



# Semester Long Internship Spring 2026

On

## eSim Copilot: Performance Optimisation and Configuration Layer

Submitted by

**Nidhi Bhawari**
Vidyalankar Institute of Technology, Mumbai

Under the guidance of

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

# 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 closely with open-source EDA tools, develop IC subcircuits in eSim, and gain exposure to real-world circuit modeling and simulation workflows. 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

Large language models (LLMs) – AI systems trained on large corpora of text to understand and generate natural language – have made it practical to embed conversational assistants directly into domain-specific software tools. This capability is particularly relevant in Electronic Design Automation (EDA), where users must reason simultaneously about component specifications, SPICE syntax, simulation parameters, and circuit behaviour. The FOSSEE (Free and Open Source Software for Education) project at IIT Bombay develops and distributes free engineering tools for undergraduate education across India. Its primary EDA tool, eSim, is a complete open-source environment for analogue, digital, and mixed-signal circuit design, built on KiCad and NgSpice. An earlier internship at FOSSEE delivered an in-application AI assistant for eSim – the *eSim Copilot* – which runs a local LLM via Ollama and exposes a chat interface inside the eSim PyQt5 application. The present internship extends that Copilot along three axes: token-by-token streaming for faster perceived responses, a JSON configuration layer for customisable system rules and runtime parameters, and context-window tuning for local-LLM efficiency.

## 1.2 Overview of eSim

eSim integrates the following open-source components under a unified Python/PyQt5 shell:

- **KiCad** – schematic capture, component assignment, and netlist generation.
- **NgSpice** – SPICE-standard circuit simulation, supporting transient (`.tran`), AC (`.ac`), DC sweep (`.dc`), operating point (`.op`), and noise (`.noise`) analyses.
- **Python-based post-processing** tools for waveform plotting and result extraction.

eSim is cross-platform (Linux, Windows, macOS) and distributed under open-source licences.

## 1.3 Overview of eSim Copilot

The eSim Copilot is the AI assistant integrated inside the eSim application. It is implemented as a set of Python modules organised across two locations:

- `src/chatbot/chatbot_thread.py` – background worker threads (`OllamaWorker`, `OllamaVisionWorker`, `MicWorker`, `OllamaStatusWorker`, `ModelFetchWorker`) that talk to the local Ollama HTTP API on `localhost:11434`. This file is also the entry point of the configuration layer introduced in this internship.
- `src/frontEnd/Chatbot.py` – the PyQt5 chat panel: message rendering, session sidebar, image staging, settings panel, and the streaming logic that fills the bot bubble token by token.
- `src/chatbot/config.json` – runtime configuration introduced in this internship; described in detail in Chapter 5.

All Copilot operations run locally; no user data is transmitted to any external service.

## 1.4 Project Objectives

The principal objectives of this internship were:

1. **Enable live token streaming** of LLM responses so that tokens appear in the chat bubble as they are generated, reducing perceived latency from tens of seconds to under two seconds.
2. **Introduce a configuration layer** (`config.json`) so that the system prompt, context window, sampling parameters, `keep_alive` setting, and history depth can be customised without editing source code.
3. **Tune the runtime** of the local LLM – the context window, history length, output-token budget, and model-keep-alive policy – so the Copilot remains responsive on commodity hardware.

## 1.5 Methodology

The project was executed in five sequential phases. Phases 1–4 produced PR #495 (the foundation) and PR #513 (the live-streaming and conflict resolution update). Phase 5 covered upstream integration.

| Phase | Description |
|---|---|
| 1. Codebase study | Review of the existing eSim Copilot codebase (worker threads in `chatbot_thread.py`, chat UI in `Chatbot.py`); identification of UI latency, configuration rigidity, and inefficient context-window sizing as the principal gaps. |
| 2. Streaming implementation | Switching `OllamaWorker` and `OllamaVisionWorker` from a blocking single-response call to emitting individual tokens through a new `chunk_signal`, then consuming that signal in `Chatbot.py` to update an anchored bot bubble in place. |
| 3. Configuration layer | Introduction of `src/chatbot/config.json`; refactor of `chatbot_thread.py` to load this file at startup and use its values for the system prompt, context window, sampling parameters, `keep_alive`, and history depth, with graceful fall-back to built-in defaults on a missing or malformed file. |
| 4. Latency tuning | Reducing `num_ctx` from 2048 to 1024, trimming the chat-history window from 10 to 6 lines, lowering `num_predict` tiers (128/256/512 selected by question complexity), and setting `keep_alive` to `"-1m"` to avoid the model-reload cost between consecutive turns. |
| 5. Upstream integration | Resolving merge conflicts in PR #513 by accepting upstream's refactors (`_reset_session_state`, `_launch_text_worker`, `_auto_switch_model`) while preserving the streaming machinery and the configuration layer. |

# Chapter 2

# Literature Survey

## 2.1 Large Language Models

The transformer architecture introduced the self-attention mechanism, enabling parallel processing of entire sequences and effective capture of long-range dependencies. Subsequent open-weight models – LLaMA, Qwen, and Mistral – achieved task performance comparable to much larger closed models at 3–13B parameters, making local deployment on consumer hardware viable.

