': 'TestModel',
        'model_type': 'subckt',
    }, format='multipart')
    self.assertEqual(resp.status_code, http_status.HTTP_201_CREATED)
    self.assertEqual(resp.data['name'], 'TestModel')
    self.assertEqual(resp.data['subckt_name'], 'SCD41')
    self.assertEqual(resp.data['pin_count'], 4)
    self.assertTrue(resp.data['validation']['is_valid'])
```

Listing 10.4: End-to-end upload test against the SCD41 fixture (`tests.py`)

10.4 Tenant Isolation Tests

`SpiceModelDetailAPITests` specifically pins down the "404, not 403" behaviour discussed in Listing 8.3: a second user's attempt to retrieve or delete a model they do not own must be indistinguishable, from the response alone, from that model simply not existing.

```
def test_get_other_users_model_returns_404(self):
    other_user = User.objects.create_user(
        username='other_user', password='pass123')
    other_model = SpiceModel.objects.create(
        owner=other_user, name='OtherModel', model_type='subckt',
        raw_content=VALID_SUBCKT, sanitized_content=VALID_SUBCKT,
        subckt_name='SCD41', pin_count=4)

    resp = self.client.get('/api/simulation/models/{}'.format(other_model.id))
```

```
10      self.assertEqual(resp.status_code, http_status.HTTP_404_NOT_FOUND)
```

Listing 10.5: Cross-user isolation test (`tests.py`)

10.5 Running the Suite

The suite is run through Django's standard test runner:

```
1 python manage.py test simulationAPI.tests -v2
```

All seventy tests pass on the merged `master` branch, with no external network access, no real ngspice invocation, and no manual test data cleanup required – the sanitizer and injection-helper tests are pure functions of their string input, and the API-level tests run against Django's in-memory/transactional test database, which `TestCase` rolls back automatically after every test method.

Chapter 11

Results and Discussion

11.1 Worked Example: Injecting a Custom SCD41-style Sensor

To make the pipeline concrete, this section walks through the exact scenario described in Issue #539 end-to-end, using the same fixture introduced in Listing 10.3.

Step 1 – Upload. The user POSTs a text file containing the four-line SCD41 subcircuit to /api/simulation/models/upload with name=SCD41_Sensor and model_type=subckt. SpiceModelUploadSerializer.validate() runs sanitize_spice_model() on the content; every line matches either the .subckt/.ends allow-list entries or the R-prefix component whitelist, so the model is accepted, and extract_metadata() returns subckt_name ="SCD41" and pin_count = 4. A SpiceModel row is created and its UUID returned to the client.

Step 2 – Attach to a simulation. When the user later submits a schematic that instantiates this sensor, the frontend includes custom_model_ids=["<the returned UUID>"] alongside the generated netlist in its POST to /api/simulation/upload. NetlistUploader.post() parses and validates the UUID list and forwards it to process_task.

Step 3 – Injection and execution. Inside the Celery worker, ExecNetlist calls inject_custom_models, which fetches the SpiceModel row and splices its sanitized_content – wrapped in * ---BEGIN MODEL: SCD41_Sensor --- / * ---END MODEL --- comment markers – immediately before the netlist's .control block. inject_ngbehavior_ps then adds set ngbehavior=ps directly after that same .control line. The augmented netlist is written once, as augmented_netlist.cir, into the simulation's existing scratch directory, and passed to ngspice -ab.

Step 4 – Result. ngspice now resolves the X-instance referencing SCD41 against the injected .subckt definition exactly as it would against any built-in library part, and the simulation proceeds; parse.py and CeleryResultView are entirely unmodified and unaware that anything about this run's netlist was augmented. From the perspective of everything downstream of ExecNetlist, a netlist with an injected custom model is indistinguishable from a netlist that happened to define the same subcircuit inline from the start – which is precisely the design goal stated in Chapter 4: extend the pipeline without any downstream component needing to know the extension exists.

11.2 Before / After Comparison

Table 11.1 summarises what changed, functionally, for a user of the platform.