## 2.2 Local LLM Inference and Ollama

A 7B parameter model at full `float32` precision requires approximately 28 GB of memory. Post-training quantisation methods such as GPTQ reduce weights to 4-bit integers with minimal quality loss. The `llama.cpp` project combines quantisation with a C++ inference engine for efficient CPU execution and introduced the GGUF model format. Ollama builds on this foundation, exposing a streaming REST API (`/api/chat`) that returns generated tokens incrementally – essential for a responsive chat interface.

## 2.3 Streaming Generation in Chat Interfaces

LLM inference is autoregressive: tokens are produced one at a time and the total wall-clock time of a response grows linearly with response length. Without streaming, a chat interface must wait for the entire response before displaying anything; with streaming, each token is rendered as soon as it is emitted. This dramatically reduces *perceived* latency even when the underlying generation time is unchanged. Ollama's `/api/chat` endpoint – accessed in this project via the official `ollama` Python package – supports streaming by setting the request field `stream=True`, after which the response yields a sequence of chunks, each containing a partial token sequence.

## 2.4 Configuration-Driven Application Design

Externalising tunable behaviour into a configuration file is a well-established practice that decouples *policy* (what the system should do) from *mechanism* (the code that does it). For an AI assistant, the most impactful settings to externalise are the system prompt, the context window, the sampling parameters, the `keep_alive` policy, and the depth of conversation history sent on each turn. A JSON file is a natural choice in a Python project because it maps directly to Python dictionaries via the standard-library `json` module and is straightforward for end-users to edit. To avoid catastrophic failures when the file is hand-edited, the loader implemented in this project performs a deep merge of the on-disk file over built-in defaults, and falls back silently to those defaults on a `json.JSONDecodeError` – printing a diagnostic line to the terminal in either case.

## 2.5 Context Window in LLMs

The *context window* of an LLM is the maximum number of tokens (input plus output) that the model can attend to in a single generation. Larger context windows allow longer conversations and more retrieved context to be included, but they consume more memory and slow inference, because the attention computation is quadratic in the sequence length. Ollama exposes the context window as the `num_ctx` option. Choosing an appropriate `num_ctx` value is a deployment decision: too small, and the model forgets earlier turns; too large, and the assistant becomes sluggish on the user's machine. Measurements during this project confirmed that for the eSim Copilot's interactive workload (a 3B-parameter quantised model on a standard laptop CPU), reducing `num_ctx` from 2048 to 1024 – combined with a smaller chat history – shortened time-to-first-token noticeably without harming answer quality.

## 2.6 Qt Document Anchors for In-Place Bubble Updates

The chat panel uses a `QTextBrowser` whose document is composed of HTML tables (the bubbles). When a token arrives, the streaming bubble must be rewritten in place rather than appended to. Naive approaches based on absolute character positions are fragile: Qt re-flows the HTML document whenever the widget gains or loses focus, shifting all positions. The implementation in this project addresses this with a sentinel HTML anchor (`<a name="_stream_anchor_"></a>`) which is located on every update via a search over the document fragments. The cursor is positioned at the anchor and a selection is extended to the end of the document, then the bubble HTML is rewritten in one operation. The same pattern is used for the typing-indicator bubble.

# Chapter 3

# User Guide

This chapter describes how to install, launch, and use the enhanced eSim Copilot. All steps are documented in the order a first-time user would encounter them.

## 3.1 Prerequisites

Before launching the Copilot, ensure the following are installed on your system:

- **eSim** (version 2.x or later) installed and working.
- **Python** 3.9 or newer, with the eSim virtual environment activated.
- **Git** for cloning the chatbot branch.
- **Ollama** installed and running locally. Download from https://ollama.com.
- At least one Ollama model pulled. The Copilot defaults to a `qwen2.5`-family text model and (optionally) a vision model such as `llava` or `moondream`:

```
ollama pull qwen2.5-coder:3b
ollama pull llava       # only if image analysis is needed
```

## 3.2 Cloning the Chatbot Branch

The Copilot is maintained on a dedicated branch of the official eSim repository. The branch is cloned as shown in Listing 3.1.

```
git clone --branch eSim-Chat-Bot-Semester-Long-Internship_Autumn-2025 \
--single-branch \
https://github.com/FOSSEE/eSim.git eSim-copilot
cd eSim-copilot
```

Listing 3.1: Cloning the eSim Copilot branch.

## 3.3 Installing Python Dependencies

A dedicated virtual environment is recommended:

```
python3 -m venv .venv
source .venv/bin/activate            # Linux / macOS
# .\.venv\Scripts\Activate.ps1       # Windows (PowerShell)

python -m pip install --upgrade pip
pip install -r requirements.txt
```

Listing 3.2: Creating the Python virtual environment and installing dependencies.

The dependency set relevant to the Copilot is:

```
# LLM + UI
PyQt5
PyQt5-Qt5
PyQt5-sip
PyQtWebEngine
ollama>=0.3.0

# Optional speech-to-text path
faster-whisper>=1.0.0
SpeechRecognition>=3.10.0
PyAudio>=0.2.13
```

Listing 3.3: Dependencies relevant to the Copilot (excerpt of `requirements.txt`).

## 3.4 Starting Ollama and Pulling a Model

Ollama runs as a background service exposing an HTTP API on `localhost:11434`.

```
# Linux / macOS
curl -fsSL https://ollama.com/install.sh | sh
ollama serve &
ollama pull qwen2.5-coder:3b
```

Listing 3.4: Installing Ollama and pulling a model.

A quick smoke test (`ollama run qwen2.5-coder:3b "Hello"`) should return a response.

## 3.5 Editing `config.json`

Runtime behaviour of the Copilot is controlled by `src/chatbot/config.json`. A representative example is shown in Listing 3.5.

```
{
        "system_rules": {
                "text_system_prompt": "You are eSim Copilot, an AI assistant for the eSim EDA tool. Be practical, direct, and concise.",
                "vision_system_prompt": "You are an electronics engineer analysing schematic images from eSim or KiCad. Use the visible reference designators (R1, C3, U2, ...)."
        },
        "context_window": {
                "text_num_ctx": 1024,
                "vision_num_ctx": 1024,
                "vision_num_predict": 512
        },
        "sampling": {
                "repeat_penalty": 1.08,
                "vision_temperature": 0.15,
                "vision_repeat_penalty": 1.05
        },
        "runtime": {
                "keep_alive": "-1m"
        },
        "history": {
                "max_lines": 6
        }
}
```

Listing 3.5: Example `src/chatbot/config.json`.

Each section is explained below:

- `system_rules.text_system_prompt` – the system prompt prepended to every text-only conversation. An instructor can rewrite this to specialise the Copilot for a particular course.
- `system_rules.vision_system_prompt` – the system prompt used when an image is attached.
- `context_window.*_num_ctx` – the context window (input + output tokens) for text and vision requests.
- `context_window.vision_num_predict` – the output token cap for vision answers.

- `sampling.repeat_penalty` – discourages the model from looping on the same phrase.
- `runtime.keep_alive` – duration to keep the model resident in RAM after a request. The default `"-1m"` keeps the model loaded indefinitely so subsequent questions skip the 30–60 second reload cost.
- `history.max_lines` – how many lines of conversation history are prepended to each request.

Changes to `config.json` take effect on the next eSim launch (the file is read once at startup). On launch the loader prints a diagnostic line to the terminal: either "`[CONFIG] Loaded .../config.json`" on success, or "`[CONFIG] Failed to read config.json (...) -- using defaults.`" if the JSON is malformed.

## 3.6 Launching eSim with the Copilot

```
source .venv/bin/activate
pgrep -x ollama > /dev/null || ollama serve &
PYTHONPATH=src python src/frontEnd/
   Application.py
```

Listing 3.6: Launching eSim with the Copilot.

Open any project, click the Copilot button in the eSim toolbar (or *View* → *eSim Copilot*), and the chat panel appears. Tokens stream into the bot bubble as the model generates them.

## 3.7 Verification

The installation can be verified through two quick checks:

1. **Service check.** `curl http://localhost:11434/api/tags` returns a JSON list of installed Ollama models.
2. **End-to-end check.** Asking the Copilot "Explain Ohm's Law" should produce a streamed response whose first tokens appear within a second or two, with the bubble visibly filling word by word until the answer is complete.

# Chapter 4

# Problem Statement

## 4.1 Problem Definition

The eSim Copilot prototype inherited from the previous internship runs locally through Ollama, but it suffers from three usability gaps that prevent practical day-to-day use:

- **No live streaming in the chat panel.** Although the underlying `OllamaWorker` thread internally calls `ollama.chat(stream=True)`, it accumulates all chunks and emits a single `response_signal` only when generation finishes. The chat panel therefore shows the typing-indicator dots for the full duration, then dumps the whole answer at once.
- **Hard-coded behaviour.** The system prompt, `num_ctx`, sampling parameters, `keep_alive`, and the number of history lines sent to the model are embedded in Python source. Customising the assistant for a course, a workflow, or a different hardware budget requires editing source files.
- **Unconstrained runtime cost.** The Copilot uses a 2048-token context window, sends ten lines of history per request, and allows the model to unload after ten minutes of inactivity. On consumer CPUs this triggers an avoidable 30–60 second reload cost on the next question and pushes the KV-cache allocation higher than the conversation actually needs.