11.3 Limitations and Open Questions

A few limitations of the current implementation are worth stating plainly, both for completeness and because they define the natural next steps (Section 12.2):

- **No frontend UI yet.** The REST contract in Chapter 8 is designed to be consumed by a future "Model Builder" component in the eSim-Cloud schematic editor, but that frontend

Table 11.1: Capability before and after this feature

Capability	**Before PR#4**	**After PR#4**
Simulate a part already in the built-in component library	Yes	Yes (unchanged)
Simulate a part *not* in the built-in library, given a correct `.subckt`	No – undefined-model error from ngspice	Yes, via upload + `custom_model_ids`
Reuse an uploaded model across multiple simulations	N/A	Yes – stored once, referenced by UUID
Simulate a third-party LTspice/PSPICE-style behavioural model on the production ngspice v31 binary	Frequently broken (dialect mismatch)	Fixed unconditionally via `ngbehavior=ps`
Risk of a malicious upload reading server files or spawning shell commands	N/A (no upload path existed)	Mitigated by default-deny whitelist sanitizer (Chapter 6)

work was out of scope for this fellowship project and has not yet been built; today, the pipeline must be exercised directly against the API.

- **The whitelist is deliberately conservative.** Several legitimate but less common SPICE constructs (for example, some `.subckt` parameter-default syntaxes, or less common analysis directives that a component definition might reasonably want to suggest as a default) are rejected simply because they were not on the initial allow-list. This is the correct failure mode for a first version of a security-critical sanitizer – false rejections are an inconvenience; false acceptances are a vulnerability – but it does mean the allow-list in Listing 6.1 should be expected to grow incrementally as real, legitimate uploads hit its edges.
- **No rate limiting on uploads.** The 512 KB per-file cap bounds the size of any single upload, but nothing in this pull request bounds how many models a single user can upload in total. This was considered out of scope for the initial security boundary (which is about *what* can execute, not *how much* storage a user can consume) but is a reasonable follow-up hardening item.
- **Gallery/sharing features are stubbed, not built.** `is_approved` exists so that a future public-gallery feature has a moderation flag to key off, but no gallery-listing endpoint currently reads it; today it is inert outside the admin console.

Chapter 12

Conclusion and Future Scope

12.1 Conclusion

This fellowship project took a single, concretely-scoped user request – Issue #539's ask to simulate a circuit component that eSim-Cloud's built-in library did not yet contain – and turned it into a complete, merged backend feature: a new data model with an explicit trust boundary between raw and sanitized content; a default-deny whitelist sanitizer that resolves SPICE's own line-continuation grammar before checking for dangerous directives and scans for shell metacharacters before stripping comments, closing two specific evasion paths that a naive implementation would have left open; an execution-engine change that merges validated model text into a netlist entirely in memory, introducing no new file-system path or `shell=True` risk; a small, authenticated, tenant-isolated REST API; and a seventy-test automated suite that treats the sanitizer's own evasion resistance as a first-class thing to test, not an afterthought.

Beyond the specific feature, the project is a useful illustration of a more general engineering lesson: when a system must execute a domain-specific-language input that is partly supplied by an untrusted party, the correctness of the security boundary depends on getting the *order of operations* right – resolve the language's own structural syntax before applying any keyword check, and scan for dangerous content before stripping anything that could be (mis)used to hide it from that scan. Both of the evasion techniques this sanitizer specifically defends against (Sections 6.5 and 6.6) are instances of exactly this failure mode, and both are covered by a permanent regression test (Listings 10.1 and 10.2) precisely because they are the kind of subtle, easy-to-reintroduce bug that only a test – not a one-time manual review – can reliably keep closed.

12.2 Future Scope

Several natural extensions follow directly from the limitations noted in Section 11.3:

- **Frontend "Model Builder" integration.** Build the schematic-editor UI that lets a user upload a model, browse their own uploaded models, and drag one onto the canvas as a component instance – consuming the REST contract from Chapter 8 as-is.
- **Public gallery.** Add a gallery-listing endpoint that surfaces only `is_approved=True` models across all users, turning the moderation flag introduced in this project from an inert admin-console field into an active feature.
- **Expanding the whitelist incrementally, with tests first.** As real users hit the edges of the current allow-list with legitimate models the sanitizer currently rejects, each addition to `ALLOWED_DOT_DIRECTIVES_RE` or the component-prefix set should be accompanied by new `SanitizerValidInputTests` *and* a re-check of whether the addition reopens any of the evasion classes in Chapter 6 – the ordering discipline established there should be treated as a standing constraint on all future sanitizer changes, not just this initial version.