## 4.2 Specific Gaps Addressed

The three usability gaps map directly to the three contributions of this internship:

| # | Gap | Addressed by |
|---|---|---|
| 1 | No live streaming in the chat panel | A new `chunk_signal` on `OllamaWorker` / `OllamaVisionWorker`; an anchor-based bubble rewriter (`_on_stream_chunk`) in `Chatbot.py` (Section 5.2). |
| 2 | Hard-coded behaviour | `src/chatbot/config.json` loaded at startup and consumed by `chatbot_thread.py` for the system prompt, `num_ctx`, sampling, `keep_alive`, and history depth (Section 5.3). |
| 3 | Unconstrained runtime cost | `num_ctx` tuned to 1024, history trimmed from 10 to 6 lines, `num_predict` budgeted to 128/256/512 by complexity, `keep_alive` set to `"-1m"` (Section 5.4). |

# Chapter 5

# Implementation

## 5.1 System Overview

The Copilot is implemented across the following Python modules:

- `src/chatbot/chatbot_thread.py` – `OllamaWorker` and `OllamaVisionWorker` (the streaming text and vision workers), plus `MicWorker`, `OllamaStatusWorker`, `ModelFetchWorker`. This is where the configuration layer is loaded and where the streaming `chunk_signal` is emitted.
- `src/frontEnd/Chatbot.py` – the PyQt5 chat panel (`ChatbotGUI`): bubble rendering, sidebar with saved sessions, image staging, settings panel with temperature and max-tokens sliders, and the receiving side of the streaming signal (`_on_stream_chunk`).
- `src/chatbot/config.json` – new in this internship; the single source of truth for the assistant's runtime behaviour.

## 5.2 Live Token Streaming

The pre-existing `OllamaWorker` internally consumed an `ollama.chat(stream=True)` generator, but only emitted the assembled text via `response_signal` at the very end. The UI therefore had no visibility into intermediate progress.

**Worker side (`chatbot_thread.py`).** A new Qt signal `chunk_signal = pyqtSignal(str)` was added to both `OllamaWorker` and `OllamaVisionWorker`. Inside the streaming loop, each token is emitted as it arrives:

```
class OllamaWorker(QThread):
response_signal = pyqtSignal(str)
status_signal   = pyqtSignal(str)
chunk_signal    = pyqtSignal(str)   # NEW:
   emits each streamed token

def run(self):
...
```

```
                stream = ollama.chat(
                model=self.model,
                messages=messages,
                stream=True,
                options={
                        "temperature":    self.temperature,
                        "num_predict":    budget,
                        "num_ctx":        num_ctx,
                        "repeat_penalty": repeat_pen,
                        "keep_alive":     keep_alive,
                },
                )
                bot_response = ""
                for chunk in stream:
                if self._stop_requested:
                bot_response += "\n\n[stopped]"
                break
                piece = chunk["message"]["content"]
                bot_response += piece
                self.chunk_signal.emit(piece)    # NEW
                self.response_signal.emit(bot_response.strip
                   ())
```

Listing 5.1: Streaming emission inside `OllamaWorker.run()` (excerpt).

**UI side (`Chatbot.py`).** On the receiving side, the chat panel maintains a small piece of state for the in-progress bubble (`_stream_buf`, `_stream_ts`, `_stream_idx`). When the first chunk arrives, the typing indicator is removed and an empty bot bubble is inserted along with a sentinel HTML anchor. Each subsequent chunk locates the anchor, selects everything from the anchor to the end of the document, and rewrites the bubble in place:

```
                _STREAM_ANCHOR = '<a name="_stream_anchor_
                   "></a>'

                def _begin_streaming_bubble(self):
                self._remove_typing_bubble()
                self._stream_buf = ""
                self._stream_ts  = _get_time()
                self._stream_idx = self._response_counter
                cursor = QTextCursor(self.chat_display.
                   document())
                cursor.movePosition(QTextCursor.End)
                cursor.insertHtml(
                self._STREAM_ANCHOR
                + _bot_bubble("...", self._stream_ts, self.
                   _stream_idx)
                )
                self._scroll_to_bottom()
```

```
15
16                  def _on_stream_chunk(self, piece: str):
17                  if self._stream_buf is None:
18                  self._begin_streaming_bubble()
19                  self._stream_buf += piece
20                  anchor_cursor = self.
                        _find_stream_anchor_cursor()
21                  if anchor_cursor is None:
22                  return
23                  anchor_cursor.movePosition(QTextCursor.End,
                        QTextCursor.KeepAnchor)
24                  anchor_cursor.removeSelectedText()
25                  anchor_cursor.insertHtml(
26                  self._STREAM_ANCHOR
27                  + _bot_bubble(self._stream_buf, self.
                        _stream_ts, self._stream_idx)
28                  )
```

Listing 5.2: Anchor-based bubble update in `Chatbot.py` (excerpt).

When the worker finally emits `response_signal`, the `display_response` slot locates the same anchor and overwrites the in-progress bubble with the authoritative final text – guaranteeing that the visible bubble matches the text that is saved into the chat history.

The Stop button works because the worker checks `self._stop_requested` on every token; when the user clicks Stop mid-generation the partial bubble freezes with a "`[stopped]`" suffix.

## 5.3 `config.json` for Customizable System Rules

Prior to this internship, the system prompt and the model options were hard-coded constants in `chatbot_thread.py`. Editing them required a code change and a re-launch from source. The PR introduces `src/chatbot/config.json` as the single source of truth and loads it once at module import:

```
1                   import json, os
2
3                   _CONFIG_PATH = os.path.join(
4                   os.path.dirname(os.path.abspath(__file__)), "
                        config.json"
5                   )
6
7                   _DEFAULT_CONFIG = {
8                           "system_rules": {
9                                   "text_system_prompt":
                                        _DEFAULT_SYSTEM_PROMPT,
10                                  "vision_system_prompt":
                                        _DEFAULT_VISION_SYSTEM_PROMPT
                                        ,
```

```
        },
        "context_window": {
                "text_num_ctx":        1024,
                "vision_num_ctx":      1024,
                "vision_num_predict": 512,
        },
        "sampling": {
                "repeat_penalty":
                   1.08,
                "vision_temperature":
                   0.15,
                "vision_repeat_penalty":
                   1.05,
        },
        "runtime": {"keep_alive": "-1m"},
        "history": {"max_lines": 6},
}

def _deep_merge(base, override):
out = dict(base)
for k, v in (override or {}).items():
if k in out and isinstance(out[k], dict) and
   isinstance(v, dict):
out[k] = _deep_merge(out[k], v)
else:
out[k] = v
return out

def load_config():
cfg = dict(_DEFAULT_CONFIG)
try:
if os.path.isfile(_CONFIG_PATH):
with open(_CONFIG_PATH, "r", encoding="utf-8"
   ) as f:
cfg = _deep_merge(_DEFAULT_CONFIG, json.load(
   f))
print(f"[CONFIG] Loaded {_CONFIG_PATH}")
else:
print(f"[CONFIG] No config.json found --
   using defaults.")
except Exception as e:
print(f"[CONFIG] Failed to read config.json
   ({e}) -- using defaults.")
return cfg

CONFIG = load_config()
_SYSTEM_PROMPT        = CONFIG["system_rules"
   ]["text_system_prompt"]
_VISION_SYSTEM_PROMPT = CONFIG["system_rules"
```

```
][ "vision_system_prompt"]
```

Listing 5.3: Configuration loader in `chatbot_thread.py` (simplified).

Two design decisions are worth noting. First, the loader performs a *deep merge* of the user's file over the built-in defaults; a user who supplies only `"runtime": {"keep_alive": "5m"}` still gets the rest of the config from the defaults rather than seeing the assistant break. Second, any `JSONDecodeError` (typically caused by hand-edits that leave a stray quote or an unescaped newline) is caught, logged, and the defaults are used – the application never fails to start because of a bad config file.

The values in `CONFIG` are then used inside the worker's `run()` method to construct each Ollama request:

```
                max_lines   = int(CONFIG["history"]["max_lines
                    "])
                num_ctx     = int(CONFIG["context_window"]["
                    text_num_ctx"])
                repeat_pen  = float(CONFIG["sampling"]["
                    repeat_penalty"])
                keep_alive  = CONFIG["runtime"]["keep_alive"]

                messages = [{"role": "system", "content":
                    _SYSTEM_PROMPT}]
                for line in self.chat_history[-max_lines:]:
                if line.startswith("User:"):
                messages.append({"role": "user",       "
                    content": line[5:].strip()})
                elif line.startswith("Bot:"):
                messages.append({"role": "assistant", "
                    content": line[4:].strip()})

                stream = ollama.chat(
                model=self.model,
                messages=messages,
                stream=True,
                options={
                        "temperature":    self.temperature,
                        "num_predict":    _smart_num_predict(
                            self.chat_history,
                        self.num_predict),
                        "num_ctx":        num_ctx,
                        "repeat_penalty": repeat_pen,
                        "keep_alive":     keep_alive,
                },
                )
```

Listing 5.4: Consuming `CONFIG` inside the streaming chat request (excerpt).

Figures 5.1 and 5.2 illustrate the same Copilot answering the same question under two different `system_rules.text_system_prompt` values – no source change

was needed between the two screenshots.