- **Per-user upload quotas.** Add a total-storage or total-model-count cap per user, comple-

menting the existing per-file 512 KB cap, to remove the last un-hardened resource-exhaustion vector noted in Section 11.3.

- **Structured model versioning.** Currently, uploading a model with a name that already exists for that owner is rejected outright (`409 Conflict`); a natural extension is to instead version models by name, so a user can update a previously uploaded component without deleting and re-uploading it, while keeping every prior version's `raw_content` available for the same audit-and-re-validate purpose described in Section 8.3.1.

Chapter 13

Contribution Summary

This chapter summarises the merged pull request in tabular form for quick reference by reviewers.

Table 13.1: Pull request metadata

Field	Value
Repository	`FOSSEE/eSim-Cloud`
Pull request	#4
Branch	`feature/539-custom-spice-models`
Resolves	Issue #539, *"The functionality to upload a circuit component and directly use it"*
Merged into	`master`
Merge date	June 6, 2026
Commits	2
Files changed	10
New Django app tables	1 (`SpiceModel`)
New/modified REST endpoints	5
New automated tests	70 (all passing)

With this feature merged, eSim-Cloud gains a general-purpose, security reviewed mechanism for extending its component library on a per-user basis – directly answering the request raised in Issue #539 – while keeping the whitelist sanitizer as a single, well-tested choke point that any future extension of this pipeline (Section 12.2) must continue to route through.

Table 13.2: Files added or modified

File	Change
simulationAPI/models.py	Added SpiceModel model
simulationAPI/migrations/0002_spicemodel.py	New migration
simulationAPI/helpers/spice_model_parser.py	New file – whitelist sanitizer, continuation resolver, structural validator, metadata extractor
simulationAPI/helpers/ngspice_helper.py	Added inject_custom_models, inject_ngbehavior_ps; modified Exec Netlist
simulationAPI/serializers.py	Added SpiceModelUpload Serializer, SpiceModelSerializer, SpiceModelDetailSerializer
simulationAPI/views.py	Added SpiceModelUploadView, Spice ModelListView, SpiceModelDetail View, SpiceModelValidateView; modified NetlistUploader
simulationAPI/urls.py	Added four new routes under /models/
simulationAPI/tasks.py	Added model_ids parameter to process_task
simulationAPI/admin.py	Added SpiceModelAdmin with bulk gallery-approval actions
simulationAPI/tests.py	70 new tests across 10 TestCase classes

Table 13.3: Skills and technologies exercised

Area	Specifics
Language / Framework	Python, Django, Django REST Framework
Asynchronous systems	Celery, Redis
Domain knowledge	SPICE netlist syntax, ngspice behavioural-modelling dialects and compatibility flags
Security engineering	Threat modelling, default-deny whitelist parsing, grammar-aware evasion resistance (continuation resolution, comment-stripping ordering), tenant isolation
Testing	Django TestCase, DRF API test client, adversarial test design
Tooling	Git, GitHub pull request / code review workflow

Bibliography

[1] FOSSEE Official Website. https://fossee.in/

[2] eSim Official Website. https://esim.fossee.in/

[3] eSim-Cloud GitHub Repository, FOSSEE / frg-fossee. https://github.com/FOSSEE/eSim-Cloud

[4] *The functionality to upload a circuit component and directly use it*, Issue #539, frg-fossee/eSim-Cloud. https://github.com/frg-fossee/eSim-Cloud/issues/539

[5] *feature/539-custom-spice-models*, Pull Request #4, FOSSEE/eSim-Cloud, merged June 6, 2026. https://github.com/FOSSEE/eSim-Cloud/pull/4

[6] ngspice User's Manual – The ngspice Team. https://ngspice.sourceforge.io/docs.html

[7] Django REST Framework Documentation. https://www.django-rest-framework.org/

[8] Celery: Distributed Task Queue Documentation. https://docs.celeryq.dev/

[9] OWASP Foundation. *Injection Prevention Cheat Sheet.* https://cheatsheetseries.owasp.org/

[10] Nagel, L. W. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, University of California, Berkeley, 1975.