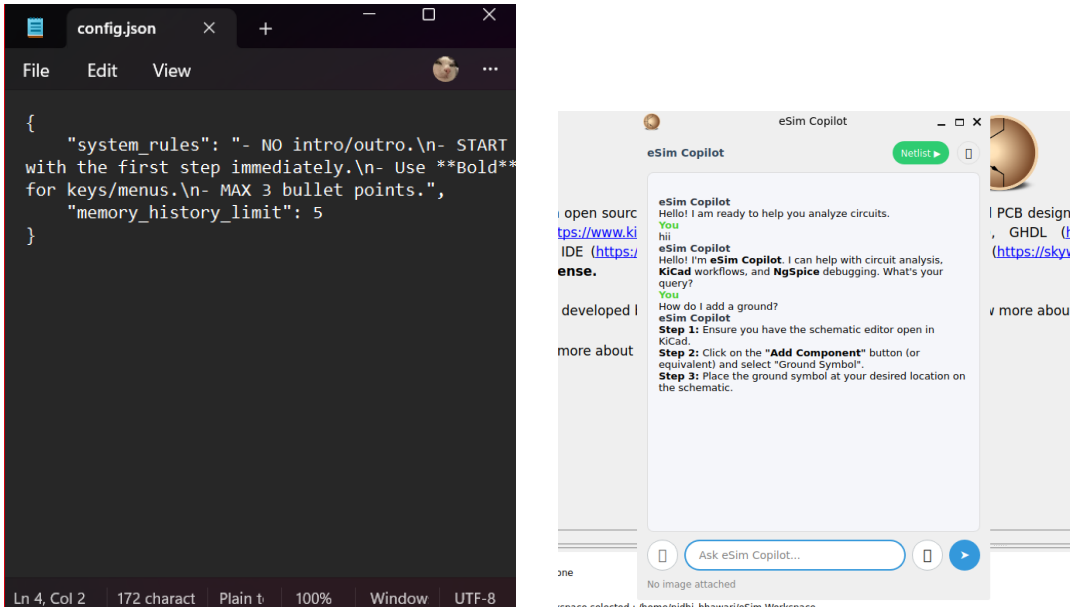


Figure 5.1: Concise mode. *Left:* `config.json` with rules instructing the Copilot to omit introductions, start with the first step, and cap the answer at three bullets. *Right:* the resulting stepwise response in the eSim Copilot panel.

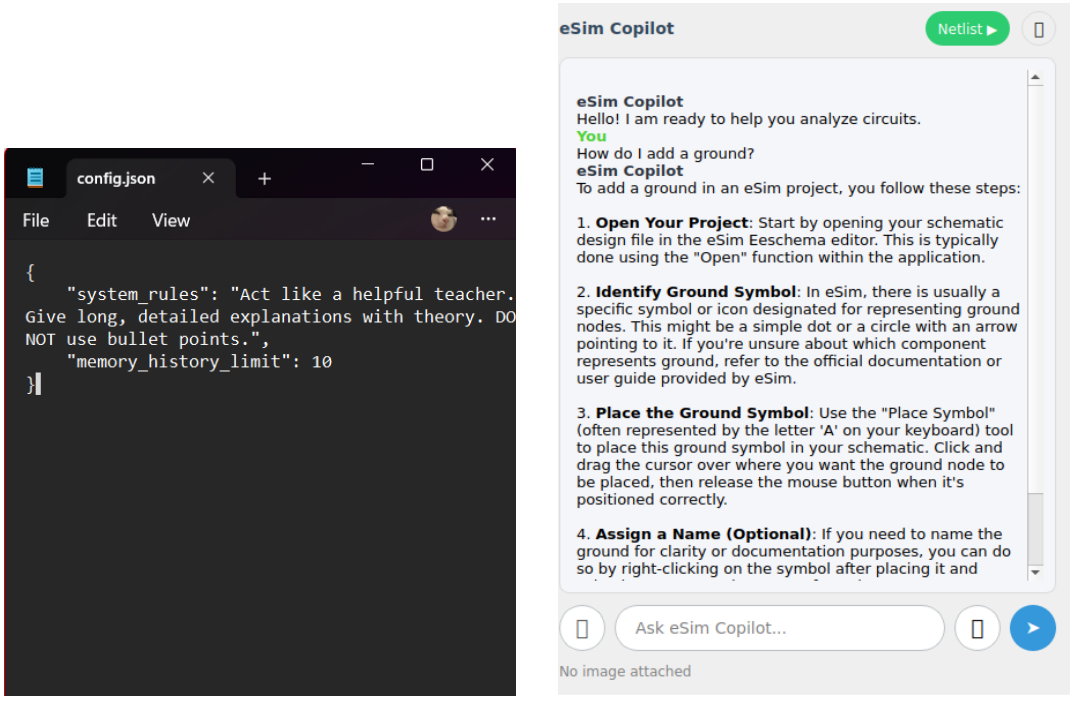


Figure 5.2: Teacher mode. *Left:* `config.json` with rules asking for long, detailed teacher-style explanations and forbidding bullet points. *Right:* the resulting verbose, numbered response in the eSim Copilot panel.

## 5.4 Latency Tuning

On top of the streaming and configuration work, four small tuning changes were applied that together cut the perceived response time on a standard laptop CPU running `qwen2.5-coder:3b`:

1. `num_ctx` **2048 → 1024.** The chat panel only sends 6 history lines plus the system prompt and the current question; 1024 tokens are sufficient and the smaller KV cache means the model produces its first token sooner.

2. **History 10 → 6 lines.** Older turns rarely affect the answer to a specific eSim question and they consume tokens that the model must re-process every turn.

3. **Smart** `num_predict` **budget.** A small classifier scans the recent user messages for keywords and picks one of three tiers: 128 tokens for definitional questions ("what is a netlist"), 512 tokens for complex/long questions (simulation errors, schematic descriptions), and 256 tokens for everything in between. The user's slider value is treated as an upper bound.

4. `keep_alive "10m"` → `"-1m".` Ollama's default behaviour is to unload the model after a configurable idle interval. Setting `keep_alive` to `"-1m"` pins the model in RAM indefinitely, so the second question in a conversation does not pay the 30–60 second cold-start cost of reloading ˜2 GB of quantised weights from disk.

All four knobs are exposed in `config.json`, so a user on a memory-constrained machine can raise the unload threshold (`"5m"`) or lower `num_ctx` further.

## 5.5 Upstream Integration

After the streaming and configuration work was published as PR #513, the upstream `eSim-Chat-Bot-Semester-Long-Internship_Autumn-2025` branch advanced in parallel: scaffolding files (`chatbot_core.py`, `ollama_runner.py`, `image_handler.py`, `stt_handler.py`) were removed and the chat panel was refactored to use helper methods (`_reset_session_state`, `_launch_text_worker`, `_auto_switch_model`). This produced merge conflicts in `src/chatbot/chatbot_thread.py`, `src/frontEnd/Chatbot.py`, and `src/frontEnd/DockArea.py`.

The conflicts were resolved by accepting upstream's refactors and then re-applying the streaming and configuration features on top. Specifically:

- The `_launch_text_worker` and `_launch_vision_worker` helpers were modified to also connect `chunk_signal` → `_on_stream_chunk`, so live streaming continues to work everywhere a worker is started (chat send, netlist analysis, retry, regenerate, debug).

- The `_reset_session_state` helper was extended to call `_reset_stream_state` at the end, so the streaming-bubble state is also cleared on "New Chat" and "Clear".

- The vision worker's options dict was kept on its config-driven form (rather than upstream's hard-coded values), so `config.json` continues to control image-analysis behaviour as well.
- Upstream's deletion of the four scaffolding files was accepted, since the live code path uses `chatbot_thread.py` and `Chatbot.py`; no functionality is lost.

After resolution, all three files parse cleanly (`python -c "import ast; ast.parse(open(...).r` and end-to-end testing in the eSim GUI confirmed that streaming, the `config.json` layer, and the latency tuning all remain functional.

## 5.6 Summary of Contributions

The table below provides a consolidated view of the contributions delivered during this internship.

## 5.7 Project Artifacts

The complete source for the contributions described above is available on GitHub at the following links:

- **Repository (branch):**
  https://github.com/FOSSEE/eSim/tree/eSim-Chat-Bot-Semester-Long-Internship_Autumn-2025
- **Pull Request #495** – foundation (streaming, `config.json` layer, context-window tuning):
  https://github.com/FOSSEE/eSim/pull/495
- **Pull Request #513** – live token streaming in the chat panel, the working `config.json` integration, latency tuning, and upstream conflict resolution:
  https://github.com/FOSSEE/eSim/pull/513

Table 5.1: Complete work summary – eSim Copilot Performance Optimisation & Configuration Layer (PRs #495 and #513).

| **Contribution** | **Status** | **Files touched / details** |
|---|---|---|
| Live token streaming (text) | Implemented | `OllamaWorker` in `chatbot_thread.py` emits a new `chunk_signal` per token; `Chatbot.py` consumes it in `_on_stream_chunk` and rewrites the bot bubble in place via a sentinel anchor. |
| Live token streaming (vision) | Implemented | `OllamaVisionWorker` emits the same `chunk_signal` so image-analysis answers also stream token by token. |
| Stop mid-generation | Implemented | The streaming loop checks `_stop_requested` per token; the partial bubble freezes with a "`[stopped]`" suffix when the user clicks the Stop button. |
| `config.json` layer | Implemented | New `src/chatbot/config.json`; loaded by `chatbot_thread.py` at startup; values used for the system prompt, `num_ctx`, sampling parameters, `keep_alive`, and history depth. Graceful fall-back to defaults on missing/malformed JSON. |
| Customisable system rules | Implemented | `system_rules.text_system_prompt` and `vision_system_prompt` fields in `config.json` replace the hard-coded prompts. |
| Latency tuning | Implemented | `num_ctx` 2048 → 1024; history window 10 → 6 lines; smart `num_predict` tiers (128/256/512 by question complexity); `keep_alive` `"10m"` → `"-1m"`. |
| Upstream integration | Implemented | Resolved merge conflicts in PR #513 across `chatbot_thread.py`, `Chatbot.py`, and `DockArea.py`; streaming hookups added to upstream's helper methods. |

Total: 7 contributions implemented across the Copilot stack; source files modified: `src/chatbot/chatbot_thread.py`, `src/frontEnd/Chatbot.py`, `src/frontEnd/DockArea.py`; new file added: `src/chatbot/config.json`; supporting files updated as part of upstream integration: `src/chatbot/__init__.py` and `src/frontEnd/Application.py`.

# Chapter 6

# Conclusion and Future Scope

## Conclusion

This project delivered three concrete improvements to the eSim Copilot, all built on free and open-source components. First, the chat panel was made truly streaming: each token is rendered in its bot bubble as it is emitted by the model, instead of the user waiting in silence for the entire answer. The implementation uses a small Qt signal (`chunk_signal`) on the worker side and an anchor-based bubble rewriter on the UI side, both of which survive Qt's document re-flows and the user clicking away mid-generation. Second, a configuration layer (`src/chatbot/config.json`) was introduced as the single source of truth for the Copilot's runtime behaviour – the system prompt, the context window, the sampling parameters, the `keep_alive` policy, and the history depth. The loader merges the user's file over built-in defaults and falls back gracefully on malformed JSON, so the assistant never fails to start because of a hand-edit. Third, the runtime itself was tuned: a smaller context window, a smaller history slice, a complexity-based output budget, and an indefinite `keep_alive` together cut the visible delay on a standard laptop without harming answer quality. Across both PR #495 (the foundation) and PR #513 (the live-streaming and conflict resolution update), the Copilot now feels noticeably faster, is straightforward to customise, and is predictable on a wider range of hardware.

## Future Scope

1. **Retrieval-Augmented Generation (RAG).** Indexing the eSim documentation in a local vector database (e.g., ChromaDB) and fetching the top-$k$ chunks at query time would ground the Copilot's answers in the official manual and reduce hallucination on tool-specific questions.
2. **In-app config editor.** A small settings dialog backed by `config.json`, so that users can change the system prompt and the runtime options without leaving eSim and without hand-editing JSON.
3. **Per-project configuration.** Allowing a project-level `config.json` to override the global one, so that a teacher can ship a course-specific Copilot persona alongside a lab kit.

4. **Live config reload.** A "Reload config" menu item that re-runs `load_config()` without restarting eSim, so that edits to `config.json` take effect immediately.

5. **Quantitative evaluation.** An automated benchmark that measures time-to-first-token and total response time across a curated set of eSim questions, so that regressions are caught when the model, prompt, or `num_ctx` change.

6. **Containerisation.** Packaging the Copilot stack (Ollama, model weights, Python environment) into a Docker image for reproducibility across operating systems.

# Bibliography


[1] FOSSEE Team, IIT Bombay. *eSim: An Open Source EDA Tool for Circuit Design, Simulation, Analysis and PCB Design*. Available: https://esim.fossee.in/ [Accessed May 2026]

[2] FOSSEE Team, IIT Bombay. *eSim Chat-Bot – Semester Long Internship Autumn 2025 (branch)*. Available: https://github.com/FOSSEE/eSim/tree/eSim-Chat-Bot-Semester-Long-Internship_Autumn-2025 [Accessed May 2026]

[3] nidhiii128. *[eSim Copilot] Performance Optimization & Configuration Layer – Pull Request #495*. Available: https://github.com/FOSSEE/eSim/pull/495 [Accessed May 2026]

[4] nidhiii128. *Update: Live streaming, working config.json layer, latency tuning – Pull Request #513*. Available: https://github.com/FOSSEE/eSim/pull/513 [Accessed May 2026]

[5] Vaswani, A. et al. (2017). *Attention Is All You Need*. arXiv:1706.03762. Available: https://arxiv.org/abs/1706.03762

[6] Brown, T. et al. (2020). *Language Models are Few-Shot Learners*. arXiv:2005.14165. Available: https://arxiv.org/abs/2005.14165

[7] Ollama (2023). *Ollama: Get up and running with large language models locally*. Available: https://ollama.com/ [Accessed May 2026]

[8] Ollama (2024). *API Reference – `/api/chat` and the `num_ctx`, `keep_alive` options*. Available: https://github.com/ollama/ollama/blob/main/docs/api.md [Accessed May 2026]

[9] Gerganov, G. (2023). *llama.cpp: Inference of Meta's LLaMA model in pure C/C++*. Available: https://github.com/ggerganov/llama.cpp [Accessed May 2026]

[10] Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. (2022). *GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers*. arXiv:2210.17323. Available: https://arxiv.org/abs/2210.17323

[11] Riverbank Computing Limited (2023). *PyQt5 Reference Guide – `QTextBrowser` and `QTextCursor`*. Available: https://www.riverbankcomputing.com/static/Docs/PyQt5/ [Accessed May 2026]

[12] KiCad Development Team (2024). *KiCad EDA: Open Source Electronics Design Automation*. Available: https://www.kicad.org/ [Accessed May 2026]

[13] Ngspice Project. *Ngspice – Open Source Mixed-Mode/Mixed-Level Circuit Simulator*. Available: https://ngspice.sourceforge.io/ [Accessed May 2026